Python ile Bilimsel Hesaplama

Açık Matematik

Açık Matematik — açık kaynaklı, reklamsız Türkçe matematik notları
acik-matematik.com
Bu çalışma Creative Commons BY-NC-SA 4.0 lisansıyla paylaşılmıştır: kaynak göstererek ve aynı lisansla, ticari olmayan amaçlarla kopyalayabilir, dağıtabilir ve uyarlayabilirsiniz.
Bu sürüm: 4 Kasım 2026

İçindekiler

Giriş 8

1 Python ile İlk Adımlar: Sayılar, Değişkenler ve İfadeler 9
1.1 Python'u Çalıştırmak 9
1.2 İfadeler ve İşlem Önceliği 10
1.3 Değişkenler ve Atama 11
1.4 Çıktıyı Biçimlendirmek 13
1.5 Modüller ve math Modülü 15
1.6 Tam Sayılar 16
1.7 Tam Bölme ve Kalan 17
1.8 Kayan Noktalı Sayılar 19
1.9 Makine Epsilonu ve Sayıları Karşılaştırmak 21
1.10 Karmaşık Sayılar 24
1.11 Tam Kesirler 27
1.12 Ondalık Hesap 29
1.13 Alıştırmalar 30

2 Kontrol Yapıları ve Fonksiyonlar 37
2.1 Koşullar ve if Deyimi 37
2.2 for Döngüsü ve range 40
2.3 while Döngüsü 44
2.4 break, continue ve else 47
2.5 Fonksiyonlar 50
2.6 Özyineleme 56
2.7 Alıştırmalar 59

3 Veri Yapıları ve Comprehension 68
3.1 Listeler 68
3.2 Dilimleme 70
3.3 Değiştirilebilirlik ve Takma Adlar 72
3.4 Demetler 75
3.5 Sözlükler 76
3.6 Kümeler 78
3.7 Comprehension 80
3.8 zip, enumerate ve sorted 82
3.9 itertools ile Sayma 85
3.10 Eratosthenes Kalburu 87
3.11 Alıştırmalar 90

 CC BY-NC-SA 4.0

4 Sınıflar, Hata Yakalama ve Dosyalar . . . 98
4.1 Modüller ve import . . . 98
4.2 Hatalar ve istisnalar . . . 101
4.3 Dosyalarla çalışmak . . . 108
4.4 Sınıflar . . . 113
4.5 Veri sınıfları: dataclass . . . 120
4.6 Hepsi bir arada: bir ölçüm dosyasını işlemek . . . 122
4.7 Alıştırmalar . . . 125

5 NumPy Dizileri . . . 134
5.1 Neden NumPy? . . . 134
5.2 Dizi Kurmak . . . 135
5.3 Eşit Aralıklı Noktalar . . . 137
5.4 Veri Türü . . . 139
5.5 Şekil ve Yeniden Şekillendirme . . . 141
5.6 İndeksleme ve Dilimleme . . . 143
5.7 Görünüm ve Kopya . . . 146
5.8 Fancy İndeksleme . . . 148
5.9 Boolean İndeksleme . . . 150
5.10 Dizileri Birleştirme . . . 153
5.11 Alıştırmalar . . . 155

6 Vektörizasyon ve Broadcasting . . . 163
6.1 Evrensel Fonksiyonlar . . . 163
6.2 İndirgeme İşlemleri ve axis Parametresi . . . 166
6.3 Boolean Maskeler ve np.where . . . 169
6.4 Broadcasting . . . 174
6.5 Uygulama: Uzaklık Matrisi . . . 179
6.6 Döngü mü, Vektör mü? . . . 180
6.7 Uygulama: Sayısal Türev . . . 182
6.8 Alıştırmalar . . . 184

7 NumPy ile Lineer Cebir . . . 191
7.1 Matris Çarpımı ve Temel İşlemler . . . 191
7.2 Lineer Sistemlerin Çözümü . . . 194
7.3 Ters Matris, Determinant ve Rank . . . 196
7.4 Vektör ve Matris Normları . . . 197
7.5 Koşul Sayısı . . . 199
7.6 Neden Ters Matrisle Çözmüyoruz? . . . 202
7.7 Özdeğerler ve Özvektörler . . . 204
7.8 Simetrik Matrisler . . . 206
7.9 En Küçük Kareler . . . 208

 CC BY-NC-SA 4.0

7.10 Tekil Değer Ayrışımı . . . 210
7.11 Alıştırmalar . . . 213

8 Matplotlib ile Grafik Çizimi . . . 222
8.1 Figure ve Axes . . . 222
8.2 Çizgiler, Renkler ve Açıklamalar . . . 224
8.3 Alt Grafikler ve Logaritmik Eksenler . . . 225
8.4 Parametrik ve Kutupsal Eğriler . . . 229
8.5 Saçılım Grafikleri ve Histogramlar . . . 230
8.6 Kontur Çizimleri . . . 234
8.7 Üç Boyutlu Yüzeyler . . . 237
8.8 Grafikleri Kaydetmek . . . 239
8.9 Alıştırmalar . . . 240

9 SymPy ile Sembolik Hesap . . . 249
9.1 Sayısal ve Sembolik Hesap . . . 249
9.2 Semboller ve İfadeler . . . 250
9.3 Varsayımlar . . . 252
9.4 İfadeleri Dönüştürmek . . . 253
9.5 Limit . . . 255
9.6 Türev . . . 257
9.7 İntegral . . . 258
9.8 Seriler ve Toplamlar . . . 261
9.9 Denklem Çözmek . . . 262
9.10 Diferansiyel Denklem Çözmek . . . 264
9.11 Matrisler . . . 265
9.12 Sembolikten Sayısala: lambdify . . . 267
9.13 Alıştırmalar . . . 269

10 Kök Bulma ve Optimizasyon . . . 276
10.1 Kökleri Ayırmak: İşaret Değişimi . . . 276
10.2 İkiye Bölme Yöntemini Yazmak . . . 278
10.3 Newton Yöntemini Yazmak . . . 279
10.4 SciPy ile Tek Değişkenli Denklemler . . . 283
10.5 Denklem Sistemleri . . . 288
10.6 Tek Değişkenli Optimizasyon . . . 290
10.7 Çok Değişkenli Optimizasyon . . . 292
10.8 Kısıtlı Optimizasyon . . . 295
10.9 Alıştırmalar . . . 297

11 Sayısal Türev ve İntegral . . . 307
11.1 Fark Bölümleriyle Türev . . . 307
11.2 Adımı Seçmek: Kesme ve Yuvarlama Hatası . . . 308

 CC BY-NC-SA 4.0

11.3 Izgarada Türev: np.gradient . . . 313
11.4 Yamuk ve Simpson Kuralları . . . 315
11.5 Örneklenmiş Veriyle İntegral . . . 318
11.6 Uyarlamalı İntegral: quad . . . 321
11.7 Katlı İntegraller: dblquad ve tplquad . . . 327
11.8 Alıştırmalar . . . 330

12 Polinomlar, İnterpolasyon ve Eğri Uydurma . . . 339
12.1 NumPy'da Polinomlar . . . 339
12.2 Kökler . . . 341
12.3 Köklerin Duyarlılığı . . . 343
12.4 İnterpolasyon Polinomu . . . 344
12.5 Runge Olayı . . . 348
12.6 Çebişev Düğümleri . . . 349
12.7 Parçalı Polinomlar ve Spline . . . 353
12.8 En Küçük Kareler ile Polinom Uydurma . . . 357
12.9 Doğrusal Olmayan Modeller . . . 360
12.10 Alıştırmalar . . . 363

13 Diferansiyel Denklemlerin Sayısal Çözümü . . . 372
13.1 Başlangıç Değer Problemi ve Yön Alanı . . . 372
13.2 Euler Yöntemi . . . 373
13.3 Dördüncü Mertebe Runge–Kutta Yöntemi . . . 377
13.4 Adım Uzunluğu ve Hata . . . 379
13.5 SciPy ile Çözüm: solve_ivp . . . 382
13.6 Katı Denklemler . . . 387
13.7 Denklem Sistemleri ve Yüksek Mertebeden Denklemler . . . 389
13.8 Faz Düzlemi . . . 391
13.9 Lotka–Volterra Av-Avcı Modeli . . . 392
13.10 Sarkaç . . . 395
13.11 Alıştırmalar . . . 398

14 Rastgele Sayılar ve Monte Carlo Yöntemleri . . . 408
14.1 Sözde Rastgele Sayılar . . . 408
14.2 Temel Dağılımlardan Örneklem . . . 410
14.3 Histogram ve Ters Dönüşüm Yöntemi . . . 412
14.4 Olasılıkları ve Beklenen Değerleri Kestirmek . . . 415
14.5 Büyük Sayılar Yasası . . . 416
14.6 Merkezi Limit Teoremi . . . 417
14.7 Monte Carlo ile İntegral . . . 420
14.8 Monte Carlo Ne Zaman Kazanır? . . . 424
14.9 Rastgele Yürüyüş . . . 427
14.10 Alıştırmalar . . . 429

 CC BY-NC-SA 4.0

15 Pandas ile Veri Analizi 437
15.1 Series: Etiketli Bir Boyutlu Dizi 437
15.2 DataFrame: Sütunları Adlandırılmış Tablo 442
15.3 CSV Dosyaları 444
15.4 Satır ve Sütun Seçmek 448
15.5 Eksik Değerler 451
15.6 Betimsel İstatistikler 454
15.7 Gruplama: groupby 457
15.8 Tabloları Birleştirmek 462
15.9 Zaman Serileri 465
15.10 Alıştırmalar 468

 CC BY-NC-SA 4.0

Giriş

Python ile Bilimsel Hesaplama, Python'ı matematik yapmak için bir araç olarak öğretir. Her bölüm aracı, okurun zaten tanıdığı bir matematik problemi üzerinden tanıtır: asal sayılar ve en büyük ortak bölen ile dilin temelleri, matrisler ve özdeğerlerle NumPy, kök bulma, integral ve diferansiyel denklemlerle SciPy, simülasyonlarla olasılık. Notlardaki her kod çalıştırılmış ve çıktısıyla birlikte verilmiştir; okur kodu kendi bilgisayarında ya da tarayıcıda çalışan bir Jupyter ortamında aynen deneyebilir.

> **i** Not
>
> Kodların arkasındaki matematik için Nümerik Analiz, Lineer Cebir, Diferansiyel Denklemler ve Olasılık Teorisi notlarına bakılabilir.

CC BY-NC-SA 4.0

1. Python ile İlk Adımlar: Sayılar, Değişkenler ve İfadeler

2^{100} sayısının bütün basamaklarını yazmak, 100!'in kaç basamaklı olduğunu bulmak, $\sqrt{2}$'nin elli basamağını hesaplamak ya da $\left(1+\frac{1}{n}\right)^n$ dizisinin e'ye ne hızla yaklaştığını görmek elle uzun sürer; bilgisayarla ise birkaç satırlık iştir. Bu ders Python'ı tam olarak böyle bir araç olarak öğretir: her yeni yapıyı, zaten bildiğimiz bir matematik problemi üzerinde göreceğiz.

İlk bölümde Python'ı gelişmiş bir hesap makinesi gibi kullanacağız. Ama bu hesap makinesi farklı sayı türleriyle çok farklı davranır. Tam sayılarla istediğimiz büyüklükte ve hiç hata yapmadan hesap yapar; ondalıklı sayılarla ise hemen her işlemde küçük bir yuvarlama hatası yapar. $0{,}1 + 0{,}2$ toplamının Python'da tam olarak $0{,}3$ çıkmamasının nedeni budur.

Bölümün asıl amacı, hangi sayı türünün ne zaman kullanılacağını ve bir sonuca ne kadar güvenilebileceğini öğrenmektir. Bunun için tam sayıları, kayan noktalı sayıları, karmaşık sayıları, tam kesirleri ve ondalık sayıları sırayla ele alacağız.

1.1 Python'u Çalıştırmak

Python kodu birkaç farklı ortamda çalıştırılabilir; hepsinde dil aynıdır, yalnız kodu yazıp sonucu görme biçimi değişir.

Python, okunaklı söz dizimiyle bilinen genel amaçlı bir programlama dilidir. Python **yorumlanan** (interpreted) bir dildir: yazdığımız kodu **yorumlayıcı** (interpreter) adı verilen program doğrudan çalıştırır; kullanıcının ayrıca yürüttüğü bir derleme adımı yoktur. Bilimsel hesaptaki yaygınlığını büyük ölçüde NumPy, SciPy, SymPy, Matplotlib ve pandas kütüphanelerine borçludur; bu kütüphaneleri ilerleyen bölümlerde tek tek göreceğiz.

Python kodunu çalıştırmanın üç yaygın yolu vardır:

- **Etkileşimli yorumlayıcı.** Terminalde `python` yazıldığında `>>>` istemi belirir. Bu isteme yazılan her ifade hemen hesaplanır ve değeri bir alt satırda gösterilir. Kısa denemeler için en hızlı yoldur.
- **Betik.** Kod `ilk.py` gibi `.py` uzantılı bir dosyaya yazılır ve terminalde `python ilk.py` komutuyla baştan sona çalıştırılır. Betikte ifadelerin değeri kendiliğinden gösterilmez; görmek istediğimizi `print` ile yazdırırız.
- **Jupyter defteri.** Kodun, açıklamaların ve çıktıların hücreler hâlinde bir arada durduğu bir belgedir. Google Colab, Jupyter defterlerini hiçbir kurulum yapmadan tarayıcıda çalıştırır.

Etkileşimli yorumlayıcıda kısa bir oturum şöyle görünür:

```
>>> 2 + 3 * 4
14
>>> 7 / 2
3.5
>>> 2 ** 10
1024
```

> **i** Kurulum hakkında kısa bir not
>
> Python, python.org adresinden ya da bilimsel kütüphanelerle birlikte gelen Anaconda dağıtımıyla kurulabilir. Kurulumla uğraşmadan başlamak için Google Colab yeterlidir. Bu notlardaki kodlar Python 3.12, NumPy 2.5, SciPy 1.18, SymPy 1.14, Matplotlib 3.11 ve pandas 3.0 ile çalıştırılmıştır.

Bu notlarda kod betik olarak verilir; hemen altında, "Çıktı:" satırından sonra, kodun çalıştırıldığında ekrana yazdığı her şey aynen gösterilir. İlk programımız şudur:

 CC BY-NC-SA 4.0

```
# İlk programımız: bir selam ve basit bir hesap
print("Merhaba, matematik!")
print(2 + 3 * 4)
```

Çıktı:

```
Merhaba, matematik!
14
```

`#` işaretinden satır sonuna kadar olan kısım bir **yorumdur** (comment); Python onu okumaz, yalnız okura açıklama yapar. `print` parantez içinde verilen değerleri ekrana yazar. Tırnak içindeki `"Merhaba, matematik!"` gibi metinlere **dize** (string) denir; dizeler Türkçe karakter de içerebilir.

1.2 İfadeler ve İşlem Önceliği

Python'a verdiğimiz her hesap bir ifadedir; önce ifadenin ne olduğunu ve Python'un onu hangi sırayla hesapladığını görelim.

Tanım 1.1 (İfade). Sayılardan, değişken adlarından, işlem işaretlerinden ve fonksiyon çağrılarından oluşan ve bir değeri olan kod parçasına **ifade** (expression) denir. Python bir ifadeyi **değerlendirerek** (evaluate) onu tek bir değere indirger.

Yani `2 + 3 * 4` bir ifadedir ve değeri `14`'tür; `print(2 + 3 * 4)` ise bu değeri ekrana yazan bir komuttur. Etkileşimli yorumlayıcı her ifadenin değerini kendiliğinden gösterir, betik göstermez.
Python'un aritmetik işlemleri aşağıdaki tablodadır.

İşlem	Python	Örnek	Değer
toplama	`+`	`7 + 2`	`9`
çıkarma	`-`	`7 - 2`	`5`
çarpma	`*`	`7 * 2`	`14`
bölme	`/`	`7 / 2`	`3.5`
tam bölme	`//`	`7 // 2`	`3`
kalan	`%`	`7 % 2`	`1`
üs alma	`**`	`7 ** 2`	`49`

İşlem önceliği matematikteki gibidir: önce üs alma, sonra çarpma ve bölmeler, en son toplama ve çıkarmalar yapılır; parantez içi her zaman önce hesaplanır. Aynı öncelikteki işlemler soldan sağa yapılır. Tek istisna üs almadır: `2 ** 3 ** 2` ifadesi $2^{3^2} = 2^9$ olarak, yani sağdan sola hesaplanır. Üs alma, önündeki eksi işaretinden de önce gelir; `-2 ** 2` ifadesi, matematikteki -2^2 gibi, $-(2^2) = -4$'tür.

```
print(2 + 3 * 4)     # önce çarpma yapılır
print((2 + 3) * 4)   # parantez önceliği değiştirir
print(2 ** 3 ** 2)   # üs sağdan sola: 2 ** 9
print(-2 ** 2)       # -(2 ** 2)
print((-2) ** 2)
print(10 - 4 - 3)    # soldan sağa: (10 - 4) - 3
print(7 / 2, 6 / 3)  # / her zaman ondalıklı sonuç verir
```

Çıktı:

```
14
20
512
-4
4
3
3.5 2.0
```

 CC BY-NC-SA 4.0

`print`'e virgülle ayrılmış birden çok değer verilirse onları aralarına birer boşluk koyarak aynı satıra yazar. Son satırda `6 / 3` işleminin `2` değil `2.0` verdiğine dikkat edin: `/` işlemi, bölüm tam sayı olsa bile ondalıklı bir sayı üretir. Tam sayı sonucu veren `//` işlemini Bölüm 1.7 kısmında göreceğiz.

⚠ Şapka işareti üs alma değildir

Birçok hesap makinesinde ve LaTeX'te 2^3 için `2^3` yazılır, ama Python'da `^` bit düzeyinde "dışlayıcı ya da" (XOR) işlemidir ve hata vermeden bambaşka bir sayı üretir. Üs almak için daima `**` kullanılır.

```
print(2 ^ 3)    # XOR: 10 ve 11 bit dizilerinden 01 çıkar
print(2 ** 3)   # üs alma
```

Çıktı:

```
1
8
```

Örnek 1.1 (Kesirli Bir İfadeyi Python'a Çevirmek). $\dfrac{2^{10}-1}{3\cdot 11}$ sayısını Python ile hesaplayınız.

Çözüm.
Doğrudan yazım. İfadeyi kâğıttaki sırayla `2 ** 10 - 1 / 3 * 11` diye yazarsak öncelik kuralları yüzünden Python önce $1/3$'ü hesaplar, sonucu 11 ile çarpar ve bunu 2^{10}'dan çıkarır:

```
# Yanlış: yalnız 1 / 3 bölünür, sonra 11 ile çarpılır
print(2 ** 10 - 1 / 3 * 11)
```

Çıktı:

```
1020.3333333333334
```

Parantezli yazım. Kesir çizgisi payı ve paydayı birer bütün yapar; Python'da bunu parantezle söyleriz:

```
# Doğru: pay ve payda ayrı ayrı parantez içinde
print((2 ** 10 - 1) / (3 * 11))
```

Çıktı:

```
31.0
```

Sağlama. $2^{10}-1 = 1023 = 3\cdot 11\cdot 31$ olduğundan aranan değer gerçekten 31'dir. ■

1.3 Değişkenler ve Atama

Bir değeri tekrar tekrar kullanmak ya da adım adım güncellemek için ona bir ad veririz.

Tanım 1.2 (Değişken ve Atama). `ad = ifade` biçimindeki komuta **atama** (assignment) denir. Python sağdaki ifadeyi değerlendirir ve elde ettiği değeri soldaki ada bağlar. Bir değere bağlanmış ada **değişken** (variable) denir.

Yani `=` işareti matematikteki eşitlik değil, bir emirdir: "sağdakini hesapla, sonucu bu adla sakla". Bu yüzden matematikte anlamsız olan `n = n + 1` Python'da anlamlıdır: `n`'nin şimdiki değerine 1 ekler ve sonucu yine `n` adına bağlar. İki değerin eşit olup olmadığını sınamak için `==` kullanılır; sonuç `True` (doğru) ya da `False` (yanlış) olur.

 CC BY-NC-SA 4.0

```
r = 2.5                    # yarıçap
pi = 3.141592653589793
area = pi * r ** 2         # dairenin alanı
perimeter = 2 * pi * r     # dairenin çevresi
print(area, perimeter)
n = 10
n = n + 1                  # n'nin eski değerine 1 ekle
n += 1                     # aynı işin kısa yazılışı
print(n, n == 12)
```

Çıktı:

```
19.634954084936208 15.707963267948966
12 True
```

`n += 1` yazımı `n = n + 1`'in kısaltmasıdır; aynı biçimde `-=`, `*=`, `/=`, `//=`, `%=` ve `**=` de vardır.
Değişken adları harf, rakam ve alt çizgiden (_) oluşur ve rakamla başlayamaz. Büyük ve küçük harf ayrıdır: `x` ile `X` iki ayrı değişkendir. `for`, `if`, `def`, `lambda` gibi Python'un kendi **anahtar kelimeleri** ad olarak kullanılamaz. Türkçe harfler kurallara aykırı değildir, ama bu notlarda değişken adları İngilizce ya da matematikteki alışılmış adlardır (`x`, `n`, `f`, `A`, `tol`). `print`, `sum`, `max` gibi hazır fonksiyonların adlarını değişkenlere vermekten de kaçınmak gerekir; aksi hâlde o ad artık fonksiyonu değil, bizim değerimizi gösterir.
Python'da bir değişkenin türü yoktur, değerin türü vardır: aynı ad önce bir tam sayıya, sonra bir ondalıklı sayıya bağlanabilir. Bir değerin türünü `type` fonksiyonu söyler.

```
x = 7
print(type(x))
x = 7 / 2              # aynı ad şimdi başka türde bir değere bağlı
print(type(x))
print(type(3 + 4j), type("yedi"), type(7 > 2))
```

Çıktı:

```
<class 'int'>
<class 'float'>
<class 'complex'> <class 'str'> <class 'bool'>
```

Bu bölümde `int` (tam sayı), `float` (kayan noktalı sayı) ve `complex` (karmaşık sayı) türlerini ayrıntılı göreceğiz. `str` dize türüdür; `bool` ise yalnız `True` ve `False` değerlerinden oluşan **mantıksal** türdür: `7 > 2` karşılaştırmasının değeri `True`'dur.
Birden çok değişkene aynı satırda atama yapılabilir: `a, b = 1, 2` yazımı `a`'ya 1'i, `b`'ye 2'yi bağlar. Bu **eşzamanlı atamada** sağ taraftaki bütün ifadeler, hiçbir ad değişmeden önce hesaplanır. Bu yüzden `a, b = b, a` iki değişkenin değerlerini takas eder.

> **i** Bir önizleme: for döngüsü
>
> Döngüleri Kontrol Yapıları ve Fonksiyonlar bölümünde ayrıntılı göreceğiz. Bu bölümdeki birkaç örnek için şu kadarı yeterli: `for k in range(a, b):` satırının altındaki girintili (dört boşlukla içeri yazılmış) satırlar, `k` sırasıyla $a, a+1, \ldots, b-1$ değerlerini alarak tekrar tekrar çalışır. `range(n)` ise $0, 1, \ldots, n-1$ değerlerini verir.
>
> ```
> for k in range(1, 6):
>     print(k, k ** 2, k ** 3)
> ```
>
> Çıktı:
>
> ```
> 1 1 1
> 2 4 8
> 3 9 27
> 4 16 64
> 5 25 125
> ```

 CC BY-NC-SA 4.0

Örnek 1.2 (Fibonacci Sayıları ve Eşzamanlı Atama). $F_0 = 0$, $F_1 = 1$ ve $n \geq 0$ için $F_{n+2} = F_{n+1} + F_n$ ile tanımlanan Fibonacci dizisinin F_{10} terimini eşzamanlı atamayla hesaplayınız.

Çözüm.
Fikir. `a` ve `b` ardışık iki terimi, yani (F_k, F_{k+1}) çiftini tutsun. `a, b = b, a + b` ataması bu çifti $(F_{k+1}, F_k + F_{k+1}) = (F_{k+1}, F_{k+2})$ çiftine taşır. (F_0, F_1)'den başlayıp bu adımı 10 kez uygularsak `a`'da F_{10} kalır.

```
a, b = 0, 1           # F0 ve F1
for k in range(10):   # adımı 10 kez uygula
    a, b = b, a + b   # (Fk, Fk+1) -> (Fk+1, Fk+2)
print(a, b)           # F10 ve F11
```

Çıktı:

```
55 89
```

Neden eşzamanlı? Atamayı iki ayrı satıra bölersek ikinci satır `a`'nın eski değerini değil, yeni değerini kullanır:

```
a, b = 0, 1
a = b          # a artık 1
b = a + b      # yeni a kullanıldı: 1 + 1
print(a, b)    # doğrusu 1 1 olmalıydı
```

Çıktı:

```
1 2
```

Doğru bir adım $(F_0, F_1) = (0, 1)$ çiftini $(F_1, F_2) = (1, 1)$ çiftine götürmeliydi. Eşzamanlı atamayla $F_{10} = 55$ bulunur. ■

1.4 Çıktıyı Biçimlendirmek

Hesap sonuçlarını okunaklı, hizalı ve istediğimiz basamak sayısıyla yazdırmak için `print`'i biçimlendirilmiş dizelerle birlikte kullanırız.
`print` virgülle ayrılmış değerleri varsayılan olarak bir boşlukla ayırır ve sonunda satırı bitirir. Ayıracı `sep`, satır sonunu `end` ile değiştirebiliriz:

```
print(1, 2, 3)
print(1, 2, 3, sep=" < ")
print("a", end="; ")   # satırı bitirme, sonuna "; " koy
print("b")
```

Çıktı:

```
1 2 3
1 < 2 < 3
a; b
```

Tanım 1.3 (f-string ve Biçim Belirteci). Önüne `f` harfi konmuş bir dizeye **f-string** denir. Böyle bir dizede süslü parantez içine yazılan `{ifade}` değerlendirilir ve değeri dizeye yerleştirilir. `{ifade:biçim}` yazımında iki noktadan sonra gelen **biçim belirteci** (format specifier) değerin nasıl yazılacağını belirler.

Yani f-string bir şablondur: sabit metin aynen kalır, süslü parantezlerin yerine ifadelerin değerleri istenen biçimde yazılır. Bilimsel hesapta en çok kullanılan biçim belirteçleri şunlardır:

 CC BY-NC-SA 4.0

Belirteç	Anlamı
`.4f`	noktadan sonra 4 basamak
`.3e`	bilimsel gösterim, noktadan sonra 3 basamak
`.5g`	5 anlamlı basamak
`.1%`	yüzde olarak, noktadan sonra 1 basamak
`,`	binlik ayırıcı
`b`, `x`	ikili ve onaltılı taban
`>8`, `<8`, `^8`	8 karakterlik alanda sağa, sola, ortaya yasla
`10.4f`	10 karakterlik alanda, noktadan sonra 4 basamak

```
x = 2 / 3
n = 2 ** 32
print(f"x = {x}")
print(f"{x:.4f} | {x:.3e} | {x:.5g} | {x:.1%}")
print(f"{n:,} | {n:.2e}")
print(f"{10:b} | {255:x}")             # ikili ve onaltılı
print(f"[{7:>6}] [{7:<6}] [{7:^6}]")   # sağa, sola, ortaya
print(f"{x = :.3f}")                   # adı ve değeri birlikte
```

Çıktı:

```
x = 0.6666666666666666
0.6667 | 6.667e-01 | 0.66667 | 66.7%
4,294,967,296 | 4.29e+09
1010 | ff
[     7] [7     ] [  7   ]
x = 0.667
```

Son satırdaki `{x = :.3f}` yazımı, ifadenin kendisini ve değerini birlikte yazar; ara sonuçları kontrol ederken çok işe yarar.

> **i** Ondalık nokta ve virgül
>
> Python sayıları her zaman ondalık noktayla yazar ve okur: `0.5` bir sayıdır, `0,5` ise virgülle ayrılmış iki ayrı sayıdır; `print(0,5)` komutu `0 5` yazar. Bu notlarda metin ve formüllerde Türkçedeki gibi ondalık virgül (0,5), kodda ve kod çıktısında ise Python'un yazdığı biçimde ondalık nokta kullanılır.

Örnek 1.3 (Hizalı Bir Karekök Tablosu). $n = 1, 2, \ldots, 6$ için n, $\sqrt{n}$ ve $1/n^2$ değerlerini sütunları hizalı bir tablo olarak yazdırınız; $\sqrt{n}$ noktadan sonra 6 basamakla, $1/n^2$ ise bilimsel gösterimle yazılsın.

Çözüm.
Biçimleri seçelim. n için 2 karakterlik sağa yaslı alan (`>2`), $\sqrt{n} = n^{1/2}$ için 10 karakterlik alanda 6 ondalık basamak (`10.6f`), $1/n^2$ için 12 karakterlik alanda 3 basamaklı bilimsel gösterim (`12.3e`) kullanalım. Karekökü `n ** 0.5` ile alıyoruz.

```
print(f"{'n':>2} {'kök n':>10} {'1/n^2':>12}")
for n in range(1, 7):
    print(f"{n:>2} {n ** 0.5:10.6f} {1 / n ** 2:12.3e}")
```

Çıktı:

```
 n      kök n        1/n^2
 1   1.000000    1.000e+00
```

 CC BY-NC-SA 4.0

```
2   1.414214    2.500e-01
3   1.732051    1.111e-01
4   2.000000    6.250e-02
5   2.236068    4.000e-02
6   2.449490    2.778e-02
```

Hizalama. Başlık satırındaki dizeler de aynı genişliklerle sağa yaslandığından başlıklar sütunların tam üstüne düştü. Sayılar ne kadar basamaklı olursa olsun her sütun aynı genişlikte kaldı. ■

1.5 Modüller ve math Modülü

Karekök, logaritma ve trigonometrik fonksiyonlar dilin çekirdeğinde değil, Python'la birlikte gelen `math` modülündedir.

Tanım 1.4 (Modül ve İçe Aktarma). Birbiriyle ilgili fonksiyon ve sabitleri bir arada tutan Python dosyasına **modül** denir. `import math` komutu `math` modülünü yükler; modüldeki bir ada `math.sqrt` gibi `modül.ad` yazımıyla ulaşılır. `from math import sqrt, pi` komutu ise yalnız `sqrt` ve `pi` adlarını, önlerine `math.` yazmadan kullanılabilir hâle getirir.

Yani modül, hazır araçların durduğu bir kutudur; `import` bu kutuyu masaya koyar. Modül adını önde yazmak (`math.sqrt`) bir adın nereden geldiğini açıkça gösterdiği için genellikle tercih edilir.

```
import math

print(math.sqrt(2), math.pi, math.e)
print(math.exp(1), math.log(math.e), math.log10(1000))
print(math.log(8, 2), math.log2(8))     # 2 tabanında logaritma
print(math.sin(math.pi / 6), math.cos(0))
print(math.floor(-2.5), math.ceil(-2.5), math.trunc(-2.5))
print(math.factorial(10), math.comb(5, 2), math.gcd(84, 36))
```

Çıktı:

```
1.4142135623730951 3.141592653589793 2.718281828459045
2.718281828459045 1.0 3.0
3.0 3.0
0.49999999999999994 1.0
-3 -2 -2
3628800 10 12
```

Bu çıktıda birkaç noktaya dikkat edelim:

- `math.log(x)` doğal logaritmadır, yani $\ln x$'tir; başka bir taban ikinci argümanla verilir: `math.log(8, 2)`.
- Trigonometrik fonksiyonlar açıyı **radyan** cinsinden alır. Dereceyi radyana `math.radians`, radyanı dereceye `math.degrees` çevirir.
- $\sin(\pi/6) = 1/2$ olduğu hâlde Python `0.49999999999999994` yazdı. Nedeni, `math.pi`'nin π'nin kendisi değil, ona çok yakın bir ondalıklı sayı olmasıdır. Bu küçük farkları Bölüm 1.8 kısmında inceleyeceğiz.
- `floor` aşağı, `ceil` yukarı yuvarlar; `trunc` ise ondalık kısmı atar, yani sıfıra doğru yuvarlar. Negatif sayılarda `floor` ile `trunc` farklı sonuç verir.
- `factorial`, `comb(n, k)` $= \binom{n}{k}$, `perm(n, k)` $= \frac{n!}{(n-k)!}$, `gcd` ve `lcm` tam sayılarla çalışır ve sonucu tam verir.

Sık kullanılan diğer adlar şunlardır: `tan`, `asin`, `acos`, `atan`, `atan2(y, x)`, `sinh`, `cosh`, `hypot(x, y)` $= \sqrt{x^2+y^2}$, tam sayı karekökü `isqrt(n)` $= \lfloor\sqrt{n}\rfloor$, sonsuzu gösteren `inf` ve "sayı değil" anlamına gelen `nan`. Bir fonksiyonun ne yaptığını `help(math.comb)` komutu yazdırır. Mutlak değer için `math.fabs(x)` de vardır, ama sonucu her zaman `float`'tır; genellikle yerleşik `abs(x)` fonksiyonu kullanılır. `abs(x)` de $|\,x\,|$'i verir, tam sayıyı tam sayı olarak döndürür ve karmaşık sayılarla da çalışır.
`from` yazımıyla adlar doğrudan kullanılır:

```
from math import sqrt, pi

print(sqrt(16), 2 * pi)
```

Çıktı:

```
4.0 6.283185307179586
```

`from math import *` yazımı modüldeki bütün adları getirir; ama hangi adın nereden geldiğini gizlediği ve aynı adlı değişkenleri sessizce ezebildiği için önerilmez.
`math` modülü gerçel sayılarla çalışır ve tanım kümesi dışındaki bir değer hata verir. Etkileşimli yorumlayıcıda bu şöyle görünür:

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Negatif sayıların karekökü için karmaşık sayılarla çalışan `cmath` modülü vardır; onu Bölüm 1.10 kısmında göreceğiz.

Örnek 1.4 (Kosinüs Teoremiyle Üçüncü Kenar). Bir üçgenin iki kenarı $a = 7$, $b = 8$ ve bu kenarlar arasındaki açı $\gamma = 60^\circ$'dir. Üçüncü kenarı $c^2 = a^2 + b^2 - 2ab\cos\gamma$ kosinüs teoremiyle Python'da hesaplayınız.

Çözüm.
Açıyı radyana çevirelim. `math.cos` açıyı radyan cinsinden beklediğinden `math.cos(60)` yazmak 60 radyanın kosinüsünü hesaplar. Bu yüzden açıyı önce `math.radians` ile çeviriyoruz; karşılaştırma için yanlış yazımın sonucunu da yazdıralım:

```
import math

a, b = 7, 8
gamma = math.radians(60)        # 60 derece = pi/3 radyan
c = math.sqrt(a**2 + b**2 - 2 * a * b * math.cos(gamma))
print(c)
print(math.sqrt(57), math.cos(gamma))
print(math.sqrt(a**2 + b**2 - 2 * a * b * math.cos(60)))  # yanlış
```

Çıktı:

```
7.549834435270749
7.54983443527075 0.5000000000000001
14.82127706395429
```

Elle sağlama. $\cos 60^\circ = \frac{1}{2}$ olduğundan $c^2 = 49 + 64 - 56 = 57$ ve $c = \sqrt{57} \approx 7{,}5498$ olur. Python'un bulduğu `c` değeri `math.sqrt(57)` ile yalnız son basamakta farklıdır: `math.cos(gamma)` tam 0,5 değil, `0.5000000000000001` çıktığı için küçük bir yuvarlama farkı oluştu.
Yanlış satır. Son satırdaki 14,82, kosinüsü yaklaşık $-0{,}95$ olan 60 radyanlık açıyla hesaplanmıştır ve $c \approx 7{,}55$ ile hiç ilgisi yoktur. ■

1.6 Tam Sayılar

Python'un tam sayıları, matematikteki $\mathbb{Z}$'nin bilgisayardaki en sadık modelidir.

Tanım 1.5 (Tam Sayı Türü). Python'da tam sayılar `int` türündendir. Bir `int` değeri, büyüklüğüne üst sınır konmadan ve tam olarak saklanır. İki `int` arasında toplama, çıkarma, çarpma, tam bölme, kalan ve doğal sayı kuvvetine üs alma işlemlerinin sonucu yine tam olarak hesaplanmış bir `int`'tir.

 CC BY-NC-SA 4.0

Yani `int` ile yapılan hesapta yuvarlama yoktur; sayının büyüklüğünü yalnız bilgisayarın belleği sınırlar. Birçok programlama dili tam sayıları 64 bitle saklar ve $2^{63}-1$'i aşan sonuçlarda taşma yaşar (bkz. Nümerik Analiz); Python'da böyle bir sınır yoktur.

```
import math

print(2 ** 63 - 1)              # 64 bitlik işaretli en büyük tam sayı
print(2 ** 100)
print(math.factorial(30))       # 30!
print(2 ** 100 + 1 - 2 ** 100)  # tam hesap: 1 kaybolmaz
print(2 ** -1)                  # negatif üs: sonuç artık int değil
```

Çıktı:

```
9223372036854775807
1267650600228229401496703205376
265252859812191058636308480000000
1
0.5
```

Son satır, negatif kuvvetin sonucunun bir `float` olduğunu gösteriyor. Uzun sayıları okunaklı yazmak için rakamların arasına alt çizgi konabilir: `1_000_000` ile `1000000` aynı sayıdır.
Bir tam sayının basamaklarını saymak için onu `str` ile dizeye çevirir, dizenin uzunluğunu `len` ile alırız: `len(str(2 ** 100))` değeri 31'dir.

Örnek 1.5 (Yüz Faktöriyelin Basamak Sayısı). $100!$ sayısının kaç basamaklı olduğunu bulunuz ve sonucu $\log_{10}$ ile sağlayınız.

Çözüm.
Doğrudan sayım. $100!$'i tam olarak hesaplayıp dizeye çevirelim ve uzunluğuna bakalım. `math.log10` çok büyük tam sayıları da kabul eder:

```
import math

N = math.factorial(100)
print(len(str(N)))                    # basamak sayısı
print(math.log10(N))
print(math.floor(math.log10(N)) + 1)
```

Çıktı:

```
158
157.97000365471578
158
```

Logaritmayla sağlama. Pozitif bir N tam sayısı d basamaklıysa $10^{d-1} \le N < 10^{d}$, yani $d-1 \le \log_{10} N < d$ olur. Buradan $d = \lfloor \log_{10} N \rfloor + 1$ bulunur. $\log_{10}(100!) \approx 157{,}97$ olduğundan $d = 158$'dir; iki yol aynı sonucu verdi. ■

1.7 Tam Bölme ve Kalan

Sayılar teorisinin temel aracı olan bölme algoritması Python'da üç işlemle karşılanır: `//`, `%` ve `divmod`.

Tanım 1.6 (Tam Bölme ve Kalan). a ve $b \neq 0$ tam sayılar olsun. Python'da `a // b` **tam bölümü** $q = \lfloor a/b \rfloor$ sayısıdır, `a % b` **kalanı** ise $r = a - bq$ sayısıdır. `divmod(a, b)` bu ikisini `(q, r)` çifti olarak birlikte verir.

 CC BY-NC-SA 4.0

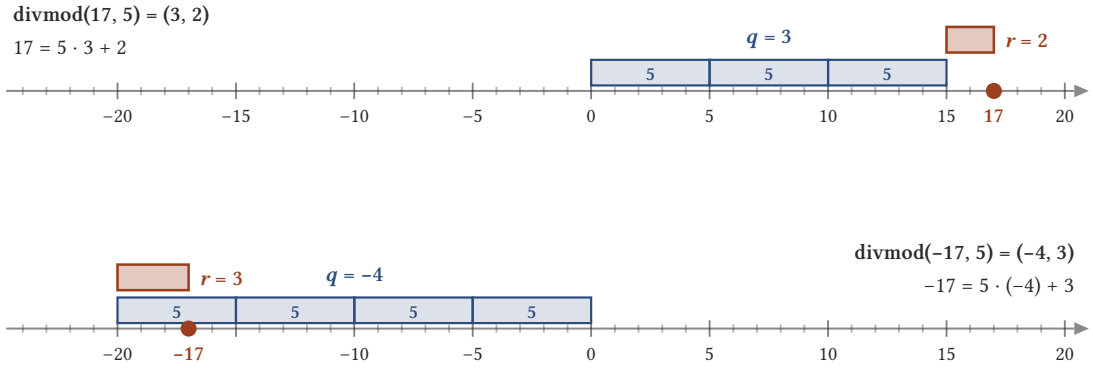
Yani `//` bölümü daima aşağı, $-\infty$'a doğru yuvarlar ve $a = bq + r$ eşitliği her zaman sağlanır. $b > 0$ ise $0 \le r < b$ olur; bu, Sayılar Teorisi notlarındaki bölme algoritmasının verdiği bölüm ve kalanın ta kendisidir, a negatif olsa bile. $b < 0$ ise $b < r \le 0$ olur, yani kalan bölenin işaretini alır.

```
print(17 // 5, 17 % 5, divmod(17, 5))
print(-17 // 5, -17 % 5, divmod(-17, 5))
print(5 * (-4) + 3)          # a = bq + r sağlaması
print(7.5 // 2, 7.5 % 2)     # ondalıklı sayılarda da tanımlı
```

Çıktı:

```
3 2 (3, 2)
-4 3 (-4, 3)
-17
3.0 1.5
```

Bölüm sıfıra doğru değil aşağı yuvarlandığından $-17/5 = -3{,}4$ için tam bölüm -3 değil -4 olur ve kalan negatif değil, 3 çıkar. (Sayının önündeki eksi işareti `//`'den önce uygulanır, yani `-17 // 5` ifadesi `(-17) // 5` demektir.) Aşağıdaki şekil iki durumu sayı doğrusu üzerinde gösteriyor: q, a'yı aşmayan en büyük 5 katının katsayısıdır.



Şekil 1.1: `divmod(17, 5)` ve `divmod(-17, 5)` sayı doğrusunda. Mavi bloklar $|q|$ tane 5 uzunluğunda adımdır ve 0'dan, a'yı aşmayan en büyük 5 katı olan $q \cdot 5$'e ulaşır; turuncu blok $r = a - 5q$ kalanıdır. İki durumda da $0 \le r < 5$ olur; -17 için bölüm -3 değil -4'tür.

Kalan işlemi, büyük kuvvetlerin modüler hesabında da kullanılır. `pow(a, n, m)` çağrısı a^n'yi hiç hesaplamadan, her adımda m'ye göre kalan alarak $a^n \bmod m$ değerini bulur. Kuvvet ardışık kare almalarla oluşturulduğundan adım sayısı yaklaşık $\log_2 n$'dir; n yüzlerce basamaklı olsa bile hesap anında biter.

Örnek 1.6 (Fermat Testiyle Bileşiklik). Fermat'nın küçük teoremini kullanarak 341'in asal olmadığını gösteriniz.

Çözüm.
Test. Fermat'nın küçük teoremine göre p asal ve $p \nmid a$ ise $a^{p-1} \equiv 1 \pmod p$ olur. Bu yüzden $341 \nmid a$ olan bir a için $a^{340} \not\equiv 1 \pmod{341}$ çıkarsa 341 asal olamaz. $a = 2$ ve $a = 3$'ü deneyelim:

```
print(pow(2, 340, 341))   # 2^340 mod 341
print(pow(3, 340, 341))   # 3^340 mod 341
print(2 ** 340 % 341)     # aynı sonuç, ama önce 2^340 hesaplanır
print(11 * 31)
```

Çıktı:

```
1
56
```

 CC BY-NC-SA 4.0

```
1
341
```

Yorum. $2^{340} \equiv 1 \pmod{341}$ olduğundan $a = 2$ ile yapılan test karar veremez; 341, 2 tabanına göre bir **sahte asal**dır. Buna karşılık $3^{340} \equiv 56 \not\equiv 1 \pmod{341}$ olduğundan 341 asal değildir. Gerçekten $341 = 11 \cdot 31$'dir. Üçüncü satırdaki doğrudan hesap da aynı kalanı verdi, ama önce 103 basamaklı 2^{340} sayısını hesapladı. ■

1.8 Kayan Noktalı Sayılar

Tam sayılardan sonra gerçel sayılara geçiyoruz: `2.5` gibi ondalık noktayla yazılan her sayı ve iki tam sayının `/` ile bölümü Python'da `float` türündendir; bu tür, gerçel sayıların sonlu bir alt kümesidir.

Tanım 1.7 (Kayan Noktalı Sayı Türü). Python'un `float` türü, IEEE 754 standardının 64 bitlik (çift duyarlıklı) makine sayılarıdır. Sıfırdan farklı her normalize `float`

$$x = \pm\,(1 + m)\,2^{e}, \qquad -1022 \le e \le 1023$$

biçimindedir; burada $m = (0{,}b_1 b_2 \ldots b_{52})_2$, yani $0 \le m < 1$ ve $2^{52} m$ bir tam sayıdır. Bir sayı yazıldığında ya da bir işlem yapıldığında sonuç, kendisine en yakın `float` değerine **yuvarlanır**.

Yani `float`, gerçel sayıları 53 anlamlı ikili basamakla, yaklaşık 16 ondalık basamakla temsil eder. Bitlerin bellekte nasıl dizildiği Nümerik Analiz notlarında ayrıntılı anlatılmıştır. Ondalık tabanda sonlu olan 0,1 sayısının ikili açılımı $0{,}000110011001100\ldots_2$ sonsuzdur; bu yüzden 0,1 bir `float` olarak tam saklanamaz, ona en yakın makine sayısı saklanır.

```
print(0.1 + 0.2)
print(0.1 + 0.2 == 0.3)
print(f"{0.1:.20f}")                 # 0.1'in saklanan değeri
print(2.0 ** 100 + 1 - 2.0 ** 100)   # 1 kayboldu
print(1e308 * 10, -1e308 * 10)       # taşma: sonsuz
```

Çıktı:

```
0.30000000000000004
False
0.10000000000000000555
0.0
inf -inf
```

Satırları sırayla okuyalım:

- Python bir `float`'ı yazarken, geri okunduğunda aynı `float`'ı veren en kısa ondalık diziyi yazar. `0.1` için bu dizi `0.1`'dir, ama saklanan değer, üçüncü satırın gösterdiği gibi, `0.1`'den biraz büyüktür. 0,1 ile 0,2'nin saklanan değerlerinin toplamı da 0,3'e en yakın `float`'a değil, onun bir sonraki komşusuna yuvarlanır.
- 2^{100} civarında ardışık iki `float` arasındaki uzaklık 2^{48}'dir. $2^{100} + 1$ bu aralığa göre 2^{100}'e o kadar yakındır ki toplam yine 2^{100}'e yuvarlanır. Aynı hesap `int` ile tam olarak 1 vermişti.
- `1e308` yazımı $1 \cdot 10^{308}$ demektir. En büyük `float`'ı aşan sonuçlar `inf`, yani sonsuz olur.

Bir `float`'ı tam sayıya `int(x)` çevirir (ondalık kısmı atarak), bir tam sayıyı `float`'a `float(n)` çevirir. `float` sisteminin sınırlarını `sys` modülündeki `sys.float_info` söyler:

```
import sys

print(sys.float_info.max)
print(sys.float_info.min)
print(sys.float_info.mant_dig, sys.float_info.dig)
print(2.0 ** 1023 * (2 - 2.0 ** -52) == sys.float_info.max)
print(2.0 ** -1022 == sys.float_info.min)
```

 CC BY-NC-SA 4.0

Çıktı:

```
1.7976931348623157e+308
2.2250738585072014e-308
53 15
True
True
```

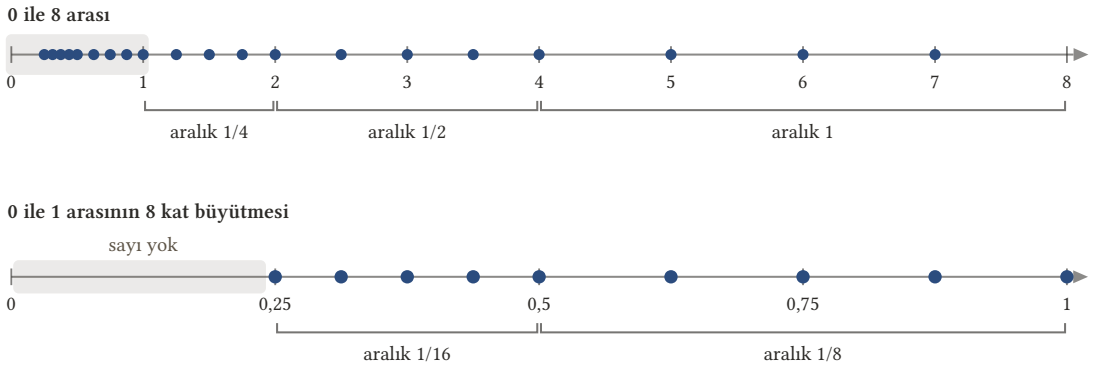
Yani en büyük `float` $2^{1023}(2-2^{-52}) \approx 1{,}798 \cdot 10^{308}$, en küçük pozitif normalize `float` ise $2^{-1022} \approx 2{,}225 \cdot 10^{-308}$'dir; bunlar Nümerik Analiz notlarında bulunan sınırlardır. `mant_dig` anlamlı ikili basamak sayısını (52 mantis biti ve gizli baştaki 1 ile 53) verir. `dig` ise 15'tir: 15 anlamlı basamaklı her ondalık sayı `float`'a çevrilip geri yazıldığında aynen elde edilir. Standart, 2^{-1022}'nin altında daha az anlamlı basamaklı **alt normal** (subnormal) sayılara da izin verir; bunların en küçüğü $2^{-1074} \approx 4{,}9 \cdot 10^{-324}$'tür.
Kayan noktalı sayılar sayı doğrusuna eşit aralıklarla dağılmaz. Bunu görmek için mantisi yalnız 2 ikili basamaklı olan oyuncak bir sistem düşünelim: sayılar $(1 + m/4) \cdot 2^e$ biçiminde olsun, burada $m \in \{0, 1, 2, 3\}$ ve $-2 \le e \le 2$'dir.

```
# 2 bitlik mantisli oyuncak sistem: (1 + m/4) * 2**e
for e in range(-2, 3):
    print(e, 2.0**e, 1.25 * 2.0**e, 1.5 * 2.0**e, 1.75 * 2.0**e)
```

Çıktı:

```
-2 0.25 0.3125 0.375 0.4375
-1 0.5 0.625 0.75 0.875
0 1.0 1.25 1.5 1.75
1 2.0 2.5 3.0 3.5
2 4.0 5.0 6.0 7.0
```

Her $[2^e, 2^{e+1})$ aralığında dört sayı vardır ve bunlar eşit aralıklıdır; ama aralık 2^{e-2}'dir ve bir sonraki ikilik aralıkta iki katına çıkar. Gerçek `float` sisteminde de her $[2^e, 2^{e+1})$ aralığında 2^{52} tane eşit aralıklı sayı vardır ve aralık 2^{e-52}'dir. Mutlak aralık büyüdükçe artar, ama sayının büyüklüğüne oranı hep aynı mertebede kalır.



Şekil 1.2: Mantisi 2 bitlik oyuncak sistemin 20 pozitif sayısı, $(1 + m/4) \cdot 2^e$. Her $[2^e, 2^{e+1})$ aralığında dört sayı eşit aralıklı durur; aralık bir sonraki ikilik aralıkta iki katına çıkar. Altta, üstteki gölgeli $[0, 1]$ parçası 8 kat büyütülmüştür: 0 ile 0,25 arasında sistemin hiçbir sayısı yoktur.

 CC BY-NC-SA 4.0

⚠ Büyük tam sayılar ve bölme

`float` yalnız 53 anlamlı ikili basamak tutar; bu yüzden 2^{53}'ten büyük tam sayıların hepsi `float` olarak saklanamaz. `/` işleminin sonucu her zaman `float` olduğundan büyük tam sayılarla bölmede `/` yerine `//` kullanılmalıdır.

```
n = 2 ** 53 + 1
print(n, float(n), float(n) == n)
print(10 ** 20 // 3)        # tam
print(int(10 ** 20 / 3))    # float üzerinden: son basamaklar bozuk
```

Çıktı:

```
9007199254740993 9007199254740992.0 False
33333333333333333333
33333333333333331968
```

1.9 Makine Epsilonu ve Sayıları Karşılaştırmak

Yuvarlama hatasının büyüklüğünü tek bir sayı belirler: 1 ile ondan büyük ilk `float` arasındaki uzaklık.

Tanım 1.8 (Makine Epsilonu). 1'den büyük en küçük `float` $1 + 2^{-52}$'dir. $\varepsilon = 2^{-52} \approx 2{,}22 \cdot 10^{-16}$ sayısına **makine epsilonu** (machine epsilon) denir.

Yani 1'in sağındaki ilk komşu $1 + \varepsilon$'dur ve 1 ile $1 + \varepsilon$ arasındaki bir sayı bu ikisinden birine yuvarlanır. Daha genel olarak, bir `x` değeri ile büyüklükçe ondan sonra gelen ilk `float` arasındaki uzaklığa `x`'in **ulp**'u (unit in the last place) denir ve `math.ulp(x)` ile bulunur; ε, `math.ulp(1.0)` değeridir.

```
import math
import sys

eps = sys.float_info.epsilon
print(eps, eps == 2 ** -52)
print(1 + eps > 1, 1 + eps / 2 == 1)
print(math.nextafter(1, 2) - 1)        # 1'den sonraki float eksi 1
print(math.ulp(1.0), math.ulp(1000.0), math.ulp(1e16))
```

Çıktı:

```
2.220446049250313e-16 True
True True
2.220446049250313e-16
2.220446049250313e-16 1.1368683772161603e-13 2.0
```

`math.nextafter(x, y)`, `x`'ten `y` yönünde ilerleyince karşılaşılan ilk `float`'tır. $1 + \varepsilon/2$ sayısı 1 ile $1 + \varepsilon$'un tam ortasındadır; IEEE 754'ün "eşitlikte son biti çift olana yuvarla" kuralıyla 1'e yuvarlanır. Son satır, aralığın sayıyla birlikte büyüdüğünü gösteriyor: 10^{16} civarında ardışık `float`'lar 2 aralıklıdır, yani $10^{16} + 1$ tam olarak saklanamaz.

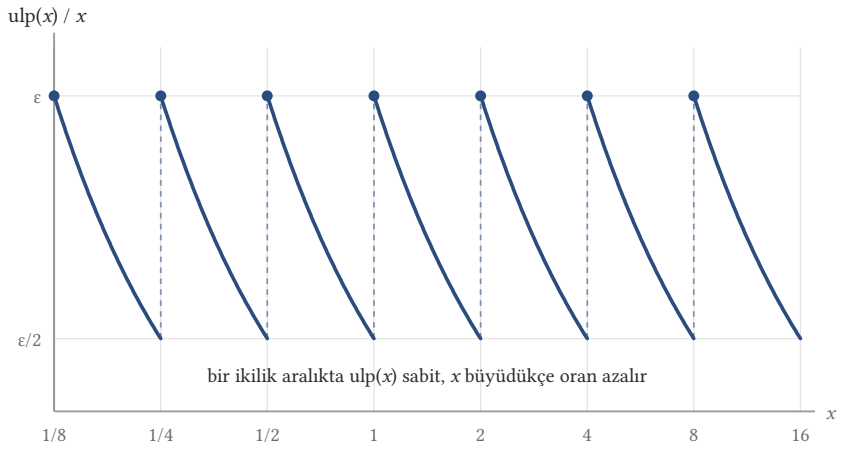
Önerme 1.1 (Yuvarlamanın Bağıl Hatası). *x, $2^{-1022} \le |\,x\,| \le 2^{1023}(2 - 2^{-52})$ koşulunu sağlayan bir gerçel sayı ve $\mathrm{fl}(x)$, x'e en yakın `float` olsun. Bu durumda*

$$\mathrm{fl}(x) = x\,(1 + \delta), \qquad |\,\delta\,| \le \frac{\varepsilon}{2} = 2^{-53}$$

olur. IEEE 754 standardı ayrıca dört işlemin ve karekökün sonucunun, tam sonucun yuvarlanmışı olmasını şart koşar: `float` sayılar x, y ve $\circ \in \{+, -, \times, /\}$ için, tam sonuç $x \circ y$ yukarıdaki normal aralıkta kaldıkça (yani taşma ya da alttan taşma olmadıkça) $\mathrm{fl}(x \circ y) = (x \circ y)(1 + \delta)$, $|\,\delta\,| \le 2^{-53}$ olur.

 CC BY-NC-SA 4.0

Yani tek bir işlemin bağıl hatası en fazla $2^{-53} \approx 1{,}1 \cdot 10^{-16}$'dır; tek bir işlemden sonra bir `float` sonucuna yaklaşık 16 anlamlı basamakla güvenebiliriz. Hatalar ise art arda yapılan işlemler boyunca birikebilir. Sınırın ispatı, ondalık tabanda k basamakla yuvarlamanın bağıl hata sınırının (bkz. Nümerik Analiz) ikili tabanda 53 basamakla aynen tekrarıdır.
Aşağıdaki şekil `math.ulp(x)/x` bağıl aralığının x'e göre nasıl değiştiğini gösteriyor. Bağıl aralık her ikilik aralığın başında ε'dur, aralığın sonuna doğru $\varepsilon/2$'ye iner ve bir sonraki 2 kuvvetinde yeniden ε'a sıçrar.



Şekil 1.3: Bağıl aralık `math.ulp(x)/x`, logaritmik x ekseninde. Bir $[2^e, 2^{e+1})$ aralığında ardışık `float`'lar arasındaki uzaklık sabit olduğundan bağıl aralık ε'dan ε/2'ye iner ve her 2 kuvvetinde yeniden ε'a sıçrar. Yuvarlamanın bağıl hatası bunun yarısıyla, en fazla ε/2 ile sınırlıdır.

Yuvarlama hatası ile yöntemin kendi hatasının birlikte nasıl davrandığını tanıdık bir limitte görelim.

Örnek 1.7 (e Sayısına Yaklaşan Dizi). $e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$ olduğunu biliyoruz (bkz. Analiz 1). $n = 10^k$, $k = 1, 2, \ldots, 16$ için $\left(1+\frac{1}{n}\right)^n$ değerini `float` ile hesaplayıp `math.e` ile arasındaki farkı inceleyiniz.

Çözüm.
Hesap.

```
import math

for k in range(1, 17):
    n = 10 ** k
    approx = (1 + 1 / n) ** n
    print(k, approx, abs(approx - math.e))
```

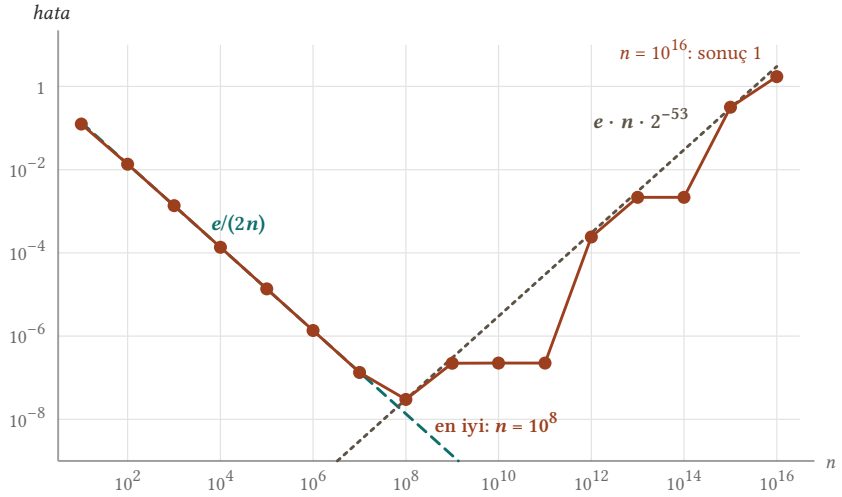
Çıktı:

```
1 2.5937424601000023 0.12453936835904278
2 2.7048138294215285 0.01346799903751661
3 2.7169239322355936 0.0013578962234515046
4 2.7181459268249255 0.00013590163411958 4
5 2.7182682371922975 1.359126674760347e-05
6 2.7182804690957534 1.359363291708604e-06
7 2.7182816941320818 1.3432696333026684e-07
8 2.7182817983473577 3.011168736577474e-08
9 2.7182820520115603 2.2355251516614771e-07
10 2.7182820532347876 2.2477574246337895e-07
11 2.71828205335711 2.248980650598753e-07
12 2.7185234960372378 0.00024166757819266138
13 2.716110034086901 0.00217179437214 4209
```

```
14 2.716110034087023 0.0021717943720220845
15 3.035035206549262 0.31675337809021675
16 1.0 1.718281828459045
```

İlk kısım: yöntemin hatası. $k \le 7$ iken hata her adımda yaklaşık on kat azalır. Bu, dizinin kendi yakınsama hızıdır: Taylor açılımıyla $e - \left(1+\frac{1}{n}\right)^n \approx \frac{e}{2n}$ olduğu gösterilebilir.
İkinci kısım: yuvarlama hatası. $k = 9$'dan sonra hata yeniden büyür. $1+\frac{1}{n}$ toplamı Önerme 1.1 gereği 2^{-53}'e kadar bağıl hatayla saklanır; n'inci kuvvet alınırken bu bağıl hata yaklaşık n ile çarpılır. $n = 10^9$ için $n \cdot 2^{-53} \approx 1{,}1 \cdot 10^{-7}$ olur ve bu, artık yöntemin $\frac{e}{2n} \approx 1{,}4 \cdot 10^{-9}$ hatasından çok büyüktür.
Son satır. $n = 10^{16}$ için $10^{-16} < \frac{\varepsilon}{2}$ olduğundan $1 + 10^{-16}$ toplamı doğrudan 1'e yuvarlanır ve $1^n = 1$ çıkar. En iyi sonuç, iki hatanın dengelendiği $k = 8$ civarında elde edilir. ■

İki hata kaynağı log-log ölçekte açıkça ayrılır: soldaki doğru parçası yöntemin $\frac{e}{2n}$ hatasıdır, sağdaki yükselen kısım ise yuvarlama hatasının n ile büyümesidir.



Şekil 1.4: $(1 + 1/n)^n$ ile e arasındaki fark, e sayısına yaklaşan dizi örneğindeki kodun 16 değeri, log-log ölçekte. Kesikli çizgi yöntemin $e/(2n)$ hatası, noktalı çizgi yuvarlama hatasının $e \cdot n \cdot 2^{-53}$ kaba tahminidir.
Hata önce n ile azalır, $n = 10^8$ civarında en küçük değerini alır, sonra yuvarlama yüzünden büyür.

Yuvarlama hataları yüzünden matematikte eşit olan iki niceliğin `float` değerleri son basamaklarda farklı çıkabilir. Bu yüzden `float` sayılar `==` ile değil, bir toleransla karşılaştırılır.

Tanım 1.9 (Toleransla Karşılaştırma). `math.isclose(a, b, rel_tol=1e-09, abs_tol=0.0)` çağrısı, `rel_tol` değerine ρ ve `abs_tol` değerine τ dersek,

$$|\, a - b \,| \le \max\big(\, \rho \cdot \max(|\, a \,|, |\, b \,|), \tau \,\big)$$

eşitsizliği sağlanıyorsa `True`, sağlanmıyorsa `False` verir. ρ'ya **bağıl tolerans**, τ'ya **mutlak tolerans** denir; verilmeyen tolerans için varsayılan değer kullanılır.

Yani bağıl tolerans farkı sayıların kendi büyüklüğüyle ölçer: varsayılan $\rho = 10^{-9}$, iki sayının yaklaşık dokuz anlamlı basamağının aynı olmasını ister. Mutlak tolerans ise büyüklükten bağımsız bir eşiktir. Sayılardan biri 0 ise bağıl koşul $|\, a \,| \le \rho\,|\, a \,|$ biçimine iner ve yalnız $a = 0$ iken sağlanır; bu yüzden sıfırla karşılaştırırken mutlak tolerans şarttır. `abs_tol=1e-12` yazımında olduğu gibi, bir argümanı adıyla vermek mümkündür.

 CC BY-NC-SA 4.0

Üç adımda iki float'ı karşılaştırmak

1. **`==` kullanma.** Matematikte eşit olan iki niceliğin `float` değerleri son basamaklarda farklı olabilir.
2. **Bağıl tolerans seç.** Sayıların büyüklüğüne uygun bir `rel_tol` ile `math.isclose(a, b, rel_tol=...)` kullan; varsayılan 10^{-9} çoğu hesap için yeterlidir.
3. **Sıfırın yakınında mutlak tolerans ekle.** Beklenen değer 0 ise problemin ölçeğine uygun bir `abs_tol` ver.

Örnek 1.8 (Sıfırla Karşılaştırma). $\sin \pi = 0$ eşitliğini Python'da `math.sin` ve `math.pi` ile sınayınız.

Çözüm.
Değeri ve üç karşılaştırmayı yazdıralım.

```
import math

y = math.sin(math.pi)
print(y)
print(y == 0)
print(math.isclose(y, 0))                  # yalnız bağıl tolerans
print(math.isclose(y, 0, abs_tol=1e-12))   # mutlak tolerans eklendi
```

Çıktı:

```
1.2246467991473532e-16
False
False
True
```

Neden sıfır değil? `math.pi` π'nin kendisi değil, ona en yakın `float`'tır ve π ile farkı yaklaşık $1{,}22 \cdot 10^{-16}$'dır. π civarında $\sin(\pi - h) = \sin h \approx h$ olduğundan Python bu küçük farkı doğru olarak hesapladı; sonuç 0 değil, bu yüzden `==` karşılaştırması `False` verdi.
Toleranslar. Sayılardan biri 0 olduğu için yalnız bağıl toleransla yapılan karşılaştırma da `False` verdi. Sinüs değerleri 1 mertebesinde olduğundan $\tau = 10^{-12}$ mutlak toleransı makul bir eşiktir ve bununla karşılaştırma `True` verdi. ■

1.10 Karmaşık Sayılar

Python karmaşık sayıları da yerleşik bir tür olarak tanır; temel işlemler için bir modül yüklemek gerekmez.

Tanım 1.10 (Karmaşık Sayı Türü). Python'da $a + bi$ karmaşık sayısı `a + bj` biçiminde yazılır ve `complex` türündendir; sanal birim `1j` ile gösterilir. Bir `z` karmaşık sayısının gerçel ve sanal kısımları `z.real` ve `z.imag`, eşleniği `z.conjugate()`, modülü `abs(z)` ile bulunur.

Yani Python, mühendislikteki geleneği izleyerek i yerine j harfini kullanır. Tek başına `j` bir değişken adıdır; sanal birim daima önünde bir sayıyla, `1j` olarak yazılır. Gerçel ve sanal kısımlar `float` olarak saklanır, bu yüzden karmaşık hesap da `float` hesabının yuvarlama hatalarını taşır. `z.conjugate()` gibi bir değerin arkasına nokta konarak çağrılan fonksiyonlara **metot** denir.

```
z = 3 + 4j
w = 1 - 2j
print(z + w, z * w, z / w)
print(z.real, z.imag, z.conjugate(), abs(z))
print(1j ** 2)
```

Çıktı:

 CC BY-NC-SA 4.0

```
(4+2j) (11-2j) (-1+2j)
3.0 4.0 (3-4j) 5.0
(-1+0j)
```

Karekök, üstel fonksiyon ve logaritmanın karmaşık sürümleri `cmath` modülündedir:

```
import cmath

print(cmath.sqrt(-1), cmath.sqrt(-4))
print(cmath.exp(1j * cmath.pi))      # Euler: e^(i pi) = -1
print(cmath.polar(3 + 4j))           # (modül, argüman)
print(cmath.rect(2, cmath.pi / 2))   # 2 e^(i pi/2) = 2i
```

Çıktı:

```
1j 2j
(-1+1.2246467991473532e-16j)
(5.0, 0.9272952180016122)
(1.2246467991473532e-16+2j)
```

`cmath.polar(z)`, $z = re^{i\theta}$ olacak biçimdeki (r, θ) çiftini $\theta \in [-\pi, \pi]$ olacak biçimde verir; `cmath.rect(r, theta)` bunun tersidir. Euler formülüne göre $e^{i\pi} = -1$'dir. Python'un sonucundaki `1.2246467991473532e-16j` sanal kısmı yine `cmath.pi` ile π arasındaki farktan gelir; aynı fark `cmath.rect` satırının gerçel kısmında da görünür.

Örnek 1.9 (Karmaşık Kökleri Olan İkinci Derece Denklem). $x^2 + 2x + 5 = 0$ denkleminin köklerini Python ile bulunuz ve sağlamasını yapınız.

Çözüm.

Formül. Kökler $x_{1,2} = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ ile verilir. Diskriminant $\Delta = 4 - 20 = -16 < 0$ olduğundan `math.sqrt` hata verir; `cmath.sqrt` ise $\sqrt{-16} = 4i$ sonucunu döndürür.

```
import cmath

a, b, c = 1, 2, 5
d = cmath.sqrt(b**2 - 4 * a * c)   # diskriminantın karekökü
x1 = (-b + d) / (2 * a)
x2 = (-b - d) / (2 * a)
print(x1, x2)
print(x1**2 + 2 * x1 + 5, x2**2 + 2 * x2 + 5)   # sağlama
```

Çıktı:

```
(-1+2j) (-1-2j)
0j 0j
```

Sonuç. Kökler $-1 + 2i$ ve $-1 - 2i$'dir; katsayılar gerçel olduğundan birbirinin eşleniğidir. Her iki kökte de polinomun değeri tam olarak 0 çıktı, çünkü bu hesaptaki bütün ara değerler `float` olarak tam saklanabilen küçük tam sayılardır. ■

Örnek 1.10 (Birimin Altıncı Kökleri). $z^6 = 1$ denkleminin altı kökünü $\omega_k = e^{2\pi i k/6}$, $k = 0, 1, \ldots, 5$ formülüyle hesaplayınız ve her birinin denklemi ne kadar iyi sağladığını inceleyiniz.

Çözüm.

Hesap. Formül, Kompleks Analiz notlarındaki n-inci kök formülünün $z_0 = 1$, $n = 6$ hâlidir.

 CC BY-NC-SA 4.0

```python
import cmath

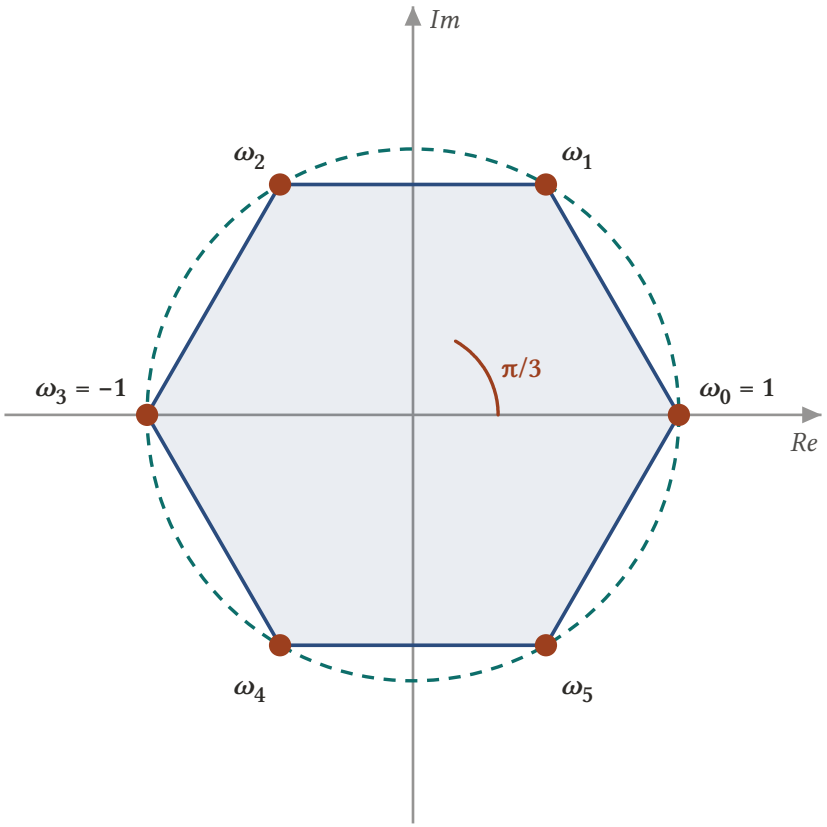
n = 6
for k in range(n):
    w = cmath.exp(2j * cmath.pi * k / n)
    print(k, w, abs(w ** n - 1))
```

Çıktı:

```
0 (1+0j) 0.0
1 (0.5000000000000001+0.8660254037844386j) 8.617648093045562e-16
2 (-0.4999999999999998+0.8660254037844387j) 1.2412670766236366e-15
3 (-1+1.2246467991473532e-16j) 7.347880794884119e-16
4 (-0.5000000000000004-0.8660254037844385j) 2.975418419039454e-15
5 (0.5000000000000001-0.8660254037844386j) 8.617648093045562e-16
```

Yorum. Tam değerler ± 1 ve $\pm\frac{1}{2} \pm \frac{\sqrt{3}}{2}\,i$'dir; $\frac{\sqrt{3}}{2} \approx 0{,}8660254$. Hesaplanan değerler son basamaklarda bunlardan biraz farklıdır ve $\left|\,\omega_k^6 - 1\,\right|$ farkları 0 değil, 10^{-15} mertebesindedir: her kök ε mertebesinde bir bağıl hatayla bulunmuştur ve altıncı kuvvet bu hatayı birkaç kat büyütür. ■

Altı kök, birim çember üzerinde düzgün bir altıgenin köşelerine dizilir; ardışık iki kök arasındaki açı $\frac{2\pi}{6} = \frac{\pi}{3}$'tür.



Şekil 1.5: $z^6 = 1$ denkleminin `cmath.exp(2j * cmath.pi * k / 6)` ile hesaplanan altı kökü. Kökler birim çember üzerinde düzgün bir altıgenin köşeleridir ve ardışık iki kök arasındaki açı $\pi/3$'tür.

 CC BY-NC-SA 4.0

1.11 Tam Kesirler

Rasyonel sayılarla hiç yuvarlama yapmadan hesap yapmak için standart kütüphanedeki `fractions` modülü kullanılır.

Tanım 1.11 (Kesir Türü). `fractions` modülündeki `Fraction(p, q)`, $q \neq 0$ olmak üzere p ve q tam sayılarından p/q rasyonel sayısını kurar ve onu paydası pozitif, en sade kesir olarak saklar. `Fraction` değerleri arasında dört işlem ve tam sayı kuvvetleri tam olarak yapılır; sonuç yine en sade hâle getirilir.

Yani `Fraction`, $\mathbb{Q}$'nun bilgisayardaki tam modelidir ve `int`'in kesirlere genişlemesi gibi düşünülebilir. Bedeli hızdır: hesap ilerledikçe pay ve paydalar büyüyebilir ve işlemler `float`'a göre çok daha yavaştır.

```
from fractions import Fraction

x = Fraction(1, 3) + Fraction(1, 6)
print(x, x.numerator, x.denominator)
print(Fraction(6, 4), Fraction("0.75"), Fraction(3, 10) * 10)
print(Fraction(1, 10) + Fraction(2, 10) == Fraction(3, 10))
print(Fraction(0.1))          # float 0.1'in tam değeri
print(float(Fraction(22, 7)))
```

Çıktı:

```
1/2 1 2
3/2 3/4 3
True
3602879701896397/36028797018963968
3.142857142857143
```

`x.numerator` ve `x.denominator` kesrin payını ve paydasını verir. `Fraction(0.1)` satırı ise bir `float`'ın aslında hangi rasyonel sayı olduğunu tam olarak gösteriyor: paydası $36028797018963968 = 2^{55}$'tir. Bu, `0.1`'in $\frac{1}{10}$ değil, ona çok yakın bir ikili kesir olduğunun kesin kanıtıdır. Bir kesri iki tam sayıdan ya da `Fraction("0.75")` gibi bir dizeden kurmak tamdır; bir `float`'tan kurmak ise `float`'ın hatasını da beraberinde getirir.

Örnek 1.11 (Harmonik Sayının Tam Değeri). $H_{10} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{10}$ toplamını tam kesir olarak bulunuz ve `float` ile yapılan toplamla karşılaştırınız.

Çözüm.
İki toplamı yan yana kuralım. `H` tam toplamı, `h` ise `float` toplamı biriktirsin:

```
from fractions import Fraction

H = Fraction(0)
h = 0.0
for k in range(1, 11):
    H += Fraction(1, k)    # tam toplam
    h += 1 / k             # float toplam
print(H)
print(float(H), h)
```

Çıktı:

```
7381/2520
2.9289682539682538 2.9289682539682538
```

Sonuç. $H_{10} = \frac{7381}{2520}$'dir; payda $2520 = \text{ekok}\,(1, 2, \ldots, 10)$ ve pay $7381 = 11^2 \cdot 61$ olduğundan kesir en sade hâldedir. Bu kısa toplamda `float` toplam, tam değere en yakın `float` ile aynı çıktı; ama tam değeri, yani pay ve paydayı, ancak `Fraction` verir. ■

 CC BY-NC-SA 4.0

Örnek 1.12 (Pi Sayısına En Yakın Kesir). Paydası 1000'i geçmeyen kesirler arasında π'ye en yakın olanı bulunuz.

Çözüm.
Araç. `Fraction` değerlerinin `limit_denominator(N)` metodu, paydası N'yi geçmeyen kesirler arasında kendisine en yakın olanı verir. `Fraction(math.pi)`, `math.pi` `float`'ının tam değeridir; π'den farkı 10^{-16} mertebesinde olduğundan bu soruda π yerine kullanılabilir.

```
import math
from fractions import Fraction

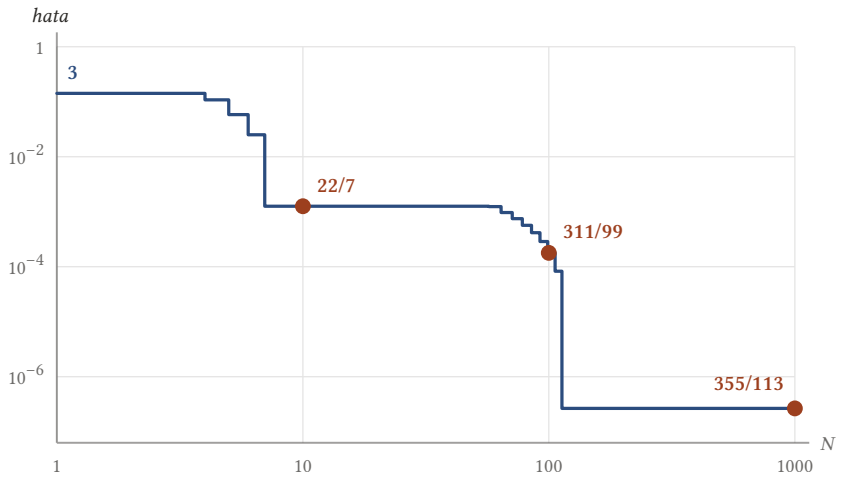
p = Fraction(math.pi)
print(p.limit_denominator(10))
print(p.limit_denominator(100))
print(p.limit_denominator(1000))
r = p.limit_denominator(1000)
print(float(r) - math.pi)
```

Çıktı:

```
22/7
311/99
355/113
2.667641894049666e-07
```

Sonuç. Aranan kesir $\frac{355}{113}$'tür ve π'den farkı yaklaşık $2{,}7 \cdot 10^{-7}$'dir. Paydası 114 ile 1000 arasında olan hiçbir kesir bundan daha iyi değildir. ■

Paydanın üst sınırı N büyüdükçe en iyi yaklaşım ancak belirli N değerlerinde değişir; aradaki N'lerde aynı kesir en iyi kalır. Aşağıdaki şekil `limit_denominator(N)` çağrısının $N = 1, 2, \ldots, 1000$ için verdiği kesirlerin π'ye uzaklığını gösteriyor.



Şekil 1.6: Paydası en fazla N olan kesirler arasında π'ye en yakın olanın hatası, `limit_denominator(N)` ile $N = 1, 2, \ldots, 1000$ için. Hata basamak basamak azalır: 22/7, $N = 7$'den 56'ya kadar en iyi kesirdir ve $N = 57$'de yerini 179/57'ye bırakır; 355/113 ise $N = 113$'ten 1000'e kadar en iyi kesir kalır. Turuncu noktalar örnekteki kodun yazdırdığı $N = 10, 100, 1000$ değerleridir.

 CC BY-NC-SA 4.0

1.12 Ondalık Hesap

Bazı hesaplarda sayıların onluk tabanda tam saklanması ya da `float`'ın sunduğundan çok daha fazla basamak gerekir; bunun için `decimal` modülü vardır.

Tanım 1.12 (Ondalık Sayı Türü). `decimal` modülündeki `Decimal` türü sayıları onluk tabanda ve belirli sayıda anlamlı basamakla saklar. İşlemlerin kaç anlamlı basamakla yapılacağını `getcontext().prec` değeri belirler (varsayılan değeri 28'dir); her işlemin sonucu bu sayıda anlamlı basamağa yuvarlanır.

Yani `Decimal` da bir kayan noktalı sayı sistemidir, ama tabanı 10'dur ve basamak sayısını biz seçeriz. `Decimal("0.1")` gibi bir dizeden kurulan ondalık sayılar tam saklanır; $\frac{1}{3}$ gibi onluk açılımı sonsuz olan sayılar ise yine yuvarlanır.

```
from decimal import Decimal, getcontext

print(Decimal("0.1") + Decimal("0.2"))
print(Decimal(0.1))               # float'tan kurulunca hatası da gelir
print(Decimal(1) / Decimal(3))    # varsayılan: 28 basamak
getcontext().prec = 50
print(Decimal(1) / Decimal(3))
```

Çıktı:

```
0.3
0.1000000000000000055511151231257827021181583404541015625
0.33333333333333333333333333
0.33333333333333333333333333333333333333333333333333
```

> ⚠ Decimal değerlerini dizeden kurun
>
> `Decimal(0.1)` yazıldığında önce `0.1` `float`'ı oluşur ve `Decimal` onun tam değerini, yani yukarıdaki uzun sayıyı saklar. Ondalık bir sayıyı tam saklamak için onu dize olarak verin: `Decimal("0.1")`.

Örnek 1.13 (Karekök İkinin Elli Basamağı). $\sqrt{2}$'yi 50 anlamlı basamakla hesaplayınız ve bulduğunuz sayının karesinin 2'den ne kadar farklı olduğunu bulunuz.

Çözüm.
Hassasiyeti ayarlayıp karekökü alalım. `Decimal` değerlerinin `sqrt` metodu karekökü, o anki hassasiyetle hesaplar:

```
from decimal import Decimal, getcontext

getcontext().prec = 50
r = Decimal(2).sqrt()
print(r)
print(r * r - 2)
print(2 ** 0.5)          # float ile karşılaştırma
```

Çıktı:

```
1.4142135623730950488016887242096980785696718753769
-1E-49
1.4142135623730951
```

Karenin farkı. `r`, $\sqrt{2}$'nin 50 anlamlı basamağa yuvarlanmış hâlidir ve $\sqrt{2}$'den küçüktür; bu yüzden karesi 2'den biraz küçük çıkar. Fark 10^{-49} mertebesindedir, yani ancak 50. anlamlı basamakta ortaya çıkar. `float` ise yalnız 17 anlamlı basamak yazdı ve sonuncusu bile $\sqrt{2}$'nin basamağı değildir: $\sqrt{2} = 1{,}41421356237309504\ldots$ olduğu hâlde `float` son basamağı 1 olarak yazdı. ■

 CC BY-NC-SA 4.0

Python'un yerleşik `round(x, n)` fonksiyonu x'i noktadan sonra n basamağa yuvarlar (n verilmezse en yakın tam sayıya). Tam ortadaki değerlerde "en yakın çift sayıya yuvarla" kuralını uygular; bu kural, çok sayıda yuvarlamada hataların hep aynı yöne birikmesini önler. Okulda öğrenilen "yarımı yukarı yuvarla" kuralı için `Decimal`'in `quantize` metodu kullanılır:

```
from decimal import Decimal, ROUND_HALF_UP

print(round(0.5), round(1.5), round(2.5))   # yarımlar çifte gider
x = Decimal("2.675")
print(x.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP))
```

Çıktı:

```
0 2 2
2.68
```

Bölümde gördüğümüz sayı türlerini şöyle özetleyebiliriz:

Tür	Değerler	Kesinlik	Ne zaman?
`int`	$\mathbb{Z}$	tam	sayma, sayılar teorisi, kombinatorik
`float`	53 ikili basamaklı sayılar	işlem başına bağıl hata $\leq 2^{-53}$	sayısal hesabın büyük çoğunluğu
`complex`	iki `float`	`float` gibi	karmaşık sayılarla sayısal hesap
`Fraction`	$\mathbb{Q}$	tam	tam kesirli sonuçlar, küçük hesaplar
`Decimal`	onluk kayan nokta	seçilen basamak sayısı	para hesabı, çok basamaklı hesap

1.13 Alıştırmalar

Aşağıdaki alıştırmalar bölümdeki sayı türlerini tanıdık matematik problemleri üzerinde kullanır; her çözüm çalıştırılmış kodu ve çıktısını içerir. Kodu önce kendiniz yazmayı deneyin, sonra çözümle karşılaştırın.

Alıştırma 1.1 (İki Üzeri Binin Basamakları). 2^{1000} sayısının kaç basamaklı olduğunu ve basamaklarının toplamını bulunuz.

Çözüm.
Basamakları dolaşalım. `for` döngüsü bir dizenin karakterlerini de tek tek dolaşabilir. `int("7")` gibi bir çağrı, tek rakamlı bir dizeyi tam sayıya çevirir.

```
n = 2 ** 1000
s = str(n)
total = 0
for c in s:           # dizenin karakterlerini tek tek dolaş
    total += int(c)   # karakteri rakama çevirip ekle
print(len(s), total)
```

Çıktı:

```
302 1366
```

Sağlama. Basamak sayısı $\lfloor 1000 \log_{10} 2 \rfloor + 1$ formülüyle de bulunur: $1000 \log_{10} 2 \approx 301{,}03$ olduğundan sonuç 302'dir. 2^{1000} sayısı 302 basamaklıdır ve basamaklarının toplamı 1366'dır. ■

Alıştırma 1.2 (Büyük Bir Kuvvetin Son Basamakları). 3^{1000} sayısının son 10 basamağını bulunuz.

Çözüm.
Kalana çevirelim. Bir sayının son 10 basamağı, onun 10^{10} ile bölümünden kalandır. Bu kalanı `pow` ile, 3^{1000}'i hesaplamadan bulabiliriz. Sağlama için doğrudan hesabı da yapalım; kalan 10 basamaktan kısa çıkarsa baştaki sıfırları da göstermek için `010d` biçimini (soldan sıfırla doldurulmuş 10 karakterlik tam sayı) kullanıyoruz:

```
r = pow(3, 1000, 10 ** 10)
print(f"{r:010d}")
print(3 ** 1000 % 10 ** 10 == r)
```

Çıktı:

```
2855220001
True
```

Sonuç. 3^{1000} sayısının son on basamağı 2855220001'dir; iki yol aynı kalanı verdi. ■

Alıştırma 1.3 (Bin Faktöriyelin Sonundaki Sıfırlar). 1000! sayısının sonunda kaç sıfır olduğunu bulunuz.

Çözüm.
Sayarak düşünelim. Sondaki her sıfır bir $10 = 2 \cdot 5$ çarpanıdır. 1000! içinde 2 çarpanları 5 çarpanlarından çok olduğundan sıfır sayısı 5 çarpanlarının sayısına eşittir. 1'den 1000'e kadar 5'in katları birer, 25'in katları birer fazla, 125'in katları bir fazla daha, 625'in katları bir fazla daha 5 çarpanı verir:

$$\left\lfloor \frac{1000}{5} \right\rfloor + \left\lfloor \frac{1000}{25} \right\rfloor + \left\lfloor \frac{1000}{125} \right\rfloor + \left\lfloor \frac{1000}{625} \right\rfloor = 200 + 40 + 8 + 1 = 249.$$

Python ile. Formülü `//` ile hesaplayalım; sağlama için 1000!'i dizeye çevirip sağdaki sıfırları sayalım. Dizenin `rstrip("0")` metodu sağdaki `0` karakterlerini atar, uzunluklar arasındaki fark atılan sıfır sayısıdır:

```
import math

n = 1000
print(n // 5 + n // 25 + n // 125 + n // 625)
s = str(math.factorial(n))
print(len(s) - len(s.rstrip("0")))   # sağdaki sıfırların sayısı
```

Çıktı:

```
249
249
```

Sonuç. 1000! sayısının sonunda 249 sıfır vardır. ■

Alıştırma 1.4 (Bir Milyon Saniye). 10^6 saniyenin kaç gün, saat, dakika ve saniye ettiğini `divmod` ile bulunuz.

Çözüm.
Birimleri sırayla ayıralım. Önce saniyeleri 60'a bölüp dakika ve artan saniyeyi, sonra dakikaları 60'a bölüp saat ve artan dakikayı, en son saatleri 24'e bölüp gün ve artan saati buluruz. `divmod`'un verdiği çifti eşzamanlı atamayla iki değişkene ayırıyoruz:

 CC BY-NC-SA 4.0

```
t = 10 ** 6
m, s = divmod(t, 60)     # dakika ve artan saniye
h, m = divmod(m, 60)     # saat ve artan dakika
d, h = divmod(h, 24)     # gün ve artan saat
print(d, h, m, s)
print(((d * 24 + h) * 60 + m) * 60 + s)   # sağlama
```

Çıktı:

```
11 13 46 40
1000000
```

Sonuç. 10^6 saniye 11 gün 13 saat 46 dakika 40 saniyedir; son satır bu parçalardan yeniden 10^6'yı elde ediyor. ■

Alıştırma 1.5 (Negatif Bölenle Tam Bölme). 17'yi -5'e Python'un `//` ve `%` işlemleriyle bölünüz ve sonucu, kalanın $0 \le r < |\,b\,|$ koşulunu sağladığı sayılar teorisi bölmesiyle karşılaştırınız.

Çözüm.
Python'un bölmesi. Tanım 1.6 gereği $q = \lfloor 17/(-5) \rfloor$, yani $q = \lfloor -3{,}4 \rfloor = -4$ ve $r = 17 - (-5)(-4) = -3$ olur; kalan bölen gibi negatiftir.
Sayılar teorisi bölmesi. Negatif bölenle bölmede kalan $0 \le r < |\,b\,| = 5$ aralığında seçilir: $17 = (-5)(-3) + 2$. Bu kalanı Python'da `a % abs(b)` verir, bölüm de $q = (a - r)/b$ eşitliğinden bulunur:

```
a, b = 17, -5
print(a // b, a % b, b * (a // b) + a % b)
r = a % abs(b)            # 0 <= r < |b| olan kalan
q = (a - r) // b
print(q, r, b * q + r)
```

Çıktı:

```
-4 -3 17
-3 2 17
```

Karşılaştırma. İki bölme de $a = bq + r$ eşitliğini sağlar, ama kalanları farklı aralıkta seçer: Python $(q, r) = (-4, -3)$, sayılar teorisi $(q, r) = (-3, 2)$ verir. Bölen pozitifken bu fark ortaya çıkmaz. ■

Alıştırma 1.6 (Makine Epsilonunu Deneyle Bulmak). $k = 50, 51, \ldots, 55$ için `1 + 2 ** -k == 1` karşılaştırmasının değerini yazdırarak $1 + 2^{-k}$ toplamının 1'e eşit çıktığı en küçük k'yi bulunuz ve sonucu makine epsilonuyla açıklayınız.

Çözüm.
Deney. Negatif kuvvet aldığımız için `2 ** -k` bir `float`'tır:

```
import sys

for k in range(50, 56):
    print(k, 1 + 2 ** -k == 1)
print(sys.float_info.epsilon == 2 ** -52)
```

Çıktı:

```
50 False
51 False
52 False
53 True
54 True
55 True
True
```

Açıklama. $k \le 52$ iken $1 + 2^{-k}$ sayısı tam olarak bir `float`'tır: mantisin k'inci ikili basamağı 1, diğerleri 0'dır. Bu yüzden toplam 1'den büyük kalır. $k = 53$ için $1 + 2^{-53} = 1 + \frac{\varepsilon}{2}$ sayısı 1 ile $1 + \varepsilon$'un tam ortasındadır ve "eşitlikte çifte yuvarla" kuralıyla 1'e yuvarlanır. Daha büyük k'lerde toplam 1'e daha da yakındır. Aranan en küçük değer $k = 53$'tür. ■

Alıştırma 1.7 (Onda Birin On Katı). `0.0`'dan başlayıp 0,1'i döngüyle on kez ekleyiniz. Sonucu 1 ile `==` ve `math.isclose` kullanarak karşılaştırınız; aynı toplamı `Fraction(1, 10)` ile de yapınız.

Çözüm.
İki toplamı birlikte kuralım.

```
import math
from fractions import Fraction

total = 0.0
exact = Fraction(0)
for k in range(10):
    total += 0.1
    exact += Fraction(1, 10)
print(total, total == 1, math.isclose(total, 1))
print(exact, exact == 1)
```

Çıktı:

```
0.9999999999999999 False True
1 True
```

Yorum. `float` toplam her adımda küçük bir yuvarlama hatası yaptı ve sonuç 1'in hemen altındaki `float` oldu; bu yüzden `== False` verdi. Fark 10^{-16} mertebesinde olduğundan `math.isclose` ile karşılaştırma `True` verdi. `Fraction` toplamı ise tam olarak 1'dir. ■

Alıştırma 1.8 (Sadeleşme Hatası). $f(x) = \dfrac{1 - \cos x}{x^2}$ fonksiyonunun $x \to 0$ iken limiti $\frac{1}{2}$'dir. $f(10^{-8})$'i Python ile doğrudan hesaplayınız ve sonucu, $1 - \cos x = 2\sin^2(x/2)$ özdeşliğiyle yeniden yazılmış formülün sonucuyla karşılaştırınız.

Çözüm.
İki formül.

```
import math

x = 1e-8
print(math.cos(x))                          # 1'e yuvarlandı
print((1 - math.cos(x)) / x ** 2)           # doğrudan formül
print(2 * math.sin(x / 2) ** 2 / x ** 2)    # özdeşlikle
```

Çıktı:

```
1.0
0.0
0.5
```

Ne oldu? $\cos x = 1 - \frac{x^2}{2} + \cdots$ olduğundan $x = 10^{-8}$ için $\cos x \approx 1 - 5 \cdot 10^{-17}$'dir. 1'in hemen altındaki `float` $1 - \frac{\varepsilon}{2}$ olduğundan, 1'e uzaklığı $\frac{\varepsilon}{4} \approx 5{,}55 \cdot 10^{-17}$'den küçük olan sayılar 1'e yuvarlanır. $5 \cdot 10^{-17}$ bu eşiğin altında kaldığı için `math.cos(x)` tam olarak 1 çıkar ve $1 - \cos x$ farkı 0 olur: birbirine çok yakın iki sayının farkı alınırken bütün anlamlı basamaklar kaybolmuştur. Bu olaya **yıkıcı sadeleşme** (catastrophic cancellation) denir. Özdeşlikle yazılan formülde fark alınmaz; $\sin(x/2)$ tam bağıl hassasiyetle hesaplanır ve sonuç doğru olarak 0,5 çıkar. ■

Alıştırma 1.9 (Bir Karmaşık Sayının Onuncu Kuvveti). $(1+i)^{10}$ sayısını Python ile hesaplayınız ve sonucu kutupsal biçimle sağlayınız.

Çözüm.
Doğrudan ve kutupsal yoldan hesap.

```
import cmath

z = 1 + 1j
print(z ** 10)
r, theta = cmath.polar(z)
print(r, theta)
print(cmath.rect(r ** 10, 10 * theta))
```

Çıktı:

```
32j
1.4142135623730951 0.7853981633974483
(9.797174393178832e-15+32.00000000000002j)
```

Elle sağlama. $|\,1+i\,| = \sqrt{2}$ ve $\arg(1+i) = \frac{\pi}{4}$ olduğundan

$$(1+i)^{10} = (\sqrt{2})^{10}\, e^{i\,10\pi/4} = 32\, e^{i\,5\pi/2} = 32\, e^{i\pi/2} = 32i$$

olur. Tam sayı kuvveti `**` ile alındığında Python ardışık çarpmalar yapar ve bu çarpımlarda bütün ara değerler küçük tam sayılar olduğundan sonuç tam olarak `32j` çıkar. Kutupsal yol ise $\sqrt{2}$ ve $\frac{\pi}{4}$'ün yuvarlanmış değerlerinden geçtiği için son basamaklarda küçük hatalar taşır. ■

Alıştırma 1.10 (Teleskopik Bir Toplam). $\displaystyle\sum_{k=1}^{20} \frac{1}{k(k+1)}$ toplamını `Fraction` ile tam olarak hesaplayınız ve $\dfrac{n}{n+1}$ formülüyle karşılaştırınız.

Çözüm.
Formül. $\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$ olduğundan toplam teleskopiktir:

$$\sum_{k=1}^{n} \frac{1}{k(k+1)} = 1 - \frac{1}{n+1} = \frac{n}{n+1}.$$

Python ile.

```
from fractions import Fraction

n = 20
```

 CC BY-NC-SA 4.0

```
S = Fraction(0)
for k in range(1, n + 1):
    S += Fraction(1, k * (k + 1))
print(S, S == Fraction(n, n + 1))
```

Çıktı:

```
20/21 True
```

Sonuç. Toplam tam olarak $\frac{20}{21}$'dir ve formülle uyuşur. ■

Alıştırma 1.11 (Beklenmedik Bir Yuvarlama). `round(2.675, 2)` çağrısının neden `2.68` değil `2.67` verdiğini açıklayınız.

Çözüm.
Saklanan değere bakalım. `Decimal(2.675)` yazımı, `2.675` `float`'ının tam değerini gösterir:

```
from decimal import Decimal

print(round(2.675, 2))
print(Decimal(2.675))     # float 2.675'in tam değeri
```

Çıktı:

```
2.67
2.674999999999999822364316059974953532218933105468 75
```

Açıklama. 2,675'in ikili açılımı sonsuzdur; ona en yakın `float` 2,675'ten biraz küçüktür. `round` bu saklanan değeri yuvarlar. Değer 2,67 ile 2,68'in tam ortasında değil, ortanın biraz altında olduğundan doğru yuvarlama 2,67'dir. Buradaki sorun bir yuvarlama kuralı değil, 2,675'in `float` olarak tam saklanamamasıdır. Ondalık sayıları tam yuvarlamak gerekiyorsa `Decimal("2.675")` ve `quantize` kullanılır. ■

Alıştırma 1.12 (Aynı Türden Beş Kart). 52 kartlık bir desteden rastgele çekilen 5 kartın hepsinin aynı türden (hepsi kupa, hepsi karo, hepsi maça ya da hepsi sinek) olma olasılığını tam kesir olarak ve yüzde olarak bulunuz.

Çözüm.
Sayalım. Bütün eşit olasılıklı eller $\binom{52}{5}$ tanedir. Aynı türden bir el için önce türü (4 yol), sonra o türün 13 kartından 5'ini ($\binom{13}{5}$ yol) seçeriz. Olasılık

$$P = \frac{4\binom{13}{5}}{\binom{52}{5}}$$

olur.
Python ile. Binom katsayıları `math.comb` ile tam sayı olarak, oran da `Fraction` ile tam kesir olarak hesaplanır:

```
import math
from fractions import Fraction

total = math.comb(52, 5)
flush = 4 * math.comb(13, 5)
P = Fraction(flush, total)
```

 CC BY-NC-SA 4.0

```
print(total, flush)
print(P, f"{float(P):.4%}")
```

Çıktı:

```
2598960 5148
33/16660 0.1981%
```

Sonuç. $P = \frac{5148}{2598960} = \frac{33}{16660} \approx 0{,}00198$, yani yaklaşık yüzde 0,198'dir: bin elden yalnız ikisi aynı türdendir. ■

Bu bölümde Python'ı bir hesap makinesi gibi kullandık: kod yukarıdan aşağıya bir kez çalıştı ve döngüyü yalnız bir önizleme olarak gördük. Gerçek programlar karar verir, bir koşul sağlanana kadar tekrar eder ve tekrar tekrar kullanılabilecek parçalar tanımlar. Bir sonraki bölüm Kontrol Yapıları ve Fonksiyonlar, `if`, `for`, `while` yapılarını ve kendi fonksiyonlarımızı yazmayı Öklid algoritması, Collatz dizisi ve Newton yöntemiyle karekök gibi tanıdık problemler üzerinde öğretiyor.

 CC BY-NC-SA 4.0

2. Kontrol Yapıları ve Fonksiyonlar

Python ile İlk Adımlar bölümünde Python'u güçlü bir hesap makinesi gibi kullandık: her satır bir kez ve yukarıdan aşağıya çalıştı. Oysa matematikteki hesapların çoğu böyle düz bir çizgi izlemez. "Diskriminant pozitifse iki gerçel kök vardır" bir kararlı. $\sum_{k=1}^{n} k^2$ aynı işlemin n kez tekrarıdır. Öklid algoritması ise "kalan sıfır olana dek böl" der ve kaç adım süreceğini önceden söylemez.
Bu bölümde bu üç kalıbın Python karşılıklarını öğreneceğiz: karar için `if`, sayısı belli tekrar için `for`, koşula bağlı tekrar için `while`. Ardından bir hesabı bir ad altında toplayan **fonksiyonlara** ve kendini çağıran fonksiyonlara, yani **özyinelemeye** geçeceğiz. Her aracı tanıdık bir matematik problemi üzerinde göreceğiz: ebob, Collatz dizisi, karekökün Newton yöntemiyle hesabı, asallık testi ve Fibonacci sayıları.
Bölümdeki her kod parçası kendi başına çalışır. Bir `.py` dosyasına ya da bir Jupyter hücresine yapıştırıp çalıştırdığınızda altındaki çıktının aynısını alırsınız.

2.1 Koşullar ve if Deyimi

Bir program, bir koşulun doğru olup olmamasına göre farklı yollar izleyebilmelidir. Python ile İlk Adımlar bölümünde `7 > 2` karşılaştırmasının değerinin `True` olduğunu görmüştük; şimdi bu tür ifadeleri, yani koşulları yakından tanıyalım.

Tanım 2.1 (Mantıksal Değer ve Koşul). Python'da **mantıksal değer** (Boolean) tipi `bool`'dur ve yalnız iki değeri vardır: `True` (doğru) ve `False` (yanlış). `==` (eşit), `!=` (eşit değil), `<`, `<=`, `>`, `>=` **karşılaştırma işleçleri** iki değeri karşılaştırıp bir `bool` üretir. Mantıksal değerler `and` (ve), `or` (veya), `not` (değil) işleçleriyle birleştirilir. Değeri `True` ya da `False` olan her ifadeye **koşul** denir.

Yani bir koşul, Python'a sorulmuş bir evet-hayır sorusudur. Matematikteki $p \wedge q$, $p \vee q$ ve $\neg p$ burada `p and q`, `p or q` ve `not p` olarak yazılır. $1 < x \leq 10$ gibi zincirleme bir eşitsizlik de matematikteki gibi tek parça yazılabilir.

```
x = 7
print(x > 5)             # 7, 5'ten büyük mü?
print(x % 2 == 0)        # 7 çift mi?
print(x != 7)            # 7, 7'den farklı mı?
print(1 < x <= 10)       # zincirleme karşılaştırma: 1 < x ve x <= 10
print(x > 5 and x % 2 == 0)
print(x > 5 or x % 2 == 0)
print(not x > 5)
print(type(x > 5))
```

Çıktı:

```
True
False
False
True
False
True
False
<class 'bool'>
```

Son satır, karşılaştırmanın sonucunun gerçekten `bool` tipinde olduğunu gösteriyor. Karşılaştırmalar aritmetik işlemlerden sonra yapılır: `x % 2 == 0` önce `x % 2`'yi hesaplar, sonra sonucu `0` ile karşılaştırır. Mantıksal işleçlerde öncelik sırası `not`, `and`, `or` biçimindedir; emin olmadığınızda parantez kullanın.

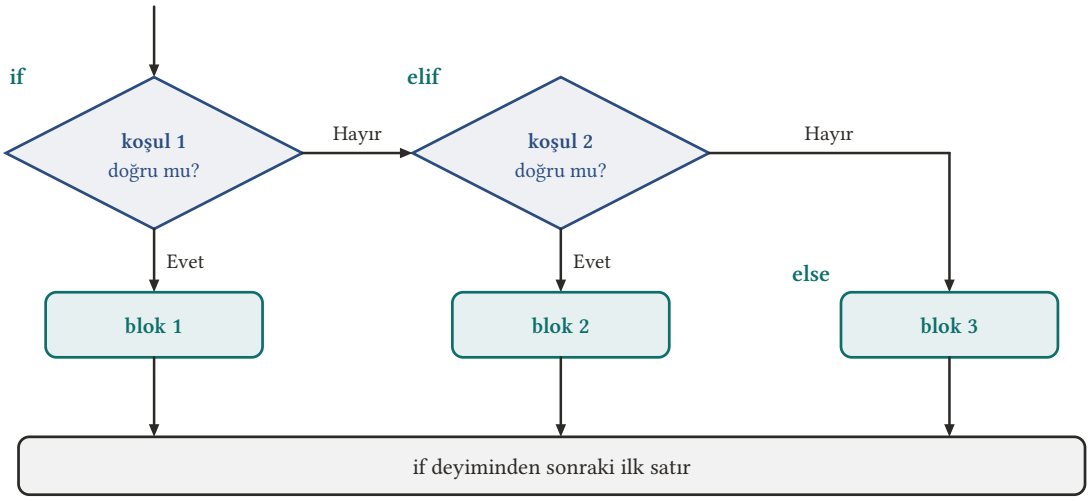
Tanım 2.2 (if, elif ve else Deyimi). Bir `if` deyiminin genel biçimi şudur:

 CC BY-NC-SA 4.0

```
if koşul_1:
    blok_1
elif koşul_2:
    blok_2
else:
    blok_3
```

Python koşulları yukarıdan aşağıya sırayla sınar ve doğru çıkan **ilk** koşulun bloğunu çalıştırır; diğer bloklar atlanır. Hiçbir koşul doğru değilse `else` bloğu çalışır. `elif` kısmı istenildiği kadar tekrarlanabilir; `elif` ve `else` kısımları hiç yazılmayabilir de. Her `if`, `elif` ve `else` satırı iki nokta (`:`) ile biter. Ona bağlı blok, aynı miktarda (gelenek olarak dört boşluk) **girintili** satırlardan oluşur.

Yani `if` deyimi, parçalı tanımlı bir fonksiyon gibi okunur: ilk uyan durum geçerlidir. Python'da blokların nerede başlayıp bittiğini süslü parantezler değil girinti belirler. Girintisi bozuk bir program ya hata verir ya da beklenmedik bir iş yapar.



Şekil 2.1: Bir `if/elif/else` deyiminin akışı. Koşullar yazıldıkları sırayla sınanır; doğru çıkan ilk koşulun bloğu çalışır ve geri kalanlar atlanır. Hiçbir koşul doğru değilse `else` bloğu çalışır. Hangi yoldan gidilirse gidilsin, sonunda deyimden sonraki satıra geçilir.

Örnek 2.1 (İkinci Dereceden Denklemin Kökleri). $x^2 - 2x + 5 = 0$ denkleminin köklerini, diskriminantın işaretine göre üç durumu ayıran bir programla bulunuz.

Çözüm.
$a \neq 0$ olmak üzere $ax^2 + bx + c = 0$ denkleminin diskriminantı $\Delta = b^2 - 4ac$'dir. $\Delta > 0$ ise iki farklı gerçel kök, $\Delta = 0$ ise çift katlı bir kök vardır. $\Delta < 0$ ise kökler, birbirinin karmaşık eşleniği olan

$$x_{1,2} = -\frac{b}{2a} \pm \frac{\sqrt{-\Delta}}{2a}\, i$$

sayılarıdır. Bu üç durum, bir `if-elif-else` deyiminin üç dalına karşılık gelir.

```
import math

a, b, c = 1, -2, 5
delta = b**2 - 4*a*c
if delta > 0:
    x1 = (-b + math.sqrt(delta)) / (2*a)
```

 CC BY-NC-SA 4.0

```
    x2 = (-b - math.sqrt(delta)) / (2*a)
    print(f"İki farklı gerçel kök: {x1} ve {x2}")
elif delta == 0:
    print(f"Çift katlı kök: {-b / (2*a)}")
else:
    re = -b / (2*a)
    im = math.sqrt(-delta) / (2*a)
    print(f"Gerçel kök yok; karmaşık kökler {re} ± {im}i")
print("Diskriminant:", delta)
```

Çıktı:

```
Gerçel kök yok; karmaşık kökler 1.0 ± 2.0i
Diskriminant: -16
```

`a, b, c = 1, -2, 5` eşzamanlı ataması üç katsayıyı birden verir. Burada $\Delta = 4 - 20 = -16 < 0$ olduğundan ilk iki koşul yanlış çıktı ve `else` bloğu çalıştı: kökler $1 \pm 2i$'dir. Katsayıları değiştirip programı yeniden çalıştırırsanız diğer dallara da girersiniz. Bunu bölümün ilerleyen kısmında bir fonksiyonla daha rahat yapacağız. ■

İki değerden birini seçmek için `if` deyiminin tek satırlık bir kısaltması da vardır. **`a if koşul else b` koşullu ifadesi**, koşul doğruysa `a`'nın, yanlışsa `b`'nin değerini alır. Örneğin

$$|\,x\,| = \begin{cases} x, & x \geq 0 \\ -x, & \text{diğer durumlarda} \end{cases}$$

tanımı bu biçimde tek satıra sığar:

```
x = -3.5
abs_x = x if x >= 0 else -x
n = 17
parity = "çift" if n % 2 == 0 else "tek"
print(abs_x, parity)
```

Çıktı:

```
3.5 tek
```

 CC BY-NC-SA 4.0

⚠ Ondalık sayıları == ile karşılaştırmayın

`=` atama, `==` karşılaştırmadır; `if x = 3:` yazmak sözdizimi hatası verir. Daha sinsi tuzak `float` sayıların eşitliğidir. Python ile İlk Adımlar bölümünde gördüğümüz gibi 0,1 ve 0,2 ikili sistemde tam gösterilemez, bu yüzden toplamları `0.3` ile tam eşit çıkmaz. Ondalık sonuçları `math.isclose` ile ya da `abs(a - b) < tol` biçiminde bir toleransla karşılaştırın.

```
import math

s = 0.1 + 0.2
if s == 0.3:
    print("eşit")
else:
    print("eşit değil, s =", s)
if math.isclose(s, 0.3):
    print("math.isclose: yeterince yakın")
```

Çıktı:

```
eşit değil, s = 0.30000000000000004
math.isclose: yeterince yakın
```

2.2 for Döngüsü ve range

Bir işlemi önceden bilinen sayıda tekrarlamak için `for` döngüsünü kullanırız. Döngünün dolaşacağı tam sayıları çoğunlukla `range` üretir; önce onu tanıyalım.

Tanım 2.3 (range Nesnesi). a, b ve $s \neq 0$ tam sayılar olsun. `range(a, b, s)` nesnesi $a, a+s, a+2s, \ldots$ tam sayılarını sırayla verir; $s > 0$ ise b'ye ulaşmadan, $s < 0$ ise b'ye inmeden durur. Örneğin $s > 0$ için verilen sayılar

$$\{\, a + ks : k = 0, 1, 2, \ldots \ \text{ ve } \ a + ks < b \,\}$$

kümesinin küçükten büyüğe sıralanmış elemanlarıdır. `range(a, b)` yazımında adım $s = 1$ alınır; `range(b)` ise `range(0, b)` demektir.

Yani `range(a, b)`, $[a, b)$ yarı açık aralığındaki tam sayıları verir: başlangıç dahil, bitiş hariç. Bu yüzden $1, 2, \ldots, n$ için `range(1, n + 1)` yazılır; `range(n)` ise $0, 1, \ldots, n-1$ olmak üzere tam n sayı verir. Bir `range`'in elemanlarını görmek için onu `list(...)` ile listeye çeviriyoruz. Listeleri Veri Yapıları ve Comprehension bölümünde ayrıntılı göreceğiz.

```
print(list(range(5)))
print(list(range(2, 11, 3)))
print(list(range(10, 0, -2)))
print(list(range(3, 3)))
print(len(range(1, 101)))
```

Çıktı:

```
[0, 1, 2, 3, 4]
[2, 5, 8]
[10, 8, 6, 4, 2]
[]
100
```

`range(3, 3)` boştur, çünkü $[3, 3)$ aralığında tam sayı yoktur. `len` ise bir `range`'in kaç sayı verdiğini söyler.

Tanım 2.4 (for Döngüsü).

 CC BY-NC-SA 4.0

```
for değişken in nesne:
    blok
```

biçimindeki **`for` döngüsü**, `nesne`'nin (örneğin bir `range`'in ya da sonraki bölümde göreceğimiz bir listenin) elemanlarını sırayla `değişken`'e atar ve her atamadan sonra girintili bloğu bir kez çalıştırır. Bloğun her çalışmasına döngünün bir **adımı** (iterasyon) denir.

Yani `for k in range(1, n + 1):` satırı, matematikteki "$k = 1, 2, \ldots, n$ için" ifadesinin birebir karşılığıdır. Döngü değişkeni sıradan bir değişkendir ve döngü bittiğinde son aldığı değeri korur.

```
for k in range(1, 6):
    print(f"{k:2} {k**2:4} {k**3:5}")
print("döngü bitti, k =", k)
```

Çıktı:

```
 1    1     1
 2    4     8
 3    9    27
 4   16    64
 5   25   125
döngü bitti, k = 5
```

f-dizgisindeki `{k**2:4}` gibi biçim belirteçleri sayıyı 4 karakterlik bir alana sağa yaslar, böylece tablo hizalı çıkar.
Döngülerin en sık kullanımı bir toplam ya da çarpım biriktirmektir. $\sum$ ve $\prod$ sembollerinin özyineli tanımı (bkz. Analiz 1) tam olarak bu kalıbı anlatır: $S_0 = 0$ ve $S_k = S_{k-1} + a_k$.

Döngüyle toplam ve çarpım üç adımda

1. Biriktirici değişkeni işlemin etkisiz elemanıyla başlatın: toplam için `s = 0`, çarpım için `p = 1`.
2. Döngünün her adımında terimi ekleyin (`s += terim`) ya da terimle çarpın (`p *= terim`).
3. Sonucu döngü bittikten sonra kullanın; biriktiriciyi döngünün içinde yeniden başlatmayın.

Örnek 2.2 (Karelerin Toplamı). $\sum_{k=1}^{100} k^2$ toplamını bir döngüyle hesaplayıp

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

formülünün verdiği değerle karşılaştırınız.

Çözüm.
Tarifin üç adımı kodda yorum olarak işaretli. `s += k**2`, `s = s + k**2` atamasının kısa yazılışıdır.

```
n = 100
s = 0                       # 1. adım: toplamın etkisiz elemanı
for k in range(1, n + 1):
    s += k**2               # 2. adım: terimi biriktiriciye ekle
print("döngüyle:", s)       # 3. adım: sonucu döngüden sonra kullan
print("formülle:", n * (n + 1) * (2*n + 1) // 6)
```

Çıktı:

```
döngüyle: 338350
formülle: 338350
```

İki değer aynıdır:

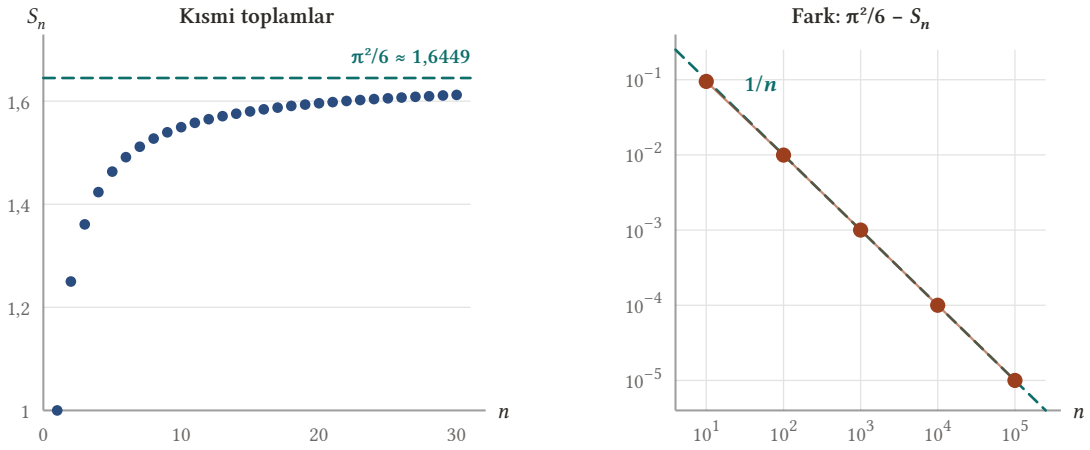
 CC BY-NC-SA 4.0

$$\frac{100 \cdot 101 \cdot 201}{6} = \frac{2030100}{6} = 338350.$$

Formülde `/` yerine `//` kullandık. $n(n+1)(2n+1)$ her zaman 6'ya bölündüğünden tam bölme kesin sonucu `int` olarak verir; `/` ise `338350.0` gibi bir `float` döndürürdü. ■

Bir döngünün bloğu başka bir döngü içerebilir; buna **iç içe döngü** denir. İç döngü, dış döngünün her adımında baştan sona bir kez çalışır.

Örnek 2.3 (Kareler Tersleri Serisinin Kısmi Toplamları). $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$ kısmi toplamlarını $n = 10, 10^2, \ldots, 10^5$ için hesaplayıp serinin toplamı olan $\pi^2/6$ ile farklarını bulunuz.



Şekil 2.2: Sol: $S_n = 1 + 1/4 + \cdots + 1/n^2$ kısmi toplamları artarak $\pi^2/6 \approx 1{,}6449$ değerine yaklaşır. Sağ: iç içe döngülü kodun yazdırdığı farklar ($n = 10, 100, \ldots, 100000$) logaritmik eksenlerde; noktalar kesikli $1/n$ doğrusunun hemen altındadır, yani fark yaklaşık $1/n$ kadardır.

Çözüm.
Serinin yakınsadığını p-serisi ölçütü (bkz. Analiz 2), toplamının $\pi^2/6$ olduğunu ise Fourier serileri (bkz. Analiz 3) verir. Dış döngü $j = 1, \ldots, 5$ için $n = 10^j$'yi seçer, iç döngü S_n'yi tarifteki gibi biriktirir. `s = 0.0` satırı dış döngünün içinde ama iç döngünün dışında durur: her n için yeni bir toplam başlar, toplamın kendisi ise iç döngü boyunca birikir.

```
import math

limit = math.pi**2 / 6
for j in range(1, 6):
    n = 10**j
    s = 0.0
    for k in range(1, n + 1):
        s += 1 / k**2
    print(f"n = {n:6}  S_n = {s:.10f}  fark = {limit - s:.4e}")
```

Çıktı:

```
n =     10  S_n = 1.5497677312  fark = 9.5166e-02
n =    100  S_n = 1.6349839002  fark = 9.9502e-03
n =   1000  S_n = 1.6439345667  fark = 9.9950e-04
n =  10000  S_n = 1.6448340718  fark = 9.9995e-05
n = 100000  S_n = 1.6449240669  fark = 9.9999e-06
```

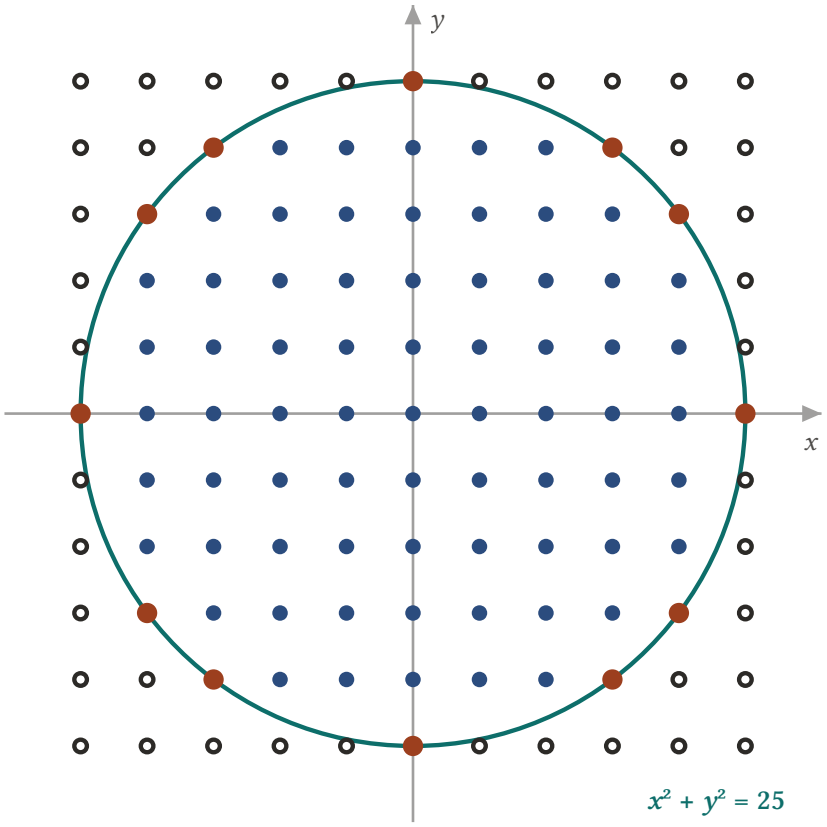
 CC BY-NC-SA 4.0

Fark her seferinde yaklaşık on kat küçülüyor; n ile farkın çarpımı hep 1'e yakın. Gerçekten de fark

$$\frac{\pi^2}{6} - S_n = \sum_{k=n+1}^{\infty} \frac{1}{k^2} \approx \int_n^{\infty} \frac{dx}{x^2} = \frac{1}{n}$$

kuyruğudur. Yani bu seriyle d ondalık basamak doğruluk için kabaca 10^d terim toplamak gerekir: seri yavaş yakınsar. ■

Örnek 2.4 (Bir Dairedeki Kafes Noktaları). $N(r)$, $x^2 + y^2 \le r^2$ eşitsizliğini sağlayan (x, y) tam sayı çiftlerinin sayısı olsun. $r = 5, 10, 15, 20$ için $N(r)$'yi hesaplayıp dairenin alanı πr^2 ile karşılaştırınız.



Şekil 2.3: $r = 5$ için kodun saydığı kafes noktaları: dairenin içindeki 69 nokta (mavi) ile çemberin tam üstündeki 12 nokta (turuncu) birlikte $N(5) = 81$ eder; $\pi \cdot 5^2 \approx 78{,}54$ ile karşılaştırın. Boş halkalar karenin içinde kalıp daireye girmeyen noktalardır.

Çözüm.
Böyle bir çiftte $|\,x\,| \le r$ ve $|\,y\,| \le r$ olmalıdır, bu yüzden x ve y'yi $-r, \dots, r$ arasında dolaştırmak yeter. İki iç döngü karedeki $(2r+1)^2$ noktanın hepsini dener; en dıştaki üçüncü döngü ise r'yi değiştirir.

```
import math

for r in range(5, 21, 5):
    count = 0
    for x in range(-r, r + 1):
        for y in range(-r, r + 1):
            if x*x + y*y <= r*r:
```

 CC BY-NC-SA 4.0

```
                count += 1
    area = math.pi * r**2
    print(f"r = {r:2}  N(r) = {count:4}  pi r^2 = {area:8.2f}")
```

Çıktı:

```
r =  5  N(r) =   81  pi r^2 =    78.54
r = 10  N(r) =  317  pi r^2 =   314.16
r = 15  N(r) =  709  pi r^2 =   706.86
r = 20  N(r) = 1257  pi r^2 =  1256.64
```

$N(r)$ ile πr^2 arasındaki fark, alanın yanında çok küçüktür. Her kafes noktasının merkezde durduğu birim kareler düşünülürse bu karelerin toplam alanı $N(r)$'dir ve daireyi neredeyse tam kaplar. $r = 20$ için iki iç döngü $41^2 = 1681$ çift dener; toplam iş, karedeki nokta sayısıyla, yani yaklaşık $4r^2$ ile orantılıdır. ■

2.3 while Döngüsü

`for` döngüsü, tekrar sayısı baştan belli olduğunda kullanışlıdır. Öklid algoritmasında ya da bir iterasyonu belirli bir hassasiyete ulaşana dek sürdürürken ise kaç adım gerektiğini önceden bilmeyiz; o zaman `while` kullanılır.

Tanım 2.5 (while Döngüsü).

```
while koşul:
    blok
```

biçimindeki **`while` döngüsü** önce koşulu sınar. Koşul doğruysa bloğu bir kez çalıştırır ve koşulu yeniden sınar. Koşul yanlış çıktığı anda döngü biter ve program döngüden sonraki satırdan devam eder.

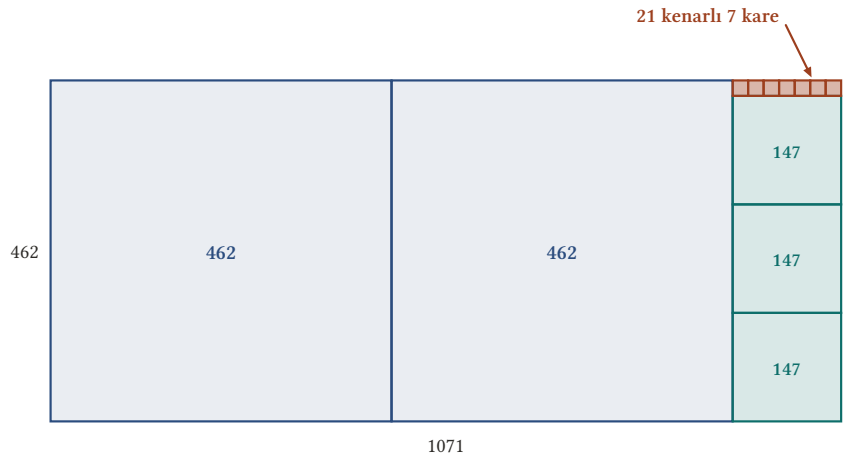
Yani `while`, "koşul sağlandığı sürece tekrarla" demektir. Koşul ilk sınamada yanlışsa blok hiç çalışmaz; koşul hiç yanlış olmazsa döngü sonsuza dek sürer. Bu yüzden blok, koşuldaki değişkenleri eninde sonunda koşulu bozacak biçimde değiştirmelidir.
Öklid algoritmasının dayandığı olgu şudur.

Önerme 2.1 (EBOB ve Bölme Kalanı). *a, b tam sayılar ve $b \neq 0$ olsun. $a = qb + r$ olacak biçimde q, r tam sayıları için $\gcd(a, b) = \gcd(b, r)$'dir. Ayrıca $\gcd(a, 0) = |\,a\,|$'dır.*

$r = a - qb$ olduğundan (a, b) ile (b, r) çiftlerinin ortak bölenleri aynıdır; ispat için bkz. Sayılar Teorisi. r olarak bölme kalanını seçip önermeyi tekrar tekrar uygulamak Öklid algoritmasıdır. Kalanlar kesin azalan doğal sayılar olduğundan bir gün sıfır olur ve son sıfırdan farklı kalan ebob'dur.

Örnek 2.5 (Öklid Algoritması). $\gcd(1071, 462)$'yi Öklid algoritmasıyla, her bölmeyi yazdırarak hesaplayınız.

 CC BY-NC-SA 4.0



Şekil 2.4: Öklid algoritması geometrik olarak: 1071 × 462 dikdörtgeninden sığdığı kadar büyük kare kesilir. 462 kenarlı 2 kare, kalan 147 × 462 şeritten 147 kenarlı 3 kare, kalan 147 × 21 şeritten 21 kenarlı 7 kare çıkar ve hiçbir şey artmaz. Bölümler 2, 3, 7 kare sayılarıdır; son karenin kenarı 21 = ebob(1071, 462)'dir.

Çözüm.
Önerme 2.1'a göre (a, b) çiftini $(b,\ a \bmod b)$ ile değiştirmek ebob'u korur. Bunu b sıfır olana dek yaparız. Kaç bölme gerektiğini önceden bilmediğimiz için döngüyü `while` ile yazarız. `divmod(a, b)` bölümü ve kalanı birlikte verir.

```
a, b = 1071, 462
while b != 0:
    q, r = divmod(a, b)
    print(f"{a} = {q} * {b} + {r}")
    a, b = b, r
print("ebob =", a)
```

Çıktı:

```
1071 = 2 * 462 + 147
462 = 3 * 147 + 21
147 = 7 * 21 + 0
ebob = 21
```

Son sıfırdan farklı kalan 21'dir ve döngü bittiğinde bu değer `a`'dadır. Gerçekten $1071 = 3^2 \cdot 7 \cdot 17$ ve $462 = 2 \cdot 3 \cdot 7 \cdot 11$ olduğundan ebob $3 \cdot 7 = 21$'dir. Şekilde kare sayıları (2, 3, 7) bölümlere, karelerin kenarları (462, 147, 21) bölenlere karşılık gelir. ■

 CC BY-NC-SA 4.0

⚠ Eşzamanlı atama ile ardışık atama farklıdır

`a, b = b, r` satırı, Python ile İlk Adımlar bölümünde gördüğümüz eşzamanlı atamadır: önce sağ taraftaki bütün değerler hesaplanır, sonra sola atanır. Aynı işi iki ayrı satırla yapmaya çalışmak yanlış sonuç verir, çünkü ikinci satır `a`'nın yeni değerini görür:

```
a, b = 1071, 462
a = b          # a artık 462
b = a % b      # yanlış: 462 % 462 hesaplanır
print(a, b)
```

Çıktı:

```
462 0
```

Bu durumda döngü ilk adımda biter ve ebob yanlışlıkla 462 çıkar.

`while` döngüsünün en ünlü örneklerinden biri, her başlangıç değeri için durup durmadığını bugün bile kimsenin bilmediği bir döngüdür.

Tanım 2.6 (Collatz Dizisi). Bir n_0 pozitif tam sayısından başlayan **Collatz dizisi**

$$n_{k+1} = \begin{cases} n_k/2, & n_k \text{ çift ise} \\ 3n_k + 1, & n_k \text{ tek ise} \end{cases}$$

kuralıyla tanımlanır. Dizinin 1'e ilk ulaştığı k indisine n_0'ın **adım sayısı** denir.

Yani çift sayılar yarıya iner, tek sayılar ise $3n+1$'e sıçrar; 1'e ulaşan dizi bundan sonra $1, 4, 2, 1, \ldots$ döngüsünde döner. **Collatz sanısı**, her başlangıç değerinden sonlu sayıda adımda 1'e ulaşıldığını söyler. Sanı çok büyük sayılara kadar bilgisayarla doğrulanmıştır ama ispatı yoktur. Adım sayısı için bir formül bilmediğimizden dizi ancak `while` ile izlenebilir.

Örnek 2.6 (Collatz Dizisi: Başlangıç Değeri 27). $n_0 = 27$ için Collatz dizisinin adım sayısını ve dizinin en büyük terimini bulunuz.



Şekil 2.5: 27'den başlayan Collatz dizisinin terimleri. Dizi 111 adımda 1'e iner; yol boyunca 77. adımda 9232 değerine kadar tırmanır. Başlangıç değeri küçük olsa da adım sayısını önceden kestirmek zordur; bu yüzden döngü `while` ile yazılır.

Çözüm.
Döngü $n \neq 1$ olduğu sürece döner. Her adımda n'yi kurala göre günceller, adım sayacını bir artırır ve o ana kadarki en büyük terimi Python'un hazır `max` fonksiyonuyla saklar. `n // 2` tam bölmedir; n çift olduğundan sonuç kesin bir `int`'tir.

 CC BY-NC-SA 4.0

```
n = 27
steps = 0
peak = n
while n != 1:
    if n % 2 == 0:
        n = n // 2
    else:
        n = 3*n + 1
    steps += 1
    peak = max(peak, n)
print("adım sayısı:", steps)
print("en büyük terim:", peak)
```

Çıktı:

```
adım sayısı: 111
en büyük terim: 9232
```

Küçük bir başlangıç değeri 111 adım sürer ve yolda 9232'ye kadar tırmanır. ■

⚠ Sonsuz döngü ve tam bölme

Koşulu hiç bozulmayan bir `while` döngüsü durmaz. Örneğin Collatz döngüsünde `n`'yi güncellemeyi unutursanız program sonsuza dek çalışır. Böyle bir programı terminalde Ctrl+C ile, Jupyter'de ise çekirdeği durdurarak kesebilirsiniz. Güvenlik için iterasyon sayısına bir üst sınır koymak iyi bir alışkanlıktır; bölümün Newton örneklerinde `max_iter` bunu yapar. Ayrıca `n / 2` yazmak `n`'yi `float`'a çevirir. 2^{53}'ten büyük tam sayıların hepsi `float` olarak tam tutulamadığından dizi bozulabilir. Tam sayı dizilerinde `//` kullanın.

2.4 break, continue ve else

Bazen döngüyü koşulun belirlediği yerden önce bitirmek ya da bir adımın geri kalanını atlamak isteriz. Bunun için iki deyim vardır.

Tanım 2.7 (break ve continue). Bir döngünün bloğunda **`break`** çalıştığında döngü hemen sona erer ve program döngüden sonraki satırdan devam eder. **`continue`** çalıştığında ise o adımın geri kalanı atlanır ve döngü bir sonraki adıma geçer. İç içe döngülerde ikisi de yalnız içinde bulundukları en içteki döngüyü etkiler.

Yani `break` "aradığımı buldum, dur", `continue` ise "bu adım beni ilgilendirmiyor, sıradakine geç" demektir. Çıkış koşulu ancak bloğun ortasında hesaplanabiliyorsa `while True:` ile sonu olmayan bir döngü kurup çıkışı içerideki bir `break` ile yapmak yaygın bir kalıptır.

Örnek 2.7 (Faktöriyelin Bir Gugolü Aştığı Yer). 10^{100} sayısına gugol (googol) denir. $n! > 10^{100}$ eşitsizliğini sağlayan en küçük n doğal sayısını bulunuz.

Çözüm.
$n!$'i `f` değişkeninde biriktirip n'yi birer artırırız; `f` ilk kez 10^{100}'ü aştığında `break` ile döngüden çıkarız. Python'un tam sayıları istenildiği kadar büyük olabildiğinden `f` kesin olarak hesaplanır.

```
import math

n = 1
f = 1                       # f = n!
```

 CC BY-NC-SA 4.0

```
while True:
    n += 1
    f *= n
    if f > 10**100:
        break               # ilk kez 10^100'ü aştı: döngüden çık
print("n =", n)
print("log10(n!) =", round(math.log10(f), 3))
print("log10((n-1)!) =", round(math.log10(f // n), 3))
```

Çıktı:

```
n = 70
log10(n!) = 100.078
log10((n-1)!) = 98.233
```

$\log_{10}(70!) \approx 100{,}078 > 100$ ve $\log_{10}(69!) \approx 98{,}233 < 100$ olduğundan aranan sayı $n = 70$'tir. Son satırda $69! = 70!/70$ eşitliğini `f // n` ile kullandık. ■

Örnek 2.8 (Bin ile Aralarında Asal Sayılar). $1 \le k \le 1000$ ve $\gcd(k, 1000) = 1$ koşullarını sağlayan k sayılarının kaç tane olduğunu ve toplamlarını bulunuz.

Çözüm.
Döngü $k = 1, \ldots, 1000$ sayılarını dolaşır. 1000 ile ortak böleni olan bir k geldiğinde `continue` adımın geri kalanını atlar; böylece sayaç ve toplam yalnız aralarında asal sayılarla güncellenir. Ebob için Python'un hazır `math.gcd` fonksiyonunu kullanıyoruz.

```
import math

n = 1000
count = 0
total = 0
for k in range(1, n + 1):
    if math.gcd(k, n) != 1:
        continue            # n ile ortak böleni var: bu k'yı atla
    count += 1
    total += k
print("n ile aralarında asal olanların sayısı:", count)
print("bunların toplamı:", total)
print("n * phi(n) / 2 =", n * count // 2)
```

Çıktı:

```
n ile aralarında asal olanların sayısı: 400
bunların toplamı: 200000
n * phi(n) / 2 = 200000
```

Sayı, Euler fonksiyonunun değeridir (bkz. Sayılar Teorisi):

$$\phi(1000) = 1000\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{5}\right) = 400.$$

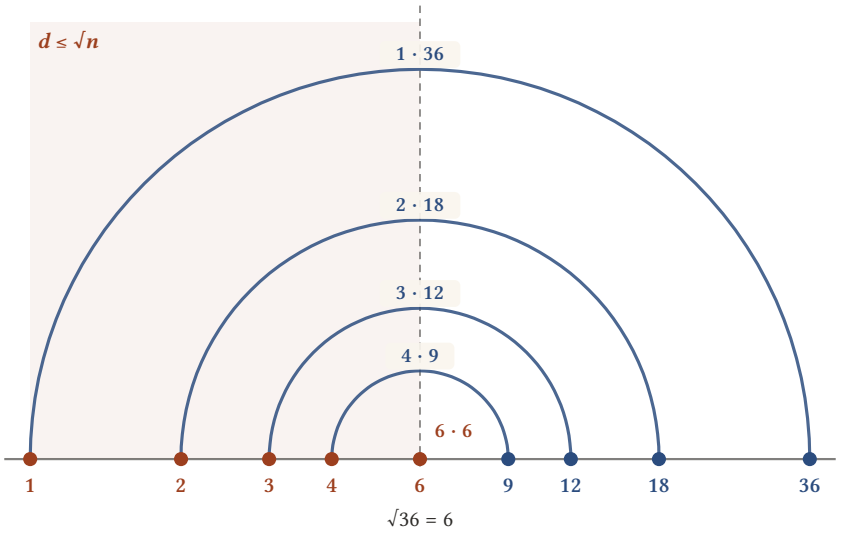
Toplamın $1000 \cdot 400/2$ çıkması tesadüf değildir. $\gcd(k, n) = 1$ ise $\gcd(n - k, n) = 1$'dir, yani bu sayılar toplamı n olan $\phi(n)/2$ çifte ayrılır. ■

Döngülerin az bilinen bir parçası da `else` kısmıdır. Bir `for` ya da `while` döngüsünün sonuna yazılan `else` bloğu, döngü `break` ile kesilmeden **kendiliğinden bittiğinde** çalışır. Bu, "aradığımı bulamadım"

 CC BY-NC-SA 4.0

durumunu yakalamanın temiz bir yoludur. Asallık testinde aradığımız şey bir bölendir; aramayı nerede bitirebileceğimizi aşağıdaki önerme söyler.

Önerme 2.2 (Bölen Aramak İçin Karekök Yeter). *$n > 1$ asal olmayan bir tam sayı ise n'nin $2 \le d \le \sqrt{n}$ koşulunu sağlayan bir d böleni vardır. Dolayısıyla $2, 3, \ldots, \lfloor\sqrt{n}\rfloor$ sayılarından hiçbiri n'yi bölmüyorsa n asaldır.*



Şekil 2.6: 36'nın bölenleri logaritmik ölçekli bir doğru üzerinde. Her d *böleni,* d · (36/d) = 36 *eşitliğindeki eşine bir yayla bağlıdır. Bütün yayların merkezi √36 = 6 noktasıdır, bu yüzden her çiftin küçük elemanı 6'yı aşmaz. Aynı eşleme her* n *için geçerlidir:* n*'nin 1 ve kendisi dışında bir böleni varsa, 2 ile √*n *arasında da bir böleni vardır.*

$n = d \cdot e$ ise d ile e'nin ikisi birden $\sqrt{n}$'den büyük olamaz, yoksa çarpımları n'yi aşardı; ispatın ayrıntısı için bkz. Sayılar Teorisi.

Örnek 2.9 (Deneme Bölmesiyle Asallık). 1001, 1005 ve 1009 sayılarının asal olup olmadığını deneme bölmesiyle belirleyiniz.

Çözüm.
Önerme 2.2'e göre $d = 2, 3, \ldots, \lfloor\sqrt{n}\rfloor$ bölenlerini denemek yeter. `math.isqrt(n)`, $\lfloor\sqrt{n}\rfloor$ tam sayı karekökünü kesin olarak verir. Bir bölen bulunduğunda `break` iç döngüyü keser ve `else` bloğu atlanır. Hiçbir bölen bulunamazsa iç döngü kendiliğinden biter ve `else` bloğu "asal" yazar. `else`'in `if` ile değil iç `for` ile aynı hizada olduğuna dikkat edin.

```
import math

for n in range(1001, 1010, 4):
    for d in range(2, math.isqrt(n) + 1):
        if n % d == 0:
            print(f"{n} = {d} * {n // d}, asal değil")
            break
    else:
        print(f"{n} asal")
```

Çıktı:

```
1001 = 7 * 143, asal değil
1005 = 3 * 335, asal değil
1009 asal
```

 CC BY-NC-SA 4.0

$\lfloor\sqrt{1009}\rfloor = 31$ olduğundan 1009 için yalnız 30 aday bölen denendi. $1001 = 7 \cdot 11 \cdot 13$'tür; kod ilk bulduğu bölen olan 7'de durdu. ■

2.5 Fonksiyonlar

Öklid algoritmasını her ebob gerektiğinde yeniden yazmak istemeyiz. **Fonksiyon**, bir hesabı bir ad altında paketler: bir kez tanımlanır, istenildiği kadar çağrılır.

Tanım 2.8 (Fonksiyon Tanımı ve Çağrısı).

```
def ad(parametre_1, parametre_2):
    blok
```

biçimindeki `def` deyimi `ad` adlı bir **fonksiyon** tanımlar; parametre sayısı herhangi bir doğal sayı olabilir. `ad(argüman_1, argüman_2)` biçimindeki bir **çağrı**, parametrelere sırasıyla argümanların değerlerini atar ve bloğu çalıştırır. Blokta bir `return ifade` satırına gelindiğinde fonksiyon hemen sona erer ve çağrının değeri `ifade`'nin değeri olur. Blok bir `return`'e rastlamadan biterse çağrının değeri özel `None` değeridir.

Yani $f(x) = x^2 - 2x + 5$ gibi bir matematiksel fonksiyon Python'da `def f(x):` satırı ve altındaki `return x**2 - 2*x + 5` satırıyla yazılır. Ama bir Python fonksiyonu bundan fazlasıdır: içinde döngüler, kararlar ve başka fonksiyon çağrıları olan bir hesap yordamıdır.

```
def f(x):
    return x**2 - 2*x + 5


print(f(0), f(1), f(3))
print(f(1 + 2j))
y = f(2) + f(-2)
print(y)
```

Çıktı:

```
5 4 8
0j
18
```

Aynı fonksiyon karmaşık sayılarla da çalışır. $f(1 + 2i) = 0$ çıkması, Örnek 2.1'de bulduğumuz kökün sağlamasıdır. Bir çağrı, değeri olan bir ifadedir; `f(2) + f(-2)` gibi başka ifadelerin içinde kullanılabilir.

Tanım 2.9 (Belge Dizgisi). Bir fonksiyon bloğunun ilk satırına yazılan dizgiye o fonksiyonun **belge dizgisi** (docstring) denir. Belge dizgisi fonksiyonun ne yaptığını, parametrelerini ve döndürdüğü değeri anlatır; `help(ad)` çağrısı onu ekrana yazar.

Yani belge dizgisi, fonksiyonun kodla birlikte taşınan kullanım kılavuzudur. Gelenek, onu üç tırnak arasında (`"""..."""`) yazmak, ilk satırda tek cümlelik bir özet vermek ve gerekiyorsa bir boş satırdan sonra ayrıntıya girmektir.

Örnek 2.10 (Ebob Fonksiyonu). Öklid algoritmasını belge dizgisi olan bir `gcd(a, b)` fonksiyonuna çeviriniz.

Çözüm.
Örnek 2.5'deki döngüyü fonksiyonun bloğuna taşırız, yazdırma satırını atar ve sonucu `return` ile döndürürüz. Negatif girdilerde de pozitif bir ebob vermek için sonucu `abs` ile döndürüyoruz.

```
def gcd(a, b):
    """a ile b'nin en büyük ortak bölenini döndürür.
```

 CC BY-NC-SA 4.0

```
    Öklid algoritması: b sıfır olana dek (a, b) çiftini
    (b, a % b) ile değiştirir.
    """
    while b != 0:
        a, b = b, a % b
    return abs(a)


help(gcd)
print(gcd(1071, 462), gcd(-12, 18), gcd(7, 0))
```

Çıktı:

```
Help on function gcd in module __main__:

gcd(a, b)
    a ile b'nin en büyük ortak bölenini döndürür.

    Öklid algoritması: b sıfır olana dek (a, b) çiftini
    (b, a % b) ile değiştirir.

21 6 7
```

`gcd(-12, 18)` çağrısında Python'un `%` işlemi kalanı bölenin işaretiyle verir: `-12 % 18` işleminin sonucu 6'dır. Ardından $(18, 6)$ çifti $(6, 0)$'a döner ve sonuç 6 olur. `gcd(7, 0)` ise döngüye hiç girmeden 7 döndürür; bu, $\gcd(a, 0) = |a|$ kuralıdır. Python'un `math` modülünde aynı işi yapan hazır bir `math.gcd` fonksiyonu da vardır. ■

Bir fonksiyon birden çok değeri virgülle ayırarak döndürebilir; çağıran taraf bunları `x1, x2 = ...` biçiminde ayrı değişkenlere açabilir. Python bu değerleri bir *demet* (tuple) içinde paketler. Demetleri de Veri Yapıları ve Comprehension bölümünde göreceğiz.

Örnek 2.11 (Kökleri Döndüren Bir Fonksiyon). Örnek 2.1'deki hesabı, $ax^2 + bx + c = 0$ denkleminin iki kökünü döndüren bir `roots(a, b, c)` fonksiyonuna çeviriniz ve üç durumu da deneyiniz.

Çözüm.
$\Delta \geq 0$ ise $\sqrt{\Delta}$ gerçeldir. $\Delta < 0$ ise $\sqrt{\Delta}$ yerine $i\sqrt{-\Delta}$ alırız ve bunu sanal birim `1j` ile `1j * math.sqrt(-delta)` olarak kurarız. Böylece kök formülü üç durumda da aynı satırla yazılır.

```
import math


def roots(a, b, c):
    """ax^2 + bx + c = 0 denkleminin köklerini döndürür (a != 0)."""
    delta = b**2 - 4*a*c
    if delta >= 0:
        r = math.sqrt(delta)
    else:
        r = 1j * math.sqrt(-delta)
    return (-b + r) / (2*a), (-b - r) / (2*a)


x1, x2 = roots(1, -3, 2)
print(x1, x2)
print(roots(1, -2, 1))
print(roots(1, -2, 5))
```

Çıktı:

 CC BY-NC-SA 4.0

```
2.0 1.0
(1.0, 1.0)
((1+2j), (1-2j))
```

$x^2 - 3x + 2 = (x-1)(x-2)$ denkleminin kökleri 2 ve 1'dir. $x^2 - 2x + 1 = (x-1)^2$ çift katlı 1 kökünü iki kez verir. $x^2 - 2x + 5$ için yine $1 \pm 2i$ bulunur. İlk çağrıda sonuçları iki değişkene açtık, diğerlerinde ise çağrının kendisini yazdırdık; Python iki değeri parantez içinde gösterdi. ■

⚠ print bir değer döndürmez

Fonksiyonun içinde sonucu `print` ile yazdırmak onu döndürmek demek değildir. `return` içermeyen bir fonksiyonun çağrısı `None` değerini alır ve bu değerle hesap yapmaya çalışmak hata verir:

```
def square_print(x):
    print(x**2)          # ekrana yazar ama değer döndürmez


def square(x):
    return x**2


a = square_print(3)
b = square(3)
print(a, b)
```

Çıktı:

```
9
None 9
```

Hesapta kullanılacak sonuçları `return` ile döndürün; `print`'i yalnız ekrana bilgi vermek için kullanın.

Fonksiyonun içindeki değişkenlerle dışarıdakiler arasındaki ilişki de net olmalıdır.

Tanım 2.10 (Yerel ve Global Değişken). Bir fonksiyonun bloğunda değer atanan değişkenler o fonksiyonun **yerel** değişkenleridir: yalnız çağrı süresince var olurlar ve fonksiyonun dışındaki aynı adlı değişkenleri etkilemezler. Parametreler de yereldir. Programın en üst düzeyinde, fonksiyonların dışında tanımlanan değişkenlere **global** değişken denir. Bir fonksiyon bir global değişkeni okuyabilir; ona yeni bir değer atayabilmesi için ise bloğun başında `global ad` bildirimi gerekir.

Yani her çağrı kendi küçük çalışma alanında çalışır; fonksiyonun içindeki `x`, dışarıdaki `x`'in yalnız adaşıdır.

```
x = 10


def g(t):
    x = 2 * t        # bu x, g'nin yerel değişkenidir
    return x + 1


print(g(5))
print(x)             # dışarıdaki x değişmedi
```

Çıktı:

```
11
10
```

 CC BY-NC-SA 4.0

Fonksiyonların yalnız parametreler ve `return` üzerinden haberleşmesi onları güvenilir yapı taşları yapar. `global` bildirimini bu bölümde yalnız bir fonksiyonun kaç kez çağrıldığını ya da kaç işlem yaptığını saymak için kullanacağız.
Sayısal yöntemlerde bir fonksiyonun asıl girdisinin yanında tolerans, en fazla adım sayısı gibi ayarları da olur. Bunları her çağrıda yazmak zorunda kalmamak için varsayılan değerler kullanılır.

Tanım 2.11 (Varsayılan Değer ve Anahtar Sözcüklü Argüman). Bir parametreye tanımda `ad=değer` biçiminde bir **varsayılan değer** verilebilir; çağrıda bu parametreye argüman yazılmazsa varsayılan değer kullanılır. Varsayılan değerli parametreler, varsayılanı olmayanlardan sonra gelir. Çağrıda bir argüman `ad=değer` biçiminde parametresinin adıyla da verilebilir; böyle bir **anahtar sözcüklü argüman**, sıraya bakılmaksızın adıyla eşleşir.

Yani sık değiştirilmeyen ayarlar varsayılan değerlerle tanımlanır, çağıran yalnız değiştirmek istediğini adıyla yazar. Bu, uzun parametre listelerini okunur kılar: `sqrt_newton(10, tol=1e-6)` çağrısı, aynı işi yapan `sqrt_newton(10, 1.0, 1e-6)` çağrısından çok daha açıktır.
Bu araçları sayısal analizin klasik bir algoritmasında birleştirelim. $a > 0$ için $\sqrt{a}$, $f(x) = x^2 - a$ fonksiyonunun pozitif köküdür. Newton yönteminin $x_{k+1} = x_k - f(x_k)/f'(x_k)$ adımı (bkz. Nümerik Analiz) bu fonksiyon için

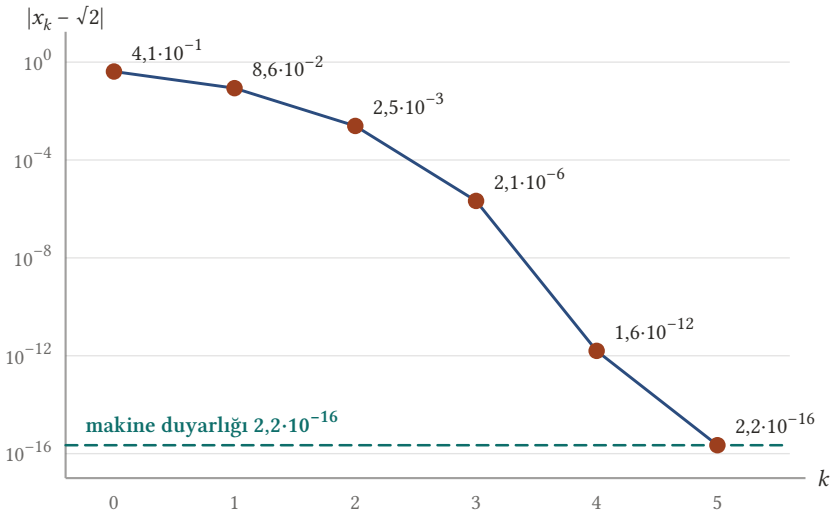
$$x_{k+1} = x_k - \frac{x_k^2 - a}{2x_k} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right)$$

olur. Bu, Babillilerden beri bilinen karekök iterasyonudur (bkz. Analiz 1).

Newton yöntemiyle karekök üç adımda

1. Bir başlangıç değeri seçin: $x_0 > 0$, örneğin $x_0 = 1$.
2. Yeni terimi $x_{k+1} = \frac{1}{2}(x_k + a/x_k)$ ile hesaplayın.
3. $|\, x_{k+1} - x_k \,|$ bir `tol` toleransından küçükse durun ve x_{k+1}'i döndürün; değilse ikinci adıma dönün. Yakınsamama ihtimaline karşı adım sayısını `max_iter` ile sınırlayın.

Örnek 2.12 (Newton Yöntemiyle Karekök). Tarifi varsayılan değerli parametreleri olan bir `sqrt_newton` fonksiyonu olarak yazıp $\sqrt{2}$'yi hesaplayınız; ara adımları da yazdırınız.



Şekil 2.7: $\sqrt{2}$ için Newton iterasyonunun ($x_0 = 1$) hatası logaritmik ölçekte. Hata üssü her adımda kabaca ikiye katlanır: 10^{-1}, 10^{-3}, 10^{-6}, 10^{-12}. Beşinci adımda hata makine duyarlığına iner; daha fazla adım bir şey kazandırmaz.

Çözüm.

 CC BY-NC-SA 4.0

Fonksiyon a'yı zorunlu, `x0`, `tol`, `max_iter` ve `verbose` parametrelerini varsayılan değerli alır. Döngü en fazla `max_iter` adım süren bir `for` döngüsüdür. Durma koşulu sağlanınca `return` hem döngüyü hem de fonksiyonu bitirir. Fonksiyon kök ile adım sayısını birlikte döndürür.

```
def sqrt_newton(a, x0=1.0, tol=1e-12, max_iter=50, verbose=False):
    """a > 0 sayısının karekökünü Newton yöntemiyle hesaplar.

    x_{k+1} = (x_k + a / x_k) / 2 iterasyonu, ardışık iki terimin
    farkı tol'dan küçük olunca durur. (kök, adım sayısı) döndürür.
    """
    x = x0
    for k in range(1, max_iter + 1):
        x_new = (x + a / x) / 2
        if verbose:
            print(f"k = {k}  x = {x_new:.16f}")
        if abs(x_new - x) < tol:
            return x_new, k
        x = x_new
    return x, max_iter


root, steps = sqrt_newton(2, verbose=True)
print("sonuç:", root, "adım:", steps)
print(sqrt_newton(10, tol=1e-6))
print(sqrt_newton(10, 3.0, 1e-6))
```

Çıktı:

```
k = 1  x = 1.5000000000000000
k = 2  x = 1.4166666666666665
k = 3  x = 1.4142156862745097
k = 4  x = 1.4142135623746899
k = 5  x = 1.4142135623730949
k = 6  x = 1.4142135623730949
sonuç: 1.414213562373095 adım: 6
(3.162277660168379, 6)
(3.162277660168379, 4)
```

İlk çağrıda yalnız `verbose`'u adıyla değiştirdik, diğer parametreler varsayılan değerlerini aldı. Doğru ondalık basamak sayısı her adımda kabaca ikiye katlanıyor: x_2'de 2, x_3'te 5, x_4'te 11 basamak doğru. Bu, Newton yönteminin karesel yakınsamasının izidir. Aynı davranış $\cos x - x = 0$ denkleminde de görülür (bkz. Nümerik Analiz); hızın kesin ifadesine Kök Bulma ve Optimizasyon bölümünde döneceğiz. Beşinci adımdan sonra terim değişmez; altıncı adımda fark sıfır olduğundan döngü durur. Sonuç, `math.sqrt(2)` değerinden yalnız son basamakta, yaklaşık $2{,}2 \cdot 10^{-16}$ kadar ayrılır; bu, makine duyarlığı düzeyinde bir farktır.
Son iki çağrının ikisi de $\sqrt{10}$ sayısını aynı 10^{-6} toleransıyla hesaplar, ama farklı başlangıç değerlerinden. `tol=1e-6` adıyla verildiğinde `x0` varsayılan 1 değerinde kalır ve altı adım gerekir. `sqrt_newton(10, 3.0, 1e-6)` çağrısında ise argümanlar sırayla eşleştiği için `x0 = 3.0` ve `tol = 1e-6` olur. $x_0 = 3$ köke yakın bir başlangıç olduğundan dört adım yeter. ■

Python'da fonksiyonlar da birer değerdir: bir değişkene atanabilir, başka bir fonksiyona argüman olarak verilebilir. Bu sayede $\sum$ sembolü gibi bir fonksiyonu girdi olarak alan kavramları doğrudan yazabiliriz. Argüman olarak verilecek kısa fonksiyonlar için ayrıca `def` yazmak da gerekmez.

Tanım 2.12 (lambda İfadesi). `lambda parametreler: ifade` biçimindeki bir **lambda ifadesi** adsız bir fonksiyon üretir; bu fonksiyon çağrıldığında `ifade`'nin değerini döndürür. Gövdesi tek bir ifadeden oluşur; içinde `if` deyimi, döngü ya da `return` bulunmaz.

 CC BY-NC-SA 4.0

Yani `lambda x: x**2`, matematikteki $x \mapsto x^2$ gösteriminin Python yazımıdır. Aşağıdaki `sigma` fonksiyonu terimi bir fonksiyon olarak alır ve $\sum_{k=m}^{n} \text{term}(k)$ toplamını hesaplar.

```
def sigma(term, m, n):
    """term(m) + term(m+1) + ... + term(n) toplamını döndürür."""
    s = 0
    for k in range(m, n + 1):
        s += term(k)
    return s


print(sigma(lambda k: k**2, 1, 100))
print(sigma(lambda k: 1 / 2**k, 0, 30))
print(sigma(lambda k: k**3, 1, 10))
print(sigma(abs, -3, 3))       # hazır bir fonksiyon da verilebilir
```

Çıktı:

```
338350
1.9999999990686774
3025
12
```

İkinci satır $\sum_{k=0}^{30} 2^{-k} = 2 - 2^{-30}$ geometrik toplamıdır, üçüncüsü $1^3 + \ldots + 10^3 = 55^2 = 3025$'tir. Son satırdaki `abs` gibi `def` ile ya da hazır tanımlı her fonksiyon da argüman olabilir. Bir lambda'yı bir ada atamak mümkündür, ama adı olacak bir fonksiyon için `def` yazmak daha okunurdur. Lambda en çok, burada olduğu gibi, tek kullanımlık argüman fonksiyonlarında işe yarar.

Örnek 2.13 (Genel Newton Fonksiyonu). f ile türevi f''yi argüman olarak alan bir `newton(f, df, x0)` fonksiyonu yazıp $\cos x = x$ denkleminin kökünü bulunuz.

Çözüm.
Örnek 2.12'teki döngü, f ve f' için yer tutan iki parametreyle her denkleme genelleşir. $f(x) = \cos x - x$ ve $f'(x) = -\sin x - 1$ fonksiyonlarını lambda ile verir, $x_0 = \pi/4$'ten başlarız. Döngü değişkenini blokta kullanmadığımız için ona gelenekteki gibi `_` adını verdik.

```
import math


def newton(f, df, x0, tol=1e-12, max_iter=50):
    """f(x) = 0 denkleminin x0 yakınındaki kökünü döndürür."""
    x = x0
    for _ in range(max_iter):
        x_new = x - f(x) / df(x)
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    return x


p = newton(lambda x: math.cos(x) - x,
           lambda x: -math.sin(x) - 1, math.pi / 4)
print(p, math.cos(p) - p)
print(newton(lambda x: x**3 - 2*x - 5, lambda x: 3*x**2 - 2, 2.0))
```

Çıktı:

```
0.7390851332151606 1.1102230246251565e-16
2.0945514815423265
```

 CC BY-NC-SA 4.0

Kök $p \approx 0{,}7390851332$'dir (bkz. Nümerik Analiz). $f(p)$'nin yaklaşık $1{,}1 \cdot 10^{-16}$ çıkması, sonucun makine duyarlığı düzeyinde doğru olduğunu gösterir. Aynı fonksiyon, Newton'un kendi örneği olan $x^3 - 2x - 5 = 0$ denkleminin $2{,}0945514815$ kökünü de verir. ■

2.6 Özyineleme

Matematikte pek çok nesne kendisi cinsinden tanımlanır: $0! = 1$ ve $n! = n \cdot (n-1)!$; Fibonacci sayıları için $F_0 = 0$, $F_1 = 1$ ve $F_n = F_{n-1} + F_{n-2}$. Python'da bir fonksiyon kendini çağırabildiğinden bu tanımlar neredeyse kelimesi kelimesine koda dönüşür.

Tanım 2.13 (Özyinelemeli Fonksiyon). Bloğunda kendisini çağıran fonksiyona **özyinelemeli** (recursive) fonksiyon denir. Böyle bir fonksiyonun en az bir **taban durumu**, yani kendini çağırmadan doğrudan sonuç döndürdüğü girdileri vardır. Diğer girdilerde fonksiyon, taban durumuna daha yakın girdilerle kendini çağırır ve bu çağrıların sonuçlarından kendi sonucunu kurar.

Yani özyineleme, tümevarımın hesaptaki karşılığıdır (bkz. Analiz 1): taban durumu başlangıç adımına, özyineli çağrı tümevarım adımına karşılık gelir. Taban durumu unutulursa ya da çağrılar tabana yaklaşmazsa fonksiyon kendini durmadan çağırır.

Örnek 2.14 (Özyinelemeli Faktöriyel). $n!$'i özyinelemeyle hesaplayan bir fonksiyon yazınız ve $20!$ için sonucu döngüyle bulunan değerle karşılaştırınız.

Çözüm.
Tanım doğrudan koda dönüşür: taban durumu $0! = 1$, özyineli adım $n! = n \cdot (n-1)!$. Karşılaştırma için tarifteki çarpım kalıbıyla yazılmış bir döngü sürümü ile Python'un `math.factorial` fonksiyonunu kullanıyoruz.

```
import math


def fact(n):
    """n! değerini özyinelemeyle hesaplar (n >= 0 tam sayı)."""
    if n == 0:
        return 1                # taban durumu
    return n * fact(n - 1)      # özyineli adım


def fact_loop(n):
    p = 1
    for k in range(2, n + 1):
        p *= k
    return p


print(fact(5), fact(20))
print(fact(20) == fact_loop(20) == math.factorial(20))
```

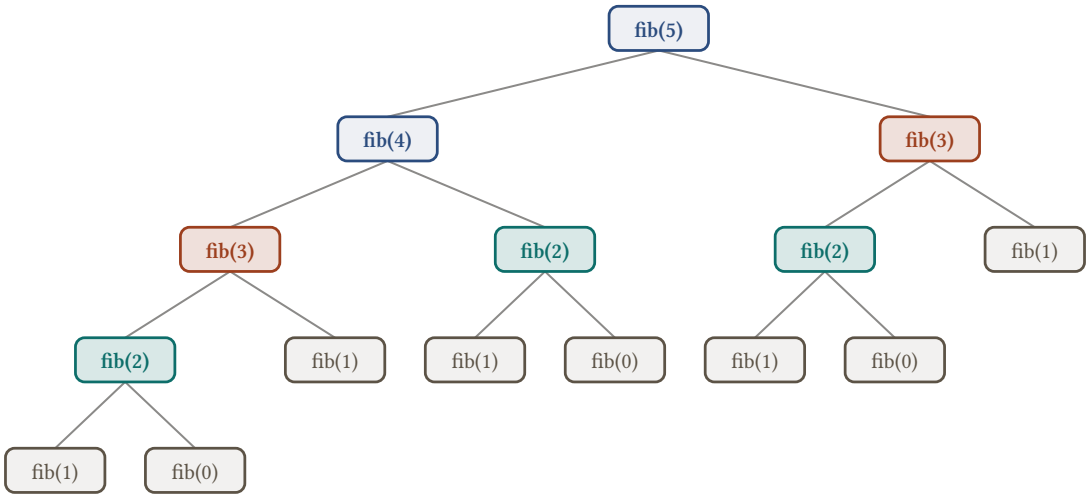
Çıktı:

```
120 2432902008176640000
True
```

`fact(5)` çağrısı `5 * fact(4)` değerini, o da `4 * fact(3)` değerini bekler. Zincir `fact(0)`'a ulaşınca geri dönüşte $1, 1, 2, 6, 24, 120$ değerleri sırayla oluşur. Üç yöntem de $20! = 2432902008176640000$ verir; zincirleme `==` üçünün eşitliğini tek satırda sınar. ■

 CC BY-NC-SA 4.0

Örnek 2.15 (Saf Özyinelemeli Fibonacci). F_n'yi tanımı doğrudan izleyen özyinelemeli bir fonksiyonla hesaplayınız ve $n = 5, 10, \ldots, 30$ için fonksiyonun kaç kez çağrıldığını sayınız.



Şekil 2.8: Saf özyinelemeli `fib(5)` çağrısının ağacı: toplam 15 çağrı. `fib(3)` (turuncu) iki kez, `fib(2)` (yeşil) üç kez baştan hesaplanır. n büyüdükçe aynı alt ağaçlar katlanarak tekrarlanır; `fib(30)` için çağrı sayısı 2692537'dir.

Çözüm.
Çağrıları saymak için global bir `calls` sayacı kullanırız. Fonksiyon bu sayaca değer atadığından blokta `global calls` bildirimi gerekir (Tanım 2.10). Her n için sayacı sıfırlayıp fonksiyonu çağırırız.

```
calls = 0


def fib(n):
    """n. Fibonacci sayısı: saf özyineleme."""
    global calls
    calls += 1
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)


for n in range(5, 31, 5):
    calls = 0
    value = fib(n)
    print(f"fib({n:2}) = {value:6}   çağrı sayısı: {calls:7}")
```

Çıktı:

```
fib( 5) =      5   çağrı sayısı:      15
fib(10) =     55   çağrı sayısı:     177
fib(15) =    610   çağrı sayısı:    1973
fib(20) =   6765   çağrı sayısı:   21891
fib(25) =  75025   çağrı sayısı:  242785
fib(30) = 832040   çağrı sayısı: 2692537
```

Çağrı sayısı $C(n)$ için $C(0) = C(1) = 1$ ve $C(n) = 1 + C(n-1) + C(n-2)$ olduğundan tümevarımla $C(n) = 2F_{n+1} - 1$ bulunur; örneğin

 CC BY-NC-SA 4.0

$$C(30) = 2 \cdot 1346269 - 1 = 2692537.$$

F_n, yaklaşık $1{,}618^n/\sqrt{5}$ hızıyla büyür ($1{,}618 \approx (1+\sqrt{5})/2$ altın orandır). Bu yüzden çağrı sayısı n ile üstel olarak artar: n'yi 5 artırmak işi yaklaşık $1{,}618^5 \approx 11$ katına çıkarır. Şekildeki ağaç sorunun kaynağını gösteriyor: aynı değerler tekrar tekrar baştan hesaplanıyor. ■

Tanım 2.14 (Hafızalama). Bir fonksiyonun hesapladığı her sonucu girdisiyle birlikte bir depoya kaydetmesine ve aynı girdiyle yeniden çağrıldığında hesabı tekrarlamadan kayıtlı sonucu döndürmesine **hafızalama** (memoization) denir.

Yani hafızalama, hesap süresini bellekle takas eder: Fibonacci ağacında her F_k yalnız bir kez hesaplanır, sonraki istekler depodan karşılanır. Depo olarak Python'un **sözlük** (dict) yapısını kullanacağız. Sözlükleri Veri Yapıları ve Comprehension bölümünde ayrıntılı göreceğiz; burada üç işlem yeter. `memo = {0: 0, 1: 1}` iki başlangıç kaydıyla bir sözlük kurar, `n in memo` n için kayıt olup olmadığını sorar, `memo[n] = değer` kaydı yazar ve `memo[n]` onu okur.

Örnek 2.16 (Hafızalı Fibonacci). Fibonacci fonksiyonunu hafızalı yazarak F_{30} ve F_{100} sayılarını hesaplayınız; çağrı sayılarını Örnek 2.15 ile karşılaştırınız.

Çözüm.
Taban durumları sözlüğün başlangıç kayıtlarıdır. Fonksiyon önce depoya bakar; kayıt yoksa değeri iki özyineli çağrıyla hesaplayıp kaydeder. `calls += 1` satırı `calls` adına yeni bir değer atadığı için yine `global calls` bildirimi gerekir. `memo[n] = ...` satırı ise `memo` adına yeni bir değer atamaz, yalnız bu adın gösterdiği sözlüğün içeriğini değiştirir; bu yüzden `memo` için bildirim gerekmez.

```
memo = {0: 0, 1: 1}    # hesaplanan değerlerin deposu (sözlük)
calls = 0


def fib(n):
    """n. Fibonacci sayısı: hafızalı özyineleme."""
    global calls
    calls += 1
    if n in memo:
        return memo[n]           # daha önce hesaplandı: hazır değer
    memo[n] = fib(n - 1) + fib(n - 2)
    return memo[n]


print(fib(30), "çağrı sayısı:", calls)
calls = 0
print(fib(100), "çağrı sayısı:", calls)
```

Çıktı:

```
832040 çağrı sayısı: 59
354224848179261915075 çağrı sayısı: 141
```

F_{30} için çağrı sayısı 2692537'den $59 = 2 \cdot 30 - 1$'e indi, çünkü her F_k bir kez hesaplanıyor. F_{100} yalnız 141 çağrı tuttu, çünkü $F_0, \ldots, F_{30}$ önceki hesaptan depoda duruyordu. Saf sürümle F_{100} için $2F_{101} - 1 \approx 1{,}1 \cdot 10^{21}$ çağrı gerekirdi.
Python'un standart kütüphanesi bu kalıbı hazır sunar. `functools` modülündeki `cache`, fonksiyon tanımının üstüne `@cache` satırı olarak yazıldığında fonksiyonu kendi deposuyla hafızalı hâle getirir. Bir fonksiyonu böyle saran `@` satırlarına **dekoratör** denir.

 CC BY-NC-SA 4.0

```python
from functools import cache


@cache
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)


print(fib(100))
print(fib.cache_info())
```

Çıktı:

```
354224848179261915075
CacheInfo(hits=98, misses=101, maxsize=None, currsize=101)
```

`cache_info()` deponun durumunu bildirir: $F_0, \ldots, F_{100}$ olmak üzere 101 değer birer kez hesaplandı (`misses`), 98 istek depodan karşılandı (`hits`). ■

⚠ Özyineleme derinliği sınırlıdır

Her özyineli çağrı, geri dönülecek yeri saklamak için bellek harcar. Python bu yüzden iç içe çağrıların derinliğini sınırlar; sınır aşılınca program `RecursionError` hatasıyla durur. Sınırı `sys` modülü söyler:

```python
import sys

print(sys.getrecursionlimit())
```

Çıktı:

```
1000
```

Bu yüzden `fact(5000)` gibi binlerce düzey inen bir özyineleme hata verir; hafızalı `fib(5000)` çağrısı da aynı nedenle başarısız olur. Derinliği girdiyle doğrusal büyüyen hesaplarda döngü kullanın. Özyinelemeyi, ebob ya da hızlı üs alma gibi derinliği logaritmik kalan hesaplara ve doğal olarak ağaç biçimli problemlere saklayın.

2.7 Alıştırmalar

Aşağıdaki alıştırmalarda bölümdeki kalıpları kendi başınıza uygulayacaksınız. Her çözümdeki kod kendi başına çalışır.

Alıştırma 2.1 (Artık Yıl Kuralı). Gregoryen takvimde 4'e bölünen yıllar artık yıldır; ancak 100'e bölünen yıllar, 400'e de bölünmedikçe artık yıl değildir. Bu kuralı bir `is_leap(year)` fonksiyonuyla yazıp 1901 ile 2000 arasındaki (ikisi de dahil) artık yılları sayınız.

Çözüm.
Kuralda en özel durum en önce sınanmalıdır: önce 400, sonra 100, en son 4. `return` fonksiyonu hemen bitirdiğinden `elif` yazmaya gerek kalmaz; bir `return`'e ulaşan çağrı alttaki satırları hiç görmez. Son satırdaki `year % 4 == 0` zaten bir `bool` olduğundan doğrudan döndürülür.

```python
def is_leap(year):
    """Gregoryen takvimde artık yıl mı?"""
    if year % 400 == 0:
```

 CC BY-NC-SA 4.0

```
        return True
    if year % 100 == 0:
        return False
    return year % 4 == 0


print(is_leap(1900), is_leap(2000), is_leap(2024), is_leap(2026))
count = 0
for y in range(1901, 2001):
    if is_leap(y):
        count += 1
print("1901-2000 arası artık yıl:", count)
```

Çıktı:

```
False True True False
1901-2000 arası artık yıl: 25
```

1900, 100'e bölünüp 400'e bölünmediğinden artık yıl değildir; 2000 ise 400'e bölündüğünden artık yıldır. 1901 ile 2000 arasında 1904, 1908, …, 2000 olmak üzere 25 artık yıl vardır. ■

Alıştırma 2.2 (Rakamlar Toplamı). Bir doğal sayının onluk tabandaki rakamlarının toplamını `while` döngüsüyle hesaplayan bir `digit_sum(n)` fonksiyonu yazınız ve 2^{100} sayısının rakamları toplamını bulunuz.

Çözüm.
n'nin son rakamı $n \bmod 10$'dur, $\lfloor n/10 \rfloor$ ise son rakamı silinmiş sayıdır. `divmod(n, 10)` ikisini birlikte verir. n sıfır olana dek son rakamı ayırıp toplama ekleriz.

```
def digit_sum(n):
    """n >= 0 tam sayısının onluk tabandaki rakamlarının toplamı."""
    s = 0
    while n > 0:
        n, d = divmod(n, 10)    # son rakamı ayır
        s += d
    return s


print(digit_sum(2026), digit_sum(9**9))
print(2**100)
print(digit_sum(2**100))
```

Çıktı:

```
10 45
1267650600228229401496703205376
115
```

$9^9 = 387420489$ sayısının rakamları toplamı 45'tir. Python'un sınırsız tam sayıları sayesinde 31 basamaklı 2^{100} de kesin hesaplanır ve rakamları toplamı 115 çıkar. ■

Alıştırma 2.3 (Mükemmel Sayılar). Kendisinden küçük pozitif bölenlerinin toplamına eşit olan sayılara mükemmel sayı denir; örneğin $6 = 1 + 2 + 3$. 10000'den küçük mükemmel sayıları bulunuz.

Çözüm.

 CC BY-NC-SA 4.0

Önerme 2.2'deki fikir bölen toplamını da hızlandırır: $d \leq \sqrt{n}$ bir bölense eşi n/d de bir bölendir. Bu yüzden d'yi yalnız $2, \ldots, \lfloor\sqrt{n}\rfloor$ arasında dolaştırıp her bölende hem d'yi hem n/d'yi ekleriz. $d = n/d$ ise, yani n tam kareyse bu bölen bir kez eklenir. 1 her zaman bölen olduğundan toplam 1'den başlar.

```
import math


def divisor_sum(n):
    """n'nin kendisinden küçük pozitif bölenlerinin toplamı (n > 1)."""
    s = 1
    for d in range(2, math.isqrt(n) + 1):
        if n % d == 0:
            s += d
            if d != n // d:
                s += n // d    # eş bölen n/d de sayılır
    return s


for n in range(2, 10001):
    if divisor_sum(n) == n:
        print(n, "mükemmel")
```

Çıktı:

```
6 mükemmel
28 mükemmel
496 mükemmel
8128 mükemmel
```

Dört mükemmel sayı bulunur: 6, 28, 496 ve 8128. Bunlar $p = 2, 3, 5, 7$ için $2^{p-1}(2^p - 1)$ biçimindedir. Öklid, $2^p - 1$ asal olduğunda bu sayının mükemmel olduğunu göstermişti; Euler de her çift mükemmel sayının bu biçimde olduğunu kanıtladı. ■

Alıştırma 2.4 (Yüzü İki Asalın Toplamı Olarak Yazmak). 100 sayısının, $p \leq q$ olmak üzere iki asalın $p + q$ toplamı olarak bütün yazılışlarını bulunuz.

Çözüm.
Önce deneme bölmesini bir `is_prime` fonksiyonuna çevirir. Bölen bulunduğu anda `return False` fonksiyonu bitirir; bu, Örnek 2.9'deki `break` ile `else` ikilisinin fonksiyon içindeki karşılığıdır. Sonra p'yi $2, \ldots, 50$ arasında dolaştırıp hem p'nin hem $100 - p$'nin asal olduğu durumları yazdırırız. $p \leq 50$ koşulu $p \leq q$ demektir.

```
import math


def is_prime(n):
    """n asal mı? (deneme bölmesi)"""
    if n < 2:
        return False
    for d in range(2, math.isqrt(n) + 1):
        if n % d == 0:
            return False
    return True


n = 100
count = 0
```

 CC BY-NC-SA 4.0

```
for p in range(2, n // 2 + 1):
    if is_prime(p) and is_prime(n - p):
        print(f"{n} = {p} + {n - p}")
        count += 1
print("yazılış sayısı:", count)
```

Çıktı:

```
100 = 3 + 97
100 = 11 + 89
100 = 17 + 83
100 = 29 + 71
100 = 41 + 59
100 = 47 + 53
yazılış sayısı: 6
```

Altı yazılış vardır. **Goldbach sanısı**, 2'den büyük her çift sayının en az bir yazılışı olduğunu söyler. Sanı çok büyük sayılara kadar bilgisayarla doğrulanmıştır ama ispatlanmamıştır. ■

Alıştırma 2.5 (Genişletilmiş Öklid Algoritması). a, b doğal sayıları için $\gcd(a,b)$ ile birlikte $ax + by = \gcd(a,b)$ eşitliğini sağlayan x, y tam sayılarını da bulan özyinelemeli bir `ext_gcd(a, b)` fonksiyonu yazınız ve $a = 240$, $b = 46$ için çalıştırınız.

Çözüm.
$b = 0$ ise $\gcd(a,0) = a$ ve $a \cdot 1 + b \cdot 0 = a$'dır; taban durumu budur. $b \neq 0$ ise $q = \lfloor a/b \rfloor$ ve $r = a - qb$ olsun. Önerme 2.1 gereği $\gcd(a,b) = \gcd(b,r) = g$'dir. Özyineli çağrı $bx' + ry' = g$ eşitliğini sağlayan x', y' sayılarını verirse

$$g = bx' + (a - qb)y' = ay' + b(x' - qy')$$

olduğundan $x = y'$ ve $y = x' - qy'$ alınır.

```
def ext_gcd(a, b):
    """(g, x, y) döndürür: g = ebob(a, b) ve a*x + b*y = g."""
    if b == 0:
        return a, 1, 0
    g, x, y = ext_gcd(b, a % b)
    return g, y, x - (a // b) * y


g, x, y = ext_gcd(240, 46)
print(g, x, y)
print(240 * x + 46 * y)
g, x, y = ext_gcd(1071, 462)
print(g, x, y, 1071 * x + 462 * y)
```

Çıktı:

```
2 -9 47
2
21 -3 7 21
```

Sağlamalar:

$$240 \cdot (-9) + 46 \cdot 47 = -2160 + 2162 = 2,$$
$$1071 \cdot (-3) + 462 \cdot 7 = -3213 + 3234 = 21.$$

 CC BY-NC-SA 4.0

Böyle x, y sayılarının varlığı Bézout teoremidir (bkz. Sayılar Teorisi); bu fonksiyon onları açıkça kurar. Özyinelemenin derinliği Öklid algoritmasının adım sayısı kadardır ve Lamé teoremine göre bu sayı en fazla $\log b$ ile orantılı büyür. ■

Alıştırma 2.6 (Hızlı Üs Alma). n çiftse $a^n = (a^{n/2})^2$, tekse $a^n = (a^{\lfloor n/2 \rfloor})^2 \cdot a$ eşitliklerini kullanarak a^n'yi hesaplayan özyinelemeli bir `power(a, n)` fonksiyonu yazınız. $n = 100, 400, 700, 1000$ için 3^n hesabında kaç çarpma yapıldığını sayınız.

Çözüm.
Taban durumu $a^0 = 1$'dir. Her özyineli adım n'yi yarıya indirir; yarım kuvvet bir kez hesaplanıp kendisiyle çarpılır, n tekse bir çarpma daha yapılır. Çarpmaları global bir sayaçla sayarız.

```
mults = 0


def power(a, n):
    """a**n değerini n'yi ikiye bölerek hesaplar (n >= 0 tam sayı)."""
    global mults
    if n == 0:
        return 1
    half = power(a, n // 2)
    mults += 1
    result = half * half
    if n % 2 == 1:
        mults += 1
        result *= a
    return result


print(power(3, 13), 3**13)
for n in range(100, 1001, 300):
    mults = 0
    ok = power(3, n) == 3**n
    print(f"n = {n:4}  çarpma sayısı: {mults:2}  doğru mu: {ok}")
```

Çıktı:

```
1594323 1594323
n =  100  çarpma sayısı: 10  doğru mu: True
n =  400  çarpma sayısı: 12  doğru mu: True
n =  700  çarpma sayısı: 16  doğru mu: True
n = 1000  çarpma sayısı: 16  doğru mu: True
```

Doğrudan $a \cdot a \cdots a$ çarpımı $n - 1$ çarpma ister; $n = 1000$ için bu 999 demektir. Bu yöntem ise en fazla $2(\lfloor \log_2 n \rfloor + 1)$ çarpma yapar ve $n = 1000$ için 16 çarpmayla yetinir. Python'un `**` işleci de tam sayı kuvvetlerini aynı fikre dayanan bir yöntemle hesaplar. ■

Alıştırma 2.7 (Euler Sayısının Serisi). $e = \sum_{k=0}^{\infty} \dfrac{1}{k!}$ serisini, sıradaki terim 10^{-17}'den küçük olana dek toplayınız ve sonucu `math.e` ile karşılaştırınız.

Çözüm.
Her terimi faktöriyeli baştan hesaplayarak bulmak gereksizdir. $\dfrac{1}{k!} = \dfrac{1}{(k-1)!} \cdot \dfrac{1}{k}$ olduğundan terimi her adımda k'ya bölmek yeter. Kaç terim gerektiğini bilmediğimiz için `while` kullanırız.

 CC BY-NC-SA 4.0

```python
import math

total = 0.0
term = 1.0          # 1/0! = 1
k = 0
while term > 1e-17:
    total += term
    k += 1
    term /= k       # 1/k! = (1/(k-1)!) / k
print("toplanan terim sayısı:", k)
print("seri    :", total)
print("math.e  :", math.e)
print("fark    :", total - math.e)
```

Çıktı:

```
toplanan terim sayısı: 19
seri    : 2.7182818284590455
math.e  : 2.718281828459045
fark    : 4.440892098500626e-16
```

$k = 0, \ldots, 18$ olmak üzere 19 terim toplandı; $1/19! \approx 8{,}2 \cdot 10^{-18}$ eşiğin altında kaldığından eklenmedi. Fark yaklaşık $4{,}4 \cdot 10^{-16}$, yani son basamakta birkaç birimdir: terimler toplanırken yapılan yuvarlamalar birikir. e sayısının tanımı için bkz. Analiz 1. ■

Alıştırma 2.8 (Newton Yöntemiyle Küp Kök). $f(x) = x^3 - a$ fonksiyonuna Newton yöntemini uygulayarak $\sqrt[3]{a}$ değerini hesaplayan bir fonksiyon yazınız; başlangıç değeri, tolerans ve en fazla adım sayısı varsayılan değerli parametreler olsun. $\sqrt[3]{10}$'u hesaplayınız.

Çözüm.
$f'(x) = 3x^2$ olduğundan Newton adımı

$$x_{k+1} = x_k - \frac{x_k^3 - a}{3x_k^2} = \frac{1}{3}\left(2x_k + \frac{a}{x_k^2}\right)$$

olur. Fonksiyonun yapısı Örnek 2.12'tekiyle aynıdır.

```python
def cbrt_newton(a, x0=1.0, tol=1e-14, max_iter=100):
    """a sayısının küp kökü: x <- (2x + a/x^2) / 3."""
    x = x0
    for k in range(1, max_iter + 1):
        x_new = (2*x + a / x**2) / 3
        if abs(x_new - x) < tol:
            return x_new, k
        x = x_new
    return x, max_iter


root, steps = cbrt_newton(10)
print(root, steps)
print(root**3)
print(cbrt_newton(10, x0=2.0))
print(cbrt_newton(-27))
```

Çıktı:

 CC BY-NC-SA 4.0

```
2.154434690031884 8
10.000000000000002
(2.154434690031884, 5)
(-3.0, 9)
```

$\sqrt[3]{10} \approx 2{,}154434690031884$ sekiz adımda bulundu; küpü 10'a son basamaktaki yuvarlama farkıyla eşittir. Başlangıç değeri `x0=2.0` ile köke yaklaştırılınca beş adım yetti. Fonksiyon negatif a için de çalışır: $\sqrt[3]{-27} = -3$. ■

Alıştırma 2.9 (En Uzun Collatz Yolu). 10000'den küçük başlangıç değerleri arasında Collatz dizisi 1'e en çok adımda ulaşanı bulunuz.

Çözüm.
Örnek 2.6'daki döngüyü, adım sayısını döndüren bir fonksiyona çevirir; güncellemeyi koşullu ifadeyle tek satırda yazıyoruz. Sonra $n = 1, \ldots, 9999$ için en büyük adım sayısını ve onu veren n'yi saklarız.

```
def collatz_steps(n):
    """n'den 1'e ulaşana dek atılan Collatz adımlarının sayısı."""
    steps = 0
    while n != 1:
        n = n // 2 if n % 2 == 0 else 3*n + 1
        steps += 1
    return steps


best_n, best_steps = 1, 0
for n in range(1, 10000):
    s = collatz_steps(n)
    if s > best_steps:
        best_n, best_steps = n, s
print("en uzun yol:", best_n, "başlangıcından", best_steps, "adım")
print("27 için:", collatz_steps(27))
```

Çıktı:

```
en uzun yol: 6171 başlangıcından 261 adım
27 için: 111
```

6171 başlangıcı 261 adımla en uzun yoldur. Fonksiyonun 27 için örnekteki 111 adımı yeniden vermesi, iki yazımın tutarlı olduğunu gösteren bir sağlamadır. ■

Alıştırma 2.10 (Hanoi Kuleleri). A, B, C adlı üç çubuktan A'da, en büyüğü en altta olmak üzere n disk dizilidir. Her hamlede bir çubuğun en üstündeki disk başka bir çubuğa taşınır ve hiçbir disk kendisinden küçük bir diskin üstüne konmaz. $n = 3$ için diskleri A'dan C'ye taşıyan hamleleri özyinelemeyle yazdırınız; hamle sayısının $2^n - 1$ olduğunu birkaç n için doğrulayınız.

Çözüm.
n diski A'dan C'ye taşımak için önce üstteki $n-1$ diski C'yi yardımcı kullanarak B'ye taşırız. Sonra en büyük diski A'dan C'ye götürür, en son $n-1$ diski A'yı yardımcı kullanarak B'den C'ye taşırız. Taban durumu $n = 0$'dır: yapılacak hamle yoktur. Hamle sayısı H_n için $H_0 = 0$ ve $H_n = 2H_{n-1} + 1$ olduğundan tümevarımla $H_n = 2^n - 1$ bulunur.

```
moves = 0
```

 CC BY-NC-SA 4.0

```python
def hanoi(n, source, target, spare, show=True):
    """n diski source'tan target'a, spare'i yardımcı kullanarak taşır."""
    global moves
    if n == 0:
        return
    hanoi(n - 1, source, spare, target, show)
    moves += 1
    if show:
        print(f"disk {n}: {source} -> {target}")
    hanoi(n - 1, spare, target, source, show)


hanoi(3, "A", "C", "B")
print("hamle sayısı:", moves)
for n in range(1, 11, 3):
    moves = 0
    hanoi(n, "A", "C", "B", show=False)
    print(f"n = {n:2}  hamle: {moves:4}  2^n - 1 = {2**n - 1}")
```

Çıktı:

```
disk 1: A -> C
disk 2: A -> B
disk 1: C -> B
disk 3: A -> C
disk 1: B -> A
disk 2: B -> C
disk 1: A -> C
hamle sayısı: 7
n =  1  hamle:    1  2^n - 1 = 1
n =  4  hamle:   15  2^n - 1 = 15
n =  7  hamle:  127  2^n - 1 = 127
n = 10  hamle: 1023  2^n - 1 = 1023
```

Fonksiyonun `show` parametresinin varsayılan değeri `True`'dur. Büyük n'lerde hamleleri yazdırmamak için `show=False` veriyoruz; sayaç yine her hamlede artar. ■

Alıştırma 2.11 (Pascal Kuralıyla Binom Katsayısı). $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ ve $\binom{n}{0} = \binom{n}{n} = 1$ kurallarıyla binom katsayısını hafızalı özyinelemeyle hesaplayıp $\binom{30}{15}$ değerini bulunuz.

Çözüm.
Kural doğrudan özyinelemeli bir fonksiyon verir. Hafızasız sürümde her taban durumu sonuca 1 katkı verir; bu yüzden $\binom{30}{15}$ için en az $\binom{30}{15}$ kez taban durumuna inilirdi. `@cache` her (n, k) çiftini bir kez hesaplatır.

```python
import math
from functools import cache


@cache
def binom(n, k):
    """C(n, k): Pascal kuralıyla özyineli hesap."""
    if k == 0 or k == n:
        return 1
    return binom(n - 1, k - 1) + binom(n - 1, k)
```

 CC BY-NC-SA 4.0

```
print(binom(30, 15), math.comb(30, 15))
print(binom.cache_info())
```

Çıktı:

```
155117520 155117520
CacheInfo(hits=196, misses=255, maxsize=None, currsize=255)
```

Sonuç `math.comb` ile aynıdır. Depoda 255 farklı (n, k) çifti var; hafızasız sürümdeki çağrı sayısı ise $2\binom{30}{15} - 1 = 310235039$, yani yaklaşık $3 \cdot 10^8$ olurdu. ■

Bu bölümde programın akışını yöneten üç yapıyı (karar, sayısı belli tekrar ve koşula bağlı tekrar) ve hesapları adlandırıp yeniden kullanmamızı sağlayan fonksiyonları gördük. Şimdiye dek hep tek tek sayılarla çalıştık; Collatz dizisinin terimlerini ya da bulduğumuz asalları bir yerde saklamadık. Bir sonraki bölüm Veri Yapıları ve Comprehension, sayı topluluklarını tutan listeleri, demetleri, sözlükleri ve kümeleri ele alıyor.

 CC BY-NC-SA 4.0

3. Veri Yapıları ve Comprehension

Matematikte çoğu zaman tek tek sayılarla değil, sayı topluluklarıyla çalışırız: bir dizinin ilk on terimi, düzlemde bir nokta belirten (x, y) ikilisi, bir sayının bölenler kümesi, her asal çarpana bir üs karşılık getiren çarpanlara ayırma. Python ile İlk Adımlar ve Kontrol Yapıları ve Fonksiyonlar bölümlerinde tek tek değerlerle, döngülerle ve fonksiyonlarla çalıştık. Bu bölümde bu toplulukları tutan dört yerleşik yapıyı tanıyacağız: liste, demet, sözlük ve küme.
Bu yapıların her birinin matematikte tanıdık bir karşılığı vardır. Liste sonlu bir diziye, demet sıralı bir n'liye, küme sonlu bir kümeye, sözlük de tanım kümesi sonlu bir fonksiyona benzer. Python'un **comprehension** yazımı ise küme kurucu gösterimin, yani $\{k^2 : k \in A, k \text{ tek}\}$ gibi ifadelerin neredeyse birebir karşılığıdır.
Bölümün sonunda bu araçlarla Pascal üçgenini ve Eratosthenes kalburunu kuracak, `itertools` modülüyle permütasyonları ve kombinasyonları tek tek üretip sayacağız.

3.1 Listeler

İlk yapımız, sonlu dizilerin Python'daki karşılığı olan listedir.

Tanım 3.1 (Liste). **Liste** (list), köşeli parantez içinde virgülle ayrılarak yazılan nesnelerin sıralı topluluğudur: `[2, 3, 5, 7]`. Bir `a` listesinin eleman sayısı `len(a)` ile bulunur. n elemanlı bir `a` listesinin elemanlarına `a[0]`, `a[1]`, ..., `a[n - 1]` biçiminde **indis** (index) ile ulaşılır.

Yani liste, $(a_0, a_1, \ldots, a_{n-1})$ sonlu dizisinin Python'daki karşılığıdır. İndisler 0'dan başlar; bu yüzden son elemanın indisi n değil $n-1$'dir. Bir liste aynı elemanı birden çok kez içerebilir, elemanlarının sırası önemlidir ve farklı türden nesneleri bir arada tutabilir.
Python negatif indislere de izin verir: `a[-1]` son elemanı, `a[-2]` sondan ikinci elemanı gösterir. Genel olarak $-n \le i < 0$ için `a[i]` ile `a[i + n]` aynı elemandır. Listenin dışına düşen bir indis, örneğin n elemanlı bir listede `a[n]`, `IndexError` hatası verir.

Örnek 3.1 (Bir Listenin Elemanlarına Erişim). İlk altı asalı bir listede toplayıp elemanlarına pozitif ve negatif indislerle ulaşalım. `in` işleci bir nesnenin listede bulunup bulunmadığını sorar; `sum`, `min` ve `max` ise listenin bütün elemanları üzerinde çalışır.

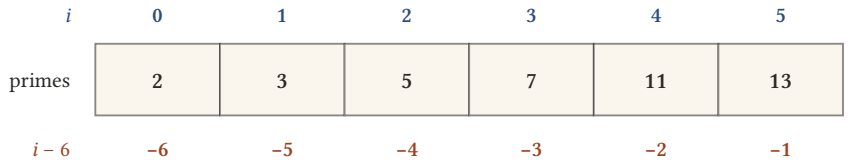
```
primes = [2, 3, 5, 7, 11, 13]
print(len(primes))      # eleman sayısı
print(primes[0])        # ilk eleman: indis 0
print(primes[5])        # son eleman: indis len - 1
print(primes[-1])       # sondan birinci
print(primes[-6])       # sondan altıncı, yani ilk eleman
print(sum(primes), min(primes), max(primes))
print(7 in primes, 9 in primes)
mixed = [1, 2.5, "üç", [4, 5]]   # farklı türler bir arada
print(len(mixed), mixed[3][0])
```

Çıktı:

```
6
2
13
13
2
41 2 13
```

```
True False
4 4
```

`mixed` listesinin son elemanı da bir listedir; `mixed[3][0]` önce bu iç listeyi, sonra onun ilk elemanını alır. Aşağıdaki şekil, `primes` listesindeki her elemanın iki ayrı indisle çağrılabildiğini gösteriyor.



Şekil 3.1: `primes` listesinin altı hücresi. Üstteki sayılar baştan sayılan indislerdir (0, 1, ..., 5), alttakiler sondan sayılan negatif indislerdir (−6, ..., −1). Aynı sütundaki iki indis aynı elemanı gösterir: `primes[i]` ile `primes[i - 6]` aynı sayıdır.

Listeler kurulduktan sonra değiştirilebilir: eleman eklenebilir, silinebilir ya da bir elemanın yerine başkası konabilir. Bu işlerin çoğu listenin **metotlarıyla** (method) yapılır. Metot, `a.append(x)` yazımındaki gibi nesnenin adından sonra nokta konularak çağrılan bir fonksiyondur. En sık kullanılanlar şunlardır:

Yazım	Etkisi
`a.append(x)`	`x`'i listenin sonuna ekler
`a.extend(b)`	`b`'nin bütün elemanlarını sona ekler
`a.insert(i, x)`	`x`'i `i` indisine yerleştirir, sonrakileri bir kaydırır
`a.pop()`	son elemanı çıkarıp döndürür; `a.pop(i)` bunu `i` indisinde yapar
`a.remove(x)`	`x`'e eşit ilk elemanı siler
`a.index(x)`	`x`'in ilk geçtiği indisi döndürür
`a.count(x)`	`x`'in kaç kez geçtiğini sayar
`a[i] = x`	`i` indisindeki elemanın yerine `x`'i koyar

Ayrıca `a + b` iki listeyi uç uca ekleyen yeni bir liste, `a * k` de `a`'nın `k` kez tekrarından oluşan yeni bir liste kurar.

```
a = [3, 1, 4]
a.append(1)         # sona bir eleman ekle
a.extend([5, 9])    # bir listenin bütün elemanlarını ekle
a.insert(0, 2)      # 0 indisine 2'yi yerleştir
print(a)
last = a.pop()      # son elemanı çıkar ve döndür
print(last, a)
a.remove(1)         # ilk 1'i sil
print(a)
print(a.index(4), a.count(1))
print([0] * 4, [1, 2] + [3])
```

Çıktı:

```
[2, 3, 1, 4, 1, 5, 9]
9 [2, 3, 1, 4, 1, 5]
[2, 3, 4, 1, 5]
2 1
[0, 0, 0, 0] [1, 2, 3]
```

 CC BY-NC-SA 4.0

`append`, `extend`, `insert` ve `remove` listeyi yerinde değiştirir ve geriye bir değer döndürmez; daha doğrusu `None` döndürür. `pop` ise hem listeyi değiştirir hem de çıkardığı elemanı döndürür.

Örnek 3.2 (Fibonacci Sayılarını Bir Listede Biriktirme). $F_0 = 0, F_1 = 1$ ve $n \geq 2$ için $F_n = F_{n-1} + F_{n-2}$ kuralıyla tanımlanan Fibonacci dizisinin ilk 15 terimini bir listede toplayalım. Listenin son iki elemanı her zaman `fib[-1]` ve `fib[-2]` olduğundan yeni terim bunların toplamıdır.

```
fib = [0, 1]
while len(fib) < 15:
    fib.append(fib[-1] + fib[-2])   # son iki terimin toplamı
print(fib)
print(fib[-1] / fib[-2])            # ardışık terimlerin oranı
print((1 + 5 ** 0.5) / 2)           # altın oran
```

Çıktı:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
1.6180257510729614
1.618033988749895
```

Son iki terimin oranı $377/233 \approx 1{,}6180258$ olup altın oran $\varphi = (1+\sqrt{5})/2 \approx 1{,}6180340$ değerinden yalnızca $8{,}2 \cdot 10^{-6}$ kadar farklıdır.

3.2 Dilimleme

Bir listenin tek bir elemanı yerine ardışık bir parçasını, ya da düzenli aralıklarla seçilmiş elemanlarını almak için dilimleme kullanılır.

Tanım 3.2 (Dilim). `a` bir liste ve `i`, `j`, `k` tam sayılar olsun ($k \neq 0$). `a[i:j:k]` ifadesine `a`'nın bir **dilimi** (slice) denir. $k > 0$ ise dilim, `a`'nın `i`, `i + k`, `i + 2k`, … indisli elemanlarından `j`'den küçük olanları bu sırayla içeren **yeni** bir listedir. `i` yazılmazsa 0, `j` yazılmazsa `len(a)`, `k` yazılmazsa 1 alınır. $k < 0$ ise dilim `i`'den başlayıp indisleri $|\,k\,|$'şar azaltarak ilerler ve `j`'den büyük indisleri alır; bu durumda `i` yazılmazsa son eleman, `j` yazılmazsa listenin başı esas alınır.

Yani `a[i:j]`, indisleri $[i, j)$ yarı açık aralığında kalan elemanları verir ve $0 \leq i \leq j \leq n$ ise tam $j - i$ eleman içerir. Kural `range` ile aynıdır: $0 \leq i \leq j \leq n$ ve $k > 0$ iken `a[i:j:k]`, tam olarak `range(i, j, k)` indislerindeki elemanlardan oluşur. Dilimdeki negatif sınırlar negatif indisler gibi sondan sayılır. İndisten farklı olarak dilim, listenin dışına taşan sınırlarda hata vermez: 8 elemanlı bir listede `a[5:100]` yalnızca var olan son üç elemanı alır.
Dilimleri, elemanların arasındaki **kesim noktalarıyla** düşünmek işi kolaylaştırır: 0 numaralı kesim ilk elemanın soluna, n numaralı kesim son elemanın sağına düşer ve `a[i:j]`, i ile j numaralı kesimlerin arasında kalan elemanlardır.

Örnek 3.3 (Bir Listeden Dilimler). 10'dan 17'ye kadar olan sayıların listesinden çeşitli dilimler alalım. `list(range(10, 18))` ifadesi, `range` nesnesinin ürettiği sayılardan bir liste kurar.

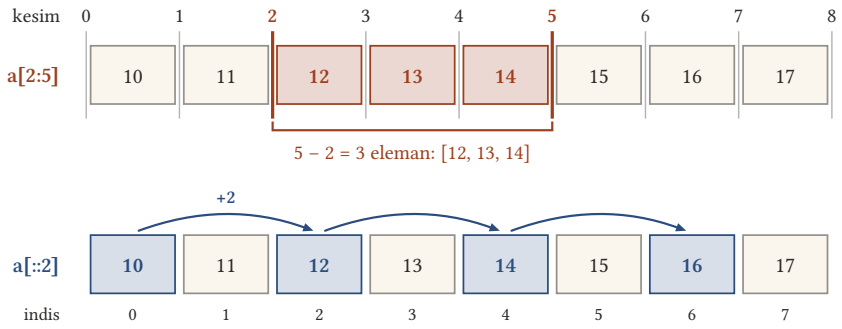
```
a = list(range(10, 18))
print(a)
print(a[2:5])     # indis 2, 3, 4
print(a[:3])      # baştan üç eleman
print(a[5:])      # indis 5'ten sona kadar
print(a[::2])     # çift indisli elemanlar
print(a[1::2])    # tek indisli elemanlar
print(a[::-1])    # ters sıra
print(a[-3:])     # son üç eleman
```

Çıktı:

 CC BY-NC-SA 4.0

```
[10, 11, 12, 13, 14, 15, 16, 17]
[12, 13, 14]
[10, 11, 12]
[15, 16, 17]
[10, 12, 14, 16]
[11, 13, 15, 17]
[17, 16, 15, 14, 13, 12, 11, 10]
[15, 16, 17]
```

`a[2:5]` diliminde 5 numaralı indis yer almaz. `a[::-1]` ise adım -1 olduğundan listeyi sondan başa doğru okur. İlk ve dördüncü dilim aşağıda gösterilmiştir.



Şekil 3.2: Dilimleme, elemanların arasındaki kesim noktalarıyla düşünülür. Üstte `a[2:5]`, 2 ve 5 numaralı kesimlerin arasında kalan üç elemanı alır; 5 numaralı indis dahil değildir. Altta `a[::2]`, baştan başlayıp ikişer atlayarak çift indisli elemanları seçer: [10, 12, 14, 16].

Dilim yalnızca okumak için değil, yazmak için de kullanılabilir. `a[i:j] = b` ataması `a`'nın o parçasını `b`'nin elemanlarıyla yerinde değiştirir. Adımsız dilimde `b`'nin uzunluğu dilimden farklı olabilir; liste buna göre uzar ya da kısalır. Adımlı bir dilime yapılan atamada ise sağ taraf dilimle aynı sayıda eleman içermelidir. `del a[i:j]` o parçayı siler.

```
a = list(range(10))
a[2:5] = [0, 0, 0]    # dilime atama listeyi yerinde değiştirir
print(a)
a[::3] = [-1] * 4     # adımlı dilimde uzunluklar tutmalı
print(a)
del a[-2:]            # son iki elemanı sil
print(a)
```

Çıktı:

```
[0, 1, 0, 0, 0, 5, 6, 7, 8, 9]
[-1, 1, 0, -1, 0, 5, -1, 7, 8, -1]
[-1, 1, 0, -1, 0, 5, -1, 7]
```

`a[::3]` dilimi $0, 3, 6, 9$ indislerini kapsar; bu yüzden sağ tarafta tam dört eleman olmalıdır. Bölümün sonunda Eratosthenes kalburunu bu adımlı dilim atamasıyla kuracağız.

Karakter dizileri (string) de birer dizidir; indeksleme ve dilimleme onlarda da aynen çalışır. Bir sayının rakamlarıyla uğraşmanın en kısa yolu, onu `str` ile karakter dizisine çevirmek, gerekirse `int` ile geri çevirmektir.

Örnek 3.4 (Ters Çevirme ve Palindromlar). Soldan ve sağdan aynı okunan dizilere **palindrom** denir; bir `s` karakter dizisinin palindrom olup olmadığını `s == s[::-1]` karşılaştırması söyler. 2^{20} sayısının rakamlarını da aynı yolla inceleyelim.

```
word = "kayak"
print(word[0], word[-1], word[1:4])
```

 CC BY-NC-SA 4.0

```
print(word[::-1], word == word[::-1])   # palindrom mu?
n = 2**20
digits = str(n)       # sayının rakamları, bir karakter dizisi
print(digits, len(digits))
print(digits[:3], digits[-3:], digits[::-1])
diff = int(digits[::-1]) - n      # ters okunuş ile fark
print(diff, diff % 9)
```

Çıktı:

```
k k aya
kayak True
1048576 7
104 576 6758401
5709825 0
```

Son satırdaki 5709825 farkı 9'a bölünür. Bu bir rastlantı değildir: bir sayı, rakamları toplamıyla 9'a bölümden aynı kalanı verir (bkz. Sayılar Teorisi). Bir sayı ile tersten okunuşu aynı rakamlardan oluştuğu için rakam toplamları eşittir; dolayısıyla farkları 9'un katıdır.

Karakter dizileri listelerden farklı olarak değiştirilemez: `word[0] = "K"` ataması `TypeError` hatası verir. Bir sonraki kısım bu farkı ele alıyor.

3.3 Değiştirilebilirlik ve Takma Adlar

Listelerle çalışırken yapılan hataların önemli bir kısmı, Python'da bir değişken adının neyi temsil ettiğini yanlış anlamaktan doğar.
Python'da `a = [1, 2, 3]` ataması bellekte bir liste nesnesi kurar ve `a` adını bu nesneye bağlar. `b = a` ataması ise yeni bir liste kurmaz; aynı nesneye ikinci bir ad bağlar. Bunun sonuçlarını anlamak için iki kavrama ihtiyacımız var.

Tanım 3.3 (Değiştirilebilir ve Değiştirilemez Nesne). Kurulduktan sonra içeriği değiştirilebilen nesnelere **değiştirilebilir** (mutable), değiştirilemeyenlere **değiştirilemez** (immutable) nesneler denir. Listeler, sözlükler ve kümeler değiştirilebilir; `int`, `float`, `complex`, `bool` türünden sayılar, karakter dizileri ve demetler değiştirilemez nesnelerdir.

Yani bir listenin elemanını değiştirdiğimizde aynı liste nesnesi yerinde kalır, yalnızca içeriği değişir. Buna karşılık bir sayıyı "değiştiren" her işlem aslında yeni bir sayı nesnesi kurar.

Tanım 3.4 (Takma Ad). Aynı nesneye bağlı iki farklı ada birbirinin **takma adı** (alias) denir. `x is y` ifadesi `x` ile `y`'nin aynı nesneye bağlı olup olmadığını, `x == y` ise içeriklerinin eşit olup olmadığını sorar.

Yani değiştirilebilir bir nesnenin iki takma adı varsa, adlardan biri üzerinden yapılan her değişiklik ötekinden de görünür. İki ayrı nesnenin içeriği eşit olabilir; bu durumda `==` doğru, `is` yanlış sonuç verir.

Örnek 3.5 (Takma Ad ile Kopya Arasındaki Fark). Bir listeyi önce yalnızca yeni bir ada bağlayalım, sonra kopyalayalım ve ardından özgün listeye bir eleman ekleyelim.

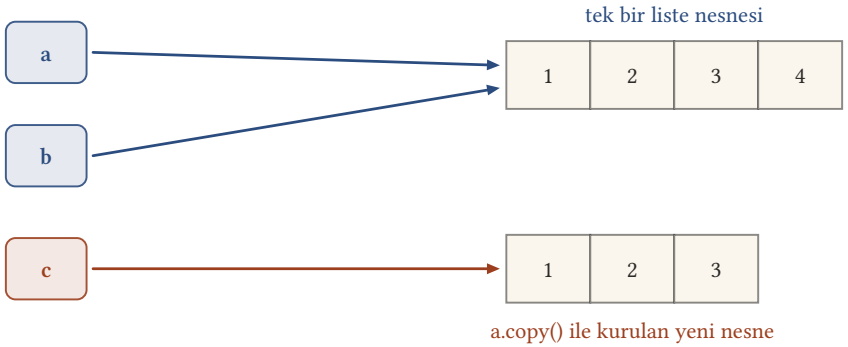
```
a = [1, 2, 3]
b = a              # b, aynı listenin ikinci adı
c = a.copy()       # c, içeriği aynı olan yeni bir liste
a.append(4)
print(a, b, c)
print(b is a, c is a)    # aynı nesne mi?
print(c == [1, 2, 3])    # içerik eşit mi?
```

Çıktı:

 CC BY-NC-SA 4.0

```
[1, 2, 3, 4] [1, 2, 3, 4] [1, 2, 3]
True False
True
```

`a.append(4)` değişikliği `b` üzerinden de görünüyor, çünkü `b` aynı nesnenin ikinci adıdır. `c` ise kopyalama anındaki içerikle kurulmuş ayrı bir nesne olduğundan değişmedi.



Şekil 3.3: Kodun sonundaki durum. `a` ve `b` adları aynı liste nesnesine bağlıdır; bu yüzden `a.append(4)` değişikliği `b` üzerinden de görünür ve `b is a` doğrudur. `c` ise kopyalama anındaki içerikle kurulmuş ayrı bir nesnedir.

Değiştirilemez nesnelerde bu sorun doğmaz, çünkü onları yerinde değiştiren bir işlem yoktur: `x += 1` ataması yeni bir `int` nesnesi kurup `x` adını ona bağlar. Listelerde ise iki yazım arasında önemli bir fark vardır. `u += [3]` listeyi yerinde genişletir; `u = u + [4]` ise yeni bir liste kurup `u` adını ona bağlar.

```
x = 5
y = x
x += 1            # yeni bir int nesnesi oluşur
print(x, y)
u = [1, 2]
v = u
u += [3]          # liste yerinde genişletilir
print(u, v)
u = u + [4]       # + yeni bir liste kurar
print(u, v)
```

Çıktı:

```
6 5
[1, 2, 3] [1, 2, 3]
[1, 2, 3, 4] [1, 2, 3]
```

⚠ Takma ad tuzağı

`b = a` bir kopya değildir. Bir listeyi değiştirmeden önce özgün hâlini saklamak istiyorsanız `a.copy()`, `a[:]` ya da `list(a)` ile bir kopya alın. Bir fonksiyona liste gönderdiğinizde fonksiyon aynı nesne üzerinde çalışır: fonksiyonun içindeki bir `append` dışarıdaki listeyi de değiştirir. Yerinde çalışan metotlar `None` döndürdüğü için `a = a.append(1)` yazmak da listeyi kaybettirir; `a` adı artık `None`'a bağlıdır.

Matrisleri satırlarının listesi olarak tutmak doğaldır: $\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$ matrisi `[[1, 2], [3, 4]]` biçiminde yazılır ve `A[i][j]`, i numaralı satırın j numaralı elemanını verir (ikisi de 0'dan sayılır). İç içe listelerde takma ad sorunu daha sinsi bir biçimde ortaya çıkar.

Örnek 3.6 (Satırları Ortak Bir Sıfır Matrisi). 3×3 sıfır matrisini iki yolla kuralım ve her birinin sol üst köşesine 1 yazalım.

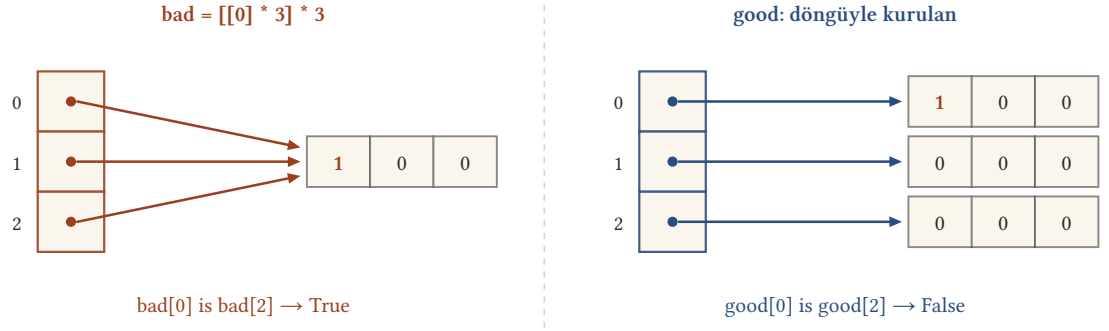
 CC BY-NC-SA 4.0

```
bad = [[0] * 3] * 3      # aynı satıra üç başvuru
good = []
for _ in range(3):
    good.append([0] * 3)  # her turda yeni bir satır
bad[0][0] = 1
good[0][0] = 1
print(bad)
print(good)
print(bad[0] is bad[2], good[0] is good[2])
```

Çıktı:

```
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
[[1, 0, 0], [0, 0, 0], [0, 0, 0]]
True False
```

`[0] * 3` tek bir satır kurar. Dıştaki `* 3` bu satırı kopyalamaz, ona üç kez başvurur; bu yüzden `bad` matrisinin üç satırı aynı nesnedir ve tek bir atama üç satırda birden görünür. Döngü ise her turda yeni bir `[0] * 3` listesi kurduğu için `good` matrisinin satırları birbirinden bağımsızdır.



Şekil 3.4: Solda `[[0] * 3] * 3` ifadesinin kurduğu yapı: dış listenin üç gözü de aynı satır nesnesini gösterir, bu yüzden `bad[0][0] = 1` ataması üç satırda birden görünür. Sağda döngü her turda yeni bir `[0] * 3` listesi kurar; atama yalnız ilk satırı değiştirir.

Aynı sorun kopyalamada da vardır. `A.copy()` yalnızca dış listeyi kopyalar: yeni dış liste, eski satır nesnelerine başvurur. Buna **sığ kopya** (shallow copy) denir. İç listeleri de kopyalayan **derin kopya** (deep copy) için `copy` modülünün `deepcopy` fonksiyonu kullanılır.

```
import copy

A = [[1, 2], [3, 4]]
B = A.copy()            # sığ kopya: satırlar ortak
C = copy.deepcopy(A)    # derin kopya: satırlar da kopyalanır
A[0][0] = 99
print(A)
print(B)
print(C)
```

Çıktı:

```
[[99, 2], [3, 4]]
[[99, 2], [3, 4]]
[[1, 2], [3, 4]]
```

`A[0][0] = 99` ataması, sığ kopya `B` ile ortak olan ilk satırı değiştirdi; derin kopya `C` ise etkilenmedi.

 CC BY-NC-SA 4.0

3.4 Demetler

Listelerin değiştirilemez kardeşi demettir. Matematikteki sıralı ikililer ve n'liler en doğal biçimde demetlerle tutulur.

Tanım 3.5 (Demet). **Demet** (tuple), virgülle ayrılmış nesnelerin sıralı ve değiştirilemez topluluğudur. Genellikle parantez içinde yazılır: `(3, 4)`. Tek elemanlı demet sondaki virgülle `(5,)`, boş demet `()` biçiminde yazılır.

Yani demet, matematikteki $(x_1, \ldots, x_n)$ sıralı n'lisinin karşılığıdır. İndeksleme, dilimleme, `len`, `in` ve `+` demetlerde de listelerdeki gibi çalışır; ancak kurulduktan sonra bir demetin elemanı değiştirilemez, demete eleman eklenemez ve demetten eleman çıkarılamaz. Demeti asıl kuran parantez değil virgüldür: `p = 3, 4` ataması da bir demet kurar.
Bir demetin elemanlarını tek satırda ayrı adlara bağlamaya **açma** (unpacking) denir: `x, y = p`. Bu yazımda sağ taraf önce tamamen hesaplanır, sonra soldaki adlara dağıtılır. Bu yüzden `a, b = b, a + b` ataması iki eski değeri de kullanır ve yardımcı bir değişkene gerek bırakmaz. Soldaki yıldızlı bir ad (`*rest`) geriye kalan elemanları bir listede toplar.

```
p = (3, 4)              # düzlemde bir nokta
x, y = p                # demeti açma
print(x, y, (x**2 + y**2) ** 0.5)
print(divmod(17, 5))    # divmod bir demet döndürür
q, r = divmod(17, 5)
print(q, r)
a, b = 0, 1
for _ in range(10):
    a, b = b, a + b     # önce sağ taraf hesaplanır
print(a, b)
first, *rest = [2, 3, 5, 7]
print(first, rest)
print(len(()), len((5,)), (1, 2) + (3,))
```

Çıktı:

```
3 4 5.0
(3, 2)
3 2
55 89
2 [3, 5, 7]
0 1 (1, 2, 3)
```

Döngüdeki `a, b = b, a + b` ataması Fibonacci dizisini ilerletir: on adımdan sonra $(a, b) = (F_{10}, F_{11}) = (55, 89)$ olur. Son satır boş demetin ve tek elemanlı demetin uzunluklarını veriyor.

Örnek 3.7 (Genişletilmiş Öklid Algoritmasının Döngülü Biçimi). Kontrol Yapıları ve Fonksiyonlar bölümünde, $\gcd(a, b)$ ile birlikte $ax + by = \gcd(a, b)$ eşitliğini sağlayan x, y tam sayılarını da bulan `ext_gcd` fonksiyonunu özyinelemeyle yazmıştık. Oradaki `return g, y, x - (a // b) * y` satırı üç değeri tek bir demet olarak döndürür; `g, x, y = ext_gcd(b, a % b)` ataması da bu demeti açar. Şimdi aynı fonksiyonu bir `while` döngüsüyle yazalım ve döngüdeki çiftleri demet atamasıyla güncelleyelim. $\gcd(252, 198)$ değerini ve Bézout katsayılarını bulalım.

```
def ext_gcd(a, b):
    """(g, x, y) döndürür: g = ebob(a, b) ve a*x + b*y = g."""
    x0, y0, x1, y1 = 1, 0, 0, 1
    while b != 0:
        q, r = divmod(a, b)
        a, b = b, r
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1
    return a, x0, y0
```

 CC BY-NC-SA 4.0

```
result = ext_gcd(252, 198)
print(result)
g, x, y = result
print(252 * x + 198 * y == g)
```

Çıktı:

```
(18, 4, -5)
True
```

Fonksiyon her adımda $a = qb + r$ bölmesini yapar. `(x0, y0)` ve `(x1, y1)` çiftleri, o anki `a` ve `b` değerlerini başlangıçtaki iki sayının doğrusal birleşimi olarak yazan katsayılardır; her adımda `a` ve `b` ile aynı kuralla güncellenirler. Sonuçta $\gcd(252, 198) = 18$ ve

$$252 \cdot 4 + 198 \cdot (-5) = 1008 - 990 = 18$$

bulunur. Özyinelemeli `ext_gcd(252, 198)` çağrısı da aynı `(18, 4, -5)` demetini döndürür.

Demetler değiştirilemez oldukları için, bir sonraki kısımda göreceğimiz sözlüklerde anahtar ve kümelerde eleman olabilirler; listeler olamaz.

3.5 Sözlükler

Bir listede elemana konumuyla, yani $0, 1, 2, \ldots$ indisleriyle ulaşırız. Pek çok problemde ise elemana bir adla ya da bir değerle ulaşmak isteriz: bir asalın üssü, bir kelimenin metinde kaç kez geçtiği, bir permütasyonun bir noktayı nereye götürdüğü gibi.

Tanım 3.6 (Sözlük). **Sözlük** (dictionary, `dict`), `anahtar: değer` çiftlerinden oluşan bir topluluktur: `{"I": 1, "V": 5, "X": 10}`. Bir sözlükte her anahtar en fazla bir kez geçer ve `d[k]` ifadesi `k` anahtarına karşılık gelen değeri verir. Sözlükler değiştirilebilir: `d[k] = v` ataması `k` anahtarı yoksa yeni bir çift ekler, varsa onun değerini değiştirir.

Yani bir sözlük, tanım kümesi sonlu olan bir fonksiyondur: anahtarlar tanım kümesini, değerler de görüntüleri oluşturur. Sözlükte olmayan bir anahtarı `d[k]` ile istemek `KeyError` hatası verir; `d.get(k, varsayılan)` ise anahtar yoksa verilen varsayılan değeri döndürür. Sözlükler çiftleri eklendikleri sırayla saklar ve o sırayla dolaşır.

Örnek 3.8 (Sonlu Bir Fonksiyonu Sözlükle Tutma). $f(x) = x^2 \bmod 7$ fonksiyonunu önce $\{0, 1, 2, 3\}$ üzerinde bir sözlük olarak yazalım, sonra tanım kümesine 4'ü ekleyelim.

```
f = {0: 0, 1: 1, 2: 4, 3: 2}    # f(x) = x**2 mod 7, x = 0, 1, 2, 3
print(f[2])                      # f(2)
f[4] = 2                         # tanım kümesine 4'ü ekle
print(f)
print(len(f), 5 in f, f.get(5, "tanımsız"))
print(list(f.keys()))
print(list(f.values()))
print(list(f.items())[:2])
```

Çıktı:

```
4
{0: 0, 1: 1, 2: 4, 3: 2, 4: 2}
5 False tanımsız
[0, 1, 2, 3, 4]
```

 CC BY-NC-SA 4.0

```
[0, 1, 4, 2, 2]
[(0, 0), (1, 1)]
```

`keys`, `values` ve `items` metotları sırasıyla anahtarları, değerleri ve `(anahtar, değer)` demetlerini verir; bunları `list` ile listeye çevirdik. $f(3) = f(4) = 2$ olduğundan f birebir değildir: değerler arasında 2 iki kez geçiyor.

Sözlüğün anahtarları her türden nesne olamaz.

Tanım 3.7 (Hashlenebilir Nesne). Python'un, içeriğinden hesaplanan ve nesnenin ömrü boyunca değişmeyen bir tam sayı atayabildiği nesnelere **hashlenebilir** (hashable) nesneler denir; bu tam sayıya nesnenin **hash değeri** denir ve `hash(x)` ile bulunur. Sayılar, karakter dizileri ve elemanları hashlenebilir olan demetler hashlenebilir; listeler, sözlükler ve kümeler hashlenebilir değildir. Sözlük anahtarlarının ve küme elemanlarının hashlenebilir olması gerekir.

Yani anahtar olarak `3`, `"asal"` ya da `(2, 3)` kullanılabilir, `[2, 3]` kullanılamaz. Bunun nedeni, sözlüğün bir anahtarı hash değerinden yararlanarak doğrudan bulmasıdır. İçeriği değişebilen bir nesnenin hash değeri de değişeceğinden sözlük onu yerinde bulamaz. Aynı mekanizma sayesinde `k in d` sorgusu, sözlük ne kadar büyük olursa olsun ortalama olarak sabit sürede yanıtlanır.

Örnek 3.9 (Asal Çarpanlara Ayırma Sözlüğü). Her $n > 1$ tam sayısı asal çarpanlarına tek türlü ayrılır (bkz. Sayılar Teorisi). $n = p_1^{k_1} \cdots p_r^{k_r}$ ayrılışını `{p1: k1, ..., pr: kr}` sözlüğü olarak döndüren bir fonksiyon yazalım. Bu sözlükten n'nin bölen sayısı

$$\tau(n) = \prod_{i=1}^{r} (k_i + 1)$$

(bkz. Sayılar Teorisi) ve bölenlerinin toplamı hemen hesaplanır. Her bölen $p_1^{m_1} \cdots p_r^{m_r}$ $(0 \le m_i \le k_i)$ biçiminde olduğundan bölenlerin toplamı, $1 + p_i + \ldots + p_i^{k_i}$ geometrik toplamlarının çarpımıdır:

$$\sigma(n) = \prod_{i=1}^{r} \frac{p_i^{k_i+1} - 1}{p_i - 1}.$$

$n = 720$ için deneyelim ve sonucu bütün bölenleri tek tek sayarak kontrol edelim.

```
def factorize(n):
    """n'nin asal çarpanlarını {asal: üs} sözlüğü olarak döndürür."""
    factors = {}
    p = 2
    while p * p <= n:
        while n % p == 0:
            factors[p] = factors.get(p, 0) + 1
            n //= p
        p += 1
    if n > 1:                    # geriye kalan büyük bir asal
        factors[n] = factors.get(n, 0) + 1
    return factors


f = factorize(720)
print(f)
tau, sigma = 1, 1
for p, e in f.items():
    tau *= e + 1                             # bölen sayısı
    sigma *= (p ** (e + 1) - 1) // (p - 1)   # bölenler toplamı
print(tau, sigma)
count, total = 0, 0                          # kaba kuvvetle kontrol
for d in range(1, 721):
```

 CC BY-NC-SA 4.0

```
    if 720 % d == 0:
        count += 1
        total += d
print(count, total)
```

Çıktı:

```
{2: 4, 3: 2, 5: 1}
30 2418
30 2418
```

$720 = 2^4 \cdot 3^2 \cdot 5$ olduğundan $\tau(720) = 5 \cdot 3 \cdot 2 = 30$ ve $\sigma(720) = 31 \cdot 13 \cdot 6 = 2418$'dir; kaba kuvvetle yapılan sayım aynı sonucu veriyor. `factors.get(p, 0) + 1` kalıbı, `p` henüz sözlükte yoksa sayımı 0'dan başlatır.

Örnek 3.10 (Kelime Sayımı). Bir metinde her kelimenin kaç kez geçtiğini bulalım. `text.split()` metni boşluklardan bölerek kelimelerin listesini verir; her kelime için sözlükteki sayacı bir artırırız.

```
from collections import Counter

text = ("asal sayı yalnız bir ve kendisine bölünen sayıdır "
        "bir sayı asal değilse bileşik sayıdır "
        "bir sayının asal çarpanları tek türlü belirlidir")
words = text.split()       # boşluklardan bölüp kelime listesi kur
counts = {}
for word in words:
    counts[word] = counts.get(word, 0) + 1
print(len(words), len(counts))
print(counts["asal"], counts["sayı"], counts.get("küme", 0))
print(Counter(words).most_common(3))
```

Çıktı:

```
21 15
3 2 0
[('asal', 3), ('bir', 3), ('sayı', 2)]
```

Metinde 21 kelime vardır ama bunların yalnız 15'i farklıdır. Bu sayma kalıbı o kadar sık kullanılır ki `collections` modülü onu hazır bir sözlük türü olarak sunar: `Counter`, bir listedeki her elemanın kaç kez geçtiğini sayar ve `most_common(k)` metoduyla en sık geçen `k` elemanı verir. Sayıları eşit olan elemanlar, ilk görüldükleri sırayla gelir.

3.6 Kümeler

Matematikteki kümelerin iki temel özelliği, elemanların sırasının önemsiz olması ve her elemanın en fazla bir kez sayılmasıdır. Python'un küme yapısı tam olarak bu iki özelliği taşır.

Tanım 3.8 (Küme). **Küme** (set), hashlenebilir nesnelerin sırasız ve tekrarsız topluluğudur. Küme parantezleri içinde yazılır: `{2, 3, 5}`. Boş küme `set()` ile kurulur, çünkü `{}` boş bir sözlüktür. `set(a)` ifadesi, yinelenebilir bir `a` nesnesinin farklı elemanlarından oluşan kümeyi verir.

Yani `{1, 2, 2, 3}` ile `{3, 2, 1}` aynı kümedir. Kümelerde indeksleme yoktur, çünkü elemanların bir sırası yoktur. Bir kümeyi yazdırdığımızda gördüğümüz sıra yalnızca iç düzenlemenin sonucudur ve ona güvenilmez. Küme elemanları hashlenebilir olduğundan `x in A` sorgusu da sözlüklerdeki gibi ortalama olarak sabit sürede yanıtlanır; aynı sorgu bir listede bütün elemanları tek tek dolaşmayı gerektirebilir.

 CC BY-NC-SA 4.0

Kümeler üzerindeki işlemler matematikteki karşılıklarıyla aynı adları taşır (bkz. Matematiğin Temelleri):

- `A | B` birleşim $A \cup B$, `A & B` kesişim $A \cap B$,
- `A - B` fark $A \setminus B$, `A ^ B` simetrik fark $A \triangle B$,
- `A <= B` alt küme $A \subseteq B$, `A < B` öz alt küme,
- `x in A` elemanlık $x \in A$, `len(A)` eleman sayısı $|A|$.

```
A = {2, 4, 6, 8, 10, 12}     # 12'ye kadar çift sayılar
B = {3, 6, 9, 12}            # 12'ye kadar 3'ün katları
print(A | B)                 # birleşim
print(A & B)                 # kesişim
print(A - B, B - A)          # farklar
print(A ^ B)                 # simetrik fark
print({6, 12} <= A, len(A | B))
print(set([1, 1, 2, 3, 3, 3]))   # tekrarlar kaybolur
print(set(), type({}))       # boş küme ve boş sözlük
```

Çıktı:

```
{2, 3, 4, 6, 8, 9, 10, 12}
{12, 6}
{8, 2, 10, 4} {9, 3}
{2, 3, 4, 8, 9, 10}
True 8
{1, 2, 3}
set() <class 'dict'>
```

Çıktıdaki `{12, 6}` ve `{8, 2, 10, 4}` sıraları kümelerin sırasız olduğunu açıkça gösteriyor. Son satır da `{}` yazımının boş küme değil boş sözlük olduğunu doğruluyor. Kümeler değiştirilebilir: `A.add(x)` kümeye `x`'i ekler (`x` zaten kümedeyse küme değişmez), `A.discard(x)` de `x`'i kümeden çıkarır. `A.remove(x)` aynı işi yapar ama `x` kümede yoksa `KeyError` hatası verir. Değiştirilebilir oldukları için bir kümenin elemanı başka bir küme olamaz; kümelerden oluşan bir küme gerektiğinde değiştirilemez `frozenset` türü kullanılır.

Örnek 3.11 (İki Kümenin Venn Bölgeleri). $U = \{1, 2, \ldots, 20\}$ evreninde A çift sayıların, B de 3'ün katlarının kümesi olsun. Venn şemasının dört bölgesini Python işlemleriyle hesaplayıp $|A \cup B| = |A| + |B| - |A \cap B|$ eşitliğini kontrol edelim.

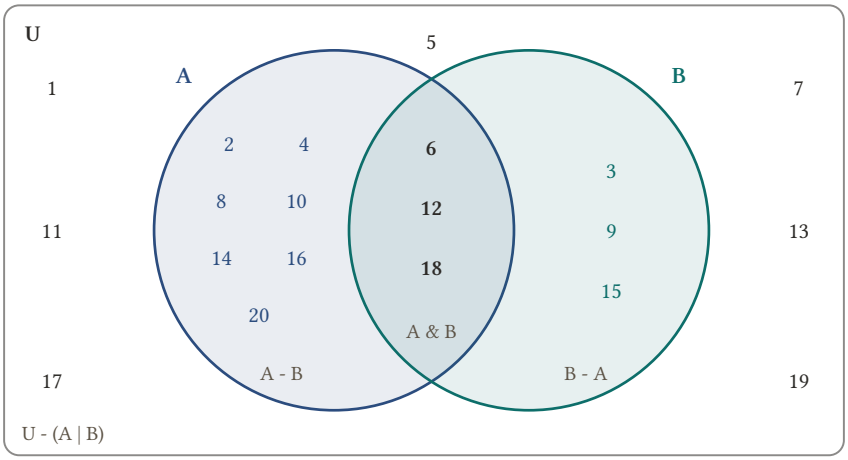
```
U = set(range(1, 21))
A = set(range(2, 21, 2))     # çift sayılar
B = set(range(3, 21, 3))     # 3'ün katları
print(A - B)                 # yalnız A'da
print(A & B)                 # ikisinde birden
print(B - A)                 # yalnız B'de
print(U - (A | B))           # hiçbirinde
print(len(A | B), len(A) + len(B) - len(A & B))
```

Çıktı:

```
{2, 4, 8, 10, 14, 16, 20}
{18, 12, 6}
{9, 3, 15}
{1, 5, 7, 11, 13, 17, 19}
13 13
```

$|A| = 10$, $|B| = 6$ ve $|A \cap B| = 3$ olduğundan iki taraf da 13'tür. Her bölgenin elemanları aşağıdaki şekilde, bölgeyi veren Python ifadesiyle birlikte görülüyor.

 CC BY-NC-SA 4.0



Şekil 3.5: $U = \{1, \ldots, 20\}$ içinde A çift sayılar, B 3'ün katları. Her bölgedeki sayılar, altındaki Python ifadesinin verdiği kümedir: `A - B` yedi, `A & B` üç, `B - A` üç, `U - (A | B)` yedi eleman içerir.

Örnek 3.12 (İçerme ve Dışarma ile Sayma). 1'den 1000'e kadar olan tam sayılardan kaçı 2, 3 ve 5'ten en az birine bölünür? Kümeleri doğrudan kurarak sayalım ve sonucu üç küme için içerme–dışarma formülüyle (bkz. Matematiğin Temelleri) karşılaştıralım.

```
N = 1000
A = set(range(2, N + 1, 2))
B = set(range(3, N + 1, 3))
C = set(range(5, N + 1, 5))
print(len(A | B | C))
print(len(A) + len(B) + len(C)
      - len(A & B) - len(A & C) - len(B & C)
      + len(A & B & C))
```

Çıktı:

```
734
734
```

Formüldeki terimlerin toplamı da aynıdır:

$$500 + 333 + 200 - 166 - 100 - 66 + 33 = 734.$$

Kesişimler ortak katlardan oluşur: örneğin $A \cap B$, 6'nın katlarının kümesidir ve $\lfloor 1000/6 \rfloor = 166$ elemanlıdır.

3.7 Comprehension

Matematikte bir kümeyi çoğu zaman elemanlarını tek tek yazmak yerine bir kuralla tanımlarız: $\{k^2 : 1 \le k \le 10\}$ ya da $\{d \in \mathbb{N} : d \mid 36\}$ gibi. Python'da da listeler, kümeler ve sözlükler aynı biçimde bir kuralla kurulabilir. Önce bu kuralın üzerinde döndüğü nesneleri adlandıralım.

Tanım 3.9 (Yinelenebilir Nesne). Bir `for` döngüsüyle elemanları tek tek dolaşılabilen nesnelere **yinelenebilir** (iterable) nesneler denir. Listeler, demetler, karakter dizileri, kümeler, sözlükler ve `range` nesneleri yinelenebilirdir; bir sözlük dolaşılırken anahtarları verilir.

Yani `for x in nesne:` yazabildiğimiz her şey yinelenebilirdir. `list`, `set`, `sum`, `min`, `max` gibi fonksiyonlar herhangi bir yinelenebilir nesneyi kabul eder.

 CC BY-NC-SA 4.0

Tanım 3.10 (Comprehension). `ifade`, `x` adını içeren bir ifade, `nesne` yinelenebilir bir nesne ve `koşul` doğru ya da yanlış değer alan bir ifade olsun. `[ifade for x in nesne if koşul]` yazımı, `nesne`'nin `koşul`'u sağlayan her `x` elemanı için `ifade`'nin değerini sırayla içeren listeyi kurar; buna **liste comprehension** denir. Köşeli parantez yerine küme parantezi yazılırsa aynı kuralla bir küme, `{anahtar: değer for x in nesne if koşul}` yazımıyla da bir sözlük kurulur. `if koşul` kısmı yazılmayabilir; birden çok `for` ve `if` art arda yazılabilir.

Yani `[k**2 for k in range(1, 11) if k % 2 == 1]` ifadesi $\{k^2 : 1 \le k \le 10, k \text{ tek}\}$ gösteriminin Python'daki karşılığıdır; tek farkı, sonucun sıralı bir liste olmasıdır. Bir comprehension, boş bir listeyle başlayıp döngüde `append` yapan kalıbın kısa yazımıdır.

```
squares = []
for k in range(1, 11):
    squares.append(k**2)
print(squares == [k**2 for k in range(1, 11)])

print([k**2 for k in range(1, 11) if k % 2 == 1])
print({k**2 % 7 for k in range(7)})      # küme: tekrarlar düşer
print({k: k**3 for k in range(1, 6)})    # sözlük
print([d for d in range(1, 37) if 36 % d == 0])
```

Çıktı:

```
True
[1, 9, 25, 49, 81]
{0, 1, 2, 4}
{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
[1, 2, 3, 4, 6, 9, 12, 18, 36]
```

İlk satır, döngüyle kurulan listenin comprehension ile kurulanla aynı olduğunu doğruluyor. Küme comprehension, $\mathbb{Z}_7$'deki tam karelerin kümesini $\{0, 1, 2, 4\}$ olarak verdi (bkz. Sayılar Teorisi). Son satır da 36'nın bölenlerini $\{d : d \mid 36\}$ tanımına harfiyen uyarak listeliyor.
Bir comprehension içinde birden çok `for` yazıldığında bunlar soldan sağa iç içe döngüler gibi okunur: ilk `for` en dıştaki döngüdür. Bir comprehension'ın ifadesi de bir comprehension olabilir; matrisleri bu yolla kurarız.

```
pairs = [(i, j) for i in range(1, 4) for j in range(1, 4) if i < j]
print(pairs)
A = [[1, 2, 3],
     [4, 5, 6]]
At = [[row[j] for row in A] for j in range(3)]   # devrik matris
print(At)
I = [[1 if i == j else 0 for j in range(3)] for i in range(3)]
print(I)
print([x for row in A for x in row])            # düzleştirme
```

Çıktı:

```
[(1, 2), (1, 3), (2, 3)]
[[1, 4], [2, 5], [3, 6]]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
[1, 2, 3, 4, 5, 6]
```

`At` için dış comprehension sütun indisi `j` üzerinde döner ve her `j` için `A`'nın `j` numaralı sütununu bir liste olarak kurar; bu liste devrik matrisin `j` numaralı satırıdır. Son satır, iki `for` ile matrisi tek bir listeye düzleştiriyor: önce satırlar, her satırın içinde de elemanlar dolaşılır.
Köşeli parantezsiz bir comprehension, `sum(k**2 for k in range(10))` örneğindeki gibi bir fonksiyonun tek argümanı olarak yazılabilir. Buna **üreteç ifadesi** (generator expression) denir: değerler bir listede biriktirilmez, fonksiyon onları sırayla birer birer tüketir. Çok terimli toplamlarda bellek harcamamanın yolu budur. `all` ve `any` fonksiyonları da üreteç ifadeleriyle sık kullanılır: `all(...)` kendisine verilen değerlerin hepsi doğruysa, `any(...)` en az biri doğruysa `True` döndürür.

 CC BY-NC-SA 4.0

Örnek 3.13 (Basel Toplamının Kısmi Toplamları). $\sum_{k=1}^{\infty} 1/k^2 = \pi^2/6$ serisinin $S_N = \sum_{k=1}^{N} 1/k^2$ kısmi toplamlarını bir üreteç ifadesiyle hesaplayalım; kalanı $R_N = \pi^2/6 - S_N$ ve $N R_N$ çarpımını da yazdıralım.

```
import math

target = math.pi**2 / 6
for N in [10, 100, 1000, 10000]:
    S = sum(1 / k**2 for k in range(1, N + 1))   # liste kurulmaz
    err = target - S
    print(f"{N:6d}  {S:.10f}  {err:.3e}  {N * err:.4f}")
```

Çıktı:

```
    10  1.5497677312  9.517e-02  0.9517
   100  1.6349839002  9.950e-03  0.9950
  1000  1.6439345667  9.995e-04  0.9995
 10000  1.6448340718  1.000e-04  1.0000
```

Son sütun 1'e yaklaşıyor, yani $R_N \approx 1/N$'dir. Bu, integral testinden gelen $\frac{1}{N+1} \le R_N \le \frac{1}{N}$ kestirimiyle uyumludur (bkz. Analiz 2). Seri yavaş yakınsar: hatayı 10^{-6}'nın altına indirmek için yaklaşık bir milyon terim gerekir.

> ⚠ Comprehension ne zaman kullanılmamalı
>
> Comprehension bir değer kurmak içindir. Yalnızca yan etkisi için, örneğin `[print(x) for x in a]` biçiminde yazdırmak için kullanılmaz; bunun için düz bir `for` döngüsü daha açıktır. İkiden fazla iç içe `for` ya da uzun koşullar içeren bir comprehension okunaksızlaşır; o durumda bir döngüye ya da yardımcı bir fonksiyona geçmek daha iyidir.

3.8 zip, enumerate ve sorted

Yinelenebilir nesnelerle çalışırken üç yerleşik fonksiyon sürekli karşımıza çıkar: birkaç diziyi eş zamanlı dolaşan `zip`, elemanları indisleriyle birlikte veren `enumerate` ve sıralayan `sorted`.

Tanım 3.11 (zip ve enumerate). `zip(a, b)`, yinelenebilir `a` ve `b` nesnelerinin aynı sıradaki elemanlarını `(a[0], b[0])`, `(a[1], b[1])`, … demetleri olarak sırayla verir ve nesnelerden kısa olanı bittiğinde durur; ikiden çok nesneyle de çalışır. `enumerate(a, start=0)` ise `a`'nın her elemanını sırasıyla `(indis, eleman)` demeti olarak verir; indisler `start`'tan başlar.

Yani `zip` iki diziyi bir fermuar gibi eşleştirir. `enumerate` ise `for i in range(len(a))` yazıp `a[i]`'ye ulaşmak yerine indisi ve elemanı birlikte verir. İkisi de demet ürettiğinden döngü değişkenleri doğrudan açılabilir: `for x, y in zip(u, v)`.

```
u = [1, 2, 3]
v = [4, -5, 6]
print(list(zip(u, v)))
print(sum(x * y for x, y in zip(u, v)))      # iç çarpım
print(list(zip("abc", [1, 2])))              # kısa olan belirler
for i, p in enumerate([2, 3, 5, 7], start=1):
    print(f"{i}. asal: {p}")
```

Çıktı:

```
[(1, 4), (2, -5), (3, 6)]
12
```

 CC BY-NC-SA 4.0

```
[('a', 1), ('b', 2)]
1. asal: 2
2. asal: 3
3. asal: 5
4. asal: 7
```

İkinci satır, $u = (1, 2, 3)$ ve $v = (4, -5, 6)$ vektörlerinin iç çarpımını $1 \cdot 4 + 2 \cdot (-5) + 3 \cdot 6 = 12$ olarak hesaplıyor.

Tanım 3.12 (sorted ve Sıralama Anahtarı). `sorted(a, key=f, reverse=False)`, yinelenebilir `a` nesnesinin elemanlarını `f(x)` değerleri artacak biçimde sıralayan **yeni** bir liste döndürür. `key` verilmezse elemanların kendileri karşılaştırılır; `reverse=True` sıralamayı azalan yapar. Listelerin `a.sort(key=f, reverse=False)` metodu aynı sıralamayı listenin kendisi üzerinde yapar ve `None` döndürür.

Yani `key`, her elemana sıralamada kullanılacak karşılaştırılabilir bir değer atayan fonksiyondur ve çoğu zaman bir `lambda` ifadesiyle yazılır. Demetler sözlük sırasıyla karşılaştırılır: önce ilk elemanlara, onlar eşitse ikinci elemanlara bakılır. Python'un sıralaması **kararlıdır**: anahtarları eşit olan elemanlar aralarındaki ilk sırayı korur. `min` ve `max` da aynı `key` argümanını kabul eder.

```
a = [5, 2, 9, 1]
b = sorted(a)                  # yeni liste; a değişmez
print(a, b)
a.sort(reverse=True)           # yerinde, büyükten küçüğe
print(a)
words = ["küme", "dizi", "an", "sözlük", "demet"]
print(sorted(words))           # alfabetik
print(sorted(words, key=len))  # uzunluğa göre
points = [(3, 4), (-1, 1), (0, -2), (2, 2)]
print(sorted(points, key=lambda p: p[0]**2 + p[1]**2))
print(max(points, key=lambda p: p[1]))
```

Çıktı:

```
[5, 2, 9, 1] [1, 2, 5, 9]
[9, 5, 2, 1]
['an', 'demet', 'dizi', 'küme', 'sözlük']
['an', 'küme', 'dizi', 'demet', 'sözlük']
[(-1, 1), (0, -2), (2, 2), (3, 4)]
(3, 4)
```

Uzunluğa göre sıralamada `küme` ile `dizi` eşit uzunluktadır ve kararlılık gereği listedeki sıralarıyla gelir. Noktalar da orijine uzaklıklarının kareleri $2, 4, 8, 25$ olacak biçimde sıralanmıştır. Karakter dizileri harflerin Unicode kod numaralarına göre karşılaştırılır. Bu yüzden `ç`, `ğ`, `ı`, `ö`, `ş`, `ü` ile başlayan kelimeler `z`'den sonraya düşer ve sıralama Türk alfabesine uymaz.

Örnek 3.14 (Kelimeleri Sıklığa Göre Sıralama). Örnek 3.10 içindeki metnin en sık geçen beş kelimesini, eşit sayıda geçenleri alfabetik sırada vererek listeleyelim. Anahtar olarak `(-sayı, kelime)` demetini kullanırsak demet karşılaştırması önce sayıya göre azalan, sonra kelimeye göre artan bir sıra kurar.

```
text = ("asal sayı yalnız bir ve kendisine bölünen sayıdır "
        "bir sayı asal değilse bileşik sayıdır "
        "bir sayının asal çarpanları tek türlü belirlidir")
counts = {}
for word in text.split():
    counts[word] = counts.get(word, 0) + 1
# önce sayıya göre azalan, eşitlikte kelimeye göre artan
ranking = sorted(counts.items(), key=lambda kv: (-kv[1], kv[0]))
for word, c in ranking[:5]:
    print(f"{word:<12}{c}")
```

Çıktı:

 CC BY-NC-SA 4.0

```
asal        3
bir         3
sayı        2
sayıdır     2
belirlidir  1
```

`counts.items()` her çifti `(kelime, sayı)` demeti olarak verir; `kv[1]` sayı, `kv[0]` kelimedir. Sayıyı eksi işaretiyle almak, `reverse=True` kullanmadan yalnız o bileşende sırayı tersine çevirir.

Örnek 3.15 (Pascal Üçgeni). Pascal üçgeninin n numaralı satırı $\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}$ binom katsayılarından oluşur. Her iç eleman, Pascal özdeşliği $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ gereği üstündeki iki elemanın toplamıdır (bkz. Olasılık Teorisi). Bir satırın başına ve sonuna birer 0 ekleyip iki listeyi `zip` ile eleman eleman toplarsak bir sonraki satırı buluruz.

```
row = [1]
for n in range(7):
    print(row)
    row = [a + b for a, b in zip([0] + row, row + [0])]
```

Çıktı:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
```

Örneğin `row = [1, 3, 3, 1]` iken `[0, 1, 3, 3, 1]` ile `[1, 3, 3, 1, 0]` listeleri eşleştirilir ve toplamları `[1, 4, 6, 4, 1]` olur. Uçlara eklenen 0'lar sayesinde satırın iki ucundaki 1'leri ayrıca yazmak gerekmez.

Örnek 3.16 (Pascal Üçgenindeki Tek Sayılar). Pascal üçgeninin ilk 16 satırını kuralım ve tek katsayıları `#`, çiftleri `.` ile göstererek yazdıralım. Satırlar `rows` listesinde biriktirilir; her yeni satır, `rows[-1]` ile alınan son satırdan üretilir. `" ".join(...)` ifadesi, karakter dizilerini aralarına birer boşluk koyarak birleştirir.

```
rows = [[1]]
for n in range(1, 16):
    prev = rows[-1]
    rows.append([a + b for a, b in zip([0] + prev, prev + [0])])
for n, row in enumerate(rows):
    marks = " ".join("#" if c % 2 == 1 else "." for c in row)
    print(" " * (15 - n) + marks)
```

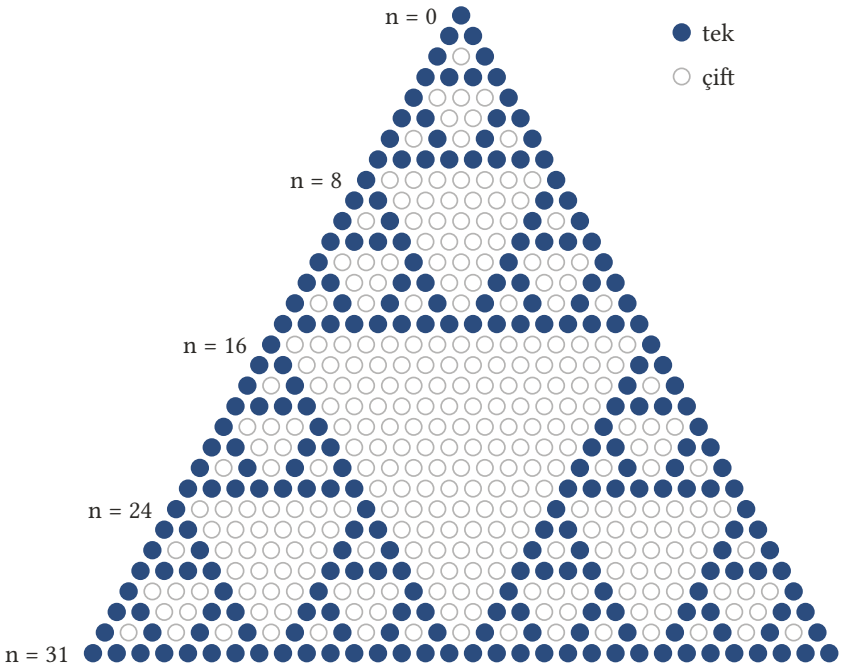
Çıktı:

```
               #
              # #
             # . #
            # # # #
           # . . . #
          # # . . # #
         # . # . # . #
        # # # # # # # #
```

 CC BY-NC-SA 4.0

```
      # . . . . . . . #
     # # . . . . . . # #
    # . # . . . . . # . #
   # # # # . . . . # # # #
  # . . . # . . . # . . . #
 # # . . # # . . # # . . # #
# . # . # . # . # . # . # . #
# # # # # # # # # # # # # # #
```

Tek katsayılar kendini tekrarlayan bir desen oluşturuyor: ilk 2^m satırın deseni, sonraki 2^m satırda yan yana iki kez yinelenir ve ortada çift sayılardan oluşan ters bir üçgen kalır. Aynı kod `range(1, 32)` ile çalıştırıldığında elde edilen 32 satır aşağıda çizilmiştir; bu desen **Sierpinski üçgeni** olarak bilinir.



Şekil 3.6: Pascal üçgeninin ilk 32 satırı (n = 0, …, 31); her nokta bir binom katsayısıdır. Dolu noktalar tek, boş noktalar çift katsayılardır. Satırlar koddaki `zip` kuralıyla üretildi; tek katsayılar Sierpinski üçgenini çizer ve sayıları $3^5 = 243$'tür.

3.9 itertools ile Sayma

Sayma problemlerinde bir formüle güvenmeden önce küçük durumları tek tek listelemek çoğu zaman öğreticidir. Standart kütüphanedeki `itertools` modülü, bir kümenin sıralamalarını, seçimlerini ve kartezyen çarpımlarını üreten hazır fonksiyonlar sunar.

Tanım 3.13 (permutations, combinations ve product). `itertools` modülünün üç fonksiyonu, yinelenebilir bir `a` nesnesinin elemanlarından demetler üretir. `permutations(a, k)`, `a`'nın farklı konumlardaki `k` elemanının bütün sıralı dizilişlerini, yani `k`'lı permütasyonlarını verir; `k` yazılmazsa `len(a)` alınır. `combinations(a, k)`, `a`'nın `k` elemanlı bütün sırasız seçimlerini verir; her seçim, elemanların `a`'daki sırasıyla bir kez üretilir. `product(a, b)` ise `a` ile `b`'nin kartezyen çarpımının bütün elemanlarını verir; `product(a, repeat=k)`, `a`'nın kendisiyle `k` katlı kartezyen çarpımıdır. Üç fonksiyon da sonuçlarını girdideki sıraya göre sözlük sırasıyla üretir.

Yani bu fonksiyonlar, sayma tekniklerinde sayısını formülle bulduğumuz nesneleri tek tek önümüze serer. n elemanlı bir kümenin k'lı permütasyonlarının sayısı $n!/(n-k)!$, kombinasyonlarının sayısı $\binom{n}{k}$, kendisiyle k katlı kartezyen çarpımının eleman sayısı da çarpma kuralı gereği n^k'dir (bkz. Olasılık Teorisi:

 CC BY-NC-SA 4.0

permütasyon sayısı, kombinasyon sayısı, çarpma kuralı). `math.perm(n, k)` ve `math.comb(n, k)` ilk iki sayıyı doğrudan hesaplar.

```
import math
from itertools import combinations, permutations, product

print(["".join(p) for p in permutations("abc")])
print(list(combinations([1, 2, 3, 4], 2)))
print(["".join(t) for t in product("01", repeat=3)])
n, k = 6, 3
print(len(list(permutations(range(n), k))), math.perm(n, k))
print(len(list(combinations(range(n), k))), math.comb(n, k))
print(len(list(product(range(n), repeat=k))), n**k)
```

Çıktı:

```
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
['000', '001', '010', '011', '100', '101', '110', '111']
120 120
20 20
216 216
```

Üç fonksiyon da sonuçlarını bir liste olarak değil, istendikçe üretir; bu yüzden hepsini görmek için `list` ile listeye çevirdik. 6 elemanlı bir kümeden 3 eleman için $6 \cdot 5 \cdot 4 = 120$ sıralı seçim, $\binom{6}{3} = 20$ sırasız seçim ve $6^3 = 216$ sıralı üçlü vardır.

Örnek 3.17 (Kuvvet Kümesini Listeleme). $S = \{1, 2, 3\}$ kümesinin bütün alt kümelerini, $k = 0, 1, 2, 3$ için k elemanlı alt kümeleri art arda `combinations` ile üreterek listeleyelim.

```
from itertools import combinations

S = [1, 2, 3]
power = [set(c) for k in range(len(S) + 1)
         for c in combinations(S, k)]
print(power)
print(len(power), 2 ** len(S))
```

Çıktı:

```
[set(), {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
8 8
```

Boş küme `set()` olarak yazdırılır. Alt küme sayısı $\binom{3}{0} + \binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 8 = 2^3$'tür (bkz. Matematiğin Temelleri).

Örnek 3.18 (İki Zarın Toplamının Dağılımı). İki zar atıldığında üste gelen sayıların toplamı hangi değeri kaç farklı biçimde alır? 36 sıralı sonucu `product` ile üretelim, toplamları `Counter` ile sayalım ve her toplamın olasılığını `Fraction` ile kesir olarak yazalım (`Fraction` için bkz. Python ile İlk Adımlar).

```
from collections import Counter
from fractions import Fraction
from itertools import product

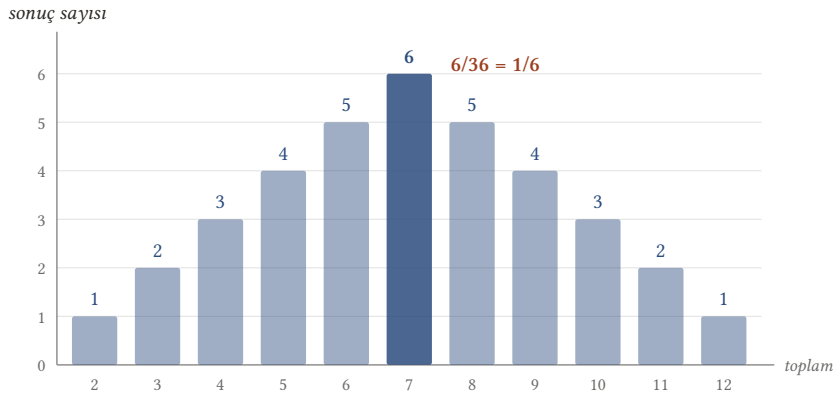
sums = Counter(a + b for a, b in product(range(1, 7), repeat=2))
for s in sorted(sums):
    print(f"{s:2d}  {sums[s]}  {Fraction(sums[s], 36)}")
```

Çıktı:

 CC BY-NC-SA 4.0

```
 2  1  1/36
 3  2  1/18
 4  3  1/12
 5  4  1/9
 6  5  5/36
 7  6  1/6
 8  5  5/36
 9  4  1/9
10  3  1/12
11  2  1/18
12  1  1/36
```

Dağılım 7 etrafında simetriktir: s toplamı $6 - |\,s - 7\,|$ farklı sonuçla elde edilir, bu yüzden en olası toplam 7'dir ve olasılığı $6/36 = 1/6$'dır.



Şekil 3.7: İki zarın toplamı için `product(range(1, 7), repeat=2)` ile sayılan 36 sonucun dağılımı. Çubuk yükseklikleri `Counter` nesnesindeki sayılardır; en olası toplam 7'dir.

3.10 Eratosthenes Kalburu

Bölümün araçlarını bir araya getiren klasik bir örnekle bitirelim: belli bir sınıra kadar bütün asalları bulmak. Her sayının asallığını tek tek bölme denemesiyle sınamak yerine, Eratosthenes kalburu bütün bileşik sayıları asalların katları olarak eler. Yöntem şu gerçeğe dayanır: asal olmayan her $n > 1$ sayısının $\sqrt{n}$'den büyük olmayan bir asal böleni vardır (bkz. Sayılar Teorisi). Bu yüzden N'ye kadar eleme yaparken $p^2 \le N$ olan asallar yeter. Ayrıca her p için eleme p^2'den başlayabilir, çünkü p^2'den küçük bileşik katlar daha küçük bir asal böleni olduğundan zaten elenmiştir.

Dört adımda Eratosthenes kalburu

1. $0, 1, \ldots, N$ indisli, bütün elemanları `True` olan bir `is_prime` listesi kurun ve 0 ile 1'in değerini `False` yapın.
2. $p = 2$'den başlayıp $p^2 \le N$ olduğu sürece p'yi birer artırın.
3. `is_prime[p]` hâlâ `True` ise p asaldır: $p^2, p^2 + p, p^2 + 2p, \ldots$ indislerini adımlı dilim atamasıyla `False` yapın.
4. Değeri `True` kalan indisleri `enumerate` ve bir comprehension ile toplayın.

Örnek 3.19 (Yüze Kadar Asallar). Kalburu bir fonksiyon olarak yazıp 100'e kadar olan asalları bulalım ve onar onar yazdıralım.

```
def sieve(n):
    """n'ye kadar (n dahil) asalların listesini döndürür."""
    is_prime = [True] * (n + 1)
```

 CC BY-NC-SA 4.0

```
    is_prime[0] = is_prime[1] = False
    p = 2
    while p * p <= n:
        if is_prime[p]:
            # p*p, p*p + p, p*p + 2p, ... sayılarını tek atamayla ele
            is_prime[p * p::p] = [False] * len(range(p * p, n + 1, p))
        p += 1
    return [k for k, flag in enumerate(is_prime) if flag]

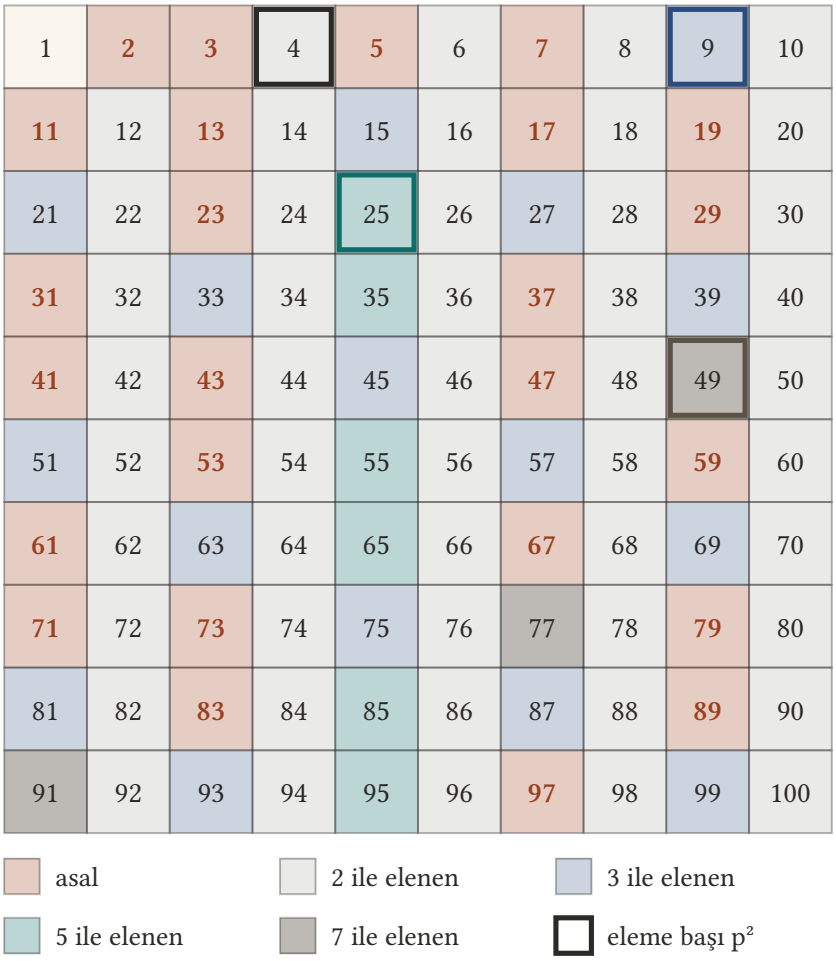

primes = sieve(100)
print(len(primes))
for i in range(0, len(primes), 10):
    print(primes[i:i + 10])
```

Çıktı:

```
25
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
[31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
[73, 79, 83, 89, 97]
```

`is_prime[p * p::p]` dilimi p^2'den başlayıp p'şer ilerleyen bütün indisleri kapsar; sağ taraftaki liste de tam o uzunluktadır. 100 için yalnız $p = 2, 3, 5, 7$ ile eleme yapılır, çünkü $11^2 = 121 > 100$'dür. Aşağıdaki şekil, her bileşik sayının hangi asalla ilk kez elendiğini gösteriyor.

 CC BY-NC-SA 4.0



Şekil 3.8: Eratosthenes kalburu 1'den 100'e kadar. Her bileşik sayı, onu ilk kez eleyen asalın rengini taşır; bu asal, sayının en küçük asal bölenidir. Kalın çerçeveli 4, 9, 25 ve 49, sırasıyla 2, 3, 5 ve 7 ile elemenin başladığı p^2 sayılarıdır. $\sqrt{100} = 10$ olduğundan 7'den sonra eleme gerekmez; 1 dışında elenmeden kalan 25 sayı asaldır.

Örnek 3.20 (Asalları Sayma). n'den büyük olmayan asalların sayısını $\pi(n)$ ile gösterelim. Kalburla 10^6'ya kadar bütün asalları bir kez bulup $\pi(10^k)$ değerlerini ve $\pi(n)$'nin $n/\ln n$'ye oranını hesaplayalım. Aşağıdaki kod, bir önceki kod bloğunun devamıdır: `sieve` fonksiyonunu yeniden tanımlamadan kullanır.

```
import math

primes = sieve(10**6)
for k in range(1, 7):
    n = 10**k
    count = sum(1 for p in primes if p <= n)
    print(f"{n:>8}  {count:>6}  {count / (n / math.log(n)):.4f}")
```

Çıktı:

```
      10       4  0.9210
     100      25  1.1513
    1000     168  1.1605
   10000    1229  1.1320
```

```
  100000    9592  1.1043
 1000000   78498  1.0845
```

Oran önce artıyor; $n = 1000$'den sonra ise 1'in üstünde kalarak yavaşça 1'e doğru iniyor. Asal sayı teoremi, $n \to \infty$ iken $\pi(n)$ ile $n/\ln n$'nin oranının 1'e gittiğini söyler.

3.11 Alıştırmalar

Aşağıdaki alıştırmaların her biri bu bölümdeki yapılardan en az birini kullanır. Çözümü açmadan önce kodu kendiniz yazmayı deneyin.

Alıştırma 3.1 (Üç Basamaklı Palindrom Asallar). Üç basamaklı palindrom asal sayıları bir liste comprehension ile bulun.

Çözüm.
Palindrom testi. Sayıyı karakter dizisine çevirip tersiyle karşılaştırırız: `s == s[::-1]`.
Asallık testi. Kontrol Yapıları ve Fonksiyonlar bölümündeki gibi 2'den $\sqrt{n}$'ye kadar bir bölen ararız.
Süzme. `range(100, 1000)` üzerinde iki koşulu `and` ile birleştiren bir comprehension yazarız.

```
def is_prime(n):
    if n < 2:
        return False
    d = 2
    while d * d <= n:
        if n % d == 0:
            return False
        d += 1
    return True


def is_palindrome(n):
    s = str(n)
    return s == s[::-1]


pals = [n for n in range(100, 1000) if is_palindrome(n) and is_prime(n)]
print(len(pals))
print(pals)
four = [n for n in range(1000, 10000) if is_palindrome(n)]
print(len(four), all(n % 11 == 0 for n in four))
```

Çıktı:

```
15
[101, 131, 151, 181, 191, 313, 353, 373, 383, 727, 757, 787, 797, 919, 929]
90 True
```

Üç basamaklı 90 palindromdan 15'i asaldır. Son satır ilginç bir gözlemi doğruluyor: dört basamaklı 90 palindromun hepsi 11'e bölünür, çünkü

$$\overline{abba} = 1001a + 110b = 11(91a + 10b)$$

olur. Bu yüzden dört basamaklı palindrom asal yoktur. $\blacksquare$

Alıştırma 3.2 (İlkel Pisagor Üçlüleri). $a < b < c \le 50$ ve $\gcd(a, b) = 1$ olmak üzere $a^2 + b^2 = c^2$ eşitliğini sağlayan bütün (a, b, c) üçlülerini iç içe bir comprehension ile listeleyin.

 CC BY-NC-SA 4.0

Çözüm.
Döngülerin sırası. Üçlüleri c'ye göre sıralı almak için en dıştaki `for` c üzerinde, içtekiler $b < c$ ve $a < b$ üzerinde döner. Böylece her üçlü bir kez, $a < b$ biçiminde üretilir.
Koşullar. Pisagor eşitliği ve `gcd(a, b) == 1` koşulu tek bir `if` içinde `and` ile birleştirilir; `gcd` fonksiyonu `math` modülünden gelir.

```
from math import gcd

triples = [(a, b, c)
           for c in range(1, 51)
           for b in range(1, c)
           for a in range(1, b)
           if a * a + b * b == c * c and gcd(a, b) == 1]
for t in triples:
    print(t)
print(len(triples))
```

Çıktı:

```
(3, 4, 5)
(5, 12, 13)
(8, 15, 17)
(7, 24, 25)
(20, 21, 29)
(12, 35, 37)
(9, 40, 41)
7
```

$c \le 50$ için 7 ilkel üçlü vardır. Örneğin $(6, 8, 10)$ eşitliği sağlar ama $\gcd(6, 8) = 2$ olduğundan ilkel değildir ve listeye girmez. ■

Alıştırma 3.3 (Comprehension ile Matris Çarpımı). $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ ve $B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$ matrislerini satırlarının listesi olarak tutun ve AB çarpımını yalnız comprehension, `zip` ve `sum` kullanarak hesaplayın.

Çözüm.
Sütunlar. AB'nin (i, j) elemanı, A'nın i numaralı satırı ile B'nin j numaralı sütununun iç çarpımıdır. B'nin sütunlarını devrik alma kalıbıyla kurarız: `cols[j]`, `B`'nin `j` numaralı sütunudur.
İç çarpım. Bir satır ile bir sütunun iç çarpımı `sum(a * b for a, b in zip(row, col))` ifadesidir.
Çarpım. Dış comprehension `A`'nın satırları, iç comprehension `B`'nin sütunları üzerinde döner; böylece `C[i][j]` doğru yere düşer.

```
A = [[1, 2, 3],
     [4, 5, 6]]
B = [[7, 8],
     [9, 10],
     [11, 12]]
cols = [[row[j] for row in B] for j in range(len(B[0]))]
C = [[sum(a * b for a, b in zip(row, col)) for col in cols]
     for row in A]
print(cols)
print(C)
```

Çıktı:

 CC BY-NC-SA 4.0

```
[[7, 9, 11], [8, 10, 12]]
[[58, 64], [139, 154]]
```

Elle kontrol edelim: sol üst eleman $1\cdot 7+2\cdot 9+3\cdot 11=58$, sağ alt eleman $4\cdot 8+5\cdot 10+6\cdot 12=154$'tür. Bu hesap NumPy ile Lineer Cebir bölümünde tek bir `A @ B` yazımına dönüşecek. ■

Alıştırma 3.4 (On Binden Küçük Mükemmel Sayılar). Kendisi dışındaki pozitif bölenlerinin toplamına eşit olan sayılara mükemmel sayı denir; örneğin $6=1+2+3$. $10\,000$'den küçük mükemmel sayıları, bölen toplamlarını bir listede biriktiren kalbur benzeri bir yöntemle bulun.

Çözüm.
Fikir. Her n için bölen aramak yerine işi tersine çevirir iz: her d için d'yi, d'nin $2d, 3d, \ldots$ katlarının hanesine ekleriz. Döngüler bittiğinde `s[m]`, m'nin kendisinden küçük bütün pozitif bölenlerinin toplamı olur.
Maliyet. Her d için yaklaşık N/d ekleme yapılır. Toplam işlem sayısı $N(1+\frac{1}{2}+\frac{1}{3}+\cdots)$ mertebesinde, yani yaklaşık $N\ln N$'dir; her sayıyı ayrı ayrı bölme denemesine sokmaktan çok daha azdır.
Süzme. `s[n] == n` olan sayılar mükemmeldir.

```
N = 10000
s = [0] * N        # s[m]: m'nin kendisinden küçük bölenlerinin toplamı
for d in range(1, N // 2):
    for m in range(2 * d, N, d):    # d'nin 2d, 3d, ... katları
        s[m] += d
print(s[1:13])
perfect = [n for n in range(2, N) if s[n] == n]
print(perfect)
print([2**(p - 1) * (2**p - 1) for p in [2, 3, 5, 7]])
```

Çıktı:

```
[0, 1, 1, 3, 1, 6, 1, 7, 4, 8, 1, 16]
[6, 28, 496, 8128]
[6, 28, 496, 8128]
```

İlk satır yöntemi küçük sayılarda doğruluyor: örneğin 12'nin kendisinden küçük bölenlerinin toplamı $1+2+3+4+6=16$'dır. Bulunan dört sayı $2^{p-1}(2^p-1)$ biçimindedir; $p=2,3,5,7$ için 2^p-1 sayıları $3,7,31,127$ asaldır. 2^p-1 asal olduğunda $2^{p-1}(2^p-1)$ sayısının mükemmel olduğunu Öklid, her çift mükemmel sayının bu biçimde olduğunu Euler göstermiştir. ■

Alıştırma 3.5 (De Morgan Kurallarını Bütün Alt Kümelerde Sınama). $U=\{1,2,3,4\}$ olsun. U'nun bütün A, B alt küme çiftleri için De Morgan kurallarının, yani

$$U\setminus(A\cup B)=(U\setminus A)\cap(U\setminus B),\qquad U\setminus(A\cap B)=(U\setminus A)\cup(U\setminus B)$$

eşitliklerinin sağlandığını Python ile doğrulayın.

Çözüm.
Alt kümeler. Örnek 3.17 içindeki gibi, $k=0,\ldots,4$ için `combinations` ile U'nun $2^4=16$ alt kümesini kurarız. Kümenin elemanlarını önce `sorted(U)` ile sıralı bir listeye çevirir iz ki alt kümeler belirli bir sırayla üretilsin.
Çiftler. İki `for` içeren bir comprehension $16\cdot 16=256$ çiftin hepsini kurar.
Sınama. Her kural için bir üreteç ifadesini `all` ile denetleriz; `all`, ancak bütün çiftlerde eşitlik sağlanıyorsa `True` döndürür.

 CC BY-NC-SA 4.0

```
from itertools import combinations

U = {1, 2, 3, 4}
subsets = [set(c) for k in range(len(U) + 1)
           for c in combinations(sorted(U), k)]
pairs = [(A, B) for A in subsets for B in subsets]
ok1 = all(U - (A | B) == (U - A) & (U - B) for A, B in pairs)
ok2 = all(U - (A & B) == (U - A) | (U - B) for A, B in pairs)
print(len(subsets), len(pairs), ok1, ok2)
wrong = [(A, B) for A, B in pairs if U - (A | B) != (U - A) | (U - B)]
print(len(wrong))
```

Çıktı:

```
16 256 True True
240
```

İki kural da 256 çiftin hepsinde sağlanıyor. Karşılaştırma için son satır, yanlış bir "kural" olan $U \setminus (A \cup B) = (U \setminus A) \cup (U \setminus B)$ eşitliğinin kaç çiftte bozulduğunu sayıyor: 240 çiftte. De Morgan kuralına göre sol taraf $(U \setminus A) \cap (U \setminus B)$'ye eşittir. İki kümenin kesişimi ancak bu iki küme eşitse birleşimlerine eşit olur; bu yüzden yanlış kural yalnızca $A = B$ olan 16 çiftte doğrudur. Sonlu bir evrende yapılan bu sınama bir ispat değildir; ispat için bkz. Matematiğin Temelleri. ■

Alıştırma 3.6 (Bir Permütasyonun Döngü Ayrışımı). σ, $\{1, 2, \ldots, 7\}$ kümesinin $1 \mapsto 3$, $2 \mapsto 5$, $3 \mapsto 1$, $4 \mapsto 6$, $5 \mapsto 2$, $6 \mapsto 7$, $7 \mapsto 4$ kuralıyla verilen permütasyonu olsun. σ'yı bir sözlük olarak tutup ayrık döngülerine ayıran bir program yazın.

Çözüm.
Gösterim. Permütasyon bir fonksiyon olduğundan sözlük olarak tutulur: `sigma[x]` değeri $\sigma(x)$'tir.
Bir döngüyü izleme. Henüz görülmemiş bir `start` noktasından başlayıp $x \mapsto \sigma(x)$ adımlarını `start`'a geri dönene kadar sürdürürüz. Uğranan noktaları hem `append` ile döngünün listesine hem de `add` ile `seen` kümesine ekleriz. Biten döngüyü `tuple(cycle)` ile saklarız: `tuple`, tıpkı `list` gibi, verilen listenin elemanlarından bir demet kurar.
Bütün noktalar. `start` sözlüğün bütün anahtarlarını dolaşır; daha önce bir döngüde görülen noktalar `seen` kümesi sayesinde atlanır. Küme kullanmak, "bu nokta görüldü mü?" sorusunu listede arama yapmadan yanıtlar.
Kontrol. Kodun ikinci yarısı σ'nın kuvvetlerini bir sözlük comprehension ile hesaplar: `power` sözlüğü σ^m'yi tutar. Başlangıçta $m = 1$'dir ve `power`, `sigma`'nın bir kopyasıdır: `dict(sigma)`, tıpkı bir listeyi kopyalayan `list(a)` gibi, aynı çiftlerden oluşan yeni bir sözlük kurar. `power` her turda $\sigma^{m+1}(x) = \sigma(\sigma^m(x))$ kuralıyla güncellenir. Döngü, `power` birim permütasyon olunca durur.

```
sigma = {1: 3, 2: 5, 3: 1, 4: 6, 5: 2, 6: 7, 7: 4}
seen = set()
cycles = []
for start in sigma:
    if start in seen:
        continue
    cycle = [start]
    seen.add(start)
    x = sigma[start]
    while x != start:          # başlangıca dönene kadar izle
        cycle.append(x)
        seen.add(x)
        x = sigma[x]
    cycles.append(tuple(cycle))
print(cycles)
```

 CC BY-NC-SA 4.0

```
power, m = dict(sigma), 1      # sigma'nın kuvvetleri
while any(power[x] != x for x in power):
    power = {x: sigma[power[x]] for x in power}
    m += 1
print(m)
```

Çıktı:

```
[(1, 3), (2, 5), (4, 6, 7)]
6
```

Ayrışım $\sigma = (1\,3)(2\,5)(4\,6\,7)$'dir. Birim permütasyona ilk kez $m = 6$'da ulaşılması da bu ayrışımla uyumludur: 6, döngü uzunlukları 2, 2 ve 3'ün en küçük ortak katıdır. ■

Alıştırma 3.7 (Hiçbir Elemanı Yerinde Bırakmayan Permütasyonlar). $\{0, 1, 2, 3, 4\}$ kümesinin her i için $p(i) \neq i$ koşulunu sağlayan p permütasyonlarını `permutations` ile sayın ve sonucu içerme–dışarma ilkesinden gelen

$$D_n = \sum_{k=0}^{n} (-1)^k \frac{n!}{k!}$$

formülüyle karşılaştırın.

Çözüm.
Permütasyonlar. `permutations(range(5))` her permütasyonu $(p(0), \ldots, p(4))$ demeti olarak verir.
Koşul. Bir demetin hiçbir elemanı yerinde değilse `all(p[i] != i for i in range(n))` doğrudur. Bu demetleri `sum(1 for ...)` kalıbıyla sayarız.
Formül. $k \le n$ için $n!/k!$ bir tam sayı olduğundan tam sayı bölmesi `//` kesin sonuç verir.

```
import math
from itertools import permutations

n = 5
count = sum(1 for p in permutations(range(n))
            if all(p[i] != i for i in range(n)))
formula = sum((-1)**k * (math.factorial(n) // math.factorial(k))
              for k in range(n + 1))
print(count, formula, math.factorial(n))
print(count / math.factorial(n), 1 / math.e)
```

Çıktı:

```
44 44 120
0.36666666666666664 0.36787944117144233
```

120 permütasyondan 44'ü hiçbir elemanı yerinde bırakmaz ve formül aynı sayıyı verir: $120 - 120 + 60 - 20 + 5 - 1 = 44$. Son satır, rastgele seçilen bir permütasyonun hiçbir elemanı yerinde bırakmama olasılığı olan $D_5/5! \approx 0{,}3667$ değerinin $1/e \approx 0{,}3679$'a şimdiden yakın olduğunu gösteriyor. Bunun nedeni, $D_n/n! = \sum_{k=0}^{n} (-1)^k/k!$ toplamının e^{-1} serisinin bir kısmi toplamı olmasıdır. ■

Alıştırma 3.8 (Üç Zarla Toplam On). Üç zar atıldığında toplamın 10 olma olasılığını, 216 sıralı sonucu `product` ile üretip sayarak kesir olarak bulun.

Çözüm.

 CC BY-NC-SA 4.0

Örnek uzay. `product(range(1, 7), repeat=3)` bütün (a, b, c) sonuçlarını üretir; eşit olasılıklı $6^3 = 216$ sonuç vardır.
Olay. Toplamı 10 olan demetleri bir comprehension ile süzeriz.
Olasılık. İstenen sonuç sayısını toplam sonuç sayısına bölen `Fraction` sadeleşmiş kesri verir.

```
from fractions import Fraction
from itertools import product

outcomes = list(product(range(1, 7), repeat=3))
favorable = [t for t in outcomes if sum(t) == 10]
print(len(outcomes), len(favorable))
print(Fraction(len(favorable), len(outcomes)))
print(favorable[:6])
```

Çıktı:

```
216 27
1/8
[(1, 3, 6), (1, 4, 5), (1, 5, 4), (1, 6, 3), (2, 2, 6), (2, 3, 5)]
```

27 sonuç toplamı 10 yapar, dolayısıyla olasılık $27/216 = 1/8$'dir. İlk altı sonuç, `product`'ın sonuçları sözlük sırasıyla ürettiğini gösteriyor: en sağdaki bileşen en hızlı değişir. ■

Alıştırma 3.9 (Toplamı Üçe Bölünen Üçlü Seçimler). $\{1, 2, \ldots, 20\}$ kümesinin, elemanlarının toplamı 3'e bölünen 3 elemanlı alt kümelerini `combinations` ile sayın ve sonucu kalanlara göre yapılan bir sayımla doğrulayın.

Çözüm.
Doğrudan sayım. `combinations(range(1, 21), 3)` bütün $\binom{20}{3} = 1140$ alt kümeyi üretir; toplamı 3'e bölünenleri süzeriz.
Kalanlara göre sayım. Üç sayının 3'e bölümünden kalanlar ya üçü de aynı olmalı ya da 0, 1, 2 kalanlarının her biri bir kez geçmelidir. Gerçekten iki kalan eşit ve üçüncüsü farklıysa, kalanlar toplamı $2a + b$ ile $b \not\equiv a$ için $2a + b \equiv b - a \not\equiv 0 \pmod 3$ olur. r_j, 3'e bölümünden j kalanını veren sayıların adedi olmak üzere istenen sayı

$$\binom{r_0}{3} + \binom{r_1}{3} + \binom{r_2}{3} + r_0 r_1 r_2$$

olur. r_j değerlerini de birer üreteç ifadesiyle sayarız.

```
import math
from itertools import combinations

triples = list(combinations(range(1, 21), 3))
good = [t for t in triples if sum(t) % 3 == 0]
print(len(triples), len(good))
r = [sum(1 for n in range(1, 21) if n % 3 == j) for j in range(3)]
by_residues = (math.comb(r[0], 3) + math.comb(r[1], 3)
               + math.comb(r[2], 3) + r[0] * r[1] * r[2])
print(r, by_residues)
```

Çıktı:

```
1140 384
[6, 7, 7] 384
```

$r_0 = 6, r_1 = r_2 = 7$ olduğundan

 CC BY-NC-SA 4.0

$$\binom{6}{3} + 2\binom{7}{3} + 6 \cdot 7 \cdot 7 = 20 + 70 + 294 = 384$$

bulunur; doğrudan sayım da aynı sonucu veriyor. ■

Alıştırma 3.10 (Farey Dizisi). Paydası 5'i geçmeyen ve $[0, 1]$ aralığında kalan bütün kesirleri bir küme comprehension ile `Fraction` nesneleri olarak kurun ve küçükten büyüğe sıralayarak F_5 Farey dizisini elde edin.

Çözüm.
Kesirler. $1 \le b \le 5$ ve $0 \le a \le b$ için `Fraction(a, b)` kendiliğinden sadeleşir: `Fraction(2, 4)` ile `Fraction(1, 2)` eşit ve aynı hash değerine sahip nesnelerdir. Küme comprehension bu tekrarları kendiliğinden atar.
Sıralama. `sorted` kümeyi artan sıralı bir listeye çevirir; `Fraction` nesneleri sayı olarak karşılaştırılır.
Kontrol. Farey dizisinde ardışık iki kesir $a/b < c/d$ için $bc - ad = 1$ olduğu bilinir. Ardışık çiftleri `zip(F5, F5[1:])` ile eşleştirip bu özelliği sınarız.

```
from fractions import Fraction

F5 = sorted({Fraction(a, b) for b in range(1, 6) for a in range(b + 1)})
print(len(F5))
print(", ".join(str(x) for x in F5))
print(all(x.denominator * y.numerator - x.numerator * y.denominator == 1
          for x, y in zip(F5, F5[1:])))
```

Çıktı:

```
11
0, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 1
True
```

F_5'in 11 terimi vardır. Bu sayı

$$1 + \sum_{b=1}^{5} \phi(b) = 1 + 1 + 1 + 2 + 2 + 4 = 11$$

ile de bulunur, çünkü $[0, 1]$'deki paydası tam olarak $b \ge 2$ olan sadeleşmiş kesirlerin sayısı Euler fonksiyonu $\phi(b)$'dir (bkz. Sayılar Teorisi); $b = 1$ için $\phi(1) = 1$ terimi $1/1$'i, baştaki 1 de $0/1$'i sayar. ■

Alıştırma 3.11 (Rakamları Aynı Olan Kareler). Üç basamaklı tam kareleri, rakamlarının sıralı demetini anahtar olarak kullanan bir sözlükte gruplayın ve aynı rakamlardan oluşan birden çok kare içeren grupları bulun.

Çözüm.
Kareler. Üç basamaklı kareler $10^2 = 100$ ile $31^2 = 961$ arasındadır; dolayısıyla $10 \le k \le 31$ yeter.
Anahtar. `sorted(str(sq))` rakamların sıralı bir listesini verir. Listeler hashlenebilir olmadığından onu `tuple` ile demete çevirirız. Rakamları aynı olan iki sayı aynı anahtarı alır.
Gruplama. `d.setdefault(k, [])` metodu, `k` anahtarı yoksa onu boş bir listeyle ekler ve her durumda `k`'ya karşılık gelen listeyi döndürür; bu listeye `append` ile ekleme yaparız.

```
groups = {}
for k in range(10, 32):                 # üç basamaklı kareler
    sq = k * k
    key = tuple(sorted(str(sq)))        # rakamların sıralı demeti
    groups.setdefault(key, []).append(sq)
```

 CC BY-NC-SA 4.0

```
print(len(groups))
print(groups[("1", "4", "4")])
print([g for g in groups.values() if len(g) > 1])
```

Çıktı:

```
18
[144, 441]
[[144, 441], [169, 196, 961], [256, 625]]
```

22 karenin rakamlarından 18 farklı anahtar çıkar. Birden çok kare içeren üç grup vardır: $144 = 12^2$ ile $441 = 21^2$; $169 = 13^2$, $196 = 14^2$ ile $961 = 31^2$; $256 = 16^2$ ile $625 = 25^2$. ■

Alıştırma 3.12 (Pascal Üçgenindeki Tek Katsayıların Sayısı). Pascal üçgeninin ilk 2^k satırındaki tek katsayıların sayısını $k = 0, 1, \ldots, 6$ için hesaplayın ve sonuçlardaki örüntüyü bulun.

Çözüm.
Satırlar. Örnek 3.15 içindeki `zip` kuralıyla ilk 64 satırı `rows` listesinde biriktiririz.
Sayım. İlk m satır `rows[:m]` dilimidir. İki `for` içeren bir üreteç ifadesi bu satırlardaki bütün katsayıları dolaşır ve tek olanları sayar.

```
rows = [[1]]
for n in range(1, 64):
    prev = rows[-1]
    rows.append([a + b for a, b in zip([0] + prev, prev + [0])])
for k in range(7):
    m = 2**k
    odd = sum(1 for row in rows[:m] for c in row if c % 2 == 1)
    print(m, odd, 3**k)
```

Çıktı:

```
1 1 1
2 3 3
4 9 9
8 27 27
16 81 81
32 243 243
64 729 729
```

İkinci ve üçüncü sütunlar her satırda eşittir: ilk 2^k satırda tam 3^k tek katsayı vardır. Bu, Örnek 3.16 içinde gördüğümüz kendini tekrarlamanın sayısal karşılığıdır. İlk 2^{k+1} satırdaki tek katsayılar, ilk 2^k satırın deseninin üç kopyasında toplanır, ortadaki ters üçgen ise yalnız çift sayılardan oluşur. Bu yüzden tek katsayıların sayısı her adımda üçe katlanır. ■

Bu bölümde Python'un dört yerleşik veri yapısını ve onları kurmanın en kısa yolu olan comprehension yazımını gördük. Bir sonraki bölüm, Sınıflar, Hata Yakalama ve Dosyalar, kendi veri türlerimizi tanımlamayı, hataları yakalamayı ve verileri dosyalara yazıp okumayı ele alıyor. Sayısal hesapta listelerin yavaş kaldığı yerde ise NumPy Dizileri devreye girecek.

 CC BY-NC-SA 4.0

4. Sınıflar, Hata Yakalama ve Dosyalar

Şimdiye kadar yazdığımız programların hepsi tek bir dosyada yaşadı, girdileri hep düzgündü ve sonuçlarını yalnız ekrana bastı. Gerçek bir hesap bundan fazlasını ister. Daha önce yazılmış fonksiyonları yeniden kullanmak isteriz; sıfıra bölme, yakınsamayan bir iterasyon ya da bozuk bir veri satırı karşısında programın anlamlı bir şey yapmasını isteriz; verileri bir dosyadan okuyup sonuçları başka bir dosyaya yazmak isteriz. Bir de matematiğin kendi nesneleri vardır: vektörler, kesirler, polinomlar, aralıklar. Bunları işlemleriyle birlikte, `u + v` ya da `p * q` yazarak kullanabilmek isteriz.
Bu bölüm bu dört ihtiyaca karşılık gelen dört aracı sırayla tanıtıyor: modüller, istisnalar, dosyalar ve sınıflar. Sonunda hepsini birleştirip bir ölçüm dosyasını okuyan, bozuk satırları ayıklayan ve sonucu yeni bir dosyaya yazan küçük bir program kuracağız. Bir sonraki bölümde başlayacağımız NumPy da bir modüldür ve onun dizileri de bir sınıfın nesneleridir; burada öğrendiklerimiz o kütüphanelerin nasıl çalıştığını anlamanın da anahtarı olacak.

4.1 Modüller ve import

Python'un gücünün büyük kısmı dilin kendisinden değil, hazır yazılmış kod kütüphanelerinden gelir; bu kodlara modüller üzerinden ulaşırız.
Modül kavramını ve `math` modülünü Python ile İlk Adımlar bölümünde tanıdık: bir modül, içinde Python kodu bulunan bir `.py` dosyasıdır ve `import` deyimi onun tanımladığı adları programa açar. Python'la birlikte gelen modüllerin oluşturduğu koleksiyona **standart kütüphane** (standard library) denir; `math`, `fractions` ve `statistics` bunlardandır. Birden çok modülü bir klasörde toplayan yapıya da **paket** (package) denir. NumPy, SciPy ve Matplotlib birer pakettir; ayrıca kurulurlar ama aynı `import` deyimiyle kullanılırlar.
`import` deyiminin dört biçimi vardır; ilk üçü aşağıdaki kodda görülüyor:

```
import math
import fractions as fr
from statistics import mean, median

print(math.lcm(4, 6, 10), math.isqrt(50))
print(fr.Fraction(3, 4) + fr.Fraction(1, 6))
data = [2, 3, 3, 5, 7, 10]
print(mean(data), median(data))
```

Çıktı:

```
60 7
11/12
5 4.0
```

- `import math`: modül `math` adıyla gelir; içindekilere `math.lcm`, `math.isqrt` diye ulaşılır.
- `import fractions as fr`: modül daha kısa bir takma adla gelir. NumPy için her yerde göreceğiniz `import numpy as np` kalıbı budur.
- `from statistics import mean, median`: modülden yalnız istenen adlar doğrudan programa gelir; `statistics.` öneki gerekmez.
- `from math import *`: modüldeki bütün adlar programa dökülür. Bu biçimden kaçınılır; nedenini birazdan bir örnekle göreceğiz.

Çıktıda $\operatorname{lcm}(4,6,10) = 60$, $\lfloor\sqrt{50}\rfloor = 7$ ve $\frac{3}{4} + \frac{1}{6} = \frac{11}{12}$ var. Verinin ortalaması tam sayı çıktığı için `5`, medyanı ise ortadaki iki değerin, 3 ile 5'in ortalaması olduğu için `4.0` olarak basıldı.
Bir modülün içinde ne olduğunu `dir` fonksiyonu listeler; `help(math.log1p)` gibi bir çağrı da bir fonksiyonun belgesini gösterir. Aşağıda `math` modülündeki adlardan `log` ile başlayanları bir list comprehension ile süzüyoruz:

 CC BY-NC-SA 4.0

```
import math

names = [name for name in dir(math) if name.startswith("log")]
print(names)
x = 1e-12
print(math.log(1 + x))
print(math.log1p(x))
```

Çıktı:

```
['log', 'log10', 'log1p', 'log2']
1.000088900581841e-12
9.999999999995e-13
```

`log1p(x)` fonksiyonu $\ln(1+x)$ değerini hesaplar. Neden ayrı bir fonksiyon gerekiyor? $x = 10^{-12}$ iken $1 + x$ toplamı kayan noktada yuvarlanır ve x'in basamaklarının çoğu kaybolur; `math.log(1 + x)` bu yüzden yaklaşık $9 \cdot 10^{-5}$ bağıl hata yapar (bkz. Nümerik Analiz). `log1p` ise toplamı hiç oluşturmadan doğru sonucu verir: $\ln(1+x) = x - x^2/2 + \cdots$ olduğundan doğru değer $10^{-12} - 5 \cdot 10^{-25}$'tir. Modülleri gezmek, bu tür hazır çözümleri bulmanın en kısa yoludur.

⚠ Yıldızlı import ad çakışmasına yol açar

`from modül import *` modüldeki bütün adları programa döker ve aynı adı taşıyan eski bir tanımın üstüne sessizce yazar; bunun ne kadar kafa karıştırıcı olabileceğini görelim. Karmaşık sayılar için matematik modülü olan `cmath` ile `math` modülünün ikisinde de `sqrt` ve `log` vardır:

```
from math import *
from cmath import *

print(sqrt(-4))
print(log(-1))
```

Çıktı:

```
2j
3.141592653589793j
```

Son import kazandığı için `sqrt` artık karmaşık karekök, `log(-1)` da $i\pi$ oldu. İki import satırının sırası değişseydi aynı kod `ValueError` verirdi. Hangi `sqrt`'nin çalıştığını anlamak için importları tek tek okumak gerekir; `math.sqrt` ve `cmath.sqrt` yazmak bu belirsizliği baştan ortadan kaldırır.

Bu derste en sık kullanacağımız standart kütüphane modülleri şunlardır:

Modül	İçerik
`math`	gerçel sayılar için fonksiyonlar ve sabitler
`cmath`	karmaşık sayılar için aynı fonksiyonlar
`fractions`, `decimal`	tam kesirler ve ondalık aritmetik
`statistics`	ortalama, medyan, varyans
`itertools`	permütasyon, kombinasyon, kartezyen çarpım
`csv`, `json`	tablo ve sözlük biçimindeki dosyalar
`pathlib`	dosya yolları
`time`	süre ölçümü

 CC BY-NC-SA 4.0

4.1.1 Kendi modülümüz

Kendi yazdığımız fonksiyonları da bir modüle koyup başka programlardan kullanabiliriz. Aşağıdaki kodu `numtools.py` adıyla kaydedelim. Öklid algoritması ve deneme bölmesi Kontrol Yapıları ve Fonksiyonlar bölümünden tanıdık; burada yalnız bir dosyada toplanıyorlar:

```
"""Sayılar teorisi için küçük yardımcı fonksiyonlar."""


def gcd(a, b):
    """Öklid algoritmasıyla en büyük ortak böleni döndürür."""
    while b != 0:
        a, b = b, a % b
    return abs(a)


def is_prime(n):
    """n asal ise True döndürür (deneme bölmesi)."""
    if n < 2:
        return False
    d = 2
    while d * d <= n:
        if n % d == 0:
            return False
        d += 1
    return True


if __name__ == "__main__":
    # Modül doğrudan çalıştırılınca küçük bir sınama yapılır
    print("gcd(182, 70) =", gcd(182, 70))
    print([p for p in range(30) if is_prime(p)])
```

Dosyanın sonundaki `if __name__ == "__main__":` bloğu yaygın bir kalıptır. Python her modüle `__name__` adında bir değişken verir. Dosya doğrudan çalıştırıldığında (`python numtools.py`) bu değişkenin değeri `"__main__"`, başka bir dosyadan import edildiğinde ise modülün adı, yani `"numtools"` olur. Böylece blok yalnız dosya doğrudan çalıştırıldığında işler ve modülü kendi kendini sınayan bir program gibi kullanmamızı sağlar. Dosya doğrudan çalıştırıldığında sınama bloğu şunu basar.
Çıktı:

```
gcd(182, 70) = 14
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

$\gcd(182, 70) = 14$ sonucunu Sayılar Teorisi notlarındaki elle yapılan hesapla karşılaştırabilirsiniz. Şimdi aynı klasörde ikinci bir dosya açıp modülü kullanalım. Program 80'den küçük ikiz asalları, yani farkı 2 olan asal çiftlerini buluyor:

```
import numtools
from numtools import is_prime

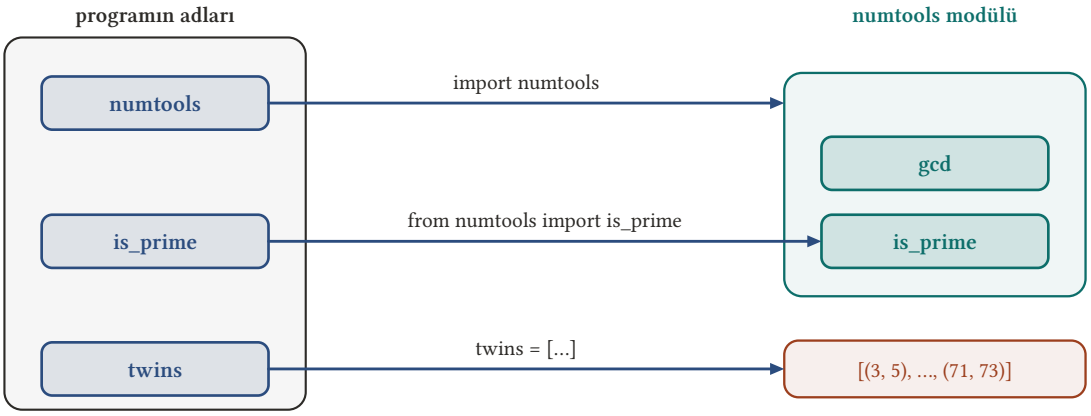
print(numtools.gcd(2**10 * 3**5, 2**4 * 3**8))
print(numtools.__name__, __name__)
twins = [(p, p + 2) for p in range(3, 80)
         if is_prime(p) and is_prime(p + 2)]
print(twins)
```

Çıktı:

```
3888
numtools __main__
[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
```

 CC BY-NC-SA 4.0

$\gcd(2^{10} \cdot 3^5, 2^4 \cdot 3^8) = 2^4 \cdot 3^5 = 3888$ beklendiği gibi çıktı. İkinci satır `__name__` değişkeninin iki dosyadaki değerini gösteriyor; modül import edildiği için `numtools.py`'deki sınama bloğu bu kez çalışmadı. Programın kullandığı adlar ve bağlandıkları nesneler aşağıdaki şekilde özetleniyor:



Şekil 4.1: İkiz asalları bulan programdaki adlar ve bağlandıkları nesneler. `import numtools` programa yalnız `numtools` adını ekler; `gcd` fonksiyonuna `numtools.gcd` diye modülün içinden ulaşılır. `from numtools import is_prime` ise modüldeki fonksiyonu programın kendi adlarına `is_prime` adıyla bağlar.

Bir modül, programın ömrü boyunca yalnız bir kez yüklenir; ikinci bir `import numtools` satırı dosyayı yeniden okumaz. Jupyter'de `numtools.py`'yi düzenledikten sonra değişikliği görmek için ya çekirdeği yeniden başlatın ya da `import importlib` satırından sonra `importlib.reload(numtools)` çağırın.

4.2 Hatalar ve istisnalar

Şimdiye kadar yazdığımız kodlar hep uygun girdilerle çalıştı; şimdi bir işlem yapılamadığında ne olduğuna bakalım. Aşağıdaki döngü 12'yi bir listedeki sayılara sırayla bölüyor:

```
values = [4, 2, 0, 5]
for v in values:
    print(12 / v)
print("bu satıra hiç gelinmez")
```

Çıktı:

```
3.0
6.0
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ZeroDivisionError: division by zero
```

İlk iki bölme yapıldı, üçüncüsü yapılamadı ve program orada durdu: listedeki 5'e de, son satıra da hiç gelinmedi. Python'un bastığı bu metne **izleme kaydı** (traceback) denir ve alttan yukarı okunur. Son satır hatanın türünü (`ZeroDivisionError`) ve açıklamasını (`division by zero`, yani sıfıra bölme) verir; üstündeki satırlar hatanın hangi dosyanın hangi satırında çıktığını gösterir. Python ile İlk Adımlar bölümünde `math.sqrt(-1)` çağrısının da böyle bir kayıt bıraktığını görmüştük. Bu çıktıda kod Python'a standart girdiden verildiği için dosya adı `<stdin>` görünüyor; kodu bir dosyadan çalıştırdığınızda o dosyanın yolu ve hatalı satırın kendisi görünür.

Tanım 4.1 (İstisna). Çalışma sırasında bir işlem gerçekleştirilemediğinde Python bir **istisna** (exception) nesnesi oluşturur ve onu **fırlatır** (raise). Her istisnanın bir **türü** (`ValueError`, `ZeroDivisionError`, ...) ve bir **mesajı** vardır. Fırlatılan istisna program tarafından **yakalanmazsa** program durur ve izleme kaydı basılır.

 CC BY-NC-SA 4.0

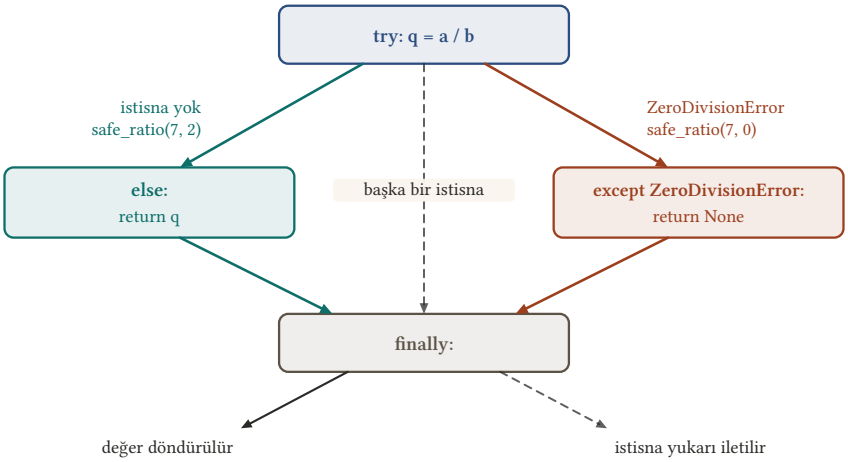
Yani istisna, "bu adımı yapamıyorum" diyen bir sinyaldir. Sinyal, onu bekleyen bir kod parçasına ulaşana kadar fonksiyon çağrılarının zincirinde yukarı doğru ilerler; böyle bir kod yoksa en üste kadar çıkar ve programı durdurur. Bir istisnayı bekleyip yakalamanın yolu `try` deyimidir.

Tanım 4.2 (try Deyimi). Bir istisnayı yakalamak için **`try` deyimi** kullanılır. Genel biçimi şudur:

```
try:
    ...               # istisna fırlatabilecek kod
except ValueError as e:
    ...               # ValueError çıkarsa çalışır; e istisna nesnesidir
except (TypeError, KeyError):
    ...               # bu iki türden biri çıkarsa çalışır
else:
    ...               # try bloğu istisnasız biterse çalışır
finally:
    ...               # her durumda, en son çalışır
```

`except` blokları yukarıdan aşağı sınanır ve istisnanın türüne uyan **ilk** blok çalışır. `else` ve `finally` blokları isteğe bağlıdır; bir `try` deyiminde en az bir `except` ya da bir `finally` bloğu bulunur.

Yani istisna fırlatabilecek kodu `try` bloğuna koyarız ve her olası hata için ne yapılacağını bir `except` bloğuna yazarız. `else` bloğu yalnız her şey yolunda gittiğinde, `finally` bloğu ise ne olursa olsun çalışır. Aşağıdaki şekil, birazdan inceleyeceğimiz `safe_ratio` fonksiyonu üzerinde blokların sırasını gösteriyor:



Şekil 4.2: `safe_ratio` fonksiyonunda blokların çalışma sırası. Önce `try` bloğu çalışır. İstisna çıkmazsa `else`, `ZeroDivisionError` çıkarsa `except` bloğu çalışır; başka türden bir istisna hiçbir `except` ile eşleşmez. Üç yolun hepsi `finally` bloğundan geçer: bu blok `return` deyiminden sonra bile çalışır.

İlk örnek olarak, ondalık ayıracı virgül olan Türkçe yazımları da okuyabilen bir fonksiyon yazalım. `float("3,75")` bir `ValueError` fırlatır; bunu yakalayıp virgülü noktaya çevirerek ikinci bir deneme yapıyoruz:

```
def parse_number(s):
    """'3.75' ya da '3,75' biçimindeki metni float'a çevirir."""
    try:
        return float(s)
    except ValueError:
        return float(s.replace(",", "."))
```

 CC BY-NC-SA 4.0

```
for s in ["2.5", "3,75", "1e-3", " 42 "]:
    print(repr(s), "->", parse_number(s))
```

Çıktı:

```
'2.5' -> 2.5
'3,75' -> 3.75
'1e-3' -> 0.001
' 42 ' -> 42.0
```

`repr(s)` metni tırnaklarıyla birlikte, boşlukları ve satır sonu gibi özel karakterleri görünür biçimde yazar; `' 42 '` metnindeki boşluklar çıktıda bu yüzden görünüyor. `float` baştaki ve sondaki boşlukları kendisi atar. `parse_number("abc")` ise ikinci denemede de `ValueError` fırlatır; `except` bloğunun içinde çıkan bu istisnayı yakalayan başka bir blok olmadığından program durur. Bu doğru davranıştır: bir metni sayıya çeviremiyorsak bunu sessizce geçiştirmek yerine haber vermeliyiz.
Bölüm boyunca karşılaşacağımız istisna türlerini, her birini tetikleyen küçük bir ifadeyle bir arada görelim. Her ifade bir `lambda` içinde duruyor ve döngü onları sırayla bir `try` bloğunda çağırıyor. `type(e).__name__` istisnanın türünün adını verir:

```
import math

tests = [
    lambda: 1 / 0,
    lambda: math.sqrt(-1),
    lambda: int("3,5"),
    lambda: "2" + 3,
    lambda: [1, 2][2],
    lambda: {"a": 1}["b"],
    lambda: math.exp(1000),
    lambda: 10.0 ** 400,
]
for f in tests:
    try:
        f()
    except Exception as e:
        print(f"{type(e).__name__}: {e}")
```

Çıktı:

```
ZeroDivisionError: division by zero
ValueError: math domain error
ValueError: invalid literal for int() with base 10: '3,5'
TypeError: can only concatenate str (not "int") to str
IndexError: list index out of range
KeyError: 'b'
OverflowError: math range error
OverflowError: (34, 'Result too large')
```

`int("3,5")` virgüllü metni tam sayı olarak okuyamadı, `"2" + 3` ise metinle sayıyı toplamaya çalıştı. Son iki satır **taşmadır** (bkz. Nümerik Analiz): $e^{1000} \approx 10^{434}$ ve 10^{400} değerleri en büyük çift duyarlıklı sayıyı, yaklaşık $1{,}8 \cdot 10^{308}$'i aşar. Python'un tam sayılarında böyle bir sınır yoktur: `10 ** 400` hatasız hesaplanır, taşan yalnız `10.0 ** 400` biçimindeki kayan noktalı hesaptır.

> ⚠ Her şeyi yakalamayın
>
> Yukarıdaki döngü gösteri amacıyla `except Exception` kullandı; bu satır neredeyse bütün hataları yakalar. Gerçek kodda yalnız beklediğiniz ve ne yapacağınızı bildiğiniz türü yakalayın: geniş bir `except`, yanlış yazılmış bir değişken adının yol açtığı `NameError`'ı da yutar ve hatanın gerçek yerini gizler. Hiçbir tür yazılmayan çıplak `except:` daha da kötüdür: programı Ctrl+C ile durdurma isteğini (`KeyboardInterrupt`) ve programdan çıkma isteğini (`SystemExit`) bile yutar.

 CC BY-NC-SA 4.0

Örnek 4.1 (Blokların Çalışma Sırası). Aşağıdaki `safe_ratio` fonksiyonu `safe_ratio(7, 2)` ve `safe_ratio(7, 0)` biçiminde çağrıldığında hangi bloklar hangi sırayla çalışır?

```
def safe_ratio(a, b):
    try:
        q = a / b
    except ZeroDivisionError:
        print("  except: sıfıra bölme")
        return None
    else:
        print("  else: bölme başarılı")
        return q
    finally:
        print("  finally: her durumda")


print(safe_ratio(7, 2))
print(safe_ratio(7, 0))
```

Çözüm.
Kod çalıştırıldığında çıktı şöyledir.
Çıktı:

```
  else: bölme başarılı
  finally: her durumda
3.5
  except: sıfıra bölme
  finally: her durumda
None
```

İlk çağrıda bölme başarılıdır: `try` bloğu istisnasız biter, `else` bloğu çalışır ve `return q` deyimine gelinir. Fonksiyon değeri döndürmeden hemen önce `finally` bloğu çalışır; `3.5` değeri bu yüzden `finally` mesajından **sonra** basıldı. İkinci çağrıda `7 / 0` bir `ZeroDivisionError` fırlatır, `try` bloğunun geri kalanı atlanır, `except` bloğu çalışır ve `None` döndürülmeden önce yine `finally` çalışır.
Yani `finally`, `try`, `except` ve `else` bloklarından hangisi `return` ile biterse bitsin en son çalışan bloktur. Açık bir dosyayı kapatmak gibi ne olursa olsun yapılması gereken işler buraya yazılır. ■

4.2.1 İstisna türlerinin soyağacı

İstisna türleri de birbirleriyle akrabadır ve bir `except` satırının hangi istisnaları yakaladığı bu akrabalığa bağlıdır. Her türün `__mro__` niteliği, o türü ve bütün üst türlerini sırayla listeler:

```
errors = [ZeroDivisionError, OverflowError, IndexError, KeyError,
          ValueError, TypeError]
for cls in errors:
    print(" -> ".join(c.__name__ for c in cls.__mro__))
```

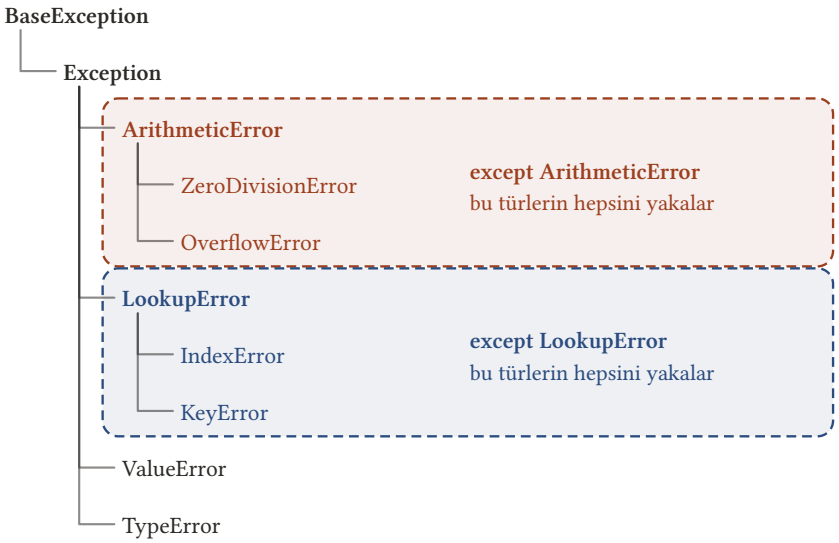
Çıktı:

```
ZeroDivisionError -> ArithmeticError -> Exception -> BaseException -> object
OverflowError -> ArithmeticError -> Exception -> BaseException -> object
IndexError -> LookupError -> Exception -> BaseException -> object
KeyError -> LookupError -> Exception -> BaseException -> object
ValueError -> Exception -> BaseException -> object
TypeError -> Exception -> BaseException -> object
```

Her satır bir türden başlayıp onun üst türlerini sıralıyor. `ZeroDivisionError` ile `OverflowError`'ın ortak üst türü `ArithmeticError`, `IndexError` ile `KeyError`'ınki `LookupError`'dır; hepsi `Exception`'dan, o da

 CC BY-NC-SA 4.0

`BaseException`'dan türer. En sondaki `object`, Python'daki her sınıfın ortak atasıdır. Bir `except T` satırı `T` türündeki istisnaları **ve** `T`'nin bütün alt türlerini yakalar:



Şekil 4.3: Bölümde karşılaştığımız istisna türlerinin soyağacı; her türün üst türü `__mro__` zincirinden okunmuştur. Bir `except` satırı, yazılan türü ve onun altındaki bütün türleri yakalar: `except ArithmeticError` hem sıfıra bölmeyi hem taşmayı, `except LookupError` hem liste indisi hem sözlük anahtarı hatasını yakalar.

Örnek 4.2 (Bir Üst Türü Yakalamak). Bir fonksiyonu verilen bir noktada hesaplayan, sıfıra bölme ya da taşma olursa programı durdurmak yerine hatanın türünü bildiren bir metin döndüren `try_compute(f, x)` fonksiyonunu, iki hata türünü tek bir `except` satırıyla yakalayarak yazınız.

Çözüm.
İki tür de `ArithmeticError`'ın alt türü olduğundan bu üst türü yakalamak yeterlidir:

```
import math


def try_compute(f, x):
    try:
        return f(x)
    except ArithmeticError as e:
        return f"{type(e).__name__} yakalandı"


print(try_compute(lambda x: 1 / x, 0))
print(try_compute(math.exp, 1000))
print(try_compute(math.exp, 1))
```

Çıktı:

```
ZeroDivisionError yakalandı
OverflowError yakalandı
2.718281828459045
```

`except ArithmeticError` hem $1/0$'daki `ZeroDivisionError`'ı hem e^{1000}'deki `OverflowError`'ı yakaladı; üçüncü çağrıda istisna çıkmadı ve e döndü. Aynı fonksiyon `math.sqrt` ile -1'de çağrılsaydı `ValueError` yakalanmaz ve programı durdururdu, çünkü `ValueError`, `ArithmeticError`'ın alt türü

 CC BY-NC-SA 4.0

değildir. Akraba olmayan türleri birlikte yakalamak için bir demet yazılır: `except (ValueError, ArithmeticError)`. ■

4.2.2 İstisna fırlatmak

Kendi fonksiyonlarımız da anlamsız bir girdi aldıklarında istisna fırlatabilir; bu, sessizce yanlış bir sonuç döndürmekten her zaman iyidir.

Tanım 4.3 (raise Deyimi). `raise T(mesaj)` deyimi `T` türünden, verilen mesajı taşıyan bir istisna oluşturup fırlatır. Bir `except` bloğunun içinde tek başına yazılan `raise` ise o an yakalanmış olan istisnayı olduğu gibi yeniden fırlatır.

Yani `raise`, Python'un kendi fonksiyonlarının hata bildirdiği mekanizmanın aynısını bizim kodumuza açar. Tür seçerken yerleşik türlerin anlamına uyulur: değer uygunsuzsa `ValueError`, tür uygunsuzsa `TypeError`, bir iterasyon sonuç vermiyorsa `RuntimeError`.

Örnek 4.3 (Girdisini Denetleyen İkiye Bölme). İkiye bölme yönteminin (bkz. Nümerik Analiz) bir kök bulacağı, f sürekli, $a < b$ ve $f(a)f(b) \le 0$ iken güvencededir. Süreklilik kodla denetlenemez; son iki koşul sağlanmadığında `ValueError` fırlatan bir `bisect(f, a, b)` fonksiyonu yazıp $f(x) = x^3 - 2x - 5$ fonksiyonuyla deneyiniz.

Çözüm.
Koşulları fonksiyonun başında denetleyip ihlal edildiklerinde `raise` ile çıkıyoruz; geri kalanı, her adımda kökü içeren aralığı yarılayan bilinen döngüdür:

```
def bisect(f, a, b, tol=1e-12):
    """f'nin [a, b] aralığındaki bir kökünü ikiye bölmeyle bulur."""
    if not a < b:
        raise ValueError(f"a < b olmalı: a={a}, b={b}")
    fa = f(a)
    if fa * f(b) > 0:
        raise ValueError("f(a) ile f(b) aynı işaretli")
    while b - a > tol:
        m = (a + b) / 2
        fm = f(m)
        if fa * fm <= 0:
            b = m
        else:
            a, fa = m, fm
    return (a + b) / 2


def f(x):
    return x**3 - 2 * x - 5


print(bisect(f, 2, 3))
for a, b in [(3, 4), (3, 2)]:
    try:
        bisect(f, a, b)
    except ValueError as e:
        print("Hata:", e)
```

Çıktı:

```
2.0945514815425668
Hata: f(a) ile f(b) aynı işaretli
Hata: a < b olmalı: a=3, b=2
```

 CC BY-NC-SA 4.0

$[2,3]$ aralığında $f(2) = -1 < 0 < 16 = f(3)$ olduğundan kök bulundu: $x \approx 2{,}0945514815$. $[3,4]$ aralığında $f(3) = 16$ ile $f(4) = 51$ aynı işaretli olduğu, $(3,2)$ çiftinde ise $a < b$ sağlanmadığı için fonksiyon hesaba hiç girmeden istisna fırlattı; çağıran kod da onu yakalayıp mesajı bastı.
Denetim olmasaydı $[3,4]$ için döngü her adımda sol ucu orta noktaya taşır ve sessizce 4'e çok yakın bir sayı döndürürdü. Oysa $f(4) = 51$ olduğundan bu sayı bir kök değildir. ■

4.2.3 Doğrulama: assert

Bir fonksiyonun girdisini değil, kendi hesabının doğru gittiğini denetlemek için `assert` deyimi kullanılır.

Tanım 4.4 (assert Deyimi). `assert koşul, mesaj` deyimi koşulu hesaplar. Koşul doğruysa hiçbir şey yapmaz; yanlışsa verilen mesajla bir `AssertionError` fırlatır. Mesaj isteğe bağlıdır.

Yani `assert`, koda yazılmış bir "burada şu kesinlikle doğru olmalı" notudur; not yanlış çıkarsa program yüksek sesle durur. Bir hesabın, doğru olduğunu bildiğimiz bir özelliğini sınamak için idealdir. Örneğin $\sqrt{2}$ için Newton iterasyonundan (bkz. Kontrol Yapıları ve Fonksiyonlar) yalnız üç adım atarsak sonuç henüz yeterince doğru değildir ve `assert` bunu yakalar:

```
def newton_sqrt(a, steps):
    x = a
    for _ in range(steps):
        x = (x + a / x) / 2
    return x


r = newton_sqrt(2, 3)
print(r)
assert abs(r * r - 2) < 1e-12, f"yakınsama yetersiz: {r * r - 2:.1e}"
print("buraya gelinmez")
```

Çıktı:

```
1.4142156862745097
Traceback (most recent call last):
  File "<stdin>", line 10, in <module>
AssertionError: yakınsama yetersiz: 6.0e-06
```

Üç adımın sonunda $r^2 - 2 \approx 6 \cdot 10^{-6}$ kaldı; koşul sağlanmadığı için program son satıra gelmeden durdu.

Örnek 4.4 (Bölme Algoritmasını assert ile Güvenceye Almak). Bölme Algoritması'na göre (bkz. Sayılar Teorisi) her a ve $b \neq 0$ tam sayısı için $a = bq + r$ ve $0 \leq r < |\,b\,|$ koşullarını sağlayan tek bir (q, r) çifti vardır. Python'un `divmod(a, b)` fonksiyonu negatif bölende farklı bir kalan kuralı izler. Bu çifti her işaret için doğru veren ve sonucunu `assert` ile denetleyen bir `division(a, b)` fonksiyonu yazınız.

Çözüm.
Python ile İlk Adımlar bölümünde gördüğümüz gibi `a // b` bölümü aşağı yuvarlanır, yani $\lfloor a/b \rfloor$'dir, ve `a % b` kalanı **bölenin işaretini** taşır. $b > 0$ iken bu tam istediğimiz $0 \leq r < b$ koşuludur. $b < 0$ iken ise $b < r \leq 0$ olur; $r = 0$ ise düzeltme gerekmez. $b < 0$ ve $r < 0$ iken bölümü 1 artırıp kalandan b çıkarırsak

$$a = bq + r = b(q+1) + (r - b)$$

eşitliği korunur ve yeni kalan $0 < r - b < |\,b\,|$ aralığına girer:

```
def division(a, b):
    """a = b*q + r ve 0 <= r < |b| olacak biçimde (q, r) döndürür."""
    if b == 0:
        raise ZeroDivisionError("b sıfır olamaz")
    q, r = divmod(a, b)
    if r < 0:              # b < 0 iken Python'un kalanı r <= 0 olur
```

 CC BY-NC-SA 4.0

```
        q, r = q + 1, r - b
    assert a == b * q + r and 0 <= r < abs(b), (a, b, q, r)
    return q, r


for a, b in [(17, 5), (-17, 5), (17, -5), (-17, -5)]:
    print((a, b), "divmod:", divmod(a, b), "division:", division(a, b))
```

Çıktı:

```
(17, 5) divmod: (3, 2) division: (3, 2)
(-17, 5) divmod: (-4, 3) division: (-4, 3)
(17, -5) divmod: (-4, -3) division: (-3, 2)
(-17, -5) divmod: (3, -2) division: (4, 3)
```

$b = 5$ iken iki fonksiyon aynı sonucu veriyor. $b = -5$ iken `divmod` kalanları -3 ve -2 olarak, yani negatif veriyor; `division` ise $17 = (-5)(-3) + 2$ ve $-17 = (-5) \cdot 4 + 3$ ayrışımlarını buluyor. `assert` satırı her çağrıda hem eşitliği hem kalanın aralığını denetledi; dört durumda da koşul sağlandığı için sessiz kaldı. Mesaj olarak `(a, b, q, r)` demetini verdik: koşul bir gün bozulursa hangi girdide bozulduğunu hemen görürüz. ■

> ⚠ assert girdi denetimi için değildir
>
> Python `-O` seçeneğiyle (`python -O program.py`) çalıştırıldığında bütün `assert` satırları atlanır. Bu yüzden kullanıcının verdiği girdiyi `assert` ile değil, `if` ve `raise` ile denetleyin. `bisect`'teki işaret denetimi bu yüzden bir `raise`'dir, `division`'daki son denetim ise bir `assert`. Kural basittir: hatalı girdi `raise` ile bildirilir, kodun kendi yanlışı `assert` ile yakalanır.

4.3 Dosyalarla çalışmak

Hesapların girdisi çoğu zaman bir dosyadan gelir; sonuçları da kaybolmasınlar diye bir dosyaya yazarız.

Tanım 4.5 (Metin Dosyası). Karakterlerden oluşan ve satır sonu karakteriyle (`"\n"`) satırlara ayrılan dosyaya **metin dosyası** denir. Diskte karakterler bayt olarak saklanır; karakterden bayta çeviri kuralına **kodlama** (encoding) denir. Türkçe karakterleri de kapsayan en yaygın kodlama UTF-8'dir.

Yani bir metin dosyası, `"\n"` karakterleriyle bölünmüş uzun bir karakter dizisidir; sayılar bile dosyada rakam rakam, metin olarak durur. Dosyayı okurken bu metni yeniden sayıya çevirmek bizim işimizdir. Python'da bir dosya `open(yol, kip, encoding="utf-8")` ile açılır. Kip (mode), dosyayla ne yapacağımızı söyler:

Kip	Anlamı
`"r"`	okumak için aç (varsayılan); dosya yoksa `FileNotFoundError`
`"w"`	yazmak için aç; dosya varsa içeriği silinir
`"a"`	sonuna eklemek için aç
`"x"`	yeni dosya oluştur; dosya varsa `FileExistsError`

`open` bir dosya nesnesi döndürür ve işimiz bitince dosyanın kapatılması gerekir. Bunu `with` bloğu kendiliğinden yapar: `with open(...) as f:` satırının altındaki blok bittiğinde, blokta bir istisna çıksa bile dosya kapatılır. Bu, bir `try` ve `finally` çifti yazmanın kısa ve güvenli yoludur.

 CC BY-NC-SA 4.0

Üç adımda dosyaya yazmak ve dosyadan okumak

1. `with open(yol, kip, encoding="utf-8") as f:` ile dosyayı açın; yazmak için `"w"`, okumak için `"r"` kipini kullanın.
2. Yazarken `f.write(metin)` kullanın ve satır sonlarını (`"\n"`) kendiniz ekleyin; okurken `for line in f:` ile satır satır dolaşın.
3. Okuduğunuz her satırı `int`, `float` ya da `split` ile ihtiyacınız olan biçime çevirin.

Örnek 4.5 (Fibonacci Sayılarını Dosyaya Yazıp Okumak). İlk 30 Fibonacci sayısını, her satıra bir sayı gelecek biçimde `fibonacci.txt` dosyasına yazınız. Sonra dosyayı okuyup son iki sayının oranını altın oran $\varphi = \frac{1+\sqrt{5}}{2}$ ile karşılaştırınız.

Çözüm.
Yazmak. Listeyi kurup her sayıyı bir f-string ile, satır sonuyla birlikte yazıyoruz:

```
F = [1, 1]
while len(F) < 30:
    F.append(F[-1] + F[-2])

with open("fibonacci.txt", "w", encoding="utf-8") as f:
    for n in F:
        f.write(f"{n}\n")
print("yazılan satır sayısı:", len(F))
```

Çıktı:

```
yazılan satır sayısı: 30
```

Okumak. Dosya nesnesi üzerinde kurulan bir `for` döngüsü dosyayı satır satır dolaşır; `readline()` ise tek bir satır okur. Dosyanın nasıl göründüğünü incelemek için ilk satırı bu yolla ayrı okuyalım. Bu kod, önceki kodun yazdığı dosyayı okur:

```
with open("fibonacci.txt", encoding="utf-8") as f:
    first = f.readline()
    rest = [int(line) for line in f]
print(repr(first))
print(len(rest), rest[:6], rest[-1])
phi = (1 + 5**0.5) / 2
print(rest[-1] / rest[-2], phi)
```

Çıktı:

```
'1\n'
29 [1, 2, 3, 5, 8, 13] 832040
1.6180339887482036 1.618033988749895
```

`repr` ilk satırın sonunda `"\n"` karakterinin durduğunu gösteriyor. `int` baştaki ve sondaki boşlukları ve satır sonunu kendisi attığı için `int(line)` doğrudan çalıştı. Okuma `readline`'ın kaldığı yerden devam ettiği için listede 29 sayı var. $F_{30}/F_{29} = 832040/514229$ oranı φ ile virgülden sonra 11 basamak örtüşüyor. ■

Okunacak dosya yoksa `open` bir `FileNotFoundError` fırlatır. Bu, yakalamanın anlamlı olduğu tipik bir durumdur:

 CC BY-NC-SA 4.0

```
from pathlib import Path

path = Path("deney_verisi.txt")
try:
    with open(path, encoding="utf-8") as f:
        text = f.read()
except FileNotFoundError as e:
    print("Dosya bulunamadı:", e.filename)
    text = ""
print(path.exists(), len(text))
```

Çıktı:

```
Dosya bulunamadı: deney_verisi.txt
False 0
```

`pathlib` modülündeki `Path` nesnesi bir dosya yolunu temsil eder. `exists()` metodu dosyanın var olup olmadığını söyler; `read_text` ve `write_text` metotları da küçük dosyaları tek satırda okuyup yazar.

⚠ Kodlamayı her zaman yazın

`encoding` verilmezse Python işletim sisteminin varsayılan kodlamasını kullanır; Türkçe Windows'ta bu genellikle UTF-8 değil, `cp1254`'tür. UTF-8 ile yazılmış bir dosya başka bir kodlamayla okunursa Türkçe karakterler bozulur:

```
text = "ağırlık, sıcaklık, ölçüm"
with open("kodlama.txt", "w", encoding="utf-8") as f:
    f.write(text)

with open("kodlama.txt", encoding="cp1254") as f:
    print(f.read())
with open("kodlama.txt", encoding="utf-8") as f:
    print(f.read())
```

Çıktı:

```
aÄŸÄ±rlÄ±k, sÄ±caklÄ±k, Ã¶lÃ§Ã¼m
ağırlık, sıcaklık, ölçüm
```

Diskteki baytlar iki okumada da aynıdır; değişen yalnız onları karakterlere çeviren kuraldır. Her `open` çağrısına `encoding="utf-8"` yazmak bu sorunu baştan önler.

4.3.1 CSV dosyaları

Sayısal veriler çoğu zaman tablo biçiminde saklanır; bunun en yaygın düz metin biçimi CSV'dir.

Tanım 4.6 (CSV Dosyası). Her satırı bir kayıt olan ve bir satırdaki alanların belirli bir ayraçla, çoğunlukla virgülle ayrıldığı metin dosyasına **CSV dosyası** (comma-separated values) denir. İlk satır genellikle sütun adlarını taşıyan **başlık satırıdır**.

Yani CSV, bir tablonun düz metin hâlidir: satırlar tablonun satırları, ayraçlar da sütun çizgileridir. Satırı `split(",")` ile bölmek basit durumlarda işe yarar ama bir alanın içinde virgül geçtiğinde (`"Ankara, Türkiye"`) bozulur. Standart kütüphanedeki `csv` modülü tırnak içindeki alanlar gibi ayrıntıları bizim yerimize halleder.
Aşağıdaki kod $[0, \pi/2]$ aralığında $\pi/12$ adımla sin ve cos değerlerinin tablosunu `trig.csv` dosyasına yazıyor ve dosyanın içeriğini olduğu gibi basıyor. `csv` modülüyle kullanılan dosyalar `newline=""` ile açılır; satır sonlarını modülün kendisi yönetir:

```
import csv
import math
```

```
with open("trig.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["x", "sin", "cos"])
    for k in range(7):
        x = k * math.pi / 12
        writer.writerow([f"{x:.6f}", f"{math.sin(x):.6f}",
                         f"{math.cos(x):.6f}"])

with open("trig.csv", encoding="utf-8") as f:
    print(f.read(), end="")
```

Çıktı:

```
x,sin,cos
0.000000,0.000000,1.000000
0.261799,0.258819,0.965926
0.523599,0.500000,0.866025
0.785398,0.707107,0.707107
1.047198,0.866025,0.500000
1.308997,0.965926,0.258819
1.570796,1.000000,0.000000
```

Dosyayı geri okumanın en rahat yolu `csv.DictReader`'dır: başlık satırını okur ve sonraki her satırı, anahtarları sütun adları olan bir sözlüğe çevirir. Aşağıdaki kod önceki kodun yazdığı dosyayı okuyup her satırda $\sin^2 x + \cos^2 x = 1$ özdeşliğinden sapmayı ölçüyor:

```
import csv

with open("trig.csv", newline="", encoding="utf-8") as f:
    rows = list(csv.DictReader(f))
print(len(rows), rows[2])
err = max(abs(float(r["sin"])**2 + float(r["cos"])**2 - 1)
          for r in rows)
print(f"en büyük sapma: {err:.1e}")
```

Çıktı:

```
7 {'x': '0.523599', 'sin': '0.500000', 'cos': '0.866025'}
en büyük sapma: 7.0e-07
```

Alanların hâlâ metin (`'0.500000'`) olduğuna dikkat edin; `float` ile sayıya çevirmek bize kalıyor. Sapma, değerleri altı ondalık basamağa yuvarlamamızdan geliyor: her değerdeki yuvarlama hatası en çok $5 \cdot 10^{-7}$ olduğundan özdeşlikten sapma da 10^{-6} mertebesinde kalıyor.

Örnek 4.6 (Noktalı Virgül ve Ondalık Virgül). Türkçe ayarlı bir tablolama programı CSV dosyalarını ondalık virgülle yazar ve alanları virgül yerine noktalı virgülle ayırır. İçeriği

```
x;f(x)
0,5;0,4794
1,0;0,8415
1,5;0,9975
```

olan `tr.csv` dosyasını okuyup $(x, f(x))$ sayı çiftlerinin listesini oluşturunuz.

> **Çözüm.**
> Kod önce dosyayı verilen içerikle oluşturuyor. Sonra `csv.reader`'a `delimiter=";"` vererek alanları noktalı virgülden böldürüyoruz. Başlık satırını, okuyucudan bir sonraki satırı çeken `next(reader)` ile ayrı alıyoruz. Kalan her satırda iki alanın virgülünü noktaya çevirip `float` uyguluyoruz:

 CC BY-NC-SA 4.0

```python
import csv

text = "x;f(x)\n0,5;0,4794\n1,0;0,8415\n1,5;0,9975\n"
with open("tr.csv", "w", encoding="utf-8") as f:
    f.write(text)

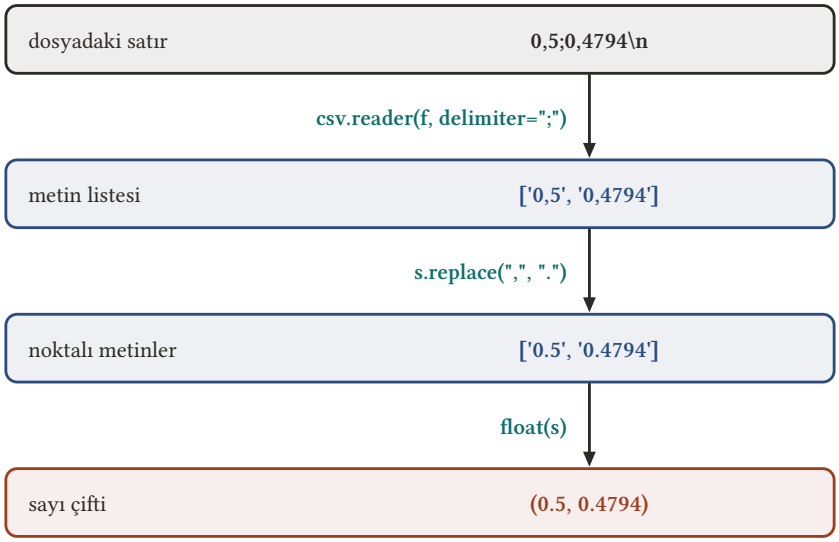
points = []
with open("tr.csv", newline="", encoding="utf-8") as f:
    reader = csv.reader(f, delimiter=";")
    header = next(reader)
    for row in reader:
        x, y = (float(s.replace(",", ".")) for s in row)
        points.append((x, y))
print(header)
print(points)
```

Çıktı:

```
['x', 'f(x)']
[(0.5, 0.4794), (1.0, 0.8415), (1.5, 0.9975)]
```

Değerler, sin x'in dört basamağa yuvarlanmış değerleridir. Ayraç olarak yanlışlıkla varsayılan virgül kullanılsaydı `0,5;0,4794` satırı `['0', '5;0', '4794']` biçiminde üç anlamsız alana bölünürdü. ■

Bir satırın dosyadan sayı çiftine dönüşmesi, ara adımlarıyla birlikte aşağıda görülüyor:



Şekil 4.4: `tr.csv` dosyasının ikinci satırının yolculuğu. `csv.reader` satırı noktalı virgülden böler ve satır sonunu atar; alanlar hâlâ metindir. Ondalık virgül noktaya çevrildikten sonra `float` her alanı sayıya dönüştürür.

4.3.2 Sözlükleri saklamak: JSON

Tablo olmayan verileri, örneğin bir hesabın ayarlarını ya da iç içe sonuçları saklamak için JSON biçimi daha uygundur. `json` modülü sözlükleri, listeleri, sayıları ve metinleri insanın da okuyabildiği bir metne çevirip dosyaya yazar (`json.dump`) ve geri okur (`json.load`):

 CC BY-NC-SA 4.0

```
import json

params = {"method": "simpson", "a": 0, "b": 3.14159, "n": [10, 20, 40]}
with open("ayarlar.json", "w", encoding="utf-8") as f:
    json.dump(params, f, indent=2)

with open("ayarlar.json", encoding="utf-8") as f:
    loaded = json.load(f)
print(loaded == params, loaded["n"][-1])
```

Çıktı:

```
True 40
```

Çıktının ilk değeri, okunan sözlüğün yazılanla aynı olduğunu söylüyor. JSON'un demet, küme ya da `Fraction` gibi türleri yoktur; bir demet yazılırsa listeye dönüşerek geri gelir.

4.4 Sınıflar

Python'daki her değer bir nesnedir ve her nesnenin bir sınıfı vardır; şimdi bu kavramı kendi matematik nesnelerimizi tanımlamak için kullanacağız.

```
from fractions import Fraction

q = Fraction(6, 8)
print(type(7), type(0.75), type(q))
print(q, q.numerator, q.denominator)
print((0.75).as_integer_ratio(), isinstance(q, Fraction))
```

Çıktı:

```
<class 'int'> <class 'float'> <class 'fractions.Fraction'>
3/4 3 4
(3, 4) True
```

`type` bir nesnenin sınıfını söyler. `Fraction(6, 8)` nesnesi kendi verisini, yani pay ve paydasını, `numerator` ve `denominator` **niteliklerinde** (attribute) taşır. `as_integer_ratio` gibi, nesneye nokta ile bağlanan fonksiyonlar ise **metotlardır**. `isinstance(q, Fraction)` bir nesnenin verilen sınıftan olup olmadığını sınar.

Tanım 4.7 (Sınıf ve Nesne). Bir **sınıf** (class), nesnelerinin hangi verileri (**nitelikler**) ve hangi işlemleri (**metotlar**) taşıyacağını tanımlayan bir kalıptır. Bu kalıptan üretilen her değere o sınıfın bir **nesnesi** (object, instance) denir. Sınıf `class Ad:` bloğuyla tanımlanır; `Ad(...)` çağrısı yeni bir nesne üretir ve onu ilk değerleriyle donatmak için sınıfın `__init__` metodunu çalıştırır.

Yani sınıf bir türün tarifidir, nesne ise o tariften yapılmış somut bir örnek. Matematikte "düzlemde bir vektör" demek bir sınıf tanımlamaya, $u = (3, 4)$ demek de o sınıftan bir nesne üretmeye benzer. İlk sınıfımız da düzlemde bir vektör olsun:

```
import math


class Vec2:
    """Düzlemde bir vektör."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        """Öklid normu."""
```

 CC BY-NC-SA 4.0

```
        return math.hypot(self.x, self.y)

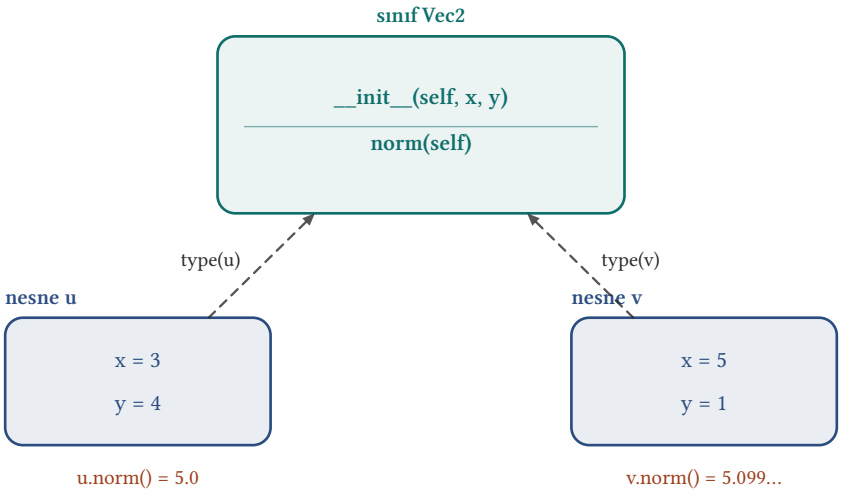

u = Vec2(3, 4)
v = Vec2(1, 1)
v.x = 5
print(u.x, u.y, u.norm())
print(v.x, v.y, v.norm(), Vec2.norm(v))
```

Çıktı:

```
3 4 5.0
5 1 5.0990195135927845 5.0990195135927845
```

Kodu satır satır okuyalım:

- `class Vec2:` sınıfı başlatır. Sınıf adları geleneksel olarak büyük harfle başlar; altındaki üç tırnaklı metin sınıfın belgesidir.
- Metotlar, sınıfın içinde `def` ile tanımlanan fonksiyonlardır. İlk parametreleri **üzerinde çalışılan nesnedir** ve geleneksel adı `self`'tir.
- `Vec2(3, 4)` çağrısı yeni bir nesne oluşturur ve onun için `__init__(self, 3, 4)` metodunu çalıştırır; `self.x = x` satırı nesneye `x` niteliğini ekler.
- `v.norm()` yazdığımızda Python bunu `Vec2.norm(v)` olarak çalıştırır, yani `self` yerine `v` konur. Çıktının son iki değeri bu iki yazımın aynı olduğunu gösteriyor.
- Nitelikler sonradan değiştirilebilir: `v.x = 5` yalnız `v` nesnesini değiştirdi, `u`'ya dokunmadı.



Şekil 4.5: Bir sınıf ve ondan üretilen iki nesne. Metotlar (`__init__`, `norm`) sınıfta bir kez tanımlıdır; her nesne kendi `x`, `y` niteliklerini taşır. `u.norm()` çağrısı `Vec2.norm(u)` demektir: metot, `self` yerine `u` konarak çalışır.

Bu sınıfın bir nesnesini `print` ile basmayı denerseniz `<__main__.Vec2 object at 0x...>` gibi, sonu bellekteki adrese göre değişen anlamsız bir metin görürsünüz; `u + v` yazarsanız da `TypeError` alırsınız. Python iki `Vec2` nesnesinin nasıl yazılacağını ve toplanacağını henüz bilmiyor. Bunu ona özel metotlarla öğretiriz.

Tanım 4.8 (Özel Metot). Adı iki alt çizgiyle başlayıp iki alt çizgiyle biten metotlara **özel metot** (special method) denir. Python bu metotları dilin kendi sözdizimi için çağırır: `a + b` ifadesi `a.__add__(b)` çağrısına, `a == b` ifadesi `a.__eq__(b)` çağrısına, `repr(a)` de `a.__repr__()` çağrısına dönüşür.

Yani bir sınıfa özel metotlar ekleyerek onun nesnelerine `+`, `*`, `==`, `abs` gibi tanıdık işlemleri öğretebiliriz. Sık kullanılanlar şunlardır:

Yazım	Çağrılan metot
`a + b`, `a - b`	`a.__add__(b)`, `a.__sub__(b)`
`a * b`	`a.__mul__(b)`; o işe yaramazsa ve `b` başka türden-se `b.__rmul__(a)`
`-a`, `abs(a)`	`a.__neg__()`, `a.__abs__()`
`a == b`	`a.__eq__(b)`
`repr(a)`, `print(a)`	`a.__repr__()`
`a(x)`	`a.__call__(x)`
`x in a`	`a.__contains__(x)`

`print` aslında `__str__` metodunu arar; sınıfta yalnız `__repr__` varsa onu kullanır. `Vec2`'yi bu metotlarla yeniden yazalım:

```
import math


class Vec2:
    """Düzlemde, işlemleriyle birlikte bir vektör."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Vec2({self.x}, {self.y})"

    def __add__(self, other):
        return Vec2(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vec2(self.x - other.x, self.y - other.y)

    def __neg__(self):
        return Vec2(-self.x, -self.y)

    def __mul__(self, k):
        """Skalerle çarpım: v * k."""
        if not isinstance(k, (int, float)):
            return NotImplemented
        return Vec2(k * self.x, k * self.y)

    __rmul__ = __mul__          # k * v de aynı sonucu versin

    def __eq__(self, other):
        if not isinstance(other, Vec2):
            return NotImplemented
        return self.x == other.x and self.y == other.y

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def dot(self, other):
        """İç çarpım."""
        return self.x * other.x + self.y * other.y
```

Birkaç ayrıntı:

 CC BY-NC-SA 4.0

- `__repr__` bir metin döndürür; Python'un konsolu ve `print` nesneyi bu metinle gösterir. Metni, nesneyi yeniden üreten koda benzetmek (`Vec2(4, 3)`) iyi bir alışkanlıktır.
- İşlem metotları **yeni** bir `Vec2` döndürür; `u + v` işlemi `u`'yu değiştirmez.
- `__mul__` yalnız sayılarla çarpımı tanımlar ve çarpan sayı değilse `NotImplemented` döndürür. Bu özel değer Python'a "bu işlemi bilmiyorum" der: öbür terim farklı türdense Python o zaman onun metodunu dener, o da bilmiyorsa `TypeError` fırlatır.
- `2 * u` yazıldığında önce `int` sınıfının `__mul__` metodu denenir. O bir `Vec2` ile çarpmayı bilmediği için Python `u.__rmul__(2)` çağrısına geçer. Skalerle çarpma değişmeli olduğundan `__rmul__ = __mul__` satırı aynı metodu ikinci bir adla bağlıyor.

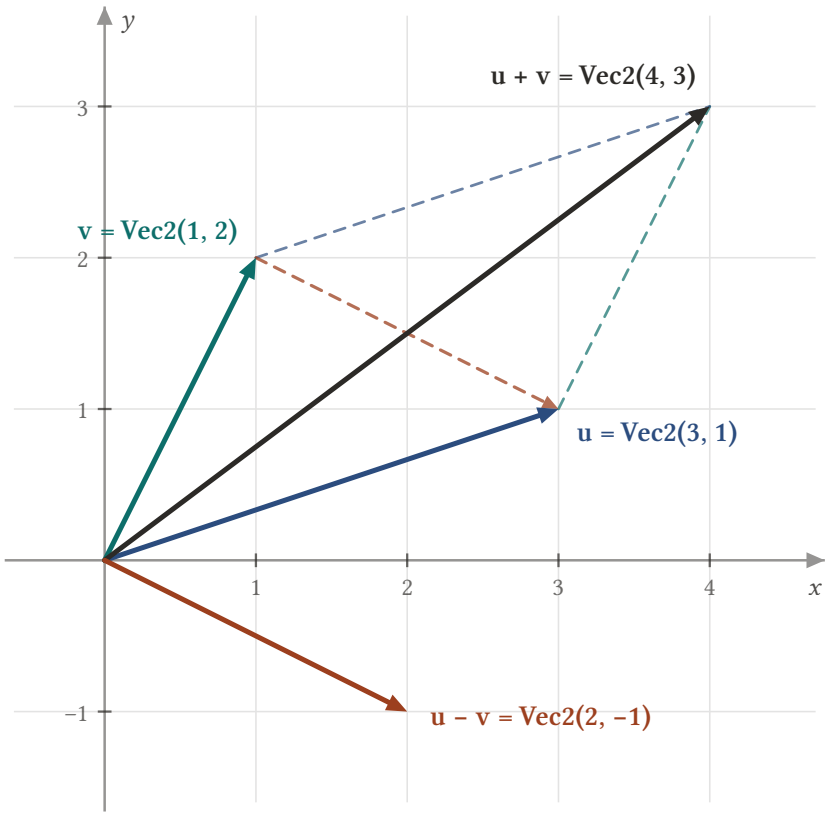
Sınıfı kullanalım. Bu kod önceki kodun devamıdır:

```
u, v = Vec2(3, 1), Vec2(1, 2)
print(u + v, u - v, -v)
print(2 * u, u * 0.5)
print(abs(Vec2(3, 4)), u.dot(v))
print(u + v == v + u, u == Vec2(3.0, 1.0))
```

Çıktı:

```
Vec2(4, 3) Vec2(2, -1) Vec2(-1, -2)
Vec2(6, 2) Vec2(1.5, 0.5)
5.0 5
True True
```

`u + v == v + u` toplamanın değişmeliliğini sınadı; `Vec2(3.0, 1.0)` ise `3 == 3.0` olduğundan `u`'ya eşit çıktı. `Vec2`'nin toplama ve skalerle çarpma işlemleri, $\mathbb{R}^2$'yi bir vektör uzayı yapan işlemlerin ta kendisidir (bkz. Lineer Cebir). Toplamın ve farkın geometrisi paralelkenar kuralıdır:



Şekil 4.6: `Vec2` ile hesaplanan `u + v` ve `u - v`. Toplam, `u` ile `v` üzerine kurulan paralelkenarın köşegenidir; fark ise `v`'nin ucundan `u`'nun ucuna giden vektördür (kesikli ok), başlangıç noktası orijine taşınmış olarak.

 CC BY-NC-SA 4.0

Tanımlamadığımız işlemler ise hâlâ hata verir, `__eq__` metodunun da bir yan etkisi vardır. Bu kod da önceki iki kodun devamıdır:

```
u, v = Vec2(3, 1), Vec2(1, 2)
try:
    u * v
except TypeError as e:
    print("TypeError:", e)
try:
    {u, v}
except TypeError as e:
    print("TypeError:", e)
w = Vec2(0.1, 0.2) + Vec2(0.2, 0.1)
print(w, w == Vec2(0.3, 0.3))
```

Çıktı:

```
TypeError: unsupported operand type(s) for *: 'Vec2' and 'Vec2'
TypeError: unhashable type: 'Vec2'
Vec2(0.30000000000000004, 0.30000000000000004) False
```

- İki vektörün `*` ile çarpımını tanımlamadık: `u.__mul__(v)` çağrısı `NotImplemented` döndürdü. İki terim aynı türden olduğu için Python `v.__rmul__(u)` çağrısını hiç denemez ve doğrudan `TypeError` fırlatır. İç çarpım için ayrı bir `dot` metodu yazmamız bilinçli bir seçimdir.
- `__eq__` tanımlayan bir sınıfın nesneleri, ayrıca bir `__hash__` metodu yazılmadıkça **hashlenebilir** değildir (bkz. Veri Yapıları ve Comprehension), yani kümeye konamaz ve sözlük anahtarı olamaz. Eşit nesnelerin hash değerleri de eşit olmalıdır ve nitelikleri değiştirilebilen bir nesnede bu koşul zamanla bozulabilir. Birazdan göreceğimiz `dataclass(frozen=True)` bu sorunu çözer.
- Son satır, kayan nokta aritmetiğinin bilinen tuzağıdır.

> ⚠ Kayan noktalı nitelikleri == ile karşılaştırmayın
>
> $0{,}1 + 0{,}2$ makinede tam olarak $0{,}3$ etmediği için (bkz. Python ile İlk Adımlar) `Vec2(0.1, 0.2) + Vec2(0.2, 0.1)` toplamı `Vec2(0.3, 0.3)` nesnesine eşit çıkmadı. `__eq__` tam eşitlik sorar ve tam sayı bileşenlerde doğru çalışır. Ölçümden ya da hesaptan gelen vektörleri karşılaştırmak için `abs(u - v) < tol` biçiminde bir tolerans ya da bileşenler üzerinde `math.isclose` (bkz. Python ile İlk Adımlar) kullanılır.

Örnek 4.7 (Kendi Kesir Sınıfımız). Rasyonel sayıları p/q biçiminde, hep sadeleşmiş ve paydası pozitif olarak tutan bir `Rational` sınıfı yazınız. Payda sıfırsa `ZeroDivisionError` fırlatılsın; toplama, çarpma, eşitlik ve `repr` tanımlansın. Sınıfı $H_{10} = 1 + \frac{1}{2} + \ldots + \frac{1}{10}$ harmonik sayısıyla deneyip sonucu `fractions.Fraction` ile karşılaştırınız.

Çözüm.
Sadeleştirmeyi `__init__` içinde bir kez yapıyoruz: pay ve paydayı $g = \gcd(p, q)$'ya böleriz, payda negatifse ikisinin de işaretini değiştirmek için $-g$'ye böleriz. Böylece her kesrin tek bir gösterimi olur ve iki kesrin eşitliği, pay ve paydalarının eşitliğine indirgenir. Toplama ve çarpma

$$\frac{p}{q} + \frac{r}{s} = \frac{ps + rq}{qs}, \qquad \frac{p}{q} \cdot \frac{r}{s} = \frac{pr}{qs}$$

formüllerinden gelir; sonuç yeni bir `Rational` olarak kurulduğu için kendiliğinden sadeleşir.

```
from fractions import Fraction
from math import gcd


class Rational:
    """p/q kesri; hep sadeleşmiş ve paydası pozitif tutulur."""
```

 CC BY-NC-SA 4.0

```
    def __init__(self, p, q=1):
        if q == 0:
            raise ZeroDivisionError("payda sıfır olamaz")
        g = gcd(p, q)
        if q < 0:
            g = -g
        self.p = p // g
        self.q = q // g

    def __repr__(self):
        return f"{self.p}/{self.q}"

    def __add__(self, other):
        return Rational(self.p * other.q + other.p * self.q,
                        self.q * other.q)

    def __mul__(self, other):
        return Rational(self.p * other.p, self.q * other.q)

    def __eq__(self, other):
        return self.p == other.p and self.q == other.q


print(Rational(6, -8), Rational(-3, 4) == Rational(6, -8))
H = Rational(0)
for k in range(1, 11):
    H = H + Rational(1, k)
print("H_10 =", H)
print("Fraction ile:", sum(Fraction(1, k) for k in range(1, 11)))
try:
    Rational(1, 0)
except ZeroDivisionError as e:
    print("Hata:", e)
```

Çıktı:

```
-3/4 True
H_10 = 7381/2520
Fraction ile: 7381/2520
Hata: payda sıfır olamaz
```

$6/(-8)$ kesri $-3/4$ olarak saklandı ve $-3/4$'e eşit çıktı. Harmonik sayı $H_{10} = 7381/2520$ bulundu ve `Fraction` ile aynı. Sıfır paydalı bir kesir oluşturma denemesi, kendi yazdığımız mesajla durduruldu. Sadeleştirmedeki gcd, `math.gcd`'den geliyor ve Öklid algoritmasıyla hesaplanıyor (bkz. Sayılar Teorisi). Gerçek bir hesapta elbette hazır `Fraction` sınıfı kullanılır; buradaki amaç, bir sınıfın içini kendi elimizle kurmaktı. ■

4.4.1 Kalıtım ve kendi istisna türlerimiz

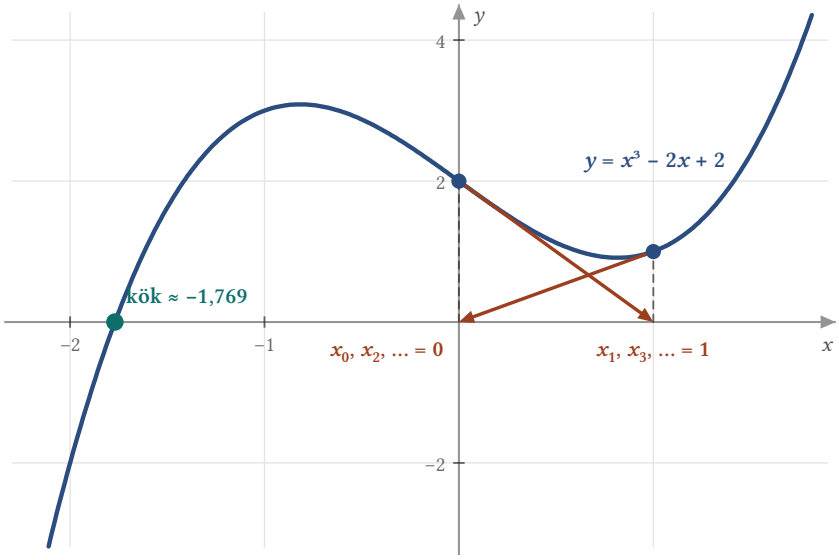
Bir sınıf, var olan bir sınıfı genişleterek de tanımlanabilir; kendi istisna türlerimizi tanımlamanın yolu budur.

Tanım 4.9 (Kalıtım). `class B(A):` biçiminde tanımlanan `B` sınıfı, `A` sınıfından **kalıtım** (inheritance) yoluyla türer: `A`'nın bütün metotlarına sahip olur, bunlara yenilerini ekleyebilir ya da aynı adla yeniden tanımlayarak onları değiştirebilir. `A`'ya **üst sınıf**, `B`'ye **alt sınıf** denir ve her `B` nesnesi aynı zamanda bir `A` nesnesidir.

 CC BY-NC-SA 4.0

Yani alt sınıf, üst sınıfın özel bir hâlidir; istisna soyağacında `ZeroDivisionError`'ın bir `ArithmeticError` olması tam olarak budur. Kendi istisna türümüzü uygun bir yerleşik türden türetirsek onu hem kendi adıyla hem de üst türünün adıyla yakalayabiliriz. Alt sınıfın `__init__` metodunda `super().__init__(...)` çağrısı üst sınıfın `__init__` metodunu çalıştırır; böylece istisnanın mesajı gibi standart bilgiler üst sınıfça saklanır.

Örnek 4.8 (Yakınsamayan Newton İterasyonu). $f(x) = x^3 - 2x + 2$ için Newton yöntemi (bkz. Nümerik Analiz) $x_0 = 0$'dan başlatıldığında 0 ile 1 arasında gidip gelir ve hiç yakınsamaz:



Şekil 4.7: $f(x) = x^3 - 2x + 2$ için Newton yöntemi $x_0 = 0$'dan başlayınca: 0'daki teğet ekseni 1'de, 1'deki teğet ekseni yine 0'da keser. İterasyon 0, 1, 0, 1, … döngüsüne girer ve tek kök olan −1,769… noktasına hiç yaklaşmaz; `newton` fonksiyonu bu yüzden `ConvergenceError` fırlatır.

İzin verilen adım sayısında yakınsamazsa son iterasyonları da taşıyan bir `ConvergenceError` fırlatan bir `newton` fonksiyonu yazıp $x_0 = -2$ ve $x_0 = 0$ için deneyiniz.

Çözüm.
`ConvergenceError` sınıfını `RuntimeError`'dan türetiyoruz ve `__init__` metoduna iterasyon listesini saklayan bir nitelik ekliyoruz:

```
class ConvergenceError(RuntimeError):
    """Bir iterasyon izin verilen adım sayısında yakınsamazsa."""

    def __init__(self, message, iterates):
        super().__init__(message)
        self.iterates = iterates


def newton(f, df, x0, tol=1e-12, maxiter=20):
    xs = [x0]
    for _ in range(maxiter):
        x = xs[-1]
        x_new = x - f(x) / df(x)
        xs.append(x_new)
        if abs(x_new - x) < tol:
            return x_new
    raise ConvergenceError(f"{maxiter} adımda yakınsamadı", xs)
```

 CC BY-NC-SA 4.0

```
def f(x):
    return x**3 - 2 * x + 2


def df(x):
    return 3 * x**2 - 2


print(newton(f, df, -2.0))
try:
    newton(f, df, 0.0)
except ConvergenceError as e:
    print("Hata:", e)
    print("son iterasyonlar:", e.iterates[-4:])
print(issubclass(ConvergenceError, RuntimeError))
```

Çıktı:

```
-1.7692923542386314
Hata: 20 adımda yakınsamadı
son iterasyonlar: [1.0, 0.0, 1.0, 0.0]
True
```

$x_0 = -2$'den başlayan iterasyon, tek gerçel kök olan $-1{,}7692923542\ldots$ sayısına yakınsadı. $x_0 = 0$ için

$$x_1 = 0 - \frac{f(0)}{f'(0)} = 0 - \frac{2}{-2} = 1, \qquad x_2 = 1 - \frac{f(1)}{f'(1)} = 1 - \frac{1}{1} = 0$$

olduğundan dizi $0, 1, 0, 1, \ldots$ döngüsüne girer. 20 adımın sonunda fonksiyon istisnayı fırlattı ve çağıran kod, istisna nesnesinin `iterates` niteliğinden döngüyü okuyabildi. Son satır `ConvergenceError`'ın bir `RuntimeError` olduğunu doğruluyor: `except RuntimeError` yazan bir kod da bu istisnayı yakalardı. ■

4.5 Veri sınıfları: dataclass

`Vec2`'yi yazarken `__init__`, `__repr__` ve `__eq__` için hep aynı kalıbı tekrarladık; yalnız veri taşıyan sınıflar için standart kütüphane bu kalıbı bizim yerimize yazar.

Tanım 4.10 (Veri Sınıfı). `dataclasses` modülündeki `@dataclass` **dekoratörü** bir sınıfın önüne yazıldığında, sınıfta `ad: tür` biçiminde bildirilen **alanlardan** `__init__`, `__repr__` ve `__eq__` metotlarını kendiliğinden üretir. Böyle bir sınıfa **veri sınıfı** (data class) denir. `@dataclass(frozen=True)` yazımı alanları salt okunur yapar ve nesneleri hashlenebilir kılar; `@dataclass(order=True)` ise nesneleri alanlarına göre sözlük sırasıyla karşılaştıran `<`, `<=` gibi metotları da ekler.

Yani veri sınıfı, alanlarını saymamızın yettiği bir sınıftır: tekrarlanan metotları Python yazar, biz yalnız matematiğe özgü metotları ekleriz. Dekoratör, `@` ile başlayıp bir tanımın üstüne yazılan ve o tanımı alıp değiştiren bir fonksiyondur. Tür bildirimleri (`x: float`) okur için bir belgedir; Python çalışırken bunları denetlemez.

```
import math
from dataclasses import dataclass


@dataclass(frozen=True)
class Point:
    x: float
    y: float
```

 CC BY-NC-SA 4.0

```
    def dist(self, other):
        return math.hypot(self.x - other.x, self.y - other.y)


A, B, C = Point(0, 0), Point(4, 0), Point(0, 3)
print(A, B == Point(4.0, 0.0))
print(A.dist(B), B.dist(C), C.dist(A))
print(len({A, B, C, Point(0, 0)}))
try:
    A.x = 1
except AttributeError as e:
    print(type(e).__name__, "-", e)
```

Çıktı:

```
Point(x=0, y=0) True
4.0 5.0 3.0
3
FrozenInstanceError - cannot assign to field 'x'
```

`Point(0, 0)` yazdırılınca alan adlarıyla birlikte göründü. Üretilen `__eq__` alanları sırayla karşılaştırdığı için `Point(4, 0) == Point(4.0, 0.0)` doğru çıktı ve dik üçgenin kenarları 4, 5, 3 bulundu. `frozen=True` sayesinde noktalar kümeye konabildi; iki eşit $(0, 0)$ noktası kümede tek eleman sayıldığı için küme 3 elemanlı. Buna karşılık bir alanı değiştirme denemesi `FrozenInstanceError` fırlattı; bu tür `AttributeError`'ın alt türü olduğu için onu üst türünün adıyla yakaladık.

Bir veri sınıfına alanlarını denetleyen bir metot da eklenebilir: `__post_init__` metodu, üretilen `__init__` alanları atadıktan hemen sonra çağrılır. Bir koşul bozuksa burada `raise` ile istisna fırlatmak, geçersiz bir nesnenin hiç oluşmamasını sağlar. Bölümün son örneğinde bunu kullanacağız.

 CC BY-NC-SA 4.0

⚠ Değiştirilebilen varsayılan değerler

Bir alana varsayılan değer olarak boş bir liste vermek, bütün nesnelerin **aynı** listeyi paylaşmasına, yani her nesnede o listenin bir takma adının bulunmasına yol açardı (bkz. Veri Yapıları ve Comprehension). `dataclass` bu hatayı daha sınıf tanımlanırken yakalar. Doğrusu, her nesne için yeni bir liste üreten `field(default_factory=list)` yazımıdır:

```
from dataclasses import dataclass, field

try:
    @dataclass
    class Run:
        method: str
        history: list = []
except ValueError as e:
    print("ValueError:", e)


@dataclass
class Run:
    method: str
    history: list = field(default_factory=list)


r1, r2 = Run("newton"), Run("bisect")
r1.history.append(1.5)
print(r1, r2)
```

Çıktı:

```
ValueError: mutable default <class 'list'> for field history is not allowed: use
default_factory
Run(method='newton', history=[1.5]) Run(method='bisect', history=[])
```

İki nesnenin listeleri ayrıdır: birine eklenen değer öbürünü etkilemedi.

4.6 Hepsi bir arada: bir ölçüm dosyasını işlemek

Bölümün araçlarını tek bir gerçekçi görevde birleştirelim: hatalı satırlar da içeren bir ölçüm dosyasından bir fiziksel sabiti tahmin edeceğiz.

💡 Dört adımda bir veri dosyasını güvenle okumak

1. Bir kaydı temsil eden bir veri sınıfı tanımlayın; fiziksel olarak imkânsız değerleri `__post_init__` içinde `ValueError` ile reddedin.
2. Dosyayı `with` ve `encoding="utf-8"` ile açın, `csv.reader` ile satırlara ayırın ve başlık satırını `next` ile atlayın.
3. Her satırı bir `try` bloğu içinde kayda dönüştürün; `except ValueError` bloğunda satır numarasını ve hatayı bir listeye yazıp sonraki satıra geçin.
4. Hesabı geçerli kayıtlarla yapın, atlanan satırları raporlayın ve sonucu bir dosyaya yazın.

Örnek 4.9 (Soğuma Ölçümlerinden Bir Sabit Tahmin Etmek). 90 °C'deki bir sıvı 20 °C'lik bir odada soğuyor. Newton'un soğuma yasasına göre sıcaklık $T(t) = 20 + 70e^{-kt}$ biçiminde azalır; burada t dakika cinsinden zaman, $k > 0$ bir sabittir. İki dakikada bir alınan ölçümler Türkçe biçimde `sogutma.csv` dosyasına kaydedilmiş, ama bazı satırlar hatalı girilmiş. Dosyayı okuyup hatalı satırları raporlayan, her geçerli ölçümden

$$k = \frac{1}{t}\ln\frac{70}{T - 20}$$

 CC BY-NC-SA 4.0

değerini hesaplayıp bunların ortalamasını bir dosyaya yazan bir program yazınız.

Çözüm.
Dört adımı sırayla uyguluyoruz. Program önce dosyayı kendisi oluşturuyor; gerçek bir işte bu dosya bir ölçüm cihazından gelirdi.

```
import csv
import math
from dataclasses import dataclass

DATA = """t;T
0;90,0
2;77,7
4;67,2
6;58,2
8;
10;45,8
12;4l,3
14;37,4
16;34,6
18;3,18
20;29,6
"""
with open("sogutma.csv", "w", encoding="utf-8") as f:
    f.write(DATA)


@dataclass
class Measurement:
    t: float        # dakika
    T: float        # santigrat derece

    def __post_init__(self):
        if self.T <= 20:
            raise ValueError(f"T = {self.T} oda sıcaklığının altında")


def to_float(s):
    return float(s.replace(",", "."))


def read_measurements(path):
    good, bad = [], []
    with open(path, newline="", encoding="utf-8") as f:
        reader = csv.reader(f, delimiter=";")
        next(reader)                          # başlık satırı
        for line_no, row in enumerate(reader, start=2):
            try:
                t, T = row
                good.append(Measurement(to_float(t), to_float(T)))
            except ValueError as e:
                bad.append((line_no, e))
    return good, bad


good, bad = read_measurements("sogutma.csv")
for line_no, e in bad:
    print(f"satır {line_no} atlandı: {e}")
ks = [math.log(70 / (m.T - 20)) / m.t for m in good if m.t > 0]
k = sum(ks) / len(ks)
```

 CC BY-NC-SA 4.0

```
print(f"{len(good)} geçerli ölçüm, k ≈ {k:.4f}")
with open("sogutma_ozet.txt", "w", encoding="utf-8") as f:
    f.write(f"gecerli={len(good)}\nk={k:.6f}\n")
```

Çıktı:

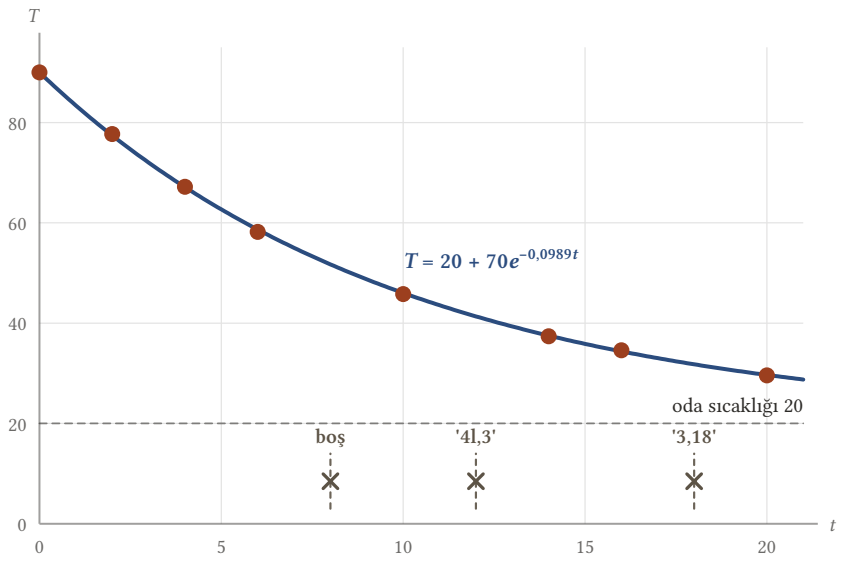
```
satır 6 atlandı: could not convert string to float: ''
satır 8 atlandı: could not convert string to float: '4l.3'
satır 11 atlandı: T = 3.18 oda sıcaklığının altında
8 geçerli ölçüm, k ≈ 0.0989
```

Hatalı satırlar. Üç satır üç farklı nedenle atlandı:

- 6. satırdaki T alanı boştu ve `float("")` bir `ValueError` fırlattı.
- 8. satırda bir rakamın yerine küçük L harfi girilmişti (`4l,3`). Mesajdaki `'4l.3'`, virgülün noktaya çevrilmiş hâlidir.
- 11. satırdaki `3,18` sayıya çevrilebildi ama oda sıcaklığının altında kaldığı için `__post_init__` onu reddetti; büyük olasılıkla `31,8` yazılmak istenmişti.

Üç hata da aynı `except ValueError` satırıyla yakalandı, çünkü ikisini `float`, birini bizim `raise` deyimimiz fırlattı. Satırda iki alan olmasaydı `t, T = row` ataması da bir `ValueError` fırlatırdı ve o durum da aynı yere düşerdi.
Tahmin. Formül, $T - 20 = 70e^{-kt}$ eşitliğinin iki tarafının logaritması alınarak bulunur. $t = 0$ satırı formüle giremeyeceği için hesaba katılmadı; kalan yedi ölçümün ortalaması $k \approx 0{,}0989$ verdi. Yani sıvı ile oda arasındaki sıcaklık farkı her dakika yaklaşık %9,4 azalıyor, çünkü $e^{-0{,}0989} \approx 0{,}906$'dır. Sonuç `sogutma_ozet.txt` dosyasına yazıldı. ■

Programın okuduğu ölçümler, atlanan satırlar ve tahmin edilen model aşağıdaki şekilde bir arada görülüyor:



Şekil 4.8: `sogutma.csv` dosyasından okunan sekiz geçerli ölçüm (noktalar; t dakika, T santigrat derece) ve ortalama $k \approx 0{,}0989$ ile çizilen $T = 20 + 70e^{-kt}$ modeli. Alttaki çarpılar atlanan üç satırın zamanlarıdır: boş alan, harfli değer ve oda sıcaklığının altında kalan değer.

 CC BY-NC-SA 4.0

4.7 Alıştırmalar

Aşağıdaki alıştırmalar bölümün araçlarını tek tek ve birlikte kullanıyor; her çözüm, çalıştırılmış kodu ve çıktısını içeriyor.

Alıştırma 4.1 (Hiperbolik Fonksiyonlar ve Kayıp Basamaklar). `math` modülündeki hiperbolik fonksiyonları `dir` ile listeleyiniz. Sonra $\cosh^2 x - \sinh^2 x$ ifadesini $x = 0, 5, 10, 15, 20$ için hesaplayıp sonuçların 1'den neden saptığını açıklayınız.

Çözüm.
Hiperbolik fonksiyonların adları `h` harfiyle bittiği için `dir(math)` listesini bu harfe göre süzmek yeterli:

```
import math

print([name for name in dir(math) if name.endswith("h")])
for x in range(0, 21, 5):
    d = math.cosh(x) ** 2 - math.sinh(x) ** 2
    print(x, d)
```

Çıktı:

```
['acosh', 'asinh', 'atanh', 'cosh', 'sinh', 'tanh']
0 1.0
5 1.000000000001819
10 0.9999999850988388
15 1.0
20 0.0
```

Özdeşliğe göre her satırda 1 çıkmalıydı. Büyük x için $\cosh^2 x$ ve $\sinh^2 x$ yaklaşık $e^{2x}/4$ büyüklüğündedir; $x = 20$ için bu yaklaşık $5{,}9 \cdot 10^{16}$'dır. Bu büyüklükteki iki ardışık kayan noktalı sayının arası 8'dir, yani iki kare yuvarlanırken 1'lik fark tamamen kaybolur ve sonuç 0 çıkar. $x = 10$ için iki ardışık sayının arası yaklaşık $1{,}5 \cdot 10^{-8}$'dir ve sapma tam bu büyüklüktedir; $x = 15$'te sonucun 1 çıkması yuvarlamaların şanslı birleşimidir. Birbirine çok yakın iki büyük sayıyı çıkarmak öncü basamakları yok eder (bkz. Nümerik Analiz). ■

Alıştırma 4.2 (Esnek Bir Sayı Okuyucu). `'12'`, `'3,5'`, `'7/2'` ve `'1e3'` gibi metinleri `float`'a çeviren bir `to_number` fonksiyonu yazınız; kesir biçimindeki yazımlar için `fractions.Fraction` kullanınız. Fonksiyonu `["12", "3,5", "7/2", "abc", "", "1e3", "1/0"]` listesine uygulayıp çevrilebilen değerleri bir listede, çevrilemeyen metinleri de hatanın türüyle birlikte başka bir listede toplayınız.

Çözüm.
Önce `float` deneniyor; o başarısız olursa `Fraction` deneniyor. `Fraction` da metni anlamazsa `ValueError`, payda sıfırsa `ZeroDivisionError` fırlatır. Çağıran döngü bu iki türü bir demetle birlikte yakalıyor:

```
from fractions import Fraction


def to_number(s):
    """'12', '3,5', '7/2', '1e3' gibi metinleri float'a çevirir."""
    s = s.strip().replace(",", ".")
    try:
        return float(s)
    except ValueError:
        return float(Fraction(s))
```

 CC BY-NC-SA 4.0

```
inputs = ["12", "3,5", "7/2", "abc", "", "1e3", "1/0"]
values, failed = [], []
for s in inputs:
    try:
        values.append(to_number(s))
    except (ValueError, ZeroDivisionError) as e:
        failed.append((s, type(e).__name__))
print(values)
print(failed)
```

Çıktı:

```
[12.0, 3.5, 3.5, 1000.0]
[('abc', 'ValueError'), ('', 'ValueError'), ('1/0', 'ZeroDivisionError')]
```

`"7/2"` metni `float` ile okunamadı ama `Fraction("7/2")` onu $7/2$ olarak tanıdı. `"abc"` ve boş metin iki denemede de `ValueError` verdi; `"1/0"` ise metin olarak geçerli bir kesir olduğu hâlde paydası sıfır olduğundan `ZeroDivisionError` fırlattı. Hiçbir hata programı durdurmadı ve her biri türüyle kaydedildi. ■

Alıştırma 4.3 (Girdisini Denetleyen Binom Katsayısı). Pascal üçgeninin satırlarını kurarak $\binom{n}{k}$ değerini hesaplayan bir `binomial(n, k)` fonksiyonu yazınız. Fonksiyon tam sayı olmayan girdide `TypeError`, negatif girdide `ValueError` fırlatsın ve $k > n$ için 0 döndürsün. Sonuçları $0 \le n < 30$, $0 \le k < 32$ için `math.comb` ile karşılaştırınız.

Çözüm.
Pascal üçgeninin bir sonraki satırı, satırın başına ve sonuna 0 eklenmiş iki kopyasının terim terim toplamıdır: $[1,2,1]$ satırından $[0,1,2,1]$ ile $[1,2,1,0]$ toplanarak $[1,3,3,1]$ elde edilir. Denetimler hesaptan önce yapılıyor:

```
import math


def binomial(n, k):
    """C(n, k) değerini Pascal üçgeninin n. satırından hesaplar."""
    if not isinstance(n, int) or not isinstance(k, int):
        raise TypeError("n ve k tam sayı olmalı")
    if n < 0 or k < 0:
        raise ValueError("n ve k negatif olamaz")
    if k > n:
        return 0
    row = [1]
    for _ in range(n):
        row = [a + b for a, b in zip([0] + row, row + [0])]
    return row[k]


print(all(binomial(n, k) == math.comb(n, k)
          for n in range(30) for k in range(32)))
for n, k in [(5.0, 2), (-1, 2), (10, 3)]:
    try:
        print((n, k), "->", binomial(n, k))
    except (TypeError, ValueError) as e:
        print((n, k), "->", type(e).__name__, "-", e)
```

Çıktı:

 CC BY-NC-SA 4.0

```
True
(5.0, 2) -> TypeError - n ve k tam sayı olmalı
(-1, 2) -> ValueError - n ve k negatif olamaz
(10, 3) -> 120
```

$30 \cdot 32 = 960$ karşılaştırmanın hepsi tuttu. `math.comb` da aynı kuralları izler: `math.comb(5.0, 2)` bir `TypeError`, `math.comb(-1, 2)` bir `ValueError` fırlatır ve $k > n$ için 0 döndürür. ■

Alıştırma 4.4 (Tam Sayı Karekökü). Newton iterasyonunun tam sayı sürümü $x \mapsto \lfloor (x + \lfloor n/x \rfloor)/2 \rfloor$ ile, yalnız tam sayı aritmetiği kullanarak $\lfloor \sqrt{n} \rfloor$ değerini hesaplayan bir `isqrt(n)` fonksiyonu yazınız. Sonucun $r^2 \le n < (r+1)^2$ koşulunu sağladığını `assert` ile denetleyiniz ve sonucu $n < 10^5$ için `math.isqrt` ile, $n = (2^{53}+1)^2$ için ayrıca `int(math.sqrt(n))` ile karşılaştırınız.

Çözüm.
$x_0 = n$'den başlayıp $y = \lfloor (x + \lfloor n/x \rfloor)/2 \rfloor$ değerini hesaplıyoruz ve y küçülmeyi bıraktığında duruyoruz. $s = \lfloor \sqrt{n} \rfloor$ diyelim. Bu kuralın neden s'de durduğu iki gözlemden çıkar:

- x tam sayı olduğundan $y = \lfloor (x + n/x)/2 \rfloor$'dir ve aritmetik-geometrik ortalama eşitsizliğinden $(x + n/x)/2 \ge \sqrt{n}$ olur. Dolayısıyla her zaman $y \ge s$'dir.
- $x > s$ ise $x > \sqrt{n}$, yani $n/x < x$ olur; bu durumda $y \le (x + n/x)/2 < x$'tir ve dizi kesin azalır.

Dizi s'nin altına inemediğinden ve s'nin üstündeyken kesin azaldığından sonlu adımda $x = s$ değerine ulaşır; o noktada $y \ge s = x$ olduğu için döngü durur. `assert` satırı bu akıl yürütmenin sonucunu her çağrıda yeniden sınıyor:

```
import math


def isqrt(n):
    """n >= 0 için karekökün tam kısmı; yalnız tam sayı aritmetiği."""
    if n < 0:
        raise ValueError("negatif sayının karekökü alınamaz")
    if n < 2:
        return n
    x = n
    y = (x + n // x) // 2
    while y < x:
        x = y
        y = (x + n // x) // 2
    assert x * x <= n < (x + 1) ** 2, n
    return x


print(all(isqrt(n) == math.isqrt(n) for n in range(10**5)))
N = (2**53 + 1) ** 2
print(isqrt(N), math.isqrt(N))
print(int(math.sqrt(N)))
```

Çıktı:

```
True
9007199254740993 9007199254740993
9007199254740992
```

10^5 değerin hepsinde sonuç `math.isqrt` ile aynı. $n = (2^{53}+1)^2 \approx 8{,}1 \cdot 10^{31}$ için iki tam sayı fonksiyonu doğru kök olan $2^{53}+1$'i verirken `int(math.sqrt(n))` bir eksiğini verdi. Asıl neden, doğru kökün kendisinin bir kayan noktalı sayı olmamasıdır: $2^{53}+1$ sayısı 54 anlamlı bit ister ve 53 bitlik

 CC BY-NC-SA 4.0

mantise sığmaz; `float(2**53 + 1)` zaten 2^{53}'tür. 2^{53} civarında ardışık iki kayan noktalı sayının arası 2 olduğundan `math.sqrt`, n kendisine tam olarak verilse bile $2^{53}+1$ döndüremez ve sonuç 2^{53}'e yuvarlanır. n'nin kayan noktaya çevrilirken $2^{106}+2^{54}$'e yuvarlanması ise ikincil bir etkidir: o sayının karekökü yaklaşık $2^{53}+1-2^{-54}$'tür ve yine 2^{53}'e yuvarlanır. Büyük tam sayılarla çalışırken kayan noktaya hiç geçmemek bu yüzden önemlidir. ■

Alıştırma 4.5 (Blokların İzini Sürmek). Aşağıdaki kodun çıktısını çalıştırmadan tahmin ediniz, sonra çalıştırıp tahmininizi denetleyiniz.

```
def g(x):
    try:
        r = 10 // x
    except ZeroDivisionError:
        print("A")
        r = 0
    except TypeError:
        print("B")
        raise
    else:
        print("C")
    finally:
        print("D")
    return r


print(g(3))
print(g(0))
try:
    g("5")
except TypeError:
    print("E")
```

Çözüm.
Üç çağrıyı tek tek izleyelim.

- `g(3)`: `10 // 3 = 3` hatasız hesaplanır; `else` bloğu `C`, `finally` bloğu `D` basar ve fonksiyon `3` döndürür.
- `g(0)`: `10 // 0` bir `ZeroDivisionError` fırlatır; ilk `except` bloğu `A` basıp `r = 0` atar, `finally` bloğu `D` basar ve fonksiyon `0` döndürür.
- `g("5")`: `10 // "5"` bir `TypeError` fırlatır; ikinci `except` bloğu `B` basar ve tek başına yazılmış `raise` aynı istisnayı yeniden fırlatır. Fonksiyondan çıkılmadan önce `finally` yine `D` basar. `return r` satırına hiç gelinmez; istisna dışarıdaki `try` tarafından yakalanır ve `E` basılır.

Kodu çalıştırmak bu tahmini doğruluyor.
Çıktı:

```
C
D
3
A
D
0
B
D
E
```

■

 CC BY-NC-SA 4.0

Alıştırma 4.6 (Pascal Üçgeni Dosyada). Pascal üçgeninin 0'dan 10'a kadar olan satırlarını, sayılar boşlukla ayrılmış olarak `pascal.txt` dosyasına yazınız. Dosyayı geri okuyup her satırdaki sayıların toplamının 2^n olduğunu doğrulayınız.

Çözüm.
Her satırı `" ".join` ile tek bir metne çevirip sonuna `"\n"` ekliyoruz. Okurken argümansız `split()` satırı boşluklardan böler ve satır sonunu da atar:

```
from math import comb

with open("pascal.txt", "w", encoding="utf-8") as f:
    for n in range(11):
        row = [comb(n, k) for k in range(n + 1)]
        f.write(" ".join(str(c) for c in row) + "\n")

with open("pascal.txt", encoding="utf-8") as f:
    rows = [[int(s) for s in line.split()] for line in f]
print(rows[4], rows[10])
print(all(sum(r) == 2**n for n, r in enumerate(rows)))
```

Çıktı:

```
[1, 4, 6, 4, 1] [1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]
True
```

Dördüncü ve onuncu satırlar beklenen katsayıları taşıyor ve on bir satırın hepsinde toplam 2^n çıktı. Bu, binom teoreminde $a = b = 1$ almanın sonucudur: $\sum_{k=0}^{n} \binom{n}{k} = (1+1)^n = 2^n$. $\blacksquare$

Alıştırma 4.7 (Faktöriyellerin Basamak Sayıları). $n = 1, 2, \ldots, 100$ için $n!$ sayısının basamak sayısını `n` ve `basamak` sütunlu bir CSV dosyasına yazınız. Dosyayı `csv.DictReader` ile okuyup basamak sayısı ilk kez 100'e ulaşan ya da onu aşan n değerini bulunuz.

Çözüm.
Python'un tam sayıları sınırsız olduğundan $n!$ tam olarak hesaplanır ve basamak sayısı `len(str(...))` ile okunur. Dosyayı okurken satırları bir sözlük comprehension ile `n → basamak` sözlüğünde topluyoruz:

```
import csv
import math

with open("faktoriyel.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["n", "basamak"])
    for n in range(1, 101):
        writer.writerow([n, len(str(math.factorial(n)))])

with open("faktoriyel.csv", newline="", encoding="utf-8") as f:
    digits = {int(r["n"]): int(r["basamak"])
              for r in csv.DictReader(f)}
first = min(n for n, d in digits.items() if d >= 100)
print(first, digits[first - 1], digits[first])
```

Çıktı:

```
70 99 101
```

Aranan değer $n = 70$'tir. İlginç olan, 100 basamaklı hiçbir faktöriyelin olmamasıdır: $69!$ sayısı 99, $70!$ sayısı 101 basamaklıdır. Basamak sayısı $\lfloor \log_{10} n! \rfloor + 1$ olduğundan 70 ile çarpmak logaritmayı

$\log_{10} 70 \approx 1{,}85$ kadar artırır; $\log_{10} 69! \approx 98{,}23$ iken $\log_{10} 70! \approx 100{,}08$ olur ve basamak sayısı bir adımda iki artar. ■

Alıştırma 4.8 (Bir Polinom Sınıfı). Katsayıları artan derece sırasıyla bir listede tutan bir `Poly` sınıfı yazınız. Sınıf, `p(x)` yazımıyla Horner yöntemine göre değer hesaplasın (`__call__`), `+` ve `*` işlemlerini ve okunaklı bir `repr` metnini desteklesin. $(1+x)^3$ polinomunu hesaplayıp $x = 2$'deki değerini 3^3 ile karşılaştırınız.

Çözüm.
Çarpımın x^m katsayısı, $i + j = m$ olan bütün $a_i b_j$ çarpımlarının toplamıdır; iç içe döngü her $a_i b_j$ çarpımını `prod[i + j]` hücresine ekliyor. Horner yöntemi $a_0 + a_1 x + a_2 x^2 + a_3 x^3$ ifadesini $a_0 + x(a_1 + x(a_2 + x a_3))$ biçiminde, en yüksek dereceden başlayarak hesaplar. `reversed(self.c)` katsayı listesini sondan başa doğru dolaştığı için döngü en yüksek dereceli katsayıdan başlar:

```
class Poly:
    """Katsayıları artan derece sırasıyla tutulan polinom."""

    def __init__(self, coeffs):
        self.c = list(coeffs)

    def __repr__(self):
        terms = []
        for k, a in enumerate(self.c):
            if a == 0:
                continue
            coef = "" if a == 1 and k > 0 else str(a)
            power = "" if k == 0 else "x" if k == 1 else f"x^{k}"
            terms.append(coef + power)
        return " + ".join(terms) or "0"

    def __call__(self, x):
        result = 0
        for a in reversed(self.c):      # Horner yöntemi
            result = result * x + a
        return result

    def __add__(self, other):
        n = max(len(self.c), len(other.c))
        a = self.c + [0] * (n - len(self.c))
        b = other.c + [0] * (n - len(other.c))
        return Poly([s + t for s, t in zip(a, b)])

    def __mul__(self, other):
        prod = [0] * (len(self.c) + len(other.c) - 1)
        for i, a in enumerate(self.c):
            for j, b in enumerate(other.c):
                prod[i + j] += a * b
        return Poly(prod)


p = Poly([1, 1])                    # 1 + x
q = p * p * p
print(q)
print(q(2), 3**3)
print(q + Poly([0, -3, -3]))
```

Çıktı:

```
1 + 3x + 3x^2 + x^3
27 27
1 + x^3
```

$(1+x)^3 = 1 + 3x + 3x^2 + x^3$ bulundu ve $x = 2$'deki değeri $3^3 = 27$ ile aynı. Son satırda $-3x - 3x^2$ eklenince sıfır olan katsayılar `repr` tarafından atlandı. NumPy'nin `numpy.polynomial.Polynomial` sınıfı aynı fikrin hazır ve çok daha kapsamlı hâlidir (bkz. Polinomlar, İnterpolasyon ve Eğri Uydurma). ■

Alıştırma 4.9 (Aralık Aritmetiği). Kapalı bir $[a, b]$ aralığını temsil eden, `frozen=True` ile tanımlanmış bir `Interval` veri sınıfı yazınız. $a > b$ ise `__post_init__` bir `ValueError` fırlatsın; işlemler

$$\begin{aligned} [a,b] + [c,d] &= [a+c,\, b+d], \\ [a,b] - [c,d] &= [a-d,\, b-c], \\ [a,b] \cdot [c,d] &= [\min P,\, \max P], \quad P = \{ac, ad, bc, bd\} \end{aligned}$$

kurallarıyla tanımlansın ve `x in I` yazımı desteklensin. $X = [0,1]$ için $X(1-X)$ ve $X - X^2$ ifadelerini hesaplayıp $x(1-x)$ fonksiyonunun $[0,1]$'deki gerçek değer kümesiyle karşılaştırınız.

Çözüm.
`__contains__` metodu `x in I` yazımını, `__post_init__` de geçersiz aralık denetimini sağlıyor. Sabit 1'i $[1,1]$ aralığı olarak yazıyoruz:

```
from dataclasses import dataclass


@dataclass(frozen=True)
class Interval:
    lo: float
    hi: float

    def __post_init__(self):
        if self.lo > self.hi:
            raise ValueError(f"boş aralık: [{self.lo}, {self.hi}]")

    def __add__(self, other):
        return Interval(self.lo + other.lo, self.hi + other.hi)

    def __sub__(self, other):
        return Interval(self.lo - other.hi, self.hi - other.lo)

    def __mul__(self, other):
        p = [self.lo * other.lo, self.lo * other.hi,
             self.hi * other.lo, self.hi * other.hi]
        return Interval(min(p), max(p))

    def __contains__(self, x):
        return self.lo <= x <= self.hi


X = Interval(0, 1)
one = Interval(1, 1)
print(X * (one - X))
print(X - X * X)
print(0.25 in X * (one - X), 2 in X - X * X)
try:
    Interval(2, 1)
```

 CC BY-NC-SA 4.0

```
except ValueError as e:
    print("Hata:", e)
```

Çıktı:

```
Interval(lo=0, hi=1)
Interval(lo=-1, hi=1)
True False
Hata: boş aralık: [2, 1]
```

$x(1-x)$ fonksiyonunun $[0,1]$'deki gerçek değer kümesi $[0,\tfrac{1}{4}]$'tür; en büyük değer $x=\tfrac{1}{2}$'de alınır. Aralık aritmetiği iki yazım için de bu kümeyi **kapsayan** bir aralık verdi ($\tfrac{1}{4}\in X(1-X)$), ama kümeyi büyüttü: $X(1-X)=[0,1]$ ve $X-X^2=[-1,1]$. Üstelik aynı fonksiyonun iki yazımı farklı sonuç verdi. Bunun nedeni, aralık aritmetiğinin X'in her geçişini birbirinden bağımsız bir sayı gibi ele almasıdır: $X-X^2$ hesabında ilk X için 0, X^2 için 1 seçilebilir ve -1 alt sınırı böyle oluşur; oysa ikisi aynı x'tir. Bu olguya aralık aritmetiğinde **bağımlılık sorunu** denir. ■

Alıştırma 4.10 (Iraksamayı Bildiren Sabit Nokta İterasyonu). $x_{n+1}=g(x_n)$ sabit nokta iterasyonunu (bkz. Nümerik Analiz) uygulayan bir `fixed_point(g, x0)` fonksiyonu yazınız. İterasyon $|x|>10^8$ değerine ulaşırsa `ArithmeticError`'dan türeyen bir `DivergenceError`, izin verilen adım sayısında yakınsamazsa bir `RuntimeError` fırlatılsın. Fonksiyonu $g(x)=\cos x$ ve $x_0=1$, $g(x)=x^2-2$ ve $x_0=2{,}1$, $g(x)=-x$ ve $x_0=1$ için deneyiniz.

Çözüm.
`DivergenceError` sınıfının gövdesinde yalnız belgesi var; davranışını tamamen `ArithmeticError`'dan alıyor. Fonksiyon her adımda önce büyüklüğü, sonra yakınsamayı denetliyor:

```
import math


class DivergenceError(ArithmeticError):
    """İterasyon sınırsız büyüdüğünde fırlatılır."""


def fixed_point(g, x0, tol=1e-12, maxiter=200, bound=1e8):
    x = x0
    for n in range(1, maxiter + 1):
        x_new = g(x)
        if abs(x_new) > bound:
            raise DivergenceError(f"{n}. adımda |x| > {bound:g}")
        if abs(x_new - x) < tol:
            return x_new, n
        x = x_new
    raise RuntimeError(f"{maxiter} adımda yakınsamadı")


print(fixed_point(math.cos, 1.0))
for g, x0 in [(lambda x: x * x - 2, 2.1), (lambda x: -x, 1.0)]:
    try:
        fixed_point(g, x0)
    except (DivergenceError, RuntimeError) as e:
        print(type(e).__name__, "-", e)
```

Çıktı:

```
(0.7390851332147725, 69)
DivergenceError - 6. adımda |x| > 1e+08
RuntimeError - 200 adımda yakınsamadı
```

Üç davranış, sabit noktadaki türevin büyüklüğüyle açıklanır (bkz. Nümerik Analiz):

- $\cos x = x$ denkleminin kökü $p \approx 0{,}7390851332$'dir ve $\mid g'(p) \mid = \sin p \approx 0{,}67 < 1$ olduğundan iterasyon yakınsar. Hata her adımda yalnız yaklaşık $0{,}67$ katına indiği için 69 adım gerekti.
- $x^2 - 2 = x$ denkleminin $2{,}1$'e yakın kökü $p = 2$'dir ve $\mid g'(2) \mid = 4 > 1$ olduğundan iterasyon kökten uzaklaşır: 2,41, 3,8081, 12,50… diye büyüyen değerler 6. adımda 10^8'i aştı.
- $g(x) = -x$ için tek sabit nokta 0'dır ve $\mid g'(0) \mid = 1$'dir. İterasyon $1, -1, 1, \ldots$ diye ne yaklaşır ne uzaklaşır; 200 adımın sonunda `RuntimeError` fırlatıldı.

İki hata türü bir demetle birlikte yakalandı ve `type(e).__name__` hangisinin çıktığını gösterdi. ■

Bu bölümle Python'un temel araçlarını tamamladık: artık kodumuzu modüllere bölebiliyor, hataları yakalayıp anlamlı biçimde bildirebiliyor, verileri dosyalardan okuyup yazabiliyor ve matematik nesnelerini kendi sınıflarımızla temsil edebiliyoruz. Bir sonraki bölümde bilimsel hesabın temel taşı olan NumPy Dizileri ile tanışacağız. NumPy bir pakettir ve dizileri, bu bölümde gördüğümüz özel metotlarla `+`, `*` ve `@` işlemlerini binlerce sayıya aynı anda uygulayan bir sınıfın nesneleridir.

 CC BY-NC-SA 4.0

5. NumPy Dizileri

Matematikte en çok kullandığımız nesneler vektörler, matrisler ve bir aralığa yayılmış nokta kümeleridir. Bir fonksiyonu bin noktada değerlendirmek, bir lineer sistemin katsayılarını saklamak ya da bir ısı dağılımını bir ızgara üzerinde tutmak aynı işe çıkar: çok sayıda sayıyı düzenli bir tabloda tutup onlarla toplu hesap yapmak. Python'ın listeleri bunun için tasarlanmamıştır. İki listeyi `+` ile "topladığımızda" uç uca eklenirler, bir listeyi 2 ile çarptığımızda iki kez tekrarlanır ve her eleman bellekte ayrı bir Python nesnesi olarak durur.

NumPy (Numerical Python) bu boşluğu dolduran kütüphanedir. Temel nesnesi olan `ndarray`, aynı türden sayıları bellekte tek bir blokta tutar ve onları vektör, matris ya da daha yüksek boyutlu bir tablo olarak okur. SciPy, Matplotlib ve pandas başta olmak üzere bu dersin geri kalanında kullanacağımız hemen her kütüphane bu nesnenin üzerine kuruludur.

Bu bölümde dizileri kurmayı, şekillerini ve veri türlerini okumayı, parçalarını seçmeyi ve dizileri birleştirmeyi öğreniyoruz. Diziler üzerinde döngü yazmadan toplu hesap yapmak bir sonraki bölümün konusudur.

5.1 Neden NumPy?

Farkı görmek için aynı iki vektörü önce liste, sonra NumPy dizisi olarak toplayalım.

```
import numpy as np

u = [1, 2, 3]
v = [10, 20, 30]
print("liste + liste:", u + v)
print("2 * liste    :", 2 * u)

x = np.array(u)
y = np.array(v)
print("dizi + dizi  :", x + y)
print("2 * dizi     :", 2 * x)
print("dizi * dizi  :", x * y)
```

Çıktı:

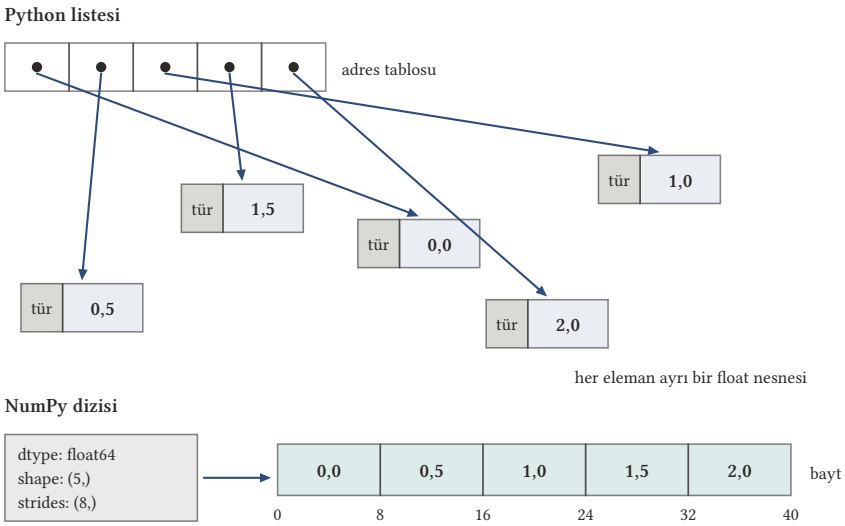
```
liste + liste: [1, 2, 3, 10, 20, 30]
2 * liste    : [1, 2, 3, 1, 2, 3]
dizi + dizi  : [11 22 33]
2 * dizi     : [2 4 6]
dizi * dizi  : [10 40 90]
```

`import numpy as np` satırı NumPy'ı yükler ve ona kısa `np` adını verir; bu kısaltma bütün Python dünyasında ortak bir alışkanlıktır. `np.array` bir listeyi diziye çevirir. Listelerde `+` birleştirme, `2 *` tekrar anlamına gelirken diziler bu işlemleri **bileşen bileşen** (elementwise) yapar: `x + y` vektör toplamı, `2 * x` skalerle çarpımdır. `x * y` ise matris çarpımı değil, karşılıklı bileşenlerin çarpımıdır; matris çarpımı için kullanılan `@` işlecini NumPy ile Lineer Cebir bölümünde göreceğiz.

Tanım 5.1 (NumPy Dizisi). Bir **NumPy dizisi** (`ndarray`, n-dimensional array), aynı veri türündeki sabit sayıda elemanın bellekteki tek bir veri bloğunda tutulduğu ve bir **şekil** (shape) yardımıyla bir, iki ya da daha çok indeksle okunduğu nesnedir.

Yani bir NumPy dizisini listeden üç özellik ayırır: bütün elemanları aynı türdendir, eleman sayısı dizi kurulurken belirlenir ve elemanlar bellekte tek bir blokta durur; yeni kurulan bir dizide yan yana dizilmişlerdir. Liste ise bellekte dağınık duran nesnelerin adreslerini tutan bir tablodur. Aşağıdaki şekil iki düzeni aynı beş sayı için karşılaştırıyor.

 CC BY-NC-SA 4.0



Şekil 5.1: Aynı beş sayı (0,0; 0,5; 1,0; 1,5 ve 2,0) iki biçimde. Üstte Python listesi: her eleman bellekte başka bir yerde duran, kendi tür ve sayaç bilgisini taşıyan bir `float` nesnesidir; liste yalnız bu nesnelerin adreslerini tutar. Altta `float64` dizisi: sayılar 8'er baytlık bitişik bir blokta yan yana durur; `dtype`, `shape` ve `strides` bu bloğun nasıl okunacağını söyler.

Bu düzenin somut bir sonucu bellek kullanımıdır. Bin ondalık sayıyı liste ve dizi olarak saklayıp kapladıkları yeri ölçelim. `sys.getsizeof` bir nesnenin bayt cinsinden boyutunu verir; listenin toplam boyutu, adres tablosu ile tek tek `float` nesnelerinin boyutlarının toplamıdır.

```
import sys
import numpy as np

n = 1000
lst = [k / 2 for k in range(n)]
arr = np.array(lst)

# liste: adres tablosu + her bir float nesnesi
total = sys.getsizeof(lst) + sum(sys.getsizeof(t) for t in lst)
print("liste:", total, "bayt")
print("dizi :", arr.nbytes, "bayt")
print("eleman başına:", arr.itemsize, "bayt")
```

Çıktı:

```
liste: 32856 bayt
dizi : 8000 bayt
eleman başına: 8 bayt
```

Liste aynı sayılar için dört kattan fazla yer kaplar: her `float` nesnesi 8 baytlık değerin yanında tür ve sayaç bilgisi de taşır, tablo da her nesne için ayrıca bir adres tutar. Dizinin `nbytes` niteliği veriye ayrılan bayt sayısını, `itemsize` niteliği de bir elemanın boyutunu verir. Bitişik düzenin asıl kazancı ise hızdır: NumPy işlemlerini derlenmiş C döngüleriyle, işlemciye yan yana duran sayıları besleyerek yapar. Bu farkı Vektörizasyon ve Broadcasting bölümünde ölçeceğiz.

5.2 Dizi Kurmak

NumPy'da dizi kurmanın iki yolu vardır: var olan bir Python listesini dönüştürmek ya da diziyi doğrudan bir kurala göre üretmek.
`np.array` bir listeyi bir boyutlu, iç içe bir listeyi iki boyutlu diziye çevirir; iç içe listede her iç liste matrisin bir satırı olur. Her dizinin `ndim` (boyut sayısı), `shape` (şekil) ve `size` (eleman sayısı) nitelikleri vardır.

 CC BY-NC-SA 4.0

```
import numpy as np

x = np.array([3, 1, 4, 1, 5])
A = np.array([[1, 2, 3],
              [4, 5, 6]])
print(x)
print(A)
print("x:", x.ndim, x.shape, x.size)
print("A:", A.ndim, A.shape, A.size)
```

Çıktı:

```
[3 1 4 1 5]
[[1 2 3]
 [4 5 6]]
x: 1 (5,) 5
A: 2 (2, 3) 6
```

Tanım 5.2 (Dizinin Şekli ve Eksenleri). Bir dizinin **şekli** (shape), her indeksin kaç değer alabileceğini sırayla veren $(n_0, n_1, \ldots, n_{d-1})$ demetidir. d sayısı dizinin **boyutu** (`ndim`), k-ıncı indeksin değiştiği yön de dizinin k numaralı **eksenidir** (axis). Dizinin eleman sayısı (`size`) $n_0 n_1 \cdots n_{d-1}$ çarpımıdır.

Yani $m \times n$ bir matris, şekli `(m, n)` olan iki boyutlu bir dizidir: 0 ekseni satır numarasını, 1 ekseni sütun numarasını sayar. Bir boyutlu bir dizinin şekli `(5,)` gibi tek elemanlı bir demettir; sondaki virgül, tek elemanlı demeti parantez içindeki bir sayıdan ayırır. İndeksler Python'ın her yerinde olduğu gibi 0'dan başlar: matematikteki a_{11} bileşeni NumPy'da `A[0, 0]`, genel olarak a_{ij} bileşeni `A[i-1, j-1]` olur.
Sık kullanılan diziler için hazır fonksiyonlar vardır. `np.zeros` ve `np.ones` istenen şekilde sıfırlardan ve birlerden, `np.full` tek bir sabitten oluşan bir dizi kurar. `np.eye(n)` $n \times n$ birim matrisi (bkz. Lineer Cebir), `np.diag` ise verilen köşegenle bir köşegen matris (bkz. Lineer Cebir) üretir. `np.zeros_like(B)`, `B` ile aynı şekil ve türde bir sıfır dizisi verir.

```
import numpy as np

print(np.zeros(4))
print(np.ones((2, 3)))
print(np.full((2, 2), 7))
print(np.eye(3))
print(np.diag([1, 2, 3]))
print(np.zeros_like(np.diag([1, 2, 3])))
```

Çıktı:

```
[0. 0. 0. 0.]
[[1. 1. 1.]
 [1. 1. 1.]]
[[7 7]
 [7 7]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[1 0 0]
 [0 2 0]
 [0 0 3]]
[[0 0 0]
 [0 0 0]
 [0 0 0]]
```

`zeros`, `ones` ve `eye` kendiliğinden ondalık (`float64`) dizi kurar; çıktıdaki `0.` ve `1.` yazımı bunun işaretidir. Şekil tek bir demet olarak verilir: `np.zeros((2, 3))` doğrudur, `np.zeros(2, 3)` ise hata verir, çünkü ikinci argüman veri türüne ayrılmıştır. Bir de `np.empty` vardır: belleği ayırır ama doldurmaz, bu yüzden

 CC BY-NC-SA 4.0

o bellek bölgesinde önceden ne varsa onu gösterir. Yalnız bütün elemanların hemen ardından yazılacağı durumlarda kullanılır.

5.3 Eşit Aralıklı Noktalar

Sayısal hesapta en sık kurduğumuz dizi, bir aralığa eşit adımlarla yayılmış noktalardır: sayısal integral, sayısal türev ve grafik çizimi hep böyle bir ızgara üzerinde yapılır. NumPy bunun için iki fonksiyon sunar; biri adımı, öteki nokta sayısını esas alır.

Tanım 5.3 (Adımla Verilen Izgara). `np.arange(a, b, h)`, a'dan başlayıp h adımıyla ilerleyen ve b'ye varmadan duran

$$a,\ a+h,\ a+2h,\ \ldots,\ a+(N-1)h, \qquad N=\left\lceil \frac{b-a}{h} \right\rceil$$

dizisini verir. Bitiş noktası b dahil değildir. Tek argümanla yazılan `np.arange(n)` ise $0, 1, \ldots, n-1$ dizisidir.

Yani `arange`, Python'ın `range` fonksiyonunun dizi döndüren ve ondalık adım da kabul eden biçimidir. Eleman sayısı N, aralığın uzunluğunun adıma oranı yukarı yuvarlanarak bulunur.

Tanım 5.4 (Nokta Sayısıyla Verilen Izgara). `np.linspace(a, b, N)`, $[a, b]$ aralığını $N-1$ eşit parçaya bölen

$$x_k = a + k\,h, \qquad h=\frac{b-a}{N-1}, \qquad k=0,1,\ldots,N-1$$

noktalarını verir. İki uç nokta da dahildir.

Yani `linspace` ile adımı değil nokta sayısını seçeriz; adım kendiliğinden hesaplanır ve istenirse `retstep=True` argümanıyla geri alınır. Sayısal integralde ve grafik çiziminde uç noktaların da hesaba girmesi gerektiğinden çoğu zaman tercih edilen budur.

```
import numpy as np

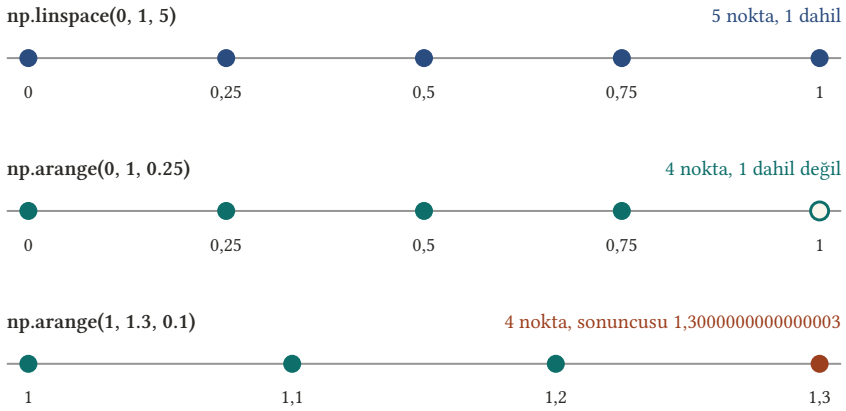
print(np.arange(5))
print(np.arange(2, 11, 3))
print(np.arange(0, 1, 0.25))
print(np.linspace(0, 1, 5))
x, h = np.linspace(0, 1, 5, retstep=True)
print("adım:", h)
```

Çıktı:

```
[0 1 2 3 4]
[2 5 8]
[0.   0.25 0.5  0.75]
[0.   0.25 0.5  0.75 1.  ]
adım: 0.25
```

`np.arange(2, 11, 3)` 2'den başlayıp 3'er artar ve 11'e varmadan durur. Aynı $[0,1]$ aralığı ve aynı 0,25 adımı için `linspace` beş, `arange` dört nokta verir; fark, bitiş noktasının alınıp alınmamasıdır. Aşağıdaki şekil bu iki ızgarayı ve hemen ardından ele alacağımız bir tuzağı gösteriyor.

 CC BY-NC-SA 4.0



Şekil 5.2: Üstte `linspace` nokta sayısıyla çalışır ve iki ucu da alır. Ortada `arange` aynı adımla 1'e varmadan durur. Altta ondalık adımlı `arange`: eleman sayısı (1,3 − 1)/0,1 oranının yukarı yuvarlanmasıyla belirlenir; bu oran yuvarlama hatası yüzünden 3 değil 3,0000000000000004 çıktığı için dördüncü bir nokta eklenir ve o nokta 1,3'ün biraz sağındadır.

⚠ Ondalık adımla arange kullanmayın

`arange` eleman sayısını $(b-a)/h$ oranından hesaplar. 0,1 gibi ondalık adımlar ikili sistemde tam temsil edilemediği için (bkz. Nümerik Analiz) bu oran bir tamsayıya çok yakın ama ondan biraz büyük çıkabilir; o zaman yukarı yuvarlama fazladan bir nokta ekler.

```
import numpy as np

x = np.arange(1, 1.3, 0.1)
print(x)
print(x[-1])                # son eleman 1.3'ten büyük
print((1.3 - 1) / 0.1)      # eleman sayısını belirleyen oran
print(np.linspace(1, 1.2, 3))
```

Çıktı:

```
[1.  1.1 1.2 1.3]
1.3000000000000003
3.0000000000000004
[1.  1.1 1.2]
```

$(1{,}3-1)/0{,}1$ oranı 3 yerine 3,0000000000000004 çıktığı için dizi dört elemanlıdır ve son eleman $b=1{,}3$ sınırının biraz sağındadır. Ondalık adım gerektiğinde `linspace` kullanıp nokta sayısını kendiniz verin; `arange` fonksiyonunu tamsayı adımlara saklayın.

Örnek 5.1 (Bir Polinomun Değer Tablosu). $f(x)=x^3-2x$ polinomunun $[-2,2]$ aralığındaki eşit aralıklı 9 noktada değer tablosunu çıkarınız ve tablodan f'nin köklerinin nerede olduğunu okuyunuz.

Çözüm.
Dokuz nokta istendiği için `np.linspace(-2, 2, 9)` kullanırız; adım $h=4/8=0{,}5$'tir. Bir dizi üzerine yazılan `x**3 - 2 * x` ifadesi her elemana ayrı ayrı uygulanır; bunun ayrıntılarını bir sonraki bölümde göreceğiz. `zip` iki diziyi eleman eleman eşler, f-string de sayıları hizalı basar.

```
import numpy as np

x = np.linspace(-2, 2, 9)
y = x**3 - 2 * x
```

 CC BY-NC-SA 4.0

```
for xk, yk in zip(x, y):
    print(f"{xk:6.2f} {yk:8.3f}")
```

Çıktı:

```
 -2.00   -4.000
 -1.50   -0.375
 -1.00    1.000
 -0.50    0.875
  0.00    0.000
  0.50   -0.875
  1.00   -1.000
  1.50    0.375
  2.00    4.000
```

f sürekli olduğundan işaret değiştirdiği her aralıkta bir kökü vardır (bkz. Nümerik Analiz). Tablo $f(0) = 0$ değerini doğrudan gösteriyor. Ayrıca f, $[-1{,}5; -1]$ aralığında $-0{,}375$'ten 1'e, $[1; 1{,}5]$ aralığında da -1'den $0{,}375$'e geçerek işaret değiştiriyor. Gerçekten $x^3 - 2x = x\,(x^2 - 2)$ olduğundan kökler 0 ve $\pm\sqrt{2} \approx \pm 1{,}414$'tür. Böyle bir tablo, ikiye bölme metoduna başlangıç aralığı seçmenin en basit yoludur. ■

5.4 Veri Türü

Bir dizinin bütün elemanları aynı türden olduğundan tür, tek tek elemanların değil dizinin bir özelliğidir ve `dtype` niteliğinde saklanır.

Tanım 5.5 (Veri Türü). Bir dizinin **veri türü** (`dtype`), elemanlarının bellekte nasıl kodlandığını ve her birinin kaç bayt tuttuğunu belirler. En sık kullanılanlar `int64` (64 bitlik işaretli tamsayı), `float64` (IEEE 754 çift duyarlıklı kayan noktalı sayı), `complex128` (gerçel ve sanal kısmı birer `float64` olan karmaşık sayı) ve `bool` (`True` ya da `False`) türleridir.

Yani `float64` dizisinin her elemanı, Python'ın `float` türüyle aynı 64 bitlik makine sayısıdır (bkz. Nümerik Analiz). `int64` ise Python'ın sınırsız `int` türünden farklıdır: yalnız -2^{63} ile $2^{63} - 1$ arasındaki tamsayıları tutabilir.
`np.array` türü elemanlara bakarak kendisi seçer. Elemanlar karışık türdeyse hepsini taşıyabilen ortak türe yükseltir: tamsayılarla ondalık sayılar birlikte `float64`, bunlarla karmaşık sayılar birlikte `complex128` olur. Tamsayıdan `float64` türüne geçiş ancak mutlak değeri 2^{53}'ü aşmayan tamsayılar için kesindir; daha büyük tamsayıların son basamakları yuvarlanabilir (bkz. Python ile İlk Adımlar). Türü `dtype` argümanıyla kendimiz de verebilir, var olan bir diziyi `astype` ile başka bir türe çevirebiliriz.

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([1, 2.5, 3])
c = np.array([1, 2 + 1j])
print(a, a.dtype)
print(b, b.dtype)
print(c, c.dtype)

d = np.array([1, 2, 3], dtype=float)
print(d, d.dtype)
print(b.astype(int))     # ondalık kısım atılır
```

Çıktı:

```
[1 2 3] int64
[1.  2.5 3. ] float64
```

 CC BY-NC-SA 4.0

```
[1.+0.j 2.+1.j] complex128
[1. 2. 3.] float64
[1 2 3]
```

`astype` yeni türde bir kopya üretir. Ondalıktan tamsayıya geçerken ondalık kısım yuvarlanmaz, atılır: 2,5 değeri 2 olur. Türlerin sınırlarını `np.iinfo` (tamsayılar) ve `np.finfo` (ondalık sayılar) verir.

```
import numpy as np

print(np.iinfo(np.int64).max)
print(np.finfo(np.float64).eps)
print(np.finfo(np.float64).max)
```

Çıktı:

```
9223372036854775807
2.220446049250313e-16
1.7976931348623157e+308
```

İlk sayı $2^{63} - 1$, ikincisi `float64` için makine epsilonu olan 2^{-52} (bkz. Python ile İlk Adımlar), üçüncüsü de temsil edilebilen en büyük sayıdır (bkz. Nümerik Analiz).

⚠ Tamsayı dizilerinin iki tuzağı

Taşma sessizdir. Python'ın `int` türü istendiği kadar büyüyebilir; ama `int64` dizisindeki bir sonuç $2^{63} - 1$'i aşarsa NumPy hata vermez. Sonuç 2^{64} modülüne göre sarılır ve anlamsız, hatta negatif bir sayı çıkar (bkz. Nümerik Analiz).

```
import numpy as np

a = np.array([10])
print(a**18, a**19)      # 10**19 int64'e sığmaz
print(10**19)            # Python int'i sınırsızdır
```

Çıktı:

```
[1000000000000000000] [-8446744073709551616]
10000000000000000000
```

Tamsayı dizisine yazılan ondalık sayı kesilir. Bir dizinin türü kurulurken belirlenir ve sonradan değişmez. Tamsayılarla kurulmuş bir diziye ondalık bir değer atanırsa değer sessizce tamsayıya kesilir.

```
import numpy as np

p = np.array([1, 2, 3])        # int64 dizisi
p[0] = 0.75                    # 0.75 sessizce 0'a kesilir
print(p)
q = np.array([1.0, 2, 3])      # float64 dizisi
q[0] = 0.75
print(q)
```

Çıktı:

```
[0 2 3]
[0.75 2.   3.  ]
```

Bir iterasyonun ara değerlerini saklayacağınız diziyi bu yüzden baştan ondalık kurun: `np.zeros(n)` ya da `np.array([1.0, 2, 3])` gibi.

 CC BY-NC-SA 4.0

5.5 Şekil ve Yeniden Şekillendirme

Aynı elemanlar farklı şekillerde okunabilir: on iki sayı bir satır, 3×4 bir matris ya da $2 \times 2 \times 3$ bir tablo olabilir. Dizinin verisine dokunmadan şeklini değiştirmek NumPy'da çok sık başvurulan bir işlemdir.

Tanım 5.6 (Yeniden Şekillendirme). `a.reshape(s)`, `a` dizisinin elemanlarını değiştirmeden onları yeni `s` şekliyle okuyan diziyi verir; bunun için `s` şeklindeki sayıların çarpımı `a.size` değerine eşit olmalıdır. Elemanlar indeks demetlerinin sözlük sırasıyla, yani **satır öncelikli** (row-major, C order) sırayla yerleşir: son indeks en hızlı, ilk indeks en yavaş değişir.

Yani `np.arange(12).reshape(3, 4)` önce ilk satırı 0, 1, 2, 3 ile, sonra ikinci satırı 4, 5, 6, 7 ile doldurur; tıpkı bir metni satır satır okur gibi. Şekil demetindeki sayılardan biri `-1` yazılabilir; NumPy onu eleman sayısından kendisi hesaplar.

```
import numpy as np

a = np.arange(12)
A = a.reshape(3, 4)
print(A)
print(a.reshape(2, -1))     # -1 yerine 12 / 2 = 6 hesaplanır
T = a.reshape(2, 2, 3)
print(T.shape, T[1, 0, 2])
print(A.T)                  # transpoze, şekli (4, 3)
print(A.ravel())            # yeniden tek boyut
```

Çıktı:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
(2, 2, 3) 8
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

`A.T`, A matrisinin transpozesidir (bkz. Lineer Cebir); `ravel` ise diziyi satır öncelikli sırayla yeniden tek boyuta indirir. Üç boyutlu `T` dizisinde `T[1, 0, 2]` elemanının 8 çıkmasının nedeni aşağıdaki önermedir.

Önerme 5.1 (Satır Öncelikli Sırada Konum). *Şekli $(n_0, n_1, \ldots, n_{d-1})$ olan bir dizinin $(i_0, i_1, \ldots, i_{d-1})$ indeksli elemanı, satır öncelikli sırada (sayma 0'dan başlar)*

$$p = i_0\, n_1 n_2 \cdots n_{d-1} + i_1\, n_2 \cdots n_{d-1} + \cdots + i_{d-2}\, n_{d-1} + i_{d-1}$$

konumundadır. Özel olarak $m \times n$ bir matrisin (i, j) elemanı $p = i\,n + j$ konumundadır.

> ***İspat.***
> *Bir elemanın konumu, sözlük sırasında ondan önce gelen indeks demetlerinin sayısıdır.*
> *Önce matris durumunu görelim. (i, j) demetinden önce, ilk indeksi $0, 1, \ldots, i-1$ olan i tam satır gelir; her satırda n eleman olduğundan bunlar $i\,n$ tanedir. İlk indeksi i olanlar arasında da $(i, 0), \ldots, (i, j-1)$ olmak üzere j demet (i, j)'den önce gelir. Toplam $p = i\,n + j$ eder.*
> *Genel durumu d üzerinden tümevarımla kuralım. $d = 1$ için $p = i_0$'dır. $d \geq 2$ olsun. İlk indeksi i_0'dan küçük olan demetler i_0 tane tam alt dizi oluşturur ve her biri $n_1 n_2 \cdots n_{d-1}$ demet içerir. İlk indeksi i_0 olan demetler ise şekli $(n_1, \ldots, n_{d-1})$ olan bir alt diziyi oluşturur ve kendi aralarında yine sözlük sırasıyla dizilir. Tümevarım hipotezine göre $(i_1, \ldots, i_{d-1})$ demetinden önce bu alt dizide*
>
> $$i_1\, n_2 \cdots n_{d-1} + \cdots + i_{d-2}\, n_{d-1} + i_{d-1}$$

 CC BY-NC-SA 4.0

> *demet vardır. İki sayının toplamı iddiadaki p'dir.* ■

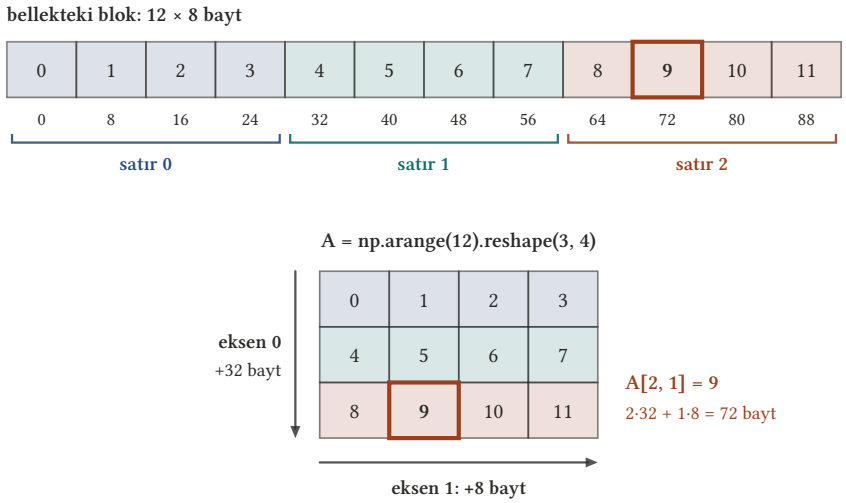
Bu konum bilgisi bellekte de aynen geçerlidir. NumPy her eksen için bir **adım** (stride) tutar: o eksende indeksi bir artırmak için bellekte kaç bayt ilerleneceği. 8 baytlık elemanlardan oluşan 3×4 bir matriste bir sütun sağa gitmek 8, bir satır aşağı inmek $4 \cdot 8 = 32$ bayttır. Böylece `A[i, j]` elemanı bloğun başından $32\,i + 8\,j = 8\,(4i + j)$ bayt ötededir; bu, Önerme 5.1 ile bulunan konumun tam 8 katıdır.

```
import numpy as np

A = np.arange(12).reshape(3, 4)
print("strides:", A.strides)
i, j = 2, 1
print("konum:", i * 4 + j, "eleman:", A[i, j])
offset = i * A.strides[0] + j * A.strides[1]
print("bayt:", offset, "=", offset // A.itemsize, "eleman")
```

Çıktı:

```
strides: (32, 8)
konum: 9 eleman: 9
bayt: 72 = 9 eleman
```



Şekil 5.3: `np.arange(12).reshape(3, 4)` dizisinin verisi bellekte tek bir bloktur; şekil yalnız bu bloğun nasıl okunacağını belirler. Satır öncelikli sırada önce 0. satırın dört elemanı, sonra 1. ve 2. satırınkiler gelir. Bir sütun sağa gitmek 8, bir satır aşağı inmek $4 \cdot 8 = 32$ bayt ilerlemektir; bu iki sayı dizinin `strides` değeridir ve `A[2, 1]` elemanı bloğun başından $2 \cdot 32 + 1 \cdot 8 = 72$ bayt ötede başlar.

Bir boyutlu bir dizi ne satır ne de sütun matrisidir; şekli yalnızca `(3,)` olur ve transpozesi kendisidir. Lineer cebirdeki satır ve sütun matrisleri (bkz. Lineer Cebir) gerektiğinde `reshape` ile açıkça kurulur.

```
import numpy as np

x = np.array([5, 6, 0])
row = x.reshape(1, -1)
col = x.reshape(-1, 1)
print(x.shape, row.shape, col.shape)
print(col)
print(x.T.shape)        # 1B dizide transpoze bir şey değiştirmez
```

Çıktı:

```
(3,) (1, 3) (3, 1)
[[5]
 [6]
 [0]]
(3,)
```

Örnek 5.2 (Üç Boyutlu Bir Dizide Konum). `T = np.arange(24).reshape(2, 3, 4)` dizisinin `T[1, 0, 2]` elemanını, diziyi yazdırmadan, Önerme 5.1 yardımıyla bulunuz.

Çözüm.
Şekil $(n_0, n_1, n_2) = (2, 3, 4)$, indeks de $(i_0, i_1, i_2) = (1, 0, 2)$'dir. Önermeye göre

$$p = i_0\, n_1 n_2 + i_1\, n_2 + i_2 = 1 \cdot 12 + 0 \cdot 4 + 2 = 14$$

olur. `np.arange(24)` dizisinin 14 numaralı konumunda 14 sayısı durduğundan `T[1, 0, 2]` elemanı 14'tür. Kodla doğrulayalım; `T[1]`, ilk indeksi 1 olan 3×4 alt diziyi verir.

```
import numpy as np

T = np.arange(24).reshape(2, 3, 4)
n0, n1, n2 = T.shape
i0, i1, i2 = 1, 0, 2
p = i0 * n1 * n2 + i1 * n2 + i2
print("formül:", p, " dizi:", T[i0, i1, i2])
print(T[1])
```

Çıktı:

```
formül: 14  dizi: 14
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

14, bu alt dizinin ilk satırının üçüncü elemanıdır. ■

5.6 İndeksleme ve Dilimleme

Listelerde bir elemana `x[i]` ile, bir parçaya `x[başla:dur:adım]` ile ulaşıyorduk (Veri Yapıları ve Comprehension). NumPy aynı sözdizimini her eksen için ayrı ayrı, virgülle ayırarak kullanır.

```
import numpy as np

A = np.arange(20).reshape(4, 5)
print(A)
print("A[2, 3]  =", A[2, 3])
print("A[-1, 0] =", A[-1, 0])    # son satır, ilk sütun
print("A[1]     =", A[1])        # ikinci satırın tamamı
print("A[:, 2]  =", A[:, 2])     # üçüncü sütunun tamamı
```

Çıktı:

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
A[2, 3]  = 13
A[-1, 0] = 15
```

 CC BY-NC-SA 4.0

```
A[1]     = [5 6 7 8 9]
A[:, 2]  = [ 2  7 12 17]
```

`A[2, 3]` üçüncü satırın dördüncü elemanıdır, yani matematikteki a_{34}. Negatif indeksler sondan sayar. Tek bir indeks verilirse ilk eksen seçilir: `A[1]` ikinci satırın tamamıdır. `A[:, 2]` yazımındaki `:` işareti "bu eksende hepsi" demektir. `A[2][3]` yazımı da aynı elemanı verir, ama önce bütün satırı ayırıp sonra içinden seçtiği için `A[2, 3]` tercih edilir.

Tanım 5.7 (Çok Boyutlu Dilimleme). d boyutlu bir dizi, köşeli parantez içinde virgülle ayrılmış en çok d indeksle seçilir. k-ıncı yerdeki indeks ya bir tamsayı ya da `başla:dur:adım` biçiminde bir **dilimdir** (slice). Tamsayı, k ekseninde tek bir konumu seçer ve bu ekseni sonuçtan kaldırır. Dilim, k ekseninde `başla`, `başla + adım`, ... konumlarını `dur` değerine varmadan seçer ve ekseni korur. Yazılmayan sondaki eksenler için `:` (hepsi) varsayılır.

Yani tamsayı indeks boyutu düşürür, dilim düşürmez: `A[1, :]` (kısaca `A[1]`) ikinci satırı şekli `(5,)` olan bir boyutlu dizi olarak, `A[1:2, :]` ise aynı satırı şekli `(1, 5)` olan bir matris olarak verir. Dilimin sınırları listelerdeki gibidir: `başla` dahil, `dur` hariçtir; yazılmayan `başla` 0, `dur` eksenin sonu, `adım` 1 demektir. İki boyutlu bir dilim, satırları ve sütunları eşit aralıklarla seçilmiş bir alt matristir (bkz. Lineer Cebir).

A[1]

0	1	2	3	4
5	**6**	**7**	**8**	**9**
10	11	12	13	14
15	16	17	18	19

şekil (5,)

A[:, 2]

0	1	**2**	3	4
5	6	**7**	8	9
10	11	**12**	13	14
15	16	**17**	18	19

şekil (4,)

A[1:3, 1:4]

0	1	2	3	4
5	**6**	**7**	**8**	9
10	**11**	**12**	**13**	14
15	16	17	18	19

şekil (2, 3)

A[::2, ::2]

0	1	**2**	3	**4**
5	6	7	8	9
10	11	**12**	13	**14**
15	16	17	18	19

şekil (2, 3)

Şekil 5.4: `A = np.arange(20).reshape(4, 5)` dizisinden dört seçim. Tek bir tamsayı indeks (`A[1]`, `A[:, 2]`) o ekseni ortadan kaldırır ve bir boyutlu bir dizi verir; dilimler eksenleri korur. `A[::2, ::2]` her iki eksende ikişer atlayarak seçer.

```
import numpy as np

A = np.arange(20).reshape(4, 5)
print(A[1:3, 1:4])
print(A[::2, ::2])
```

 CC BY-NC-SA 4.0

```
print(A[1, :].shape, A[1:2, :].shape)
print(A[::-1, 0])          # ilk sütun, ters sırada
```

Çıktı:

```
[[ 6  7  8]
 [11 12 13]]
[[ 0  2  4]
 [10 12 14]]
(5,) (1, 5)
[15 10  5  0]
```

Negatif adım sırayı ters çevirir: `A[::-1, 0]` ilk sütunu aşağıdan yukarıya okur. Dilimlere atama da yapılabilir: `A[0, :] = 0` ilk satırı sıfırlar, `A[:, -1] = [1, 2, 3, 4]` son sütunu verilen değerlerle doldurur. Atanan tek bir sayıysa seçilen her konuma yazılır.

Örnek 5.3 (Satranç Tahtası Deseni). 8×8 bir tamsayı matrisinde $i + j$ toplamı tek olan (i, j) konumlarına 1, ötekilere 0 yazınız (indeksler 0'dan başlar). Matrisi eleman eleman dolaşan bir döngüyle değil, iki dilim atamasıyla kurunuz.

Çözüm.
$i + j$ toplamının tek olması için indekslerden biri çift, öteki tek olmalıdır. Çift indeksler `::2` dilimiyle (0, 2, 4, 6), tek indeksler `1::2` dilimiyle (1, 3, 5, 7) seçilir. Dolayısıyla 1 yazılacak konumlar iki parçadır: çift satırların tek sütunları ve tek satırların çift sütunları.

```
import numpy as np

Z = np.zeros((8, 8), dtype=int)
Z[::2, 1::2] = 1     # çift satır, tek sütun
Z[1::2, ::2] = 1     # tek satır, çift sütun
print(Z)
```

Çıktı:

```
[[0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]
 [0 1 0 1 0 1 0 1]
 [1 0 1 0 1 0 1 0]]
```

Sol üst köşedeki $(0, 0)$ konumunda $0 + 0 = 0$ çift olduğundan 0, onun sağındaki $(0, 1)$ konumunda 1 durur. ■

Örnek 5.4 (Bir Matrisi Bloklara Ayırmak). M, satır satır $1, 2, \ldots, 16$ sayılarıyla doldurulmuş 4×4 matris olsun. M matrisini dört tane 2×2 bloğa ayırınız:

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}$$

Çözüm.
Blok matris kavramı için bkz. Lineer Cebir. Satırlar `:2` (0 ve 1) ile `2:` (2 ve 3) dilimlerine, sütunlar da aynı biçimde ikiye ayrılır; her blok bir satır diliminin ve bir sütun diliminin kesişimidir.

```
import numpy as np
```

 CC BY-NC-SA 4.0

```
M = np.arange(1, 17).reshape(4, 4)
M11, M12 = M[:2, :2], M[:2, 2:]
M21, M22 = M[2:, :2], M[2:, 2:]
print(M)
print(M11)
print(M12)
print(M21)
print(M22)
```

Çıktı:

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
[[1 2]
 [5 6]]
[[3 4]
 [7 8]]
[[ 9 10]
 [13 14]]
[[11 12]
 [15 16]]
```

Örneğin M_{12}, ilk iki satırın son iki sütunudur: $\begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix}$. Bu bloklar kurulurken hiçbir sayı kopyalanmaz; bunun ne demek olduğunu bir sonraki kısımda görüyoruz. ■

5.7 Görünüm ve Kopya

Dilimleme hızlıdır, çünkü NumPy seçilen elemanları kopyalamaz; yeni dizi eski dizinin belleğinin bir bölümüne bakar. Bu, listelerden alıştığımız davranışın tersidir ve dikkat ister.

```
import numpy as np

a = np.arange(6)
s = a[1:4]          # dizi dilimi: görünüm
s[0] = 100
print("a   =", a)

lst = list(range(6))
t = lst[1:4]        # liste dilimi: kopya
t[0] = 100
print("lst =", lst)
```

Çıktı:

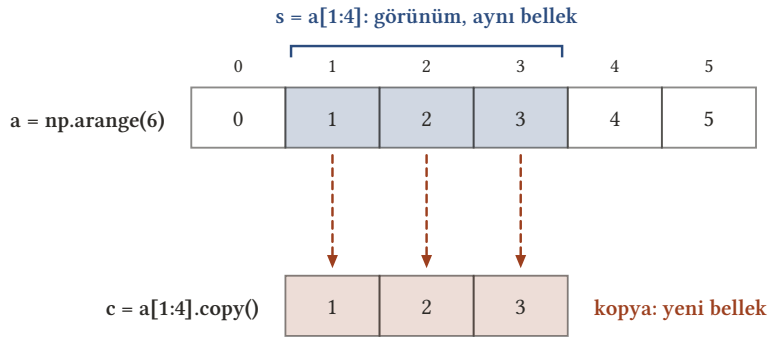
```
a   = [  0 100   2   3   4   5]
lst = [0, 1, 2, 3, 4, 5]
```

Liste dilimi yeni bir liste kurduğu için `lst` değişmedi. Dizi dilimi `s` ise `a` ile aynı belleği paylaşır; bu yüzden `s[0] = 100` ataması doğrudan `a[1]` elemanını değiştirdi.

Tanım 5.8 (Görünüm ve Kopya). Bir dizinin **görünümü** (view), kendi verisi olmayan ve başka bir dizinin belleğinin bir bölümünü kendi şekli ve adımlarıyla okuyan dizidir. **Kopya** (copy) ise verisi yeni bir bellek bloğuna yazılmış, öncekinden bağımsız dizidir.

 CC BY-NC-SA 4.0

Yani bir görünüm üzerinden yapılan atama asıl diziyi de değiştirir, bir kopya üzerinden yapılan değiştirmez. Temel dilimleme, `.T` ve mümkün olduğunda `reshape` ile `ravel` görünüm verir; `copy()`, `astype` ve aşağıda göreceğimiz fancy ve boolean indeksleme ise kopya üretir.



Şekil 5.5: `s = a[1:4]` kendi verisi olmayan bir görünümdür: `a`'nın 1, 2 ve 3 numaralı hücrelerine bakar, bu yüzden `s[0] = 100` ataması `a[1]`'i değiştirir. `c = a[1:4].copy()` ise aynı değerleri yeni bir bloğa yazar; `c` üzerinde yapılan değişiklik `a`'ya ulaşmaz.

İki dizinin bellek paylaşıp paylaşmadığını `np.shares_memory` söyler. Bir görünümün `base` niteliği de verisini ödünç aldığı diziyi gösterir; kendi verisi olan bir dizide bu nitelik `None` olur.

```
import numpy as np

a = np.arange(6)
s = a[1:4]
c = a[1:4].copy()
print(np.shares_memory(a, s), s.base is a)
print(np.shares_memory(a, c), c.base is None)
c[0] = -1
print("a =", a, " c =", c)
```

Çıktı:

```
True True
False True
a = [0 1 2 3 4 5]  c = [-1  2  3]
```

 CC BY-NC-SA 4.0

⚠ Eski değeri saklamak için copy kullanın

Listelerde `x[:]` bir kopya verir ve eski değeri saklamanın alışılmış yoludur. NumPy dizisinde ise `x[:]` bütün diziye bakan bir görünümdür; dizi yerinde güncellenince "eski" değer de değişir.

```
import numpy as np

x = np.array([1.0, 2.0, 3.0])
x_old = x[:]           # listelerdeki gibi kopya sanılır
x[:] = x / 2           # yerinde güncelleme
print(x_old)           # x_old da değişti

y = np.array([1.0, 2.0, 3.0])
y_old = y.copy()       # gerçek kopya
y[:] = y / 2
print(y_old)
```

Çıktı:

```
[0.5 1.  1.5]
[1. 2. 3.]
```

İterasyonlarda bir önceki adımın değerini `x_old = x.copy()` ile saklayın. `x = x / 2` gibi bir atama yeni bir dizi kurup `x` adını ona bağlar ve eski diziye dokunmaz; tehlike, `x[:] = ...` ya da `x += ...` gibi yerinde güncellemelerdedir.

Örnek 5.5 (Transpoze Bir Görünümdür). `A = np.zeros((2, 3), dtype=int)` ve `B = A.T` olsun. `B[2, 0] = 5` atamasından sonra `A` matrisi ne olur?

Çözüm.
`A.T` veriyi kopyalamaz; aynı belleği eksenlerin yerini değiştirerek okur. Yani `B[i, j]` ile `A[j, i]` aynı bellek hücresidir. Dolayısıyla `B[2, 0] = 5` ataması `A[0, 2]` elemanını 5 yapar.

```
import numpy as np

A = np.zeros((2, 3), dtype=int)
B = A.T
B[2, 0] = 5
print(A)
print(np.shares_memory(A, B))
```

Çıktı:

```
[[0 0 5]
 [0 0 0]]
True
```

Matematikte $B = A^t$ yazıp B matrisini değiştirmek A'yı etkilemez, çünkü B yeni bir matristir. NumPy'da bağımsız bir transpoze gerekiyorsa `A.T.copy()` yazılır. ■

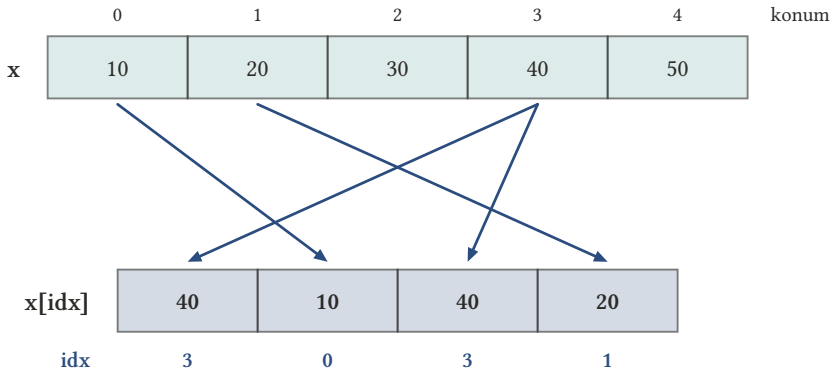
5.8 Fancy İndeksleme

Dilimler yalnızca eşit aralıklı konumları seçebilir. Konumları istediğimiz sırada, gerekirse tekrarlayarak seçmek için indeks olarak bir tamsayı listesi ya da dizisi veririz.

Tanım 5.9 (Tamsayı Dizisiyle İndeksleme). `x` bir boyutlu bir dizi, `idx` de tamsayılardan oluşan bir dizi ya da liste olsun. `x[idx]`, k-ıncı elemanı `x[idx[k]]` olan ve şekli `idx` ile aynı olan **yeni** bir dizidir. Bu seçime **fancy indeksleme** (fancy indexing) denir.

 CC BY-NC-SA 4.0

Yani `x` konumlardan değerlere, `idx` de konumlardan konumlara giden birer fonksiyon olarak düşünülürse `x[idx]` dizisi $x \circ \text{idx}$ bileşkesidir. İndeksler tekrar edebilir, sıraları serbesttir ve sonuç her zaman bir kopyadır.



Şekil 5.6: `x[idx]` ile `idx = [3, 0, 3, 1]`: sonucun k-ıncı elemanı `x[idx[k]]`'dir. İndeksler istenen sırada verilebilir ve tekrar edebilir; 3 numaralı konum iki kez seçilmiştir. Sonuç, `x`'ten bağımsız yeni bir dizidir.

```
import numpy as np

x = np.array([10, 20, 30, 40, 50])
idx = np.array([3, 0, 3, 1])
print(x[idx])
print(x[[0, -1]])               # ilk ve son eleman

A = np.arange(12).reshape(3, 4)
print(A[[2, 0]])                # üçüncü ve birinci satır
print(A[[0, 1, 2], [3, 2, 1]])  # (0,3), (1,2), (2,1) elemanları
```

Çıktı:

```
[40 10 40 20]
[10 50]
[[ 8  9 10 11]
 [ 0  1  2  3]]
[3 6 9]
```

İki boyutlu bir dizide tek bir indeks listesi satırları seçer: `A[[2, 0]]` üçüncü ve birinci satırları bu sırayla dizer. İki indeks listesi verilirse NumPy onları **çift çift** eşler: `A[[0, 1, 2], [3, 2, 1]]` bir alt matris değil, $(0,3)$, $(1,2)$ ve $(2,1)$ konumlarındaki üç elemandır. Seçilen satırlarla sütunların bütün kesişimlerinden oluşan alt matris için `np.ix_` kullanılır.

Örnek 5.6 (Bir Alt Matrisin Seçilmesi). Aşağıdaki A matrisinde ikinci satırı ve ikinci ile dördüncü sütunları silerek elde edilen alt matrisi (bkz. Lineer Cebir) NumPy ile bulunuz.

$$A = \begin{pmatrix} 1 & 2 & 0 & 4 \\ -5 & 1 & 3 & 0 \\ 0 & 2 & 1 & 7 \end{pmatrix}$$

> **Çözüm.**
> Silinmeyen satırlar birinci ve üçüncü, silinmeyen sütunlar da birinci ve üçüncüdür; 0'dan sayınca satır indeksleri `[0, 2]`, sütun indeksleri de `[0, 2]` olur. `np.ix_(rows, cols)` seçilen satırlarla sütunların bütün kesişimlerini alır. Aynı sonuç önce satırları, sonra sütunları seçerek iki adımda da bulunur.

 CC BY-NC-SA 4.0

```
import numpy as np

A = np.array([[1, 2, 0, 4],
              [-5, 1, 3, 0],
              [0, 2, 1, 7]])
rows = [0, 2]           # kalan satırlar
cols = [0, 2]           # kalan sütunlar
print(A[np.ix_(rows, cols)])
print(A[rows][:, cols])     # aynı sonuç, iki adımda
print(A[rows, cols])        # dikkat: yalnız (0,0) ve (2,2)
```

Çıktı:

```
[[1 0]
 [0 1]]
[[1 0]
 [0 1]]
[1 1]
```

Alt matris $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$'dir. Bu örnekte `A[::2, ::2]` dilimi de aynı sonucu verirdi; ama silinen satır ve sütunlar düzensiz olduğunda fancy indeksleme gerekir. Son satır tipik bir hatayı gösteriyor: `A[rows, cols]` indeksleri çift çift eşlediği için yalnız $(0,0)$ ve $(2,2)$ konumlarındaki elemanları verir. ■

5.9 Boolean İndeksleme

Çoğu zaman elemanları konumlarına göre değil bir koşula göre seçmek isteriz: pozitif olanlar, 3'e bölünenler, bir aralığa düşenler. Bir dizinin bir sayıyla karşılaştırılması her eleman için ayrı bir `True` ya da `False` değeri üretir ve bu `bool` dizisi bir indeks olarak kullanılabilir.

```
import numpy as np

x = np.arange(1, 13)
mask = x % 3 == 0
print(mask)
print(x[mask])
print(x[(x > 4) & (x % 2 == 1)])    # 4'ten büyük tekler
print(x[~mask])                     # 3'e bölünmeyenler
print("kaç tane:", mask.sum())
```

Çıktı:

```
[False False  True False False  True False False  True False False  True]
[ 3  6  9 12]
[ 5  7  9 11]
[ 1  2  4  5  7  8 10 11]
kaç tane: 4
```

Tanım 5.10 (Maske ve Boolean İndeksleme). `x` ile aynı şekilde ve `bool` türünde bir `m` dizisine **maske** (mask) denir. `x[m]`, `x` dizisinin `m` maskesinin `True` olduğu konumlardaki elemanlarını satır öncelikli sırayla toplayan, bir boyutlu **yeni** bir dizidir. Maskeye atama, yani `x[m] = c`, yalnız bu konumları değiştirir.

Yani maske bir alt kümeyi gösterir: $S = \{k : m_k = \text{True}\}$ ise `x[m]` dizisi x_k $(k \in S)$ elemanlarını sırayla verir. Koşullar `&` (ve), `|` (veya) ve `~` (değil) işleçleriyle birleştirilir ve her koşul paranteze alınır, çünkü `&` ile `|` karşılaştırmalardan önce uygulanır. `True` değeri 1, `False` değeri 0 sayıldığından `mask.sum()` koşulu sağlayan eleman sayısını verir.

 CC BY-NC-SA 4.0

⚠ and, or ve not dizilerle çalışmaz

Python'ın `and`, `or` ve `not` sözcükleri tek bir doğruluk değeri bekler. Bir diziye uygulandıklarında NumPy dizinin bütün olarak doğru mu yanlış mı sayılacağına karar veremez ve hata verir.

```
import numpy as np

x = np.arange(1, 13)
try:
    print(x[x > 4 and x < 9])
except ValueError as e:
    print("Hata:", str(e).split(".")[0])   # mesajın ilk cümlesi
print(x[(x > 4) & (x < 9)])
```

Çıktı:

```
Hata: The truth value of an array with more than one element is ambiguous
[5 6 7 8]
```

Dizilerde her zaman `&`, `|`, `~` işleçlerini ve parantezi kullanın.

Maskeye atama, bir koşulu sağlayan elemanları topluca değiştirmenin en kısa yoludur. Aşağıda -5 ile 5 arasında rastgele tamsayılar üretip negatif olanları sıfırlıyoruz. `np.random.default_rng(1)` sabit bir tohumla (seed) bir rastgele sayı üreteci kurar; böylece kod her çalıştığında aynı sayılar çıkar. Ayrıntıları Rastgele Sayılar ve Monte Carlo Yöntemleri bölümünde göreceğiz. `rng.integers(-5, 6, size=10)` çağrısı, üst sınır hariç olmak üzere 10 tamsayı üretir.

```
import numpy as np

rng = np.random.default_rng(1)
y = rng.integers(-5, 6, size=10)
print(y)
y[y < 0] = 0          # negatifleri sıfırla
print(y)
```

Çıktı:

```
[ 0  0  3  5 -5 -4  4  5 -3 -2]
[0 0 3 5 0 0 4 5 0 0]
```

Örnek 5.7 (Eratosthenes Kalburu ile Asallar). Veri Yapıları ve Comprehension bölümünde listeyle kurduğumuz Eratosthenes kalburunu bu kez bir `bool` dizisi üzerinde kurarak 100'e kadar olan asal sayıları bulunuz. Aşağıdaki şekil, çözümdeki kodun ürettiği maskeyi 10×10 bir tablo olarak gösteriyor.

 CC BY-NC-SA 4.0

1	**2**	**3**	4	**5**	6	**7**	8	9	10
11	12	**13**	14	15	16	**17**	18	**19**	20
21	22	**23**	24	25	26	27	28	**29**	30
31	32	33	34	35	36	**37**	38	39	40
41	42	**43**	44	45	46	**47**	48	49	50
51	52	**53**	54	55	56	57	58	**59**	60
61	62	63	64	65	66	**67**	68	69	70
71	72	**73**	74	75	76	77	78	**79**	80
81	82	**83**	84	85	86	87	88	**89**	90
91	92	93	94	95	96	**97**	98	99	100

Şekil 5.7: Kalburdan sonra `is_prime[1:].reshape(10, 10)` maskesi: 1'den 100'e kadar sayılar satır satır dizilmiş, maskenin `True` olduğu 25 asal vurgulanmıştır. 2 ve 5 dışındaki asalların hepsi 1, 3, 7 ya da 9 ile biten sütunlarda durur.

Çözüm.
Algoritma liste sürümündekiyle aynıdır: $p^2 \le N$ olan her asal p için $p^2, p^2+p, p^2+2p, \ldots$ konumları elenir. NumPy üç yerde işi kısaltır. Birincisi, `is_prime` bir liste değil, `np.ones(N + 1, dtype=bool)` ile kurulan ve bütün elemanları `True` olan bir `bool` dizisidir. İkincisi, dilime tek bir değer atanabildiğinden `is_prime[p * p::p] = False` yazmak yeter; sağ tarafta dilimle aynı uzunlukta bir liste kurmak gerekmez. Üçüncüsü, asallar bir comprehension ile toplanmaz; `np.arange(N + 1)` dizisinden doğrudan maskeyle seçilir.

```
import numpy as np

N = 100
is_prime = np.ones(N + 1, dtype=bool)
is_prime[:2] = False                # 0 ve 1 asal değil
for p in range(2, int(N**0.5) + 1):
    if is_prime[p]:
        is_prime[p * p::p] = False   # p*p, p*p + p, ... elenir
primes = np.arange(N + 1)[is_prime]
print(primes)
print("asal sayısı:", primes.size)
print(is_prime[1:].reshape(10, 10).astype(int))
```

Çıktı:

```
[ 2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
 97]
asal sayısı: 25
[[0 1 1 0 1 0 1 0 0 0]
 [1 0 1 0 0 0 1 0 1 0]
```

 CC BY-NC-SA 4.0

```
[0 0 1 0 0 0 0 0 1 0]
[1 0 0 0 0 0 1 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 0 1 0 0 0 0 0 1 0]
[1 0 0 0 0 0 1 0 0 0]
[1 0 1 0 0 0 0 0 1 0]
[0 0 1 0 0 0 0 0 1 0]
[0 0 0 0 0 0 1 0 0 0]]
```

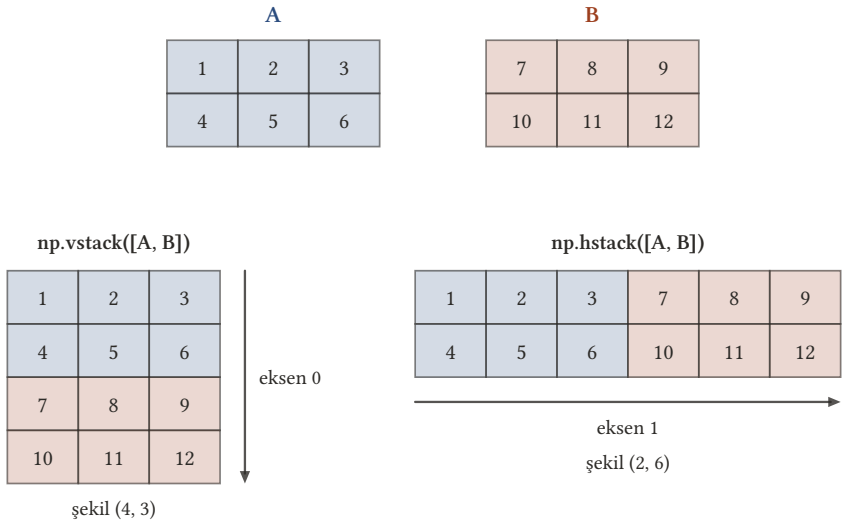
Liste sürümüyle aynı 25 asal bulunur. Son çıktı, maskenin 1'den 100'e kadar olan kısmının satır satır 10×10 okunuşudur; şekildeki tablo da budur. ■

5.10 Dizileri Birleştirme

Büyük dizileri çoğu zaman küçük parçalardan kurarız: vektörleri bir matrisin satırları ya da sütunları olarak yan yana koyar, blok matrisleri bloklarından birleştiririz.

Tanım 5.11 (Bir Eksen Boyunca Birleştirme). `np.concatenate([A, B, ...], axis=k)`, dizileri k ekseni boyunca uç uca ekler. Dizilerin şekilleri k dışındaki bütün eksenlerde aynı olmalıdır; sonucun k eksenindeki uzunluğu, parçaların bu eksendeki uzunluklarının toplamıdır.

Yani iki 2×3 matris `axis=0` ile alt alta konup 4×3, `axis=1` ile yan yana konup 2×6 bir matris verir. Bu iki durumun kısa yolları `np.vstack` (dikey yığma) ve `np.hstack` (yatay yığma) fonksiyonlarıdır. `np.stack` ise parçaları **yeni** bir eksen boyunca üst üste koyar: iki 2×3 matristen şekli `(2, 2, 3)` olan bir dizi çıkar.



Şekil 5.8: İki 2 × 3 dizi. `np.vstack` onları 0 ekseni boyunca alt alta koyar ve 4 × 3 bir dizi verir; `np.hstack` 1 ekseni boyunca yan yana koyar ve 2 × 6 bir dizi verir. Birleştirilen eksen dışındaki uzunluklar aynı olmalıdır.

```
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6]])
B = np.array([[7, 8, 9],
              [10, 11, 12]])
print(np.vstack([A, B]))
```

```
print(np.hstack([A, B]))
print(np.concatenate([A, B], axis=1).shape)
print(np.stack([A, B]).shape)       # yeni eksen: (2, 2, 3)
```

Çıktı:

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
(2, 6)
(2, 2, 3)
```

Bir boyutlu vektörlerle çalışırken `np.column_stack` vektörleri sütun, `np.vstack` satır olarak dizer. Birleştirmenin tersi `np.split` fonksiyonudur: diziyi verilen konumlardan keserek parçalarının listesini verir.

```
import numpy as np

x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
print(np.column_stack([x, y]))
print(np.vstack([x, y]))
print(np.split(np.arange(10), [3, 7]))
```

Çıktı:

```
[[1 4]
 [2 5]
 [3 6]]
[[1 2 3]
 [4 5 6]]
[array([0, 1, 2]), array([3, 4, 5, 6]), array([7, 8, 9])]
```

`np.split(np.arange(10), [3, 7])` diziyi 3 ve 7 numaralı konumlardan keser; parçalar `[0:3]`, `[3:7]` ve `[7:10]` dilimleridir.

Örnek 5.8 (Vandermonde Matrisi). $x = (1, 2, 3, 4)$ noktaları için (i, j) bileşeni x_i^j olan $(i, j = 0, 1, 2, 3)$ 4×4 Vandermonde matrisini sütunlarını yan yana koyarak kurunuz.

Çözüm.
Matrisin sütunları x vektörünün kuvvetleridir: $x^0 = (1, 1, 1, 1)$, x, x^2 ve x^3. Bir dizinin kuvveti her elemana ayrı uygulandığından `x**2` gibi ifadeler bu sütunları doğrudan verir; `np.column_stack` da onları sütun olarak dizer. Sonucu NumPy'ın hazır `np.vander` fonksiyonuyla karşılaştırıyoruz; `increasing=True` kuvvetlerin artan sırada dizilmesini ister. Karşılaştırmayı yapan `np.array_equal`, iki dizinin şekli ve bütün elemanları aynıysa `True` verir.

```
import numpy as np

x = np.array([1, 2, 3, 4])
V = np.column_stack([x**0, x, x**2, x**3])
print(V)
print(np.array_equal(V, np.vander(x, increasing=True)))
```

Çıktı:

```
[[ 1  1  1  1]
 [ 1  2  4  8]
 [ 1  3  9 27]
```

 CC BY-NC-SA 4.0

```
 [ 1  4 16 64]]
True
```

Bu matris, verilen noktalardan geçen polinomu bulma probleminin katsayı matrisidir ve Polinomlar, İnterpolasyon ve Eğri Uydurma bölümünde yeniden karşımıza çıkacak. ■

Örnek 5.9 (Blok Matrisin Transpozesi). Örnek 5.4 örneğindeki 4×4 matris M ve blokları verilsin. `np.block` ile

$$M^t = \begin{pmatrix} M_{11}^t & M_{21}^t \\ M_{12}^t & M_{22}^t \end{pmatrix}$$

eşitliğini doğrulayınız.

Çözüm.
`np.block`, iç içe bir listeyle verilen blokları blok matris düzeninde birleştirir; iç listeler blok satırlarıdır. Önce bloklardan M matrisinin kendisinin geri geldiğini, sonra transpoze bloklarla kurulan matrisin `M.T` ile aynı olduğunu `np.array_equal` ile kontrol ediyoruz.

```
import numpy as np

M = np.arange(1, 17).reshape(4, 4)
M11, M12 = M[:2, :2], M[:2, 2:]
M21, M22 = M[2:, :2], M[2:, 2:]
N = np.block([[M11, M12], [M21, M22]])
print(np.array_equal(N, M))
MT = np.block([[M11.T, M21.T], [M12.T, M22.T]])
print(np.array_equal(MT, M.T))
print(MT)
```

Çıktı:

```
True
True
[[ 1  5  9 13]
 [ 2  6 10 14]
 [ 3  7 11 15]
 [ 4  8 12 16]]
```

İki karşılaştırma da `True` verir. Kuralın nedeni şudur: transpoze, (i, j) konumundaki elemanı (j, i) konumuna taşır. Bu, blokların yerini köşegene göre değiştirir (M_{12} ile M_{21} yer değiştirir) ve her bloğun kendi içinde de transpozesini alır. ■

5.11 Alıştırmalar

Aşağıdaki alıştırmaların her birinin çözümü, kodu ve çalıştırılmış çıktısıyla birlikte verilmiştir; önce kendiniz deneyip sonra karşılaştırın.

Alıştırma 5.1 (Esas Köşegen ve İz). `A = np.arange(1, 26).reshape(5, 5)` matrisinin esas köşegenini fancy indekslemeyle seçiniz ve köşegen elemanlarının toplamı olan izi hesaplayınız.

Çözüm.
Adım 1. Esas köşegen (bkz. Lineer Cebir) $(0,0), (1,1), \ldots, (4,4)$ konumlarıdır. İki indeks dizisi çift çift eşlendiğinden satır ve sütun indeksi olarak aynı `k = np.arange(5)` dizisini vermek tam bu konumları seçer.
Adım 2. İz, seçilen elemanların toplamıdır; dizilerin `sum` metodu elemanları toplar.

 CC BY-NC-SA 4.0

Adım 3. Sonucu NumPy'ın hazır fonksiyonlarıyla karşılaştırırız: `np.diag` bir matrise uygulandığında köşegenini çıkarır, `np.trace` izi verir.

```
import numpy as np

A = np.arange(1, 26).reshape(5, 5)
k = np.arange(5)
d = A[k, k]                  # (0,0), (1,1), ..., (4,4)
print(d)
print("iz:", d.sum())
print(np.array_equal(d, np.diag(A)), np.trace(A))
```

Çıktı:

```
[ 1  7 13 19 25]
iz: 65
True 65
```

Elle de kontrol edebiliriz: köşegen elemanları $a_{kk} = 5k + k + 1 = 6k + 1$ ($k = 0, \ldots, 4$) olduğundan iz $1 + 7 + 13 + 19 + 25 = 65$'tir. ■

Alıştırma 5.2 (Çerçeve Matrisi). Kenarlarındaki elemanları 1, içindeki elemanları 0 olan 6×6 bir tamsayı matrisini tek bir dilim atamasıyla kurunuz ve matristeki 1'lerin sayısını bulunuz.

Çözüm.
Adım 1. Bütün elemanları 1 olan `np.ones((6, 6), dtype=int)` matrisiyle başlarız.
Adım 2. İç kısım, ilk ve son satır ile ilk ve son sütun dışında kalan 4×4 bloktur. Her iki eksende `1:-1` dilimi ilk ve son konumu dışarıda bırakır; bu bloğa 0 atarız.
Adım 3. 1'lerin sayısını elemanların toplamı verir.

```
import numpy as np

Z = np.ones((6, 6), dtype=int)
Z[1:-1, 1:-1] = 0        # içteki 4 x 4 blok
print(Z)
print("birlerin sayısı:", Z.sum())
```

Çıktı:

```
[[1 1 1 1 1 1]
 [1 0 0 0 0 1]
 [1 0 0 0 0 1]
 [1 0 0 0 0 1]
 [1 0 0 0 0 1]
 [1 1 1 1 1 1]]
birlerin sayısı: 20
```

Sayım elle de yapılabilir: 36 elemandan $4 \cdot 4 = 16$ tanesi iç kısımdadır, geriye $36 - 16 = 20$ eleman kalır. ■

Alıştırma 5.3 (Ondalık Adımın Eleman Sayısı). `np.arange(0.1, 0.4, 0.1)` çağrısının kaç eleman verdiğini Tanım 5.3 tanımındaki formülle açıklayınız.

Çözüm.

 CC BY-NC-SA 4.0

Adım 1. Tam aritmetikte $N = \lceil (0{,}4 - 0{,}1)/0{,}1 \rceil = \lceil 3 \rceil = 3$ olur ve dizi $0{,}1;\ 0{,}2;\ 0{,}3$ noktalarından oluşur; $0{,}4$ dahil değildir.
Adım 2. Makinede ise $0{,}1$ ve $0{,}4$ tam temsil edilemez. Çıkarma ve bölme yuvarlama hatası taşır ve oran 3 değil, ondan biraz büyük bir sayı çıkar. Yukarı yuvarlama bu durumda $N = 4$ verir.
Adım 3. Kodla görelim; aynı aralıktaki üç noktayı güvenle veren `linspace` çağrısını da ekliyoruz.

```
import numpy as np

x = np.arange(0.1, 0.4, 0.1)
print(x, x.size)
print((0.4 - 0.1) / 0.1)
print(np.linspace(0.1, 0.3, 3))
```

Çıktı:

```
[0.1 0.2 0.3 0.4] 4
3.0000000000000004
[0.1 0.2 0.3]
```

Oran $3{,}0000000000000004$ çıktığı için dizi dört elemanlıdır ve dışarıda kalması gereken $0{,}4$ de diziye girer. `np.linspace(0.1, 0.3, 3)` ise nokta sayısını doğrudan verdiğimiz için bu sorunu yaşamaz. ■

Alıştırma 5.4 (Üç Boyutlu Bir Dizide İndeks). `T = np.arange(60).reshape(3, 4, 5)` dizisinde 37 sayısı hangi (i_0, i_1, i_2) indeksinde durur?

Çözüm.
Adım 1. `np.arange(60)` dizisinde 37 sayısı 37 numaralı konumdadır. Önerme 5.1 gereği şekil $(3, 4, 5)$ iken

$$37 = 20\,i_0 + 5\,i_1 + i_2, \qquad 0 \le i_1 < 4, \quad 0 \le i_2 < 5$$

olmalıdır.
Adım 2. $5\,i_1 + i_2 \le 5 \cdot 3 + 4 = 19 < 20$ olduğundan i_0, 37'nin 20'ye bölümüdür: $37 = 20 \cdot 1 + 17$, yani $i_0 = 1$.
Adım 3. Kalan $17 = 5\,i_1 + i_2$ ve $0 \le i_2 < 5$ olduğundan i_1, 17'nin 5'e bölümüdür: $17 = 5 \cdot 3 + 2$, yani $i_1 = 3$ ve $i_2 = 2$.
Adım 4. Bölme adımlarını Python'ın `divmod` fonksiyonu (bölüm ve kalanı birlikte verir) ile yapıp sonucu doğrularız. NumPy'ın `np.unravel_index` fonksiyonu da tam bu dönüşümü yapar; çıktısını okunaklı olsun diye Python tamsayılarına çeviriyoruz.

```
import numpy as np

T = np.arange(60).reshape(3, 4, 5)
i0, r = divmod(37, 4 * 5)
i1, i2 = divmod(r, 5)
print(i0, i1, i2)
print(T[i0, i1, i2])
print([int(t) for t in np.unravel_index(37, T.shape)])
```

Çıktı:

```
1 3 2
37
[1, 3, 2]
```

37 sayısı `T[1, 3, 2]` konumundadır. ■

 CC BY-NC-SA 4.0

Alıştırma 5.5 (Görünüm mü, Kopya mı?). `a = np.arange(6)` dizisinden `b = a[::2]` ve `c = a[[1, 3]]` dizileri kuruluyor, ardından `b[0] = 100` ve `c[0] = 200` atamaları yapılıyor. Bu atamalardan sonra `a`, `b` ve `c` dizileri ne olur? Önce kodu çalıştırmadan tahmin ediniz.

Çözüm.
Adım 1. `b = a[::2]` temel bir dilimdir, dolayısıyla bir görünümdür: `b` dizisi `a` dizisinin 0, 2 ve 4 numaralı hücrelerine bakar. `b[0] = 100` ataması `a[0]` hücresine yazar.
Adım 2. `c = a[[1, 3]]` fancy indekslemedir, dolayısıyla bir kopyadır: `c` kurulurken 1 ve 3 değerleri yeni bir bloğa yazılır. `c[0] = 200` ataması yalnız bu yeni bloğu değiştirir.
Adım 3. Buna göre `a` dizisinde yalnız ilk eleman 100 olur, `b` dizisi `[100, 2, 4]`, `c` dizisi `[200, 3]` olur. Kodla doğrulayalım.

```
import numpy as np

a = np.arange(6)
b = a[::2]
c = a[[1, 3]]
b[0] = 100
c[0] = 200
print("a =", a)
print("b =", b)
print("c =", c)
```

Çıktı:

```
a = [100   1   2   3   4   5]
b = [100   2   4]
c = [200   3]
```

■

Alıştırma 5.6 (Üçün ya da Beşin Katları). 1000'den küçük olup 3'e ya da 5'e bölünen doğal sayıların toplamını bir maskeyle hesaplayınız.

Çözüm.
Adım 1. `np.arange(1000)` ile $0, 1, \ldots, 999$ sayılarını kurarız; 0 toplamı değiştirmez.
Adım 2. "3'e bölünür" koşulu `x % 3 == 0`, "5'e bölünür" koşulu `x % 5 == 0` maskesidir. "Ya da" bağlacı `|` işleciyle kurulur ve her koşul paranteze alınır.
Adım 3. Maskeyle seçilen elemanları `sum` ile toplarız.
Adım 4. Sonucu içerme–dışarma ilkesiyle (bkz. Veri Yapıları ve Comprehension) denetleriz. n'yi aşmayan k'nin katları $k, 2k, \ldots, mk$ ($m = \lfloor n/k \rfloor$) olduğundan toplamları $k\,m(m+1)/2$'dir. Hem 3'ün hem 5'in katı olanlar 15'in katlarıdır ve iki kez sayılmamaları için bir kez çıkarılırlar. $n = 999$ için m sırasıyla $\lfloor 999/3 \rfloor = 333$, $\lfloor 999/5 \rfloor = 199$ ve $\lfloor 999/15 \rfloor = 66$ olur:

$$S = 3 \cdot \frac{333 \cdot 334}{2} + 5 \cdot \frac{199 \cdot 200}{2} - 15 \cdot \frac{66 \cdot 67}{2}$$
$$= 166\,833 + 99\,500 - 33\,165 = 233\,168.$$

```
import numpy as np


def multiples_sum(k, n=999):
    """n'yi aşmayan k'nin katlarının toplamı."""
    m = n // k
    return k * m * (m + 1) // 2
```

 CC BY-NC-SA 4.0

```
x = np.arange(1000)
mask = (x % 3 == 0) | (x % 5 == 0)
print(x[mask][:10])
print("toplam:", x[mask].sum())
print("formül:", multiples_sum(3) + multiples_sum(5)
      - multiples_sum(15))
```

Çıktı:

```
[ 0  3  5  6  9 10 12 15 18 20]
toplam: 233168
formül: 233168
```

Maske ile formül aynı sonucu, 233 168'i verir. ■

Alıştırma 5.7 (Bir Permütasyonun Tersi). `perm = np.array([2, 0, 3, 1])` dizisi $\{0,1,2,3\}$ kümesinin $i \mapsto \text{perm}[i]$ permütasyonunu (bkz. Lineer Cebir) göstersin. Ters permütasyonu tek bir fancy indeksleme atamasıyla bulunuz.

Çözüm.
Adım 1. Ters permütasyon `inv`, her i için `inv[perm[i]] = i` koşuluyla belirlenir: perm, i'yi $\text{perm}[i]$'ye götürüyorsa ters permütasyon $\text{perm}[i]$'yi i'ye geri götürür.
Adım 2. Fancy indeksleme atamaya da uygulanır: `inv[perm] = v` ataması her i için `inv[perm[i]] = v[i]` yazar. `v = np.arange(4)` alınırsa Adım 1'deki koşulun hepsi tek satırda sağlanır.
Adım 3. Doğrulama: `y = x[perm]` bir diziyi permütasyona göre yeniden dizer; `y[inv]` ise eski sırayı geri getirmelidir. Ayrıca `np.argsort(perm)`, `perm` dizisini sıralayan konumları verdiği için o da tersi bulmalıdır.

```
import numpy as np

perm = np.array([2, 0, 3, 1])
inv = np.zeros_like(perm)
inv[perm] = np.arange(perm.size)    # inv[perm[i]] = i
print("ters:", inv)

x = np.array([10, 20, 30, 40])
y = x[perm]
print(y, y[inv])
print(np.array_equal(inv, np.argsort(perm)))
```

Çıktı:

```
ters: [1 3 0 2]
[30 10 40 20] [10 20 30 40]
True
```

Elle: $0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 3, 3 \mapsto 1$ olduğundan ters permütasyon $2 \mapsto 0, 0 \mapsto 1, 3 \mapsto 2, 1 \mapsto 3$, yani `[1, 3, 0, 2]` dizisidir. ■

Alıştırma 5.8 (Pascal Matrisi). (i,j) bileşeni $\binom{i}{j}$ olan ($j > i$ için 0) 6×6 alt üçgensel Pascal matrisini, her satırını bir önceki satırdan tek bir dilim toplamasıyla elde ederek kurunuz.

Çözüm.
Adım 1. Pascal üçgenini Veri Yapıları ve Comprehension bölümünde listelerle satır satır kurmuştuk; burada aynı kuralı bir matrisin satırlarına uyguluyoruz. $1 \le j \le i$ için Pascal özdeşliği (bkz. Analiz 1)

$$\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$$

der. Ayrıca her i için $\binom{i}{0} = 1$'dir.
Adım 2. i-inci satırın $1, \ldots, i$ konumları `P[i, 1:i + 1]` dilimidir. Sağ taraftaki ilk terimler $j-1 = 0, \ldots, i-1$ için `P[i - 1, 0:i]`, ikinci terimler $j = 1, \ldots, i$ için `P[i - 1, 1:i + 1]` dilimidir. Bu ikinci dilimin son elemanı $\binom{i-1}{i} = 0$'dır ve matris sıfırlarla başlatıldığı için zaten 0'dır.
Adım 3. İlk sütunu 1 yapıp satırları sırayla doldururuz, sonra her bileşeni `math.comb` ile karşılaştırırız. `math.comb(i, j)`, $j > i$ için 0 verir.

```
import math
import numpy as np

n = 6
P = np.zeros((n, n), dtype=int)
P[:, 0] = 1
for i in range(1, n):
    # C(i, j) = C(i-1, j-1) + C(i-1, j),  j = 1, ..., i
    P[i, 1:i + 1] = P[i - 1, 0:i] + P[i - 1, 1:i + 1]
print(P)
ok = all(P[i, j] == math.comb(i, j)
         for i in range(n) for j in range(n))
print("math.comb ile aynı:", ok)
```

Çıktı:

```
[[ 1  0  0  0  0  0]
 [ 1  1  0  0  0  0]
 [ 1  2  1  0  0  0]
 [ 1  3  3  1  0  0]
 [ 1  4  6  4  1  0]
 [ 1  5 10 10  5  1]]
math.comb ile aynı: True
```

Satırlar Pascal üçgeninin satırlarıdır; örneğin son satır $1, 5, 10, 10, 5, 1$ sayıları, yani $(a+b)^5$ açılımının katsayılarıdır. ■

Alıştırma 5.9 (Blok Köşegen Matris). $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ ve $B = 5I_3$ olmak üzere

$$M = \begin{pmatrix} A & O \\ O & B \end{pmatrix}$$

blok köşegen matrisini `np.block` ile kurunuz. Burada O uygun boyutlu sıfır matrislerdir.

Çözüm.
Adım 1. Aynı blok satırındaki blokların satır sayıları, aynı blok sütunundaki blokların sütun sayıları eşit olmalıdır. Sağ üstteki O, A ile aynı satır sayısına (2), B ile aynı sütun sayısına (3) sahiptir, yani 2×3 tipindedir; sol alttaki O da aynı nedenle 3×2 tipindedir.
Adım 2. $B = 5I_3$ skaler matrisini `5 * np.eye(3, dtype=int)` verir. Sıfır blokları `np.zeros` ile, tamsayı türünde kurarız.
Adım 3. Sonucun şeklinin $(2+3) \times (2+3) = 5 \times 5$ olduğunu ve köşegen bloklarının dilimlerle geri okunduğunu kontrol ederiz.

```
import numpy as np

A = np.array([[1, 2],
```

 CC BY-NC-SA 4.0

```
              [3, 4]])
B = 5 * np.eye(3, dtype=int)
M = np.block([[A, np.zeros((2, 3), dtype=int)],
              [np.zeros((3, 2), dtype=int), B]])
print(M)
print(M.shape)
print(np.array_equal(M[:2, :2], A), np.array_equal(M[2:, 2:], B))
```

Çıktı:

```
[[1 2 0 0 0]
 [3 4 0 0 0]
 [0 0 5 0 0]
 [0 0 0 5 0]
 [0 0 0 0 5]]
(5, 5)
True True
```

■

Alıştırma 5.10 (Faktöriyel ve Taşma). `np.prod(np.arange(1, n + 1))` ifadesi $n!$ değerini `int64` aritmetiğiyle hesaplar (`np.prod` bir dizinin elemanlarını çarpar). Bu ifadenin yanlış sonuç verdiği en küçük n değerini `math.factorial` ile karşılaştırarak bulunuz.

Çözüm.
Adım 1. `int64` en çok $2^{63}-1 \approx 9{,}22 \cdot 10^{18}$ değerini tutabilir. Çarpım bu sınırı aştığı anda sonuç sessizce sarılır.
Adım 2. $20! \approx 2{,}43 \cdot 10^{18}$ sınırın altında, $21! = 21 \cdot 20! \approx 5{,}11 \cdot 10^{19}$ ise üstündedir. Bu yüzden ilk hatalı değerin $n = 21$ olmasını bekleriz.
Adım 3. $n = 18, \ldots, 22$ için NumPy sonucunu, Python'ın sınırsız tamsayılarıyla tam hesap yapan `math.factorial` ile karşılaştırırız.

```
import math
import numpy as np

for n in range(18, 23):
    p = np.prod(np.arange(1, n + 1))
    print(n, p, p == math.factorial(n))
print("int64 üst sınırı:", np.iinfo(np.int64).max)
```

Çıktı:

```
18 6402373705728000 True
19 121645100408832000 True
20 2432902008176640000 True
21 -4249290049419214848 False
22 -1250660718674968576 False
int64 üst sınırı: 9223372036854775807
```

$n = 21$ için sonuç negatif çıkıyor: tahmin doğrudur. Büyük tamsayılarla tam hesap gerektiğinde NumPy yerine Python'ın `int` türü ya da `math` modülü kullanılmalıdır. ■

Bu bölümde NumPy dizilerini kurmayı, şekillerini ve veri türlerini okumayı ve parçalarını seçmeyi gördük. Şekil yalnız verinin nasıl okunacağını belirler; dilimleme görünüm, fancy ve boolean indeksleme kopya verir. Dizilerin asıl gücü ise onlar üzerinde döngü yazmadan hesap yapmaktır. Bir sonraki bölüm Vektö-

 CC BY-NC-SA 4.0

rizasyon ve Broadcasting, evrensel fonksiyonları, bir eksen boyunca toplamaları ve farklı şekillerdeki dizilerin birlikte nasıl işlendiğini anlatıyor.

 CC BY-NC-SA 4.0

6. Vektörizasyon ve Broadcasting

Matematikte bir fonksiyonu bir sayı kümesinin her elemanına uygularken "her i için $y_i = x_i^2 - \sin x_i$" deyip geçeriz; kimse bunu tek tek yazmaz. Kontrol Yapıları ve Fonksiyonlar bölümünde aynı işi bir `for` döngüsüyle yapmıştık. NumPy Dizileri bölümünde ise dizileri kurmayı, şekillerini değiştirmeyi ve indekslemeyi öğrendik. Bu bölümde dizilerle **döngü yazmadan** hesap yapacağız:

```
import math
import numpy as np

# Döngüyle: her eleman ayrı ayrı işlenir
xs = [0.0, 0.5, 1.0, 1.5, 2.0]
ys = []
for v in xs:
    ys.append(v**2 - math.sin(v))
print([round(v, 8) for v in ys])    # NumPy gibi 8 basamakla yazdır

# Vektörel: bütün dizi tek ifadeyle işlenir
x = np.array(xs)
y = x**2 - np.sin(x)
print(y)
```

Çıktı:

```
[0.0, -0.22942554, 0.15852902, 1.25250501, 3.09070257]
[ 0.         -0.22942554  0.15852902  1.25250501  3.09070257]
```

İki hesap aynı sayıları verir, ama ikincisi matematikteki yazılışa çok daha yakındır. Bir dizinin bütün elemanlarını tek bir ifadeyle işleme tekniğine **vektörizasyon** (vectorization) denir. Vektörel kod hem kısadır hem de çok hızlıdır, çünkü elemanlar üzerindeki döngü Python yorumlayıcısında değil, NumPy'ın derlenmiş C kodunda döner. Bölümün ikinci büyük konusu **broadcasting** kurallarıdır: şekilleri farklı iki diziyi, örneğin bir matrisle bir satırı, kopya oluşturmadan birlikte işlemeyi sağlarlar.

6.1 Evrensel Fonksiyonlar

Vektörel hesabın yapı taşı, bir diziye eleman eleman uygulanan fonksiyonlardır.

```
import numpy as np

x = np.linspace(0, np.pi, 5)
print(x)
print(np.sin(x))

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
print(a + b, np.add(a, b))      # aynı işlem, iki yazılış
print(a * b, a**2 / b)
print(np.maximum(a, 3))         # her i için max(a[i], 3)
print(np.sqrt(a))
```

Çıktı:

```
[0.         0.78539816 1.57079633 2.35619449 3.14159265]
[0.00000000e+00 7.07106781e-01 1.00000000e+00 7.07106781e-01
 1.22464680e-16]
[11 22 33 44] [11 22 33 44]
```

 CC BY-NC-SA 4.0

```
[ 10  40  90 160] [0.1 0.2 0.3 0.4]
[3 3 3 4]
[1.         1.41421356 1.73205081 2.        ]
```

`np.sin(x)` beş elemanın beşinin de sinüsünü verdi. Son değer $\sin \pi = 0$ olması gerekirken $1{,}22 \cdot 10^{-16}$ çıktı: `np.pi` sayısı π'nin kendisi değil, ona en yakın makine sayısıdır (bkz. Python ile İlk Adımlar). Bu küçük sayıyı da gösterebilmek için NumPy ikinci satırdaki bütün diziyi bilimsel gösterimle yazdı. `a + b` ile `np.add(a, b)` aynı işlemdir; `+`, `*`, `**`, `/` işleçleri NumPy dizilerinde arka planda bu fonksiyonları çağırır.

Tanım 6.1 (Evrensel Fonksiyon). Aynı şekle sahip dizilere eleman eleman uygulanan ve sonucu yine aynı şekle sahip bir dizi olarak veren NumPy fonksiyonuna **evrensel fonksiyon** (universal function, kısaca **ufunc**) denir. Tek girdili bir ufunc f için $y = f(x)$ dizisinin elemanları $y_i = f(x_i)$, iki girdili bir g için $z = g(x, y)$ dizisinin elemanları $z_i = g(x_i, y_i)$ olur.

Yani bir ufunc, sayılar için tanımlı bir fonksiyonu dizilere "kendiliğinden" genişletir; döngüyü biz değil, NumPy kurar. Girdi tek bir sayı da olabilir; o zaman sonuç da tek bir sayıdır. En sık kullanılanlar şunlardır:

Tür	Evrensel fonksiyonlar
Aritmetik	`np.add` (`+`), `np.subtract` (`-`), `np.multiply` (`*`), `np.divide` (`/`), `np.floor_divide` (`//`), `np.mod` (`%`), `np.power` (`**`)
Üstel ve logaritma	`np.exp`, `np.log`, `np.log2`, `np.log10`, `np.sqrt`
Trigonometrik	`np.sin`, `np.cos`, `np.tan`, `np.arcsin`, `np.arctan`, `np.arctan2`, `np.hypot`, `np.sinh`, `np.cosh`
Yuvarlama ve işaret	`np.abs`, `np.sign`, `np.floor`, `np.ceil`, `np.rint` (en yakın tamsayıya)
Karşılaştırma	`np.maximum`, `np.minimum`, `np.greater` (`>`), `np.less` (`<`), `np.equal` (`==`)

Sık kullanılan `np.round` (verilen basamağa yuvarlama) ve `np.isclose` (yaklaşık eşitlik) da eleman eleman çalışır, ama ufunc değildir. Bu yüzden ufunc'ların birazdan göreceğimiz `reduce`, `accumulate` gibi yöntemleri onlarda yoktur.
Python ile İlk Adımlar bölümündeki `math` modülünün fonksiyonları ise yalnız tek bir sayı kabul eder:

```
import math
import numpy as np

x = np.linspace(0, 1, 3)
try:
    math.sin(x)                # math yalnız tek sayı kabul eder
except TypeError as err:
    print("Hata:", err)
print(np.sin(x))               # np.sin dizinin her elemanına uygulanır
print(math.sin(0.5), np.sin(0.5))
```

Çıktı:

```
Hata: only 0-dimensional arrays can be converted to Python scalars
[0.         0.47942554 0.84147098]
0.479425538604203 0.479425538604203
```

Kural basittir: dizilerle çalışırken matematik fonksiyonlarını `np` içinden alın. Tek bir sayı için `math.sin` ile `np.sin` aynı değeri verir.

 CC BY-NC-SA 4.0

⚠ Geçersiz işlemler hata vermez, nan ve inf üretir

Python'da `math.sqrt(-1)` ya da `1 / 0` hata verip programı durdurur. NumPy ise dizinin geri kalanını hesaplamaya devam eder. Geçersiz sonucun yerine `nan` ("not a number", sayı değil), sıfıra bölmede ya da taşmada `inf` veya `-inf` yazar ve yalnız bir `RuntimeWarning` uyarısı basar. Aşağıda `np.errstate` bu uyarıları susturuyor ki çıktı yalnız sonuçları göstersin:

```
import numpy as np

x = np.array([4.0, 1.0, 0.0, -1.0])
with np.errstate(divide="ignore", invalid="ignore"):
    r = np.sqrt(x)
    lg = np.log(x)
    inv = 1 / x
print(r)
print(lg)
print(inv)
print(np.isnan(r), np.isfinite(lg))
```

Çıktı:

```
[ 2.  1.  0. nan]
[1.38629436 0.               -inf        nan]
[ 0.25  1.     inf -1.  ]
[False False False  True] [ True  True False False]
```

`nan` sonraki her işleme bulaşır: `nan + 1` de, `nan` içeren bir dizinin toplamı da `nan`'dır. Şüpheli sonuçları `np.isnan` ve `np.isfinite` ile ayıklayın; `x == np.nan` her zaman `False` verir.

İki girdili ufunc'ların birkaç yararlı yöntemi de vardır. `reduce` işlemi bütün diziye art arda uygulayıp tek sayıya indirir, `accumulate` ara sonuçları da tutar, `outer` ise bütün (x_i, y_j) çiftleri için tablo kurar:

```
import numpy as np

n = np.arange(1, 8)
print(np.add.reduce(n))             # 1 + 2 + ... + 7
print(np.add.accumulate(n))         # kısmi toplamlar
print(np.multiply.accumulate(n))    # 1!, 2!, ..., 7!
print(np.multiply.outer(n[:4], n[:4]))
```

Çıktı:

```
28
[ 1  3  6 10 15 21 28]
[   1    2    6   24  120  720 5040]
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]
 [ 4  8 12 16]]
```

`np.multiply.accumulate` ilk yedi faktöriyeli, `np.multiply.outer` ise küçük bir çarpım tablosunu tek satırda verdi. Uygulamada `np.add.reduce` yerine eşdeğeri olan `np.sum`, `np.add.accumulate` yerine `np.cumsum` yazılır; bunları birazdan indirgeme işlemleriyle birlikte göreceğiz.

Örnek 6.1 (Kartezyenden Kutupsal Koordinatlara). $(1,1)$, $(0,2)$, $(-1,0)$, $(-1,-1)$ ve $(3,-4)$ noktalarının kutupsal koordinatlarını, yani $r = \sqrt{x^2+y^2}$ uzaklığını ve $(-\pi, \pi]$ aralığındaki θ açısını döngü kullanmadan hesaplayın.

> **Çözüm.**
> Apsisleri ve ordinatları iki ayrı dizide tutarız. r için `np.hypot(x, y)` evrensel fonksiyonu $\sqrt{x_i^2+y_i^2}$ değerini her nokta için hesaplar. Açı için `np.arctan(y / x)` kullanamayız. $x=0$ olan $(0,2)$ noktasında sıfıra bölmüş oluruz. Ayrıca arctan yalnız $(-\pi/2, \pi/2)$ aralığında değer aldığından $(-1,-1)$ ile $(1,1)$

 CC BY-NC-SA 4.0

noktalarına aynı açıyı verir. `np.arctan2(y, x)` ise noktanın hangi bölgede olduğuna da bakarak doğru açıyı verir.

```
import numpy as np

x = np.array([1.0, 0.0, -1.0, -1.0, 3.0])
y = np.array([1.0, 2.0, 0.0, -1.0, -4.0])
r = np.hypot(x, y)              # sqrt(x**2 + y**2)
theta = np.arctan2(y, x)        # (-pi, pi] aralığında açı
print(r)
print(np.round(np.degrees(theta), 2))

# Geri dönüşüm: x = r cos(theta), y = r sin(theta)
print(np.allclose(r * np.cos(theta), x),
      np.allclose(r * np.sin(theta), y))
```

Çıktı:

```
[1.41421356 2.         1.         1.41421356 5.        ]
[  45.     90.    180.   -135.    -53.13]
True True
```

Açılar derece cinsinden $45°, 90°, 180°, -135°$ ve yaklaşık $-53{,}13°$'dir. $(-1,-1)$ noktası üçüncü bölgede olduğu için açısı $-135°$ çıktı, $(1,1)$ ile karışmadı. Son satır $x = r\cos\theta$ ve $y = r\sin\theta$ ile geri dönüşümün bütün noktalar için tuttuğunu `np.allclose` ile doğruluyor. Bu fonksiyon iki dizinin eleman eleman yuvarlama hatası kadar yakın olup olmadığını sınar. ■

6.2 İndirgeme İşlemleri ve axis Parametresi

Bir dizinin toplamı, en büyük elemanı ya da ortalaması gibi sayılar, diziyi tek bir sayıya **indirgeyen** işlemlerle bulunur.

```
import numpy as np

v = np.array([3.0, -1.0, 4.0, 1.0, -5.0, 9.0])
print(v.sum(), v.prod(), v.mean())
print(v.min(), v.max(), v.argmin(), v.argmax())
print(v.std(), np.median(v))
print(np.cumsum(v))
```

Çıktı:

```
11.0 540.0 1.8333333333333333
-5.0 9.0 4 5
4.336537277085896 2.0
[ 3.  2.  6.  7.  2. 11.]
```

`argmin` ve `argmax` en küçük ve en büyük elemanın değerini değil, **indeksini** verir: en büyük eleman olan 9, dizinin 5 numaralı yerindedir. `cumsum` kısmi toplamları, `std` standart sapmayı verir. `std` varsayılan olarak n'ye böler; örneklem standart sapması için `v.std(ddof=1)` yazılır. Tamsayı dizilerinde `prod` kolayca `int64` sınırını aşar. NumPy Dizileri bölümünde gördüğümüz gibi bu taşma sessizdir ve örneğin $21!$ negatif bir sayı olarak çıkar.

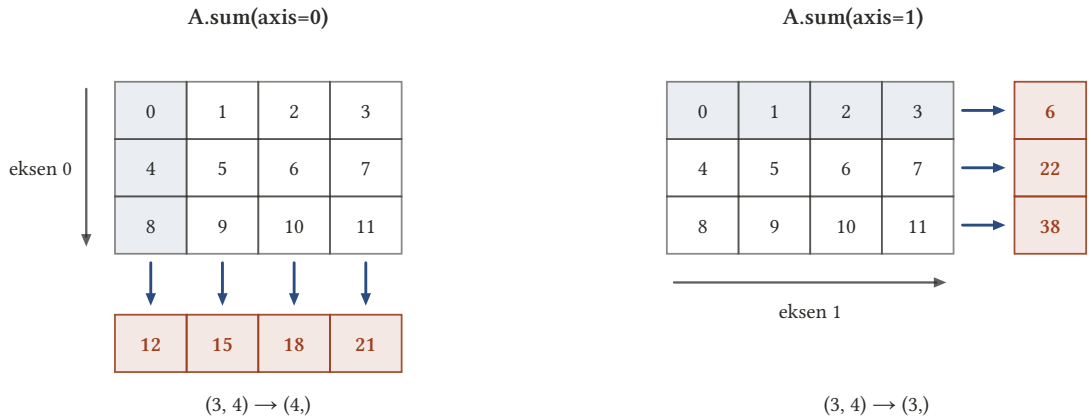
Çok boyutlu dizilerde çoğu zaman bütün elemanların toplamını değil, satır ya da sütun toplamlarını isteriz. Hangi eksen boyunca toplanacağını, yani NumPy Dizileri bölümünde tanıdığımız eksenlerden hangisinin kullanılacağını `axis` parametresi belirler.

Tanım 6.2 (Bir Eksen Boyunca İndirgeme). Şekli $(n_0, n_1, \ldots, n_{d-1})$ olan bir A dizisinin k numaralı eksen boyunca toplamı, `A.sum(axis=k)`, k-ıncı indeks üzerinden toplanarak elde edilen dizidir. Bu dizinin şekli, A'nın şeklinden n_k çıkarılarak bulunur. Bir matris ($d = 2$) için `A.sum(axis=0)` dizisinin j'inci

 CC BY-NC-SA 4.0

elemanı $\sum_i A_{ij}$, `A.sum(axis=1)` dizisinin i'inci elemanı ise $\sum_j A_{ij}$ olur. `prod`, `mean`, `std`, `min`, `max`, `argmin` ve `argmax` da `axis` parametresini aynı anlamda kullanır. `axis` verilmezse bütün elemanlar üzerinden tek bir sonuç hesaplanır.

Yani `axis` ile adı verilen eksen sonuçtan **kaybolur**. Bir matriste `axis=0` satır indeksi i'yi yok eder: her sütun aşağı doğru toplanır ve sütun sayısı kadar sonuç çıkar. `axis=1` ise sütun indeksi j'yi yok eder ve her satırın toplamını verir.



Şekil 6.1: A = np.arange(12).reshape(3, 4) matrisinde iki indirgeme. Solda `axis=0`: toplama 0. eksen boyunca, yani satırlar üzerinden yapılır, her sütun tek sayıya iner (ilk sütun 0 + 4 + 8 = 12) ve sonuç (4,) şeklindedir. Sağda `axis=1`: her satır tek sayıya iner (ilk satır 0 + 1 + 2 + 3 = 6) ve sonuç (3,) şeklindedir. Adı verilen eksen sonuçtan kaybolur.

```
import numpy as np

A = np.arange(12).reshape(3, 4)
print(A)
print(A.sum(axis=0))        # sütun toplamları, şekil (4,)
print(A.sum(axis=1))        # satır toplamları, şekil (3,)
print(A.sum())              # bütün elemanların toplamı
print(A.max(axis=0), A.argmax(axis=1))
print(A.sum(axis=1, keepdims=True))
print(A.cumsum(axis=1))     # kısmi toplamlar, şekil (3, 4) korunur

B = np.arange(24).reshape(2, 3, 4)
print(B.sum(axis=1).shape, B.sum(axis=(0, 2)))
```

Çıktı:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[12 15 18 21]
[ 6 22 38]
66
[ 8  9 10 11] [3 3 3]
[[ 6]
 [22]
 [38]]
[[ 0  1  3  6]
 [ 4  9 15 22]
```

 CC BY-NC-SA 4.0

```
 [ 8 17 27 38]]
(2, 4) [ 60  92 124]
```

`keepdims=True` verilirse kaybolan eksen silinmez, uzunluğu 1 olarak kalır: satır toplamları (3,) yerine (3, 1) şeklinde bir sütun olarak gelir. Bu, birazdan göreceğimiz broadcasting ile birlikte çok işe yarar. `cumsum` ve `cumprod` ise indirgeme değildir: `axis` ile verilen eksen boyunca kısmi toplamları (çarpımları) hesaplarlar ve dizinin şeklini korurlar. Yukarıda `A.cumsum(axis=1)` her satırın kısmi toplamlarını verdi ve sonuç yine (3, 4) şeklinde çıktı. `axis` verilmezse dizi önce düzleştirilir; `A.cumsum()` 12 elemanlı, tek eksenli bir dizidir. Son satırda üç boyutlu bir dizi var. `axis=1` ortadaki uzunluğu 3 olan ekseni yok ettiği için sonuç (2, 4) şeklindedir. `axis=(0, 2)` ise iki ekseni birden yok eder.

Örnek 6.2 (Dürer'in Sihirli Karesi). Albrecht Dürer'in 1514 tarihli *Melencolia I* gravüründeki

$$M = \begin{pmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{pmatrix}$$

karesinde bütün satır ve sütun toplamlarının, iki köşegenin toplamının ve dört 2×2 köşe bloğunun toplamlarının 34 olduğunu NumPy ile gösterin.

Çözüm.
Satır toplamları `axis=1`, sütun toplamları `axis=0` ile gelir. Köşegen toplamı izdir (`np.trace`). Ters köşegen için matrisi `np.fliplr` ile soldan sağa çevirip izini alırız. Dört köşe bloğu için 4×4 matrisi `reshape(2, 2, 2, 2)` ile dört indeksli bir diziye çevirirız. Yeni dizide M_{ij} elemanı, $i = 2a + b$ ve $j = 2c + e$ olmak üzere (a, b, c, e) konumundadır. (a, c) hangi blokta olduğumuzu, (b, e) blok içindeki konumu gösterir. Blok içi indeksler 1 ve 3 numaralı eksenlerdir, bu yüzden onlar üzerinden toplarız.

```
import numpy as np

M = np.array([[16, 3, 2, 13],
              [5, 10, 11, 8],
              [9, 6, 7, 12],
              [4, 15, 14, 1]])
print("satırlar  :", M.sum(axis=1))
print("sütunlar  :", M.sum(axis=0))
print("köşegen   :", np.trace(M))
print("ters köş. :", np.trace(np.fliplr(M)))
print("çeyrekler :", M.reshape(2, 2, 2, 2).sum(axis=(1, 3)).ravel())
```

Çıktı:

```
satırlar  : [34 34 34 34]
sütunlar  : [34 34 34 34]
köşegen   : 34
ters köş. : 34
çeyrekler : [34 34 34 34]
```

Hepsi 34'tür. Bu sayı rastgele değildir: $1, 2, \ldots, 16$ sayılarının toplamı 136'dır. Dört satırın toplamı eşitse ortak değer zorunlu olarak $136/4 = 34$ olur. Yani 1'den 16'ya kadar sayılarla kurulan her 4×4 sihirli karenin sabiti 34'tür. ■

Örnek 6.3 (Basel Serisinin Kısmi Toplamları). $S_n = \sum_{k=1}^{n} 1/k^2$ kısmi toplamlarının hepsini $n = 10^6$'ya kadar tek bir `cumsum` çağrısıyla hesaplayın. $n = 10, 100, \ldots, 10^6$ için $\pi^2/6 - S_n$ farkını yazdırıp farkın nasıl küçüldüğünü yorumlayın.

Çözüm.

 CC BY-NC-SA 4.0

Kontrol Yapıları ve Fonksiyonlar bölümünde bu toplamları iç içe döngülerle, her n için baştan hesaplamıştık. Şimdi hepsini tek seferde buluruz. $k = 1, 2, \ldots, 10^6$ dizisini `np.arange` ile kurarız. `1.0 / k**2` dizisi serinin terimlerini, bunun `cumsum`'ı da bütün kısmi toplamları verir. Python indeksleri 0'dan başladığı için S_n değeri `S[n - 1]` konumundadır.

```
import numpy as np

N = 10**6
k = np.arange(1, N + 1)
S = np.cumsum(1.0 / k**2)       # S[n - 1] = 1 + 1/4 + ... + 1/n^2
target = np.pi**2 / 6

for n in [10, 100, 1000, 10**4, 10**5, 10**6]:
    gap = target - S[n - 1]
    print(f"n = {n:>7}   S_n = {S[n - 1]:.10f}   "
          f"fark = {gap:.3e}   n * fark = {n * gap:.4f}")
```

Çıktı:

```
n =      10   S_n = 1.5497677312   fark = 9.517e-02   n * fark = 0.9517
n =     100   S_n = 1.6349839002   fark = 9.950e-03   n * fark = 0.9950
n =    1000   S_n = 1.6439345667   fark = 9.995e-04   n * fark = 0.9995
n =   10000   S_n = 1.6448340718   fark = 1.000e-04   n * fark = 1.0000
n =  100000   S_n = 1.6449240669   fark = 1.000e-05   n * fark = 1.0000
n = 1000000   S_n = 1.6449330668   fark = 1.000e-06   n * fark = 1.0000
```

Son sütundaki $n \cdot (\pi^2/6 - S_n)$ değerleri 1'e gidiyor, yani kalan yaklaşık $1/n$'dir. Bu, integral testinden gelen

$$\frac{1}{n+1} \le \frac{\pi^2}{6} - S_n \le \frac{1}{n}$$

kestirimiyle uyumludur (bkz. Analiz 2). Seri yavaş yakınsar: bir milyon terimden sonra bile hata 10^{-6} mertebesindedir. ■

6.3 Boolean Maskeler ve np.where

Karşılaştırma işleçleri de evrensel fonksiyondur ve sonuçları `True`/`False` değerlerinden oluşan **boolean** dizilerdir. Böyle bir maskeyle eleman seçmeyi (`x[mask]`), koşulları `&` (ve), `|` (veya), `~` (değil) ile birleştirmeyi ve `mask.sum()` ile saymayı NumPy Dizileri bölümünde görmüştük. Maskenin sayaç olarak birkaç kullanımı daha vardır:

```
import numpy as np

x = np.arange(-3, 4)
mask = x > 0
print(mask)
print(mask.sum(), mask.mean())      # True sayısı ve oranı
print(np.any(x > 2), np.all(x > -5))
```

Çıktı:

```
[False False False False  True  True  True]
3 0.42857142857142855
True True
```

`True` değeri 1, `False` değeri 0 sayıldığından `mask.sum()` koşulu sağlayan eleman sayısını, `mask.mean()` ise bunların oranını verir: yedi elemanın üçü pozitiftir ve oran $3/7 \approx 0{,}43$'tür. `np.any` en az bir elemanın, `np.all` bütün elemanların koşulu sağlayıp sağlamadığını söyler.

 CC BY-NC-SA 4.0

⚠ Parantezsiz koşul zinciri de hata verir

NumPy Dizileri bölümünde `and`, `or` ve `not` sözcüklerinin dizilerle çalışmadığını görmüştük. Parantezi unutmak da aynı hataya götürür. `&` işlecinin önceliği karşılaştırmalardan yüksek olduğu için `x > -2 & x < 2` ifadesi `x > (-2 & x) < 2` diye okunur. Bu bir zincirleme karşılaştırmadır ve Python onu içeride `and` ile çözer. Yani yine bir dizinin bütün olarak doğru olup olmadığını sormuş oluruz:

```
import numpy as np

x = np.arange(-3, 4)
try:
    x[x > -2 & x < 2]              # parantez yok: -2 & x önce hesaplanır
except ValueError as err:
    print("Hata:", str(err).split(".")[0])   # mesajın ilk cümlesi
print(x[(x > -2) & (x < 2)])       # doğrusu: her koşul parantez içinde
```

Çıktı:

```
Hata: The truth value of an array with more than one element is ambiguous
[-1  0  1]
```

Maskeyle eleman seçmek diziyi kısaltır. Çoğu zaman ise dizinin şekli korunsun, koşulu sağlayan ve sağlamayan elemanlar farklı değerler alsın isteriz. Parçalı tanımlı fonksiyonlar tam olarak böyledir.

Tanım 6.3 (np.where ile Eleman Eleman Seçim). c bir boolean dizisi, a ve b aynı şekle sahip diziler ya da sayılar olsun. `np.where(c, a, b)` çağrısı, her i için

$$z_i = \begin{cases} a_i, & c_i \text{ doğru ise,} \\ b_i, & \text{diğer durumlarda} \end{cases}$$

elemanlarından oluşan z dizisini verir. Tek argümanla çağrılan `np.where(c)` ise c'nin doğru olduğu indeksleri döndürür.

Yani `np.where`, "eğer ... ise ... değilse ..." kalıbının eleman eleman çalışan biçimidir. Örneğin

$$f(x) = \begin{cases} 1 - x^2, & |\,x\,| \le 1, \\ 0, & \text{diğer durumlarda} \end{cases}$$

fonksiyonu tek satırda hesaplanır:

```
import numpy as np

x = np.linspace(-2, 2, 9)
f = np.where(np.abs(x) <= 1, 1 - x**2, 0.0)
print(x)
print(f)
print(np.where(x > 0))      # tek argüman: koşulun doğru olduğu indeksler
print(np.where(x > 0, x, -x) == np.abs(x))
```

Çıktı:

```
[-2.  -1.5 -1.  -0.5  0.   0.5  1.   1.5  2. ]
[0.   0.   0.   0.75 1.   0.75 0.   0.   0.  ]
(array([5, 6, 7, 8]),)
[ True  True  True  True  True  True  True  True  True]
```

Son satır, `np.where(x > 0, x, -x)` ifadesinin tam olarak mutlak değer olduğunu doğruluyor. Tek argümanlı çağrı bir **demet** döndürür. Dizinin her ekseni için bir indeks dizisi vardır, burada eksen tek olduğu için demetin tek elemanı var.

 CC BY-NC-SA 4.0

Örnek 6.4 (Parçalı Fonksiyonun İntegrali). Yukarıdaki f fonksiyonunun $[-2, 2]$ aralığındaki integralini, aralığı $n = 4000$ eşit parçaya bölen orta nokta toplamıyla hesaplayın ve kesin değer $4/3$ ile karşılaştırın.

Çözüm.
$h = (b-a)/n$ olmak üzere alt aralıkların orta noktaları $m_i = a + (i + 1/2)h$, $i = 0, 1, \ldots, n-1$ noktalarıdır. Orta nokta toplamı $h \sum_i f(m_i)$'dir (Riemann toplamı için bkz. Analiz 2). Orta noktaların hepsi bir dizide, fonksiyon değerleri `np.where` ile, toplam da `sum` ile bulunur:

```
import numpy as np

n = 4000
a, b = -2.0, 2.0
h = (b - a) / n
m = a + h * (np.arange(n) + 0.5)        # alt aralıkların orta noktaları
f = np.where(np.abs(m) <= 1, 1 - m**2, 0.0)
approx = h * f.sum()
print(approx, 4 / 3, abs(approx - 4 / 3))
```

Çıktı:

```
1.3333335 1.3333333333333333 1.66666666689963e-07
```

Kesin değer $\int_{-1}^{1} (1 - x^2)\,dx = 4/3$'tür. Hata yaklaşık $1{,}67 \cdot 10^{-7}$'dir. Hata yalnız f'nin sıfır olmadığı $[-1, 1]$ aralığından gelir. Orta nokta kuralının bu aralıktaki hatası, aralığın uzunluğu L olmak üzere $Lh^2 \mid f'' \mid /24$ formülüyle verilir. Burada $L = 2$, $h^2 = 10^{-6}$ ve $\mid f'' \mid = 2$ olduğundan bu formül tam olarak $1{,}67 \cdot 10^{-7}$ değerini verir. Kırılma noktaları ± 1 ızgaranın düğümlerine denk geldiği için fazladan bir hata da oluşmadı. ■

⚠ np.where iki kolu da hesaplar

`np.where(c, a, b)` çağrılmadan önce `a` ve `b` dizileri **bütünüyle** hesaplanır; seçim ondan sonra yapılır. `np.where(x >= 0, np.sqrt(x), 0.0)` yazınca karekök negatif sayılar için de alınır. Sonuç doğru çıkar ama bir uyarı basılır. Aşağıda `np.errstate(invalid="raise")` uyarıyı hataya çevirerek bunu görünür kılıyor. Geçersiz değerler fonksiyona hiç gönderilmemelidir: ya girdi önceden düzeltilir (`np.maximum(x, 0)`) ya da ufunc'ın `where=` parametresi kullanılır.

```
import numpy as np

x = np.array([-4.0, -1.0, 0.0, 1.0, 4.0])
with np.errstate(invalid="raise"):      # uyarıyı hataya çevir
    try:
        y = np.where(x >= 0, np.sqrt(x), 0.0)
    except FloatingPointError as err:
        print("Hata:", err)
    y1 = np.sqrt(np.maximum(x, 0.0))
    y2 = np.sqrt(x, out=np.zeros_like(x), where=x >= 0)
print(y1)
print(y2)
```

Çıktı:

```
Hata: invalid value encountered in sqrt
[0. 0. 0. 1. 2.]
[0. 0. 0. 1. 2.]
```

Maskelerin güzel bir uygulaması, bir olayın olasılığını rastgele denemelerle kestirmektir. $[0,1) \times [0,1)$ karesine düzgün dağılımla nokta atarsak bir noktanın $x^2 + y^2 \le 1$ çeyrek dairesine düşme olasılığı alanların oranı, yani $\pi/4$ olur. Öyleyse içeri düşen noktaların oranının 4 katı π için bir kestirimdir:

```
import numpy as np

rng = np.random.default_rng(2024)
n = 1000
P = rng.random((n, 2))                  # birim karede n nokta, şekil (n, 2)
inside = (P**2).sum(axis=1) <= 1.0      # çeyrek dairenin içinde mi?
print(inside[:8])
print(inside.sum(), 4 * inside.mean())

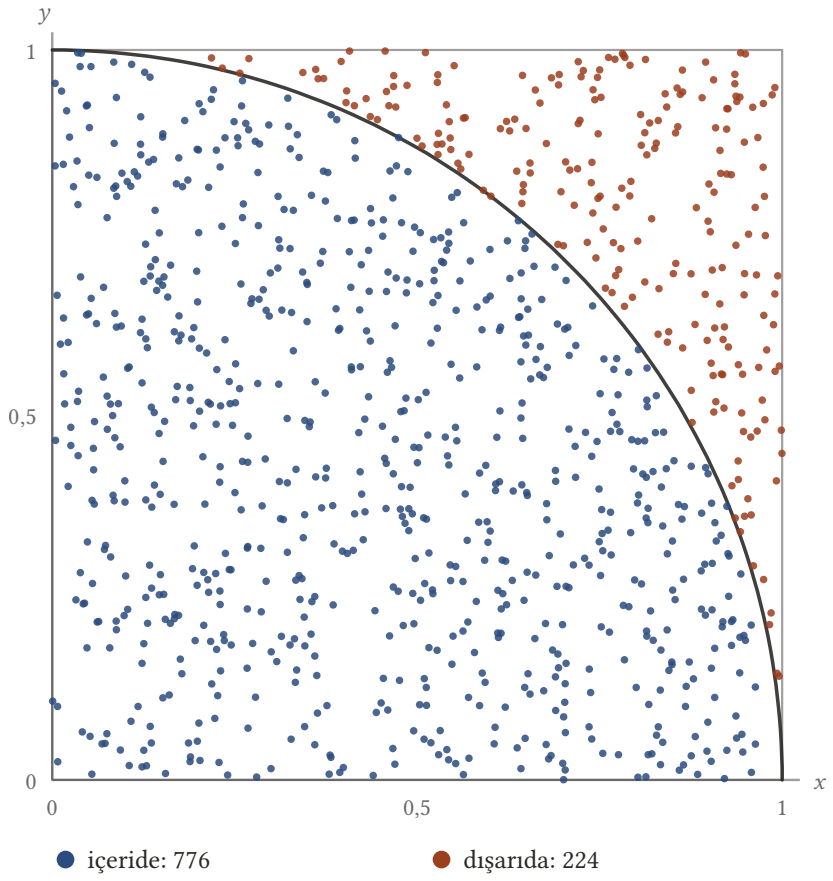
for n in [10**4, 10**5, 10**6]:
    P = rng.random((n, 2))
    print(n, 4 * np.mean((P**2).sum(axis=1) <= 1.0))
```

Çıktı:

```
[ True  True False  True  True  True  True  True]
776 3.104
10000 3.162
100000 3.1472
1000000 3.141404
```

`rng.random((n, 2))` her satırı bir nokta olan $n \times 2$ bir dizi üretir. `(P**2).sum(axis=1)` her noktanın $x^2 + y^2$ değerini verir, karşılaştırma bunu maskeye çevirir ve `mean` içerideki noktaların oranını hesaplar. Noktalar üzerinde hiçbir döngü yok; tek döngü, denenen n değerleri üzerindedir. Kestirim n büyüdükçe π'ye yaklaşır, ama yavaş: hata kabaca $1/\sqrt{n}$ ile küçülür. Rastgele sayı üretecini, bu hata hızını ve Monte Carlo yöntemlerini Rastgele Sayılar ve Monte Carlo Yöntemleri bölümünde ayrıntılı ele alacağız.

 CC BY-NC-SA 4.0



Şekil 6.2: `default_rng(2024)` ile üretilen ilk 1000 nokta. $x^2 + y^2 \le 1$ maskesi 776 noktada doğrudur (mavi). Çeyrek dairenin alanı $\pi/4$ olduğundan $4 \cdot 776/1000 = 3{,}104$ değeri π için bir kestirimdir.

Örnek 6.5 (Kaydırılmış Maskelerle İkiz Asallar). p ile $p+2$ sayılarının ikisi de asalsa bu çifte ikiz asal çifti denir. İkisi de 10^6'dan küçük olan ikiz asal çiftlerinin sayısını, Eratosthenes kalburunun maskesini kaydırarak döngüsüz bulun.

Çözüm.
Kalbur maskesini NumPy Dizileri bölümündeki gibi kurarız; yalnız N bu kez 10^6'dır. `is_prime[p]` değeri p'nin asal olup olmadığını söyler. Python döngüsü yalnız $p \le 1000$ için döner; asıl iş, yani yaklaşık 2,1 milyon silme, vektörel dilim atamalarında yapılır.
Yeni olan kısım ikiz asallardır. `is_prime[:-2]` dilimi $p = 0, 1, \ldots, N-3$ için p'nin, `is_prime[2:]` dilimi ise aynı konumda $p+2$'nin asal olup olmadığını söyler. İki dilim aynı uzunluktadır, bu yüzden `&` ile birleşimleri tam olarak p ile $p+2$'nin ikisinin de asal olduğu konumlarda `True` olur. `np.flatnonzero`, tek argümanlı `np.where` gibi, maskenin `True` olduğu indeksleri verir.

```
import numpy as np

N = 10**6
is_prime = np.ones(N, dtype=bool)       # 0, 1, ..., N - 1 için bayrak
is_prime[:2] = False                    # 0 ve 1 asal değil
for p in range(2, int(N**0.5) + 1):
    if is_prime[p]:
        is_prime[p * p::p] = False      # p*p, p*p + p, ... siliniyor
print(is_prime.sum())                   # N'den küçük asalların sayısı
```

 CC BY-NC-SA 4.0

```
twin = is_prime[:-2] & is_prime[2:]     # twin[p]: p ve p + 2 asal mı?
first = np.flatnonzero(twin)            # her çiftin küçük elemanı
print(first[:8], first[-2:])
print(twin.sum())
```

Çıktı:

```
78498
[ 3  5 11 17 29 41 59 71] [999611 999959]
8169
```

10^6'dan küçük 78498 asal ve 8169 ikiz asal çifti vardır. İlk sekiz çift, Sınıflar, Hata Yakalama ve Dosyalar bölümünde 80'e kadar deneme bölmesiyle bulduğumuz $(3, 5), (5, 7), \ldots, (71, 73)$ çiftleridir; sonuncusu $(999959, 999961)$'dir. Orada her sayı ayrı ayrı sınanıyordu. Burada bir milyon sayının hepsi tek bir `&` işlemiyle sınandı. İkiz asal çiftlerinin sonsuz sayıda olup olmadığı hâlâ açık bir sorudur. ■

6.4 Broadcasting

Şimdiye kadar ya aynı şekildeki iki diziyi ya da bir diziyle bir sayıyı işledik. `A + 5` yazınca 5 sayısı `A`'nın her elemanına eklendi. Şekilleri farklı iki dizi için de benzer bir kural vardır:

```
import numpy as np

A = np.arange(12).reshape(3, 4)
row = np.array([100, 200, 300, 400])        # şekil (4,)
col = np.array([[1000], [2000], [3000]])    # şekil (3, 1)
print(A + 5)
print(A + row)
print(A + col)
try:
    A + np.array([1, 2, 3])                 # şekil (3,)
except ValueError as err:
    print("Hata:", err)
```

Çıktı:

```
[[ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
[[100 201 302 403]
 [104 205 306 407]
 [108 209 310 411]]
[[1000 1001 1002 1003]
 [2004 2005 2006 2007]
 [3008 3009 3010 3011]]
Hata: operands could not be broadcast together with shapes (3,4) (3,)
```

(4,) şekilli `row` dizisi matrisin **her satırına**, (3, 1) şekilli `col` sütunu ise **her sütununa** eklendi. (3,) şekilli bir dizi ise eklenemedi. Hangi şekillerin birlikte işlenebileceğini aşağıdaki kurallar belirler.

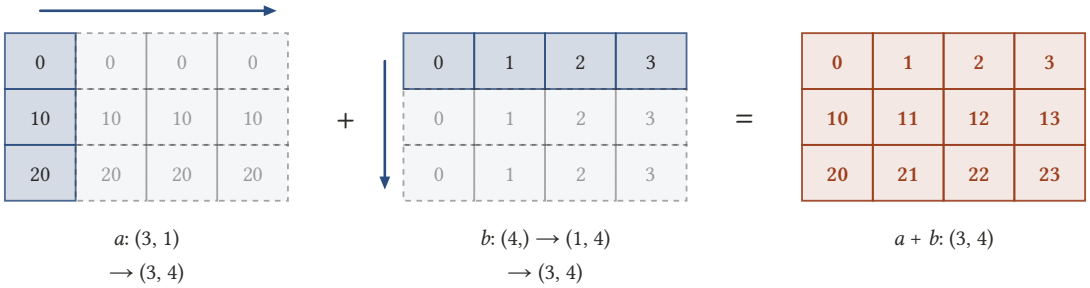
Tanım 6.4 (Broadcasting Kuralları). Şekilleri $(a_1, \ldots, a_p)$ ve $(b_1, \ldots, b_q)$ olan iki dizi bir ufunc ile birlikte işlenirken şu adımlar izlenir:

1. Kısa olan şeklin **soluna** 1'ler eklenerek iki dizinin eksen sayısı eşitlenir.
2. Şekiller sağdan sola eksen eksen karşılaştırılır. Her eksende iki uzunluk ya eşit olmalı ya da biri 1 olmalıdır. Aksi hâlde işlem `ValueError` hatası verir.
3. Sonucun her eksendeki uzunluğu iki uzunluğun büyüğüdür. Uzunluğu 1 olan eksen, o uzunluğa kadar **kopyalanmış gibi** davranır; bu kopyalar bellekte oluşturulmaz.

Bu kurallara **broadcasting** (yayma) denir. Bir sayı, şekli $()$ olan bir dizi gibi işlem görür.

 CC BY-NC-SA 4.0

Yani küçük dizi, büyük dizinin şekline “yayılır”. Uzunluğu 1 olan bir eksen istenen uzunluğa kadar uzatılır, eksik eksenler sola eklenir. Yukarıda (3, 4) ile (4,) şekilleri uyumluydu: (4,) önce (1, 4) olur, sonra satır üç kez kopyalanmış gibi davranır. (3, 4) ile (3,) ise uyumsuzdur, çünkü (3,) dizisi (1, 3) olur ve son eksende 4 ile 3 çatışır. İki dizi aynı anda da yayılabilir. (3, 1) ile (4,) şekillerinden (3, 4) şekilli bir sonuç çıkar:



Şekil 6.3: Broadcasting ile (3, 1) şekilli a ve (4,) şekilli b toplanır. Önce b'nin şekli soldan 1 eklenerek (1, 4) olur. Sonra uzunluğu 1 olan eksenler karşı tarafın uzunluğuna uzatılır: a'nın sütunu sağa, b'nin satırı aşağı kopyalanmış gibi davranır. Kesikli çizilen kopyalar bellekte yoktur. Sonuç, `a + b` çıktısındaki (3, 4) şekilli dizidir.

```
import numpy as np

a = np.array([[0], [10], [20]])     # şekil (3, 1)
b = np.arange(4)                    # şekil (4,)
print(a + b)
print(np.broadcast_shapes((3, 1), (4,)))
print(np.broadcast_shapes((5, 3), (3,)))
print(np.broadcast_shapes((8, 1, 6, 1), (7, 1, 5)))
```

Çıktı:

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]]
(3, 4)
(5, 3)
(8, 7, 6, 5)
```

`np.broadcast_shapes` bir hesap yapmadan yalnız sonucun şeklini söyler. Kuralların adım adım uygulanışı şöyledir:

Birinci	İkinci	Hizalanmış şekiller	Sonuç
(3, 1)	(4,)	(3, 1) ve (1, 4)	(3, 4)
(5, 3)	(3,)	(5, 3) ve (1, 3)	(5, 3)
(8, 1, 6, 1)	(7, 1, 5)	(8, 1, 6, 1) ve (1, 7, 1, 5)	(8, 7, 6, 5)
(3, 4)	(3,)	(3, 4) ve (1, 3)	hata: 4 ile 3

Broadcasting’den yararlanmak için çoğu zaman tek eksenli bir diziyi sütun ya da satır şekline çevirmek gerekir. Bunun için dizinin indeksine `np.newaxis` (ya da onun kısaltması olan `None`) yazılır. Bu, o konuma uzunluğu 1 olan yeni bir eksen ekler:

```
import numpy as np

x = np.array([1, 2, 3])
print(x.shape, x[:, np.newaxis].shape, x[np.newaxis, :].shape)
```

 CC BY-NC-SA 4.0

```
print(x[:, None].shape, x.reshape(-1, 1).shape)
print(x[:, None] - x)       # D[i, j] = x[i] - x[j]
```

Çıktı:

```
(3,) (3, 1) (1, 3)
(3, 1) (3, 1)
[[ 0 -1 -2]
 [ 1  0 -1]
 [ 2  1  0]]
```

`x[:, None] - x` ifadesinde (3, 1) ile (3,) şekilleri (3, 3) şekline yayılır ve $D_{ij} = x_i - x_j$ farklarının tablosu çıkar. Bu "sütun eksi satır" kalıbı, bütün çiftler üzerinden yapılan hesapların temelidir.

Örnek 6.6 (Çarpım Tablosu). $1, 2, \ldots, 9$ sayılarının 9×9 çarpım tablosunu broadcasting ile kurun. Tablonun elemanları toplamının $(1 + 2 + \cdots + 9)^2 = 2025$ olduğunu ve tablonun simetrik olduğunu doğrulayın.

Çözüm.
n dizisini sütun yaparız: `n[:, None]` (9, 1), `n` ise (9,) şeklindedir. Çarpımları (9, 9) şekline yayılır ve $T_{ij} = n_i n_j$ olur.

```
import numpy as np

n = np.arange(1, 10)            # şekil (9,)
T = n[:, None] * n              # (9, 1) ile (9,) -> (9, 9)
print(T[:4, :6])
print(T.shape, T.sum(), n.sum()**2)
print(np.array_equal(T, np.outer(n, n)), np.array_equal(T, T.T))
```

Çıktı:

```
[[ 1  2  3  4  5  6]
 [ 2  4  6  8 10 12]
 [ 3  6  9 12 15 18]
 [ 4  8 12 16 20 24]]
(9, 9) 2025 2025
True True
```

$$\sum_{i,j} n_i n_j = \Big(\sum_i n_i\Big)\Big(\sum_j n_j\Big) = 45^2 = 2025$$

olduğundan iki toplam eşit çıktı. `np.outer` aynı tabloyu kuran hazır fonksiyondur. Simetri $T = T^{T}$ eşitliğidir; `T.T` matrisin transpozudur. ■

Örnek 6.7 (Bir Veri Matrisini Standartlaştırma). Her satırı bir ölçüm, her sütunu bir değişken olan 6×3 bir X veri matrisinde her sütundan o sütunun ortalamasını çıkarıp sonucu sütunun standart sapmasına bölün. Bu işleme standartlaştırma, sonuçlara z-değerleri denir. Elde edilen Z matrisinin her sütununun ortalamasının 0, standart sapmasının 1 olduğunu doğrulayın.

Çözüm.
Sütun ortalamaları `X.mean(axis=0)` ile gelir ve (3,) şeklindedir. (6, 3) şekilli X'ten çıkarılınca (1, 3) şekline, oradan da (6, 3) şekline yayılır. Yani her satırdan aynı ortalama vektörü çıkarılır. Standart sapmalara bölme de aynı biçimde çalışır:

$$Z_{ij} = \frac{X_{ij} - \mu_j}{\sigma_j}.$$

 CC BY-NC-SA 4.0

Örnek veriyi `rng.normal` ile üretiyoruz. Bu çağrı normal dağılımlı rastgele sayılar verir: `loc` ortalamayı, `scale` yayılımı (standart sapmayı) belirler. Burada ikisi de (3,) şekilli birer dizidir ve `size=(6, 3)` şekline yayılır. Böylece her sütun kendi ortalaması ve yayılımıyla üretilir.

```
import numpy as np

rng = np.random.default_rng(7)
X = rng.normal(loc=[10.0, 50.0, -3.0], scale=[1.0, 8.0, 0.5],
               size=(6, 3))     # 6 ölçüm, 3 değişken
mu = X.mean(axis=0)             # şekil (3,): sütun ortalamaları
sigma = X.std(axis=0)           # şekil (3,): sütun std sapmaları
Z = (X - mu) / sigma            # (6, 3) ile (3,) -> (6, 3)
print(np.round(X, 2))
print(np.round(Z, 3))
print(np.allclose(Z.mean(axis=0), 0), np.allclose(Z.std(axis=0), 1))
```

Çıktı:

```
[[10.   52.39 -3.14]
 [ 9.11 46.36 -3.5 ]
 [10.06 60.72 -3.25]
 [ 9.38 53.92 -2.82]
 [10.11 42.56 -3.01]
 [10.7  39.25 -3.23]]
[[ 0.211  0.439  0.097]
 [-1.512 -0.391 -1.623]
 [ 0.325  1.587 -0.426]
 [-0.99   0.65   1.61 ]
 [ 0.413 -0.915  0.684]
 [ 1.553 -1.37  -0.343]]
True True
```

Sütunların ortalamaları (yaklaşık 10, 50 ve -3) ve yayılımları çok farklıydı. Standartlaştırmadan sonra üçü de aynı ölçeğe geldi. Ortalamalar yuvarlama hatası kadar sıfıra yakın olduğundan karşılaştırmayı `==` ile değil `np.allclose` ile yaptık. ■

⚠ (n,) ile (n, 1) aynı şey değildir

Broadcasting hata vermesi gereken bazı yerlerde sessizce çalışır. En sık görülen durum, (n,) şekilli bir vektörle (n, 1) şekilli bir sütunun farkıdır. İkisi de "n elemanlı bir vektör" gibi görünür, ama farkları (n, n) şekilli bir tablo olur:

```
import numpy as np

y = np.array([1.0, 2.0, 3.0, 4.0])                  # şekil (4,)
y_hat = np.array([[1.1], [1.9], [3.2], [3.8]])      # şekil (4, 1)
r = y - y_hat
print(r.shape)                  # (4,) beklenirken (4, 4)
r = y - y_hat.ravel()           # ya da y_hat[:, 0]
print(r.shape, r)
```

Çıktı:

```
(4, 4)
(4,) [-0.1  0.1 -0.2  0.2]
```

Bir hesabın sonucu beklenmedik biçimde büyük çıkıyorsa ilk iş ara sonuçların `.shape` değerlerini yazdırmaktır.

 CC BY-NC-SA 4.0

İki değişkenli bir fonksiyonu bir ızgarada hesaplamak broadcasting'in en doğal kullanımıdır. x değerleri satır, y değerleri sütun yapılırsa $F_{ij} = f(x_j, y_i)$ tablosu tek ifadeyle çıkar. Aynı tabloyu `np.meshgrid` ile de kurabiliriz. Bu fonksiyon iki koordinat dizisini tam ızgara şekline kopyalar; Matplotlib ile Grafik Çizimi bölümündeki eş yükselti eğrileri ve yüzeyler bu dizileri kullanır.

```
import numpy as np

x = np.linspace(-1, 1, 5)                       # şekil (5,)
y = np.linspace(0, 1, 3)                        # şekil (3,)
F = x[np.newaxis, :]**2 + y[:, np.newaxis]      # şekil (3, 5)
print(F)
X, Y = np.meshgrid(x, y)                        # ikisi de şekil (3, 5)
print(X.shape, np.array_equal(F, X**2 + Y))
```

Çıktı:

```
[[1.   0.25 0.   0.25 1.  ]
 [1.5  0.75 0.5  0.75 1.5 ]
 [2.   1.25 1.   1.25 2.  ]]
(3, 5) True
```

Burada $f(x, y) = x^2 + y$'dir. Tablonun satırları y'ye, sütunları x'e karşılık gelir; `meshgrid` de varsayılan olarak bu düzeni kullanır. Broadcasting sürümü (3, 5) şekilli iki tam diziyi bellekte oluşturmaz, bu yüzden büyük ızgaralarda daha az bellek harcar.

Örnek 6.8 (Bir Çemberin İçindeki Kafes Noktaları). $x^2 + y^2 \le r^2$ eşitsizliğini sağlayan (x, y) tamsayı çiftlerinin sayısı $N(r)$ olsun. $r = 10, 100, 1000$ için $N(r)$'yi broadcasting ile hesaplayın ve πr^2 ile karşılaştırın.

Çözüm.
Kontrol Yapıları ve Fonksiyonlar bölümünde $N(r)$'yi iç içe iki döngüyle $r = 20$'ye kadar saymıştık. Şimdi döngülerin yerini bir tablo alır. Hem x hem y, $-r, \dots, r$ tamsayılarından birini alır. k bu tamsayıların dizisi olsun. `k[:, None]**2 + k**2` ifadesi $(2r+1) \times (2r+1)$ şekilli bir tabloda bütün $x^2 + y^2$ değerlerini verir. Bunu r^2 ile karşılaştırıp maskedeki `True` değerlerini sayarız:

```
import numpy as np

for r in [10, 100, 1000]:
    k = np.arange(-r, r + 1)                    # şekil (2r + 1,)
    inside = k[:, None]**2 + k**2 <= r**2       # şekil (2r+1, 2r+1)
    N = inside.sum()
    print(f"r = {r:>4}   N(r) = {N:>7}   pi r^2 = {np.pi * r**2:12.2f}"
          f"   fark = {N - np.pi * r**2:8.2f}")
```

Çıktı:

```
r =   10   N(r) =     317   pi r^2 =       314.16   fark =     2.84
r =  100   N(r) =   31417   pi r^2 =     31415.93   fark =     1.07
r = 1000   N(r) = 3141549   pi r^2 =   3141592.65   fark =   -43.65
```

Orada birim karelerle gerekçelendirdiğimiz $N(r) \approx \pi r^2$ yaklaşımı çok daha büyük r değerlerinde de tutuyor: fark, πr^2'nin yanında çok küçüktür. Farkın büyüme hızı Gauss çember problemi olarak bilinen ünlü bir açık sorudur. $r = 1000$ için tablo yaklaşık 4 milyon elemanlıdır ve yine de bir anda hesaplanır. ■

 CC BY-NC-SA 4.0

6.5 Uygulama: Uzaklık Matrisi

Broadcasting'in en çok kullanılan uygulamalarından biri, bir nokta kümesindeki bütün çiftlerin uzaklıklarını tek seferde hesaplamaktır. Düzlemde m nokta $m \times 2$ şekilli bir P dizisinde, her satır bir nokta olacak biçimde tutulsun. Öklid uzaklığı (bkz. Analiz 4)

$$D_{ij} = \|P_i - P_j\| = \sqrt{(P_{i1} - P_{j1})^2 + (P_{i2} - P_{j2})^2}$$

formülüyle verilir. Burada P_{i1} ve P_{i2}, i'inci noktanın iki koordinatıdır. `P[:, None, :]` (m, 1, 2), `P[None, :, :]` ise (1, m, 2) şeklindedir. Farkları (m, m, 2) şekline yayılır: `diff[i, j]` dizisi $P_i - P_j$ vektörüdür. Son eksen üzerinden kareler toplanıp karekök alınınca (m, m) şekilli uzaklık matrisi çıkar.

```
import numpy as np
from scipy.spatial.distance import cdist

rng = np.random.default_rng(5)
P = np.round(10 * rng.random((6, 2)), 1)        # 6 nokta, şekil (6, 2)
diff = P[:, None, :] - P[None, :, :]            # şekil (6, 6, 2)
D = np.sqrt((diff**2).sum(axis=-1))             # şekil (6, 6)
print(P)
print(np.round(D, 2))
print(np.allclose(D, D.T), np.allclose(D, cdist(P, P)))

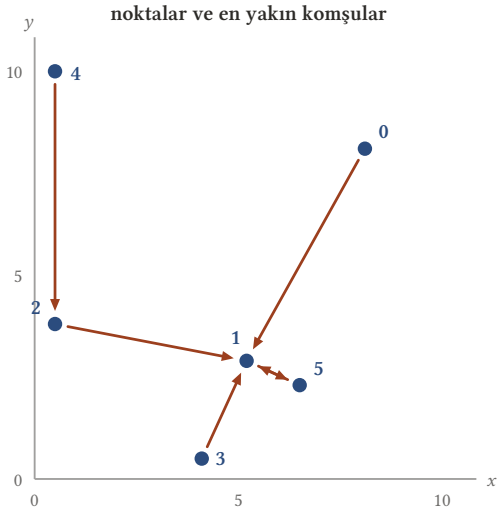
E = D.copy()
np.fill_diagonal(E, np.inf)                     # kendisi aday olmasın
print(E.argmin(axis=1))                         # en yakın komşular
```

Çıktı:

```
[[ 8.1  8.1]
 [ 5.2  2.9]
 [ 0.5  3.8]
 [ 4.1  0.5]
 [ 0.5 10. ]
 [ 6.5  2.3]]
[[ 0.    5.95  8.73  8.59  7.83  6.02]
 [ 5.95  0.    4.79  2.64  8.51  1.43]
 [ 8.73  4.79  0.    4.88  6.2   6.18]
 [ 8.59  2.64  4.88  0.   10.16  3.  ]
 [ 7.83  8.51  6.2  10.16  0.    9.76]
 [ 6.02  1.43  6.18  3.    9.76  0.  ]]
True True
[1 5 1 1 2 1]
```

Matris simetriktir ve köşegeni sıfırdır. SciPy'ın `cdist` fonksiyonu da aynı sonucu verdi. En yakın komşuyu bulmak için her satırın köşegen dışındaki en küçük elemanının yerini ararız. Köşegeni `np.inf` yaparsak bir nokta kendisini seçemez ve `argmin(axis=1)` her nokta için en yakın komşunun indeksini verir.

 CC BY-NC-SA 4.0



D[i, j] = |P[i] − P[j]|

	0	1	2	3	4	5
0	0,00	5,95	8,73	8,59	7,83	6,02
1	5,95	0,00	4,79	2,64	8,51	1,43
2	8,73	4,79	0,00	4,88	6,20	6,18
3	8,59	2,64	4,88	0,00	10,16	3,00
4	7,83	8,51	6,20	10,16	0,00	9,76
5	6,02	1,43	6,18	3,00	9,76	0,00

Şekil 6.4: Solda `default_rng(5)` ile üretilen altı nokta; her ok bir noktadan en yakın komşusuna gider (1 ile 5 birbirinin en yakın komşusudur). Sağda broadcasting ile hesaplanan uzaklık matrisi D: uzaklık küçüldükçe hücre koyulaşır, her satırda köşegen dışındaki en küçük eleman çerçevelidir. Çerçeveli sütunlar `E.argmin(axis=1)` sonucu olan [1 5 1 1 2 1] dizisidir.

Bu yöntemin bir bedeli vardır: ara dizi `diff` $m^2 d$ eleman tutar (d uzayın boyutudur). $m = 10^4$ nokta ve $d = 3$ için bu $3 \cdot 10^8$ sayı, yani 2,4 GB bellek eder. Vektörizasyon hızı çoğu zaman bellekle satın alır. Çok büyük nokta kümelerinde hesap parçalara bölünür ya da `cdist` gibi ara dizi kurmayan hazır fonksiyonlar kullanılır.

6.6 Döngü mü, Vektör mü?

Vektörel kodun daha hızlı olduğunu söyledik; şimdi bunu ölçelim. Python'un `timeit` modülü bir ifadeyi birçok kez çalıştırıp süresini ölçer. Bilgisayarda aynı anda çalışan başka işler ölçümü yalnız yavaşlatabileceği için tekrarların en küçüğünü almak âdettir. Bir milyon sayının karelerinin toplamını üç yolla hesaplayalım:

```
import timeit
import numpy as np

n = 10**6
x = np.random.default_rng(0).random(n)
xs = x.tolist()                 # aynı sayılar, Python listesi olarak


def sum_squares_index(a):
    total = 0.0
    for i in range(len(a)):         # dizinin elemanlarına tek tek eriş
        total += a[i] * a[i]
    return total


def sum_squares_loop(values):
    total = 0.0
    for v in values:
        total += v * v
    return total


print(sum_squares_index(x), sum_squares_loop(xs), np.sum(x * x))
```

```
def best_time(f, number):
    return min(timeit.repeat(f, number=number, repeat=7)) / number


t_index = best_time(lambda: sum_squares_index(x), 1)
t_loop = best_time(lambda: sum_squares_loop(xs), 1)
t_vec = best_time(lambda: np.sum(x * x), 10)
print(f"dizi üzerinde döngü  : {t_index * 1e3:6.0f} ms")
print(f"liste üzerinde döngü : {t_loop * 1e3:6.0f} ms")
print(f"vektörel             : {t_vec * 1e3:6.1f} ms")
```

Çıktı:

```
333560.6157428297 333560.6157428297 333560.61574281874
dizi üzerinde döngü  :    128 ms
liste üzerinde döngü :     19 ms
vektörel             :    1.4 ms
```

Üç yöntem de aynı toplamı verdi. Döngünün sonucuyla `np.sum`'ınki son basamaklarda farklıdır. `np.sum` sayıları gruplara ayırıp önce grup toplamlarını alır (ikili toplama), bu yüzden yuvarlama hatası daha az birikir. Süreler bu bölümün yazıldığı bilgisayarda ölçülmüştür; sizin makinenizde farklı çıkacaktır, ama oranların mertebesi genellikle benzer kalır. Vektörel toplam liste üzerindeki döngüden yaklaşık 14 kat, dizi üzerindeki döngüden yaklaşık 90 kat hızlıdır.

Dizi üzerindeki döngünün listeden bile yavaş olması şaşırtıcı görünebilir. Bunun nedeni, `a[i]` ile her erişimde NumPy'ın makine sayısını bir Python nesnesine sarmak zorunda olmasıdır. Döngü her elemanda bu paketlemeyi, tür denetimini ve yorumlayıcının işini tekrarlar. Vektörel kodda ise bu masraf dizi başına bir kez ödenir ve elemanlar üzerindeki döngü derlenmiş kodda, bellekte art arda duran sayılar üzerinde döner. Farkın n ile nasıl değiştiğini görmek için ölçümü farklı boyutlarda tekrarlayalım. Tabloya bir de `np.vectorize` ile "vektörleştirilmiş" bir Python fonksiyonu ekleyelim:

```
import timeit
import numpy as np

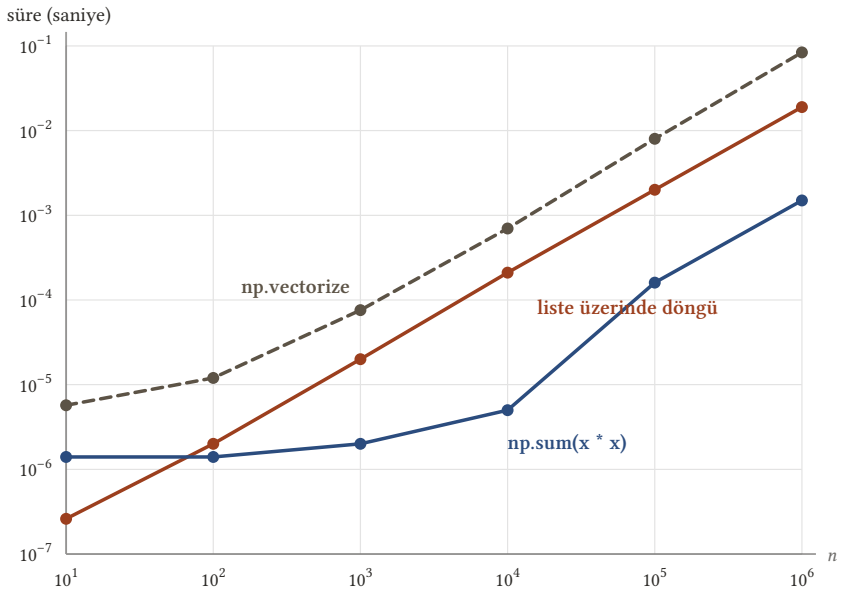

def sum_squares_loop(values):
    total = 0.0
    for v in values:
        total += v * v
    return total


square = np.vectorize(lambda v: v * v)
rng = np.random.default_rng(0)
print("     n    döngü (s)  vectorize (s)  vektörel (s)")
for n in [10, 100, 10**3, 10**4, 10**5, 10**6]:
    x = rng.random(n)
    xs = x.tolist()
    reps = max(1, 10**5 // n)
    times = []
    for f in [lambda: sum_squares_loop(xs),
              lambda: np.sum(square(x)),
              lambda: np.sum(x * x)]:
        t = min(timeit.repeat(f, number=reps, repeat=5)) / reps
        times.append(t)
    print(f"{n:>7}   {times[0]:9.1e}   {times[1]:11.1e}"
          f"   {times[2]:11.1e}")
```

Çıktı:

 CC BY-NC-SA 4.0

```
      n    döngü (s)  vectorize (s)  vektörel (s)
     10     2.6e-07       5.7e-06       1.4e-06
    100     2.0e-06       1.2e-05       1.4e-06
   1000     2.0e-05       7.6e-05       2.0e-06
  10000     2.1e-04       7.0e-04       5.0e-06
 100000     2.0e-03       8.0e-03       1.6e-04
1000000     1.9e-02       8.4e-02       1.5e-03
```



Şekil 6.5: Zaman taraması kodunun tablosu, iki ekseni de logaritmik ölçekte. Döngünün süresi n ile doğru orantılı büyür (eğim 1). Vektörel toplamın küçük n için yaklaşık 1 mikrosaniyelik sabit bir başlangıç maliyeti vardır, büyük n'de de döngüden 10 ile 40 kat arasında hızlıdır. `np.vectorize` içeride yine Python döngüsü çalıştırdığı için düz döngüden de yavaştır. Sayılar makineye göre değişir; eğrilerin biçimi değişmez.

Tablo üç şey söylüyor. Birincisi, çok küçük dizilerde (10 eleman) vektörel kodun yaklaşık 1 mikrosaniyelik sabit başlangıç maliyeti döngüden pahalıdır; vektörizasyon büyük dizilerde kazandırır. İkincisi, $n \geq 10^3$ için vektörel kod yaklaşık 10 ile 40 kat hızlıdır. Vektörel eğrinin 10^4 ile 10^5 arasındaki sıçraması, `x * x` ara dizisi için bellek ayırma ve önbellek etkilerinden gelir; ölçümün makineden makineye, hatta çalıştırmadan çalıştırmaya en çok değişen kısmı budur. Üçüncüsü, `np.vectorize` hiç hızlandırmaz.

> ⚠ np.vectorize bir hızlandırma aracı değildir
>
> `np.vectorize(f)`, tek sayı alan bir Python fonksiyonunu dizilere uygulanabilir hâle getirir. Ama bunu içeride bir Python döngüsüyle yapar. Kodu kısaltır, ama yukarıdaki tabloda görüldüğü gibi düz döngüden bile yavaştır. Gerçek vektörizasyon, hesabı ufunc'lar, dilimler, maskeler ve broadcasting ile yeniden yazmaktır.

6.7 Uygulama: Sayısal Türev

Türev tanımındaki fark bölümü, kaydırılmış dizilerin farkı olarak tek satırda yazılabilir. $[0, 2\pi]$ aralığını 16 eşit parçaya bölen $x_0, x_1, \ldots, x_{16}$ noktalarında $y_i = \sin x_i$ değerleri bilinsin ve $h = 2\pi/16$ olsun. `np.diff(y)` dizisi $y_{i+1} - y_i$ farklarını verir; bu farklar 17 değil 16 tanedir. `y[2:] - y[:-2]` ise $y_{i+1} - y_{i-1}$ farklarıdır; bunlar iç noktalara aittir ve sayıları 15'tir.

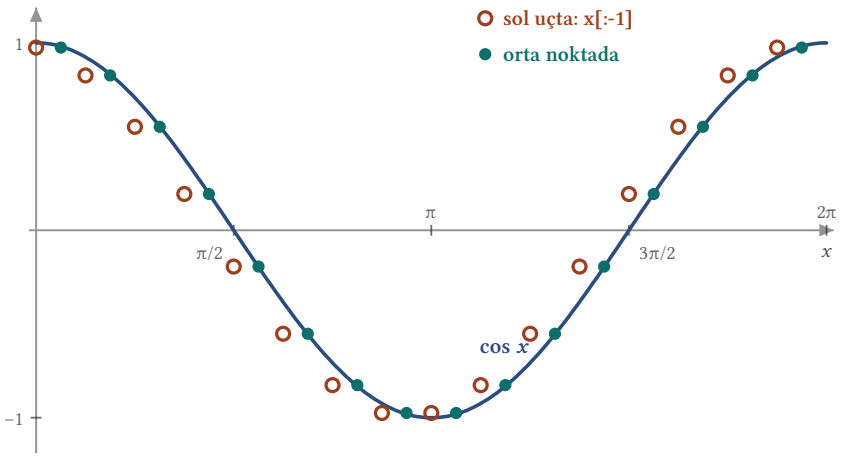
```
import numpy as np

n = 17
x = np.linspace(0, 2 * np.pi, n)        # 16 eşit alt aralık
h = x[1] - x[0]
y = np.sin(x)
d_fwd = np.diff(y) / h                  # n - 1 değer
d_cen = (y[2:] - y[:-2]) / (2 * h)      # n - 2 değer, x[1:-1] için
mid = (x[:-1] + x[1:]) / 2              # alt aralıkların orta noktaları
print(round(h, 4), len(d_fwd), len(d_cen))
print(np.abs(d_fwd - np.cos(x[:-1])).max())   # sol uca yazılırsa
print(np.abs(d_fwd - np.cos(mid)).max())      # orta noktaya yazılırsa
print(np.abs(d_cen - np.cos(x[1:-1])).max())  # merkezi fark
```

Çıktı:

```
0.3927 16 15
0.19383917874071455
0.006289921998797743
0.025504641595567312
```

İleri fark bölümü $(y_{i+1} - y_i)/h$, x_i noktasındaki türeve ancak $O(h)$ doğrulukla yaklaşır (bkz. Nümerik Analiz). Aynı sayı x_i ile x_{i+1} arasındaki **orta noktadaki** türeve ise $O(h^2)$ doğrulukla yaklaşır. Yani hangi değerin hangi noktaya ait olduğu, hesabın kendisi kadar önemlidir. Aynı 16 sayı sol uçlara yazılınca en büyük hata 0,19, orta noktalara yazılınca 0,0063 çıktı. Merkezi fark formülü $(y_{i+1} - y_{i-1})/(2h)$ iç noktalarda yine $O(h^2)$ doğruluk verir (bkz. Nümerik Analiz).



Şekil 6.6: sin x fonksiyonunun $[0, 2\pi]$ aralığındaki 17 noktalık ızgarada `np.diff(y) / h` ile bulunan 16 fark bölümü ($h \approx 0{,}39$). Değerler alt aralıkların sol uçlarına yazılınca (boş daireler) cos x eğrisinin yarım adım soluna kayar, en büyük hata 0,19 olur. Aynı sayılar orta noktalara yazılınca (dolu daireler) eğrinin üstüne oturur, en büyük hata 0,0063'e iner.

Dizi üzerinde fark formülü üç adımda

1. Noktaları bir `x` dizisinde kur ve `h` adımını belirle. Fonksiyon değerlerini tek ifadeyle hesapla: `f(x)`, `f(x + h)`, … ya da ızgarada `y = f(x)`.
2. Formülü kaydırılmış dizilerle yaz: ızgarada `y[1:]`, `y[:-1]`, `y[2:]` gibi dilimlerle, fonksiyon biliniyorsa `f(x + h) - f(x - h)` gibi ifadelerle. Birden çok `h` değerini aynı anda denemek için `h`'yi sütun yap.
3. Sonucun hangi noktalara ait olduğunu belirle (`x[1:-1]`, orta noktalar, …). Kesin türev biliniyorsa farkın mutlak değerinin en büyüğünü uygun `axis` üzerinden al.

 CC BY-NC-SA 4.0

Örnek 6.9 (Merkezi Farkın Hatası ve Adım Boyu). $f(x) = \sin x$ için merkezi fark formülü $\big(\, f(x+h) - f(x-h) \,\big)/(2h)$ ile $[0, 2\pi]$ aralığındaki 201 noktada türevi $h = 10^{-1}, 10^{-2}, 10^{-3}$ adımlarıyla hesaplayın. Her h için en büyük hatayı bulun ve hatanın h^2 ile orantılı küçüldüğünü gösterin.

Çözüm.
1. adım. `x` (201,) şekilli ızgaradır. `h` ise (3, 1) şeklinde bir sütun olarak kurulur.
2. adım. `np.sin(x + h)` ifadesinde (3, 1) ile (201,) şekilleri (3, 201) şekline yayılır. Böylece her satır bir h değeri için 201 noktadaki fark bölümlerini tutar.
3. adım. Sonuç `x` noktalarının kendisine aittir. Kesin türev $\cos x$'ten farkın mutlak değeri `axis=1` boyunca, yani her h için ayrı ayrı en büyüğe indirgenir.

```
import numpy as np

x = np.linspace(0, 2 * np.pi, 201)              # şekil (201,)
h = np.array([1e-1, 1e-2, 1e-3])[:, None]       # şekil (3, 1)
D = (np.sin(x + h) - np.sin(x - h)) / (2 * h)   # şekil (3, 201)
err = np.abs(D - np.cos(x)).max(axis=1)         # şekil (3,)
for hk, ek in zip(h.ravel(), err):
    print(f"h = {hk:.0e}   en büyük hata = {ek:.3e}"
          f"   h^2/6 = {hk**2 / 6:.3e}")
print(err[:-1] / err[1:])
```

Çıktı:

```
h = 1e-01   en büyük hata = 1.666e-03   h^2/6 = 1.667e-03
h = 1e-02   en büyük hata = 1.667e-05   h^2/6 = 1.667e-05
h = 1e-03   en büyük hata = 1.667e-07   h^2/6 = 1.667e-07
[99.95051153 99.99943903]
```

Merkezi farkın hatası $\frac{h^2}{6} f'''(\xi)$'dir ve $|\, f''' \,| = |\cos| \le 1$ olduğundan en büyük hata $h^2/6$ civarında beklenir. Tablo bunu çok iyi doğruluyor. h on kat küçülünce hata yüz kat küçülüyor (son satırdaki oranlar), yani yöntem ikinci mertebedendir. h'yi çok küçültmenin bir sınırı da vardır, çünkü bir noktadan sonra yuvarlama hatası baskın çıkar. Bunu Sayısal Türev ve İntegral bölümünde inceleyeceğiz. ■

6.8 Alıştırmalar

Aşağıdaki alıştırmalarda döngü yerine ufunc'lar, indirgemeler, maskeler ve broadcasting kullanacağız. Her çözüm, kodu ve çalıştırılmış çıktısıyla birlikte verilmiştir; önce kendiniz deneyip sonra karşılaştırın.

Alıştırma 6.1 (Wallis Çarpımı). Wallis çarpımı $\dfrac{\pi}{2} = \displaystyle\prod_{k=1}^{\infty} \frac{4k^2}{4k^2 - 1}$ eşitliğini söyler (bkz. Analiz 2). $W_n = 2\prod_{k=1}^{n} \frac{4k^2}{4k^2-1}$ kısmi çarpımlarını $n = 10^5$'e kadar `np.cumprod` ile hesaplayın ve $\pi - W_n$ farkının n ile nasıl küçüldüğünü bulun.

Çözüm.
`k` bir tamsayı dizisidir, ama `/` bölmesi tamsayı dizilerinde de `float` sonuç verir; bu yüzden çarpanlar doğru hesaplanır. `np.cumprod` bütün kısmi çarpımları verir; W_n değeri `W[n - 1]` konumundadır.

```
import numpy as np

k = np.arange(1, 10**5 + 1)             # 1, 2, ..., 10^5
factors = 4 * k**2 / (4 * k**2 - 1)
W = 2 * np.cumprod(factors)             # W[n - 1]: ilk n çarpan
for n in [10, 100, 1000, 10**4, 10**5]:
    gap = np.pi - W[n - 1]
    print(f"n = {n:>6}   W_n = {W[n - 1]:.8f}   "
          f"fark = {gap:.3e}   n * fark = {n * gap:.4f}")
```

Çıktı:

```
n =     10   W_n = 3.06770381   fark = 7.389e-02   n * fark = 0.7389
n =    100   W_n = 3.13378749   fark = 7.805e-03   n * fark = 0.7805
n =   1000   W_n = 3.14080775   fark = 7.849e-04   n * fark = 0.7849
n =  10000   W_n = 3.14151412   fark = 7.853e-05   n * fark = 0.7853
n = 100000   W_n = 3.14158480   fark = 7.854e-06   n * fark = 0.7854
```

Son sütun $n \cdot (\pi - W_n)$ değerinin $0{,}7854 \approx \pi/4$ sayısına yaklaştığını gösteriyor. Yani $\pi - W_n \approx \dfrac{\pi}{4n}$'dir. Wallis çarpımı da Basel serisi gibi $1/n$ hızıyla, yavaş yakınsar: 10^5 çarpandan sonra bile hata yaklaşık $8 \cdot 10^{-6}$'dır. ■

Alıştırma 6.2 (Sütunların En Büyük Elemanları). `rng = np.random.default_rng(3)` üreteciyle `rng.integers(0, 100, size=(5, 4))` matrisini kurun. Her sütunun en büyük elemanını ve bu elemanın bulunduğu satırın indeksini döngü kullanmadan bulun.

Çözüm.
Sütun boyunca indirgeme `axis=0` ile yapılır. `A.max(axis=0)` en büyük değerleri, `A.argmax(axis=0)` bunların satır indekslerini verir. Bir sütunda en büyük değer birden çok kez geçerse `argmax` ilkinin indeksini döndürür. Son satırda fancy indeksleme yaparak (bkz. NumPy Dizileri) $A_{i_j j}$ elemanlarını alıyor ve sonucun en büyük değerlerle aynı olduğunu doğruluyoruz.

```
import numpy as np

rng = np.random.default_rng(3)
A = rng.integers(0, 100, size=(5, 4))
print(A)
print(A.max(axis=0))            # her sütunun en büyük elemanı
print(A.argmax(axis=0))         # bu elemanların satır indeksleri
cols = np.arange(A.shape[1])
print(A[A.argmax(axis=0), cols])
```

Çıktı:

```
[[81  8 17 23]
 [18 80 86 58]
 [ 3  9 33 43]
 [62 47 26 15]
 [69 73  3 11]]
[81 80 86 58]
[0 1 1 1]
[81 80 86 58]
```

Sütunların en büyükleri $81, 80, 86, 58$'dir. İlki 0 numaralı satırda, diğer üçü 1 numaralı satırdadır. ■

Alıştırma 6.3 (Satırları Normalleştirme). Elemanları negatif olmayan

$$W = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 3 & 1 \\ 1 & 1 & 6 \end{pmatrix}$$

matrisinin her satırını kendi toplamına bölerek satır toplamları 1 olan bir P matrisi kurun. Böyle matrislere satır stokastik matris denir; Markov zincirlerinin geçiş matrisleri bu türdendir.

Çözüm.

 CC BY-NC-SA 4.0

Satır toplamları `W.sum(axis=1)` ile bulunur, ama bu (3,) şekilli bir dizidir. (3, 3) şekilli W'yi buna bölersek (3,) dizisi (1, 3) şekline yayılır ve j'inci **sütun** s_j'ye bölünür; istediğimiz bu değildir. Toplamı `keepdims=True` ile (3, 1) şeklinde bir sütun olarak tutarsak i'inci **satır** s_i'ye bölünür:

```
import numpy as np

W = np.array([[2.0, 1.0, 1.0],
              [0.0, 3.0, 1.0],
              [1.0, 1.0, 6.0]])
s = W.sum(axis=1, keepdims=True)    # şekil (3, 1)
P = W / s                           # her satır kendi toplamına bölünür
print(s.ravel())
print(P)
print(P.sum(axis=1))
print(W / W.sum(axis=1))            # yanlış: j. sütun s_j'ye bölünür
```

Çıktı:

```
[4. 4. 8.]
[[0.5   0.25  0.25 ]
 [0.    0.75  0.25 ]
 [0.125 0.125 0.75 ]]
[1. 1. 1.]
[[0.5   0.25  0.125]
 [0.    0.75  0.125]
 [0.25  0.25  0.75 ]]
```

Satır toplamları $4, 4, 8$'dir. P'nin her satırının toplamı 1 çıktı. Son çıktı yanlış yolun sonucudur. Hata vermedi, çünkü matris karedir ve şekiller uyumludur. Ama sonuç, örneğin ilk satırı $(0{,}5;\ 0{,}25;\ 0{,}125)$, istediğimiz matris değildir. ■

Alıştırma 6.4 (Sonucun Şeklini Önceden Bulmak). `A = np.ones((6, 1, 4))` ve `B = np.arange(5).reshape(5, 1)` dizileri için `A + B` toplamının şeklini broadcasting kurallarıyla elle bulun, sonra kodla doğrulayın.

Çözüm.
1. kural. `B`'nin şekli (5, 1) iki eksenlidir; soluna 1 eklenince (1, 5, 1) olur.
2. kural. Sağdan sola karşılaştırırız: son eksende 4 ile 1, ortada 1 ile 5, ilk eksende 6 ile 1. Her çiftte ya uzunluklar eşit ya da biri 1'dir, yani şekiller uyumludur.
3. kural. Her eksende büyük uzunluk alınır: sonuç (6, 5, 4) şeklindedir.

```
import numpy as np

A = np.ones((6, 1, 4))
B = np.arange(5).reshape(5, 1)
C = A + B
print(C.shape, np.broadcast_shapes(A.shape, B.shape))
print(C[0])                 # C[i, j, k] = 1 + B[j, 0]
```

Çıktı:

```
(6, 5, 4) (6, 5, 4)
[[1. 1. 1. 1.]
 [2. 2. 2. 2.]
 [3. 3. 3. 3.]
```

```
 [4. 4. 4. 4.]
 [5. 5. 5. 5.]]
```

`C[i, j, k] = A[i, 0, k] + B[j, 0] = 1 + j` olduğundan her `C[i]` dilimi, j'inci satırı $1 + j$ olan 5×4 bir matristir. ■

Alıştırma 6.5 (İşaret Değişimiyle Kök Aralığı). $f(x) = x^3 - 2x - 5$ fonksiyonunu $[-3, 3]$ aralığında 0,25 adımlı ızgarada hesaplayın ve işaretin değiştiği alt aralıkları döngü kullanmadan bulun.

Çözüm.
`np.sign(f)` her değerin işaretini (-1, 0 ya da 1) verir. Komşu iki değerin işaretleri farklıysa çarpımları negatiftir: `s[:-1] * s[1:] < 0` maskesi $[x_i, x_{i+1}]$ aralığında işaret değişimi olan i indekslerinde doğrudur. `np.flatnonzero` bu indeksleri listeler.

```
import numpy as np

x = np.linspace(-3, 3, 25)              # adım 0.25
f = x**3 - 2 * x - 5
s = np.sign(f)
idx = np.flatnonzero(s[:-1] * s[1:] < 0)
print(idx)
for i in idx:
    print(x[i], x[i + 1], f[i], f[i + 1])
```

Çıktı:

```
[20]
2.0 2.25 -1.0 1.890625
```

İşaret yalnız $[2; 2{,}25]$ aralığında değişir: $f(2) = -1 < 0$, $f(2{,}25) = 1{,}890625 > 0$. f sürekli olduğundan Ara Değer Teoremi (bkz. Nümerik Analiz) gereği bu aralıkta bir kök vardır (bu kök yaklaşık 2,0946'dır). Böyle bir aralık, ikiye bölme gibi kök bulma yöntemleri için iyi bir başlangıçtır; bkz. Kök Bulma ve Optimizasyon. ■

Alıştırma 6.6 (Izgarada Yerel Maksimumlar). $f(x) = e^{-x/5} \sin x$ fonksiyonunu $[0, 20]$ aralığında 0,01 adımlı ızgarada hesaplayın. İki komşusundan da büyük olan iç ızgara noktalarını döngü kullanmadan bulun ve $f'(x) = 0$ koşulundan gelen kesin yerel maksimum noktalarıyla karşılaştırın.

Çözüm.
1. adım. Önce kesin yerleri bulalım. $f'(x) = e^{-x/5}\left(\cos x - \frac{1}{5}\sin x\right)$ olduğundan kritik noktalar $\tan x = 5$, yani $x = \arctan 5 + k\pi$ noktalarıdır. Parantezdeki ifadenin türevi $-\sin x - \frac{1}{5}\cos x$'tir. k çiftken $\sin x$ ve $\cos x$ pozitif olduğundan bu türev negatiftir. Yani f' artıdan eksiye geçer ve nokta bir yerel maksimumdur (bkz. Analiz 2). k tekken aynı akıl yürütme yerel minimum verir. $[0, 20]$ aralığına $\arctan 5 + 2m\pi$ biçimindeki maksimumlardan $m = 0, 1, 2$ için olanlar düşer; $m = 3$ için $x \approx 20{,}22$ aralığın dışında kalır.
2. adım. Ayrık maksimumları kaydırılmış dilimlerle buluruz. `y[1:-1]` iç noktaları, `y[:-2]` sol komşuları, `y[2:]` sağ komşuları hizalar. `(y[1:-1] > y[:-2]) & (y[1:-1] > y[2:])` maskesi iki komşusundan da büyük olan iç noktalarda doğrudur.
3. adım. Maske `x[1:-1]` noktalarına aittir. Bu yüzden `np.flatnonzero` ile bulunan indekslere 1 ekleyip `x` ve `y` içindeki indeksleri elde ederiz.

```
import numpy as np

x = np.linspace(0, 20, 2001)            # adım 0.01
```

 CC BY-NC-SA 4.0

```
y = np.exp(-x / 5) * np.sin(x)
peak = (y[1:-1] > y[:-2]) & (y[1:-1] > y[2:])   # iç noktalar için
i = np.flatnonzero(peak) + 1            # x ve y içindeki indeksler
print(x[i])
print(np.round(y[i], 4))
exact = np.arctan(5) + 2 * np.pi * np.arange(3)
print(np.round(exact, 4))
print(np.abs(x[i] - exact).max())
```

Çıktı:

```
[ 1.37  7.66 13.94]
[0.7451 0.212  0.0604]
[ 1.3734  7.6566 13.9398]
0.003413925875397794
```

Izgara üç yerel maksimum buldu. Bunların kesin noktalardan en büyük uzaklığı yaklaşık $0{,}0034$'tür, yani yarım adımdan ($0{,}005$) azdır. Maksimum değerler her seferinde aynı oranda küçülür: $f(x+2\pi) = e^{-2\pi/5} f(x)$ olduğundan ardışık iki maksimumun oranı $e^{-2\pi/5} \approx 0{,}285$'tir. Gerçekten $0{,}2120/0{,}7451 \approx 0{,}285$'tir. ■

Alıştırma 6.7 (Taylor Polinomunu Vandermonde Matrisiyle Hesaplamak). e^{-x} fonksiyonunun 0 etrafındaki üçüncü dereceden Taylor polinomu $p(x) = \sum_{j=0}^{3} c_j x^j$, $c_j = (-1)^j/j!$ olsun. p'yi $x = 0;\ 0{,}25;\ 0{,}5;\ 0{,}75;\ 1$ noktalarında, $V_{ij} = x_i^j$ matrisini broadcasting ile kurarak hesaplayın ve $\lvert\, p(x) - e^{-x} \,\rvert$ hatasını $x^4/24$ sınırıyla karşılaştırın.

Çözüm.
`x[:, None]` (5, 1), `k = np.arange(4)` ise (4,) şeklindedir. `x[:, None]**k` ifadesi (5, 4) şekline yayılır ve $V_{ij} = x_i^j$ kuvvetler tablosunu verir. Bu, NumPy Dizileri bölümünde sütunları yan yana koyarak kurduğumuz Vandermonde matrisidir; broadcasting onu tek ifadeyle verir ve sonucu yine `np.vander` ile karşılaştırabiliriz. (4,) şekilli c ile çarpım satırlara yayılır. `axis=1` boyunca toplam her nokta için $\sum_j c_j x_i^j = p(x_i)$ değerini verir.

```
import math
import numpy as np

x = np.linspace(0, 1, 5)                # şekil (5,)
k = np.arange(4)                        # şekil (4,)
c = np.array([(-1)**j / math.factorial(j) for j in k])
V = x[:, None]**k                       # şekil (5, 4): V[i, j] = x_i^j
p = (V * c).sum(axis=1)                 # p(x_i) = sum_j c_j x_i^j
print(V)
print(np.array_equal(V, np.vander(x, 4, increasing=True)))
print(p)
print(np.abs(p - np.exp(-x)))
print(x**4 / 24)
```

Çıktı:

```
[[1.       0.       0.       0.      ]
 [1.       0.25     0.0625   0.015625]
 [1.       0.5      0.25     0.125   ]
 [1.       0.75     0.5625   0.421875]
 [1.       1.       1.       1.      ]]
True
```

```
[1.         0.77864583 0.60416667 0.4609375  0.33333333]
[0.         0.00015495 0.00236399 0.01142905 0.03454611]
[0.         0.00016276 0.00260417 0.01318359 0.04166667]
```

Taylor teoremine göre kalan $R_3(x) = \frac{f^{(4)}(\xi)}{4!}x^4$'tür (bkz. Nümerik Analiz). Burada $f^{(4)}(\xi) = e^{-\xi} \le 1$ olduğundan $|\,R_3(x)\,| \le x^4/24$ olur. Son iki satırı karşılaştırınca her noktada gerçek hatanın bu sınırın altında kaldığı görülür. En büyük hata $x = 1$'de, $0{,}0345 < 0{,}0417$'dir. ■

Alıştırma 6.8 (En Yakın Nokta Çifti). `rng = np.random.default_rng(11)` üreteciyle birim karede `rng.random((200, 2))` noktalarını üretin. Birbirine en yakın iki noktayı uzaklık matrisiyle bulun ve sonucu iç içe iki döngüyle doğrulayın.

Çözüm.
Uzaklık matrisini broadcasting ile kurarız. Her çift matriste iki kez geçer ($D_{ij} = D_{ji}$) ve köşegen sıfırdır. Bu yüzden köşegeni ve altını `np.tril_indices` ile `np.inf` yaparız; geriye yalnız $i < j$ çiftleri kalır. `D.argmin()` düzleştirilmiş dizideki indeksi verir, `np.unravel_index` bunu (satır, sütun) çiftine çevirir.

```
import numpy as np

rng = np.random.default_rng(11)
P = rng.random((200, 2))                        # şekil (200, 2)
diff = P[:, None, :] - P[None, :, :]            # şekil (200, 200, 2)
D = np.sqrt((diff**2).sum(axis=-1))             # şekil (200, 200)
D[np.tril_indices(len(P))] = np.inf             # i >= j çiftleri dışarıda
i, j = np.unravel_index(D.argmin(), D.shape)
print(i, j, D[i, j])

# Kontrol: iç içe iki döngü, 19900 çift
best, pair = np.inf, None
for a in range(len(P)):
    for b in range(a + 1, len(P)):
        d = np.hypot(P[a, 0] - P[b, 0], P[a, 1] - P[b, 1])
        if d < best:
            best, pair = d, (a, b)
print(pair, best == D[i, j])
```

Çıktı:

```
40 83 0.001200238111042671
(40, 83) True
```

En yakın çift 40 ve 83 numaralı noktalardır; aralarındaki uzaklık yaklaşık $0{,}0012$'dir. Döngü aynı çifti ve aynı uzaklığı buldu, ama 19900 çifti tek tek dolaşarak. Vektörel çözüm bunun için $200 \times 200 \times 2$ elemanlı bir ara dizi kurdu. ■

Alıştırma 6.9 (Birim Kürenin Hacmi). `rng = np.random.default_rng(42)` üreteciyle $[-1, 1]^3$ küpünde 10^6 düzgün dağılımlı nokta üretin. Birim kürenin içine düşen noktaların oranından kürenin hacmini kestirin ve $4\pi/3$ ile karşılaştırın.

Çözüm.
`rng.uniform(-1, 1, size=(n, 3))` her satırı küpte bir nokta olan bir dizi üretir. `(P**2).sum(axis=1) <= 1` maskesi noktanın kürenin içinde olup olmadığını söyler. Küpün hacmi 8 olduğundan kürenin hacmi yaklaşık $8 \cdot$ (içerideki oran)'dır.

 CC BY-NC-SA 4.0

```
import numpy as np

rng = np.random.default_rng(42)
n = 10**6
P = rng.uniform(-1, 1, size=(n, 3))     # [-1, 1]^3 küpü, hacmi 8
inside = (P**2).sum(axis=1) <= 1.0
V = 8 * inside.mean()
print(inside.sum(), V)
print(4 * np.pi / 3, abs(V - 4 * np.pi / 3))
```

Çıktı:

```
523685 4.18948
4.1887902047863905 0.0006897952136091234
```

Bir milyon noktanın 523685'i küreye düştü. Kestirim 4,18948, gerçek değer $4\pi/3 \approx 4{,}18879$ ve fark yaklaşık $7 \cdot 10^{-4}$'tür. Bir noktanın küreye düşme olasılığı $p = \pi/6$'dır. İçerideki noktaların oranı $\hat{p}$ ise kestirim $8\hat{p}$'dir ve standart sapması $8\sqrt{p(1-p)/n} \approx 4 \cdot 10^{-3}$'tür. Bulunan fark bu beklenen hata düzeyinin rahatça içindedir. ■

Alıştırma 6.10 (İkinci Türev İçin Fark Formülü). $[0, 1]$ aralığında 0,01 adımlı ızgarada $f(x) = e^x$ değerleri verilsin. İç noktalarda $f''(x_i) \approx (y_{i+1} - 2y_i + y_{i-1})/h^2$ yaklaşımını dilimlerle hesaplayın. En büyük hatayı $h^2 e/12$ sınırıyla karşılaştırın.

Çözüm.
`y[2:]`, `y[1:-1]` ve `y[:-2]` dilimleri sırasıyla y_{i+1}, y_i ve y_{i-1} değerlerini iç noktalar için hizalar. Sonuç `x[1:-1]` noktalarına aittir ve 99 elemanlıdır.

```
import numpy as np

x = np.linspace(0, 1, 101)
h = x[1] - x[0]
y = np.exp(x)
d2 = (y[2:] - 2 * y[1:-1] + y[:-2]) / h**2     # x[1:-1] noktalarında
err = np.abs(d2 - np.exp(x[1:-1]))
print(len(d2), err.max(), x[1:-1][err.argmax()])
print(h**2 / 12 * np.e)
```

Çıktı:

```
99 2.242702939581065e-05 0.99
2.2652348570492043e-05
```

Formülün hatası $\frac{h^2}{12} f^{(4)}(\xi)$'dir (bkz. Nümerik Analiz). $[0, 1]$ aralığında $f^{(4)}(x) = e^x \le e$ olduğundan hata en çok $h^2 e/12 \approx 2{,}27 \cdot 10^{-5}$ olabilir. Bulunan en büyük hata $2{,}24 \cdot 10^{-5}$'tir ve e^x'in en büyük olduğu sağ uca yakın $x = 0{,}99$ noktasında oluşur. ■

Bu bölümde döngüleri dizi işlemlerine çevirmeyi öğrendik. Evrensel fonksiyonlar eleman eleman hesap yapar, `axis` parametresi indirgemenin yönünü seçer, maskeler ve `np.where` koşulları döngüsüz işler. Broadcasting ise şekilleri farklı dizileri kopyalamadan birleştirir. Sıradaki NumPy ile Lineer Cebir bölümünde aynı dizileri matris ve vektör olarak ele alacak, matris çarpımı, lineer sistemler ve özdeğerler için NumPy'ın hazır araçlarını kullanacağız.

 CC BY-NC-SA 4.0

7. NumPy ile Lineer Cebir

Bilimsel hesaplamadaki problemlerin büyük çoğunluğu sonunda bir lineer cebir problemine dönüşür. Bir diferansiyel denklemi sayısal olarak çözmek, bir veri kümesine eğri uydurmak ya da bir optimizasyon problemini çözmek çoğu zaman bir $Ax = b$ sistemini çözmeye ya da bir matrisin özdeğerlerini bulmaya varır. Kâğıt üzerinde 3×3 bir sistemi Gauss yöntemiyle birkaç dakikada çözeriz; bilgisayar 2000×2000 bir sistemi, bu bölümde ölçeceğimiz gibi, saniyenin onda biri kadar bir sürede çözer.

NumPy'ın `numpy.linalg` modülü bu hesapları onlarca yıldır geliştirilen ve sınanan LAPACK kütüphanesine devreder. Bu bölümde Lineer Cebir notlarından tanıdığımız işlemleri (matris çarpımı, ters matris, determinant, rank, özdeğerler) NumPy ile yapacağız. Tanımları ve ispatları yeniden anlatmayacağız; gerektiğinde Lineer Cebir notlarına bağlantı vereceğiz. Dizilerin nasıl kurulduğunu NumPy Dizileri, eleman eleman işlemleri ve broadcasting'i Vektörizasyon ve Broadcasting bölümünde gördük.

Asıl dikkatimiz, kayan noktalı sayılarla hesap yapmanın getirdiği farklarda olacak. Tekil bir matrisin determinantı 0 yerine $6{,}7 \cdot 10^{-18}$ çıkabilir; `solve` hiçbir uyarı vermeden bileşenleri 10^{15} mertebesinde olan anlamsız bir çözüm döndürebilir; ters matrisle çözmek hem daha yavaş hem de daha az doğrudur. Bu olguları açıklayan kavram **koşul sayısıdır**. Bölümü en küçük kareler problemi ve tekil değer ayrışımıyla kapatacağız.

7.1 Matris Çarpımı ve Temel İşlemler

NumPy'da bir matris, iki eksenli bir dizidir: birinci eksen satırları, ikinci eksen sütunları sayar. Matematikteki AB çarpımının Python'daki karşılığı `@` işlecidir.

```
import numpy as np

A = np.array([[1, 2],
              [3, 4]])
B = np.array([[0, 1],
              [1, 0]])

print(A @ B)    # matris çarpımı
print(B @ A)    # sıra değişince sonuç değişir
print(A * B)    # eleman eleman çarpım: matris çarpımı DEĞİL
```

Çıktı:

```
[[2 1]
 [4 3]]
[[3 4]
 [1 2]]
[[0 2]
 [3 0]]
```

B matrisi A'yı sağdan çarpınca A'nın sütunlarının, soldan çarpınca satırlarının yerini değiştirir; bu yüzden $AB \neq BA$'dır. Son satırdaki `A * B` ise matris çarpımı değildir: karşılıklı elemanları çarpar.

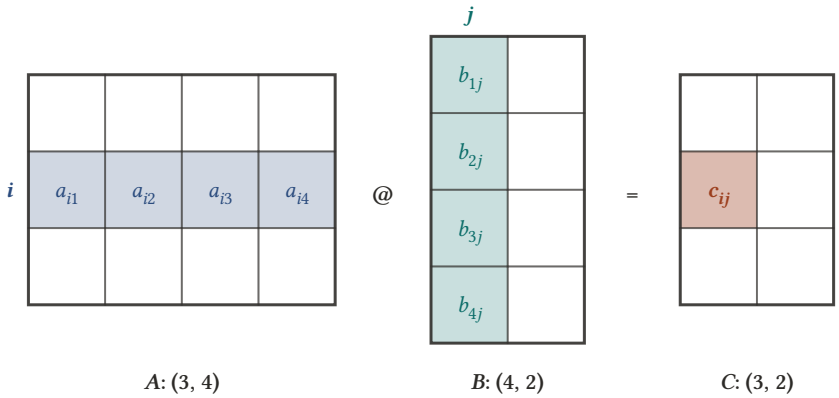
Tanım 7.1 (Matris Çarpımı İşleci). A dizisinin şekli (m, n), B dizisinin şekli (n, p) olsun. `A @ B` ifadesi, elemanları

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

olan (m, p) şekilli diziyi verir. A'nın sütun sayısı B'nin satır sayısına eşit değilse işlem `ValueError` hatası verir. `np.matmul(A, B)` aynı işlemin fonksiyon biçimidir.

 CC BY-NC-SA 4.0

Yani `@`, Lineer Cebir notlarındaki matris çarpımının ta kendisidir. Aşağıdaki şekil bu boyut kuralını gösteriyor: çarpımın her elemanı, A'nın bir satırı ile B'nin bir sütununun iç çarpımıdır.



$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + a_{i4}b_{4j}$$

iç boyutlar eşit olmalı: A'nın 4 sütunu, B'nin 4 satırı

Şekil 7.1: `A @ B` için boyut kuralı: (3, 4) şekilli A ile (4, 2) şekilli B'nin çarpımı (3, 2) şekillidir. C'nin i. satır, j. sütundaki elemanı, A'nın i. satırı ile B'nin j. sütununun iç çarpımıdır. Bu yüzden A'nın sütun sayısı B'nin satır sayısına eşit olmak zorundadır.

> ⚠ **Yıldız işareti matris çarpımı değildir**
>
> `A * B`, aynı şekilli iki diziyi eleman eleman çarpar. Şekiller farklıysa broadcasting kuralları devreye girer ve hiçbir hata almadan beklenmedik bir sonuç elde edilebilir: örneğin $(3, 3)$ şekilli `A` ile uzunluğu 3 olan `v` için `A * v` hata vermez ama Av değildir. Matematikte AB yazdığımız her yerde kodda `A @ B` yazılır. Eski kodlarda görülen `np.dot(A, B)` iki eksenli dizilerde aynı sonucu verir; `np.matrix` sınıfı ise artık önerilmez ve yeni kodda kullanılmaz.

Tek eksenli bir dizi ne satır ne de sütun vektörüdür; yalnızca bir sayı listesidir. `@` işleci onu bağlama göre yorumlar: `A @ v` çarpımında `v` bir sütun vektörü, `w @ A` çarpımında `w` bir satır vektörü gibi davranır. Aynı uzunluktaki iki vektörün `u @ v` çarpımı ise standart iç çarpımdır. Transpoz `A.T` ile alınır.

```
import numpy as np

A = np.array([[2, -1, 0],
              [1, 3, 4]])      # 2 x 3 matris
v = np.array([1, 0, -2])       # uzunluğu 3 olan vektör
w = np.array([1, 1])           # uzunluğu 2 olan vektör

print(A @ v)        # A v: v sütun gibi davranır
print(w @ A)        # w^T A: w satır gibi davranır
print(v @ v)        # iç çarpım: 1 + 0 + 4
print(A.T.shape)    # transpoz
print(A @ A.T)      # 2 x 2 ve simetrik
print(np.trace(A @ A.T))    # köşegen toplamı (iz)
```

Çıktı:

```
[ 2 -7]
[3 2 4]
5
(3, 2)
[[ 5 -1]
```

 CC BY-NC-SA 4.0

```
 [-1 26]]
31
```

AA^T her zaman simetriktir, çünkü $(AA^T)^T = (A^T)^T A^T = AA^T$'dir. `np.trace` köşegen elemanların toplamını, yani izi verir. $n \times n$ birim matris `np.eye(n)` ile, bir kare matrisin k. kuvveti `np.linalg.matrix_power(A, k)` ile kurulur.

Örnek 7.1 (Fibonacci Sayıları ve Matris Kuvveti). Fibonacci sayıları $F_0 = 0$, $F_1 = 1$ ve $F_{n+1} = F_n + F_{n-1}$ ile tanımlanır. $Q = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ matrisinin kuvvetlerini kullanarak F_{10} ve F_{100} sayılarını hesaplayınız.

Çözüm.
Önce $n \geq 1$ için

$$Q^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

olduğunu tümevarımla görelim. $n = 1$ için $F_2 = 1, F_1 = 1, F_0 = 0$ olduğundan eşitlik doğrudur. Eşitlik n için doğruysa

$$Q^{n+1} = Q^n Q = \begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

olur. Demek ki F_n, Q^n'nin birinci satır, ikinci sütundaki elemanıdır; Python'da bu eleman `[0, 1]` indeksiyle okunur.

```python
import numpy as np

Q = np.array([[1, 1],
              [1, 0]])
print(np.linalg.matrix_power(Q, 10))

Q100 = np.linalg.matrix_power(Q, 100)
print(Q100[0, 1])              # int64 taştı!

Q_obj = Q.astype(object)       # elemanlar Python tamsayısı olur
print(np.linalg.matrix_power(Q_obj, 100)[0, 1])
```

Çıktı:

```
[[89 55]
 [55 34]]
3736710778780434371
354224848179261915075
```

Q^{10}'un köşegen dışı elemanından $F_{10} = 55$ okunur. F_{100} için ilk sonuç yanlıştır; doğru değer son satırdaki $354\,224\,848\,179\,261\,915\,075$ sayısıdır. NumPy tamsayı dizilerini varsayılan olarak 64 bitlik `int64` türünde tutar. Bu türün en büyük değeri $2^{63} - 1 \approx 9{,}22 \cdot 10^{18}$'dir ve F_{93} bu sınırı aşar. Sınırı aşan bir tamsayı işlemi hata vermez: sonuç 2^{64}'e göre kalanı alınarak sessizce "başa sarar". `astype(object)` ile elemanlar, büyüklüğü sınırsız olan Python tamsayılarına çevrilir; hesap yavaşlar ama tam doğru olur. ■

 CC BY-NC-SA 4.0

⚠ Tamsayı taşması sessizdir

Python'ın kendi tamsayıları istediği kadar büyüyebilir (Python ile İlk Adımlar), NumPy'ın `int64` dizileri büyüyemez. Ayrıca `solve`, `inv`, `det` ve `eig` gibi LAPACK'e dayanan fonksiyonlar hesabı her zaman kayan noktalı sayılarla yapar. Tamsayı dizileri önce `float64` türüne çevrilir ve sonuç yine yuvarlama hatası taşır: örneğin $\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$ tamsayı matrisinin determinantı `5.000000000000001` çıkar. `object` türündeki diziler ise bu fonksiyonlarda hata verir. Tam aritmetikle matris hesabı gerekiyorsa SymPy ile Sembolik Hesap bölümündeki matrisler kullanılır.

7.2 Lineer Sistemlerin Çözümü

Lineer cebirin merkezindeki problem, A bir $n \times n$ matris ve $b \in \mathbb{R}^n$ iken lineer denklem sistemini $Ax = b$ biçiminde çözmektir. A regüler ise çözüm tektir ve NumPy'da tek satırla bulunur: `np.linalg.solve(A, b)`.

`solve`, elle yaptığımız Gauss yöntemini bir matris ayrışımı olarak düzenler. Satır değişimleriyle birlikte A'yı bir alt üçgensel L ve bir üst üçgensel U matrisinin çarpımına ayırır, sonra iki üçgensel sistemi ileriye ve geriye doğru yerine koyarak çözer. Bu **LU ayrışımı** yaklaşık $\frac{2}{3}n^3$ aritmetik işlem gerektirir. Her adımda sütundaki mutlak değerce en büyük elemanı köşegene getirmek (kısmi pivotlama, partial pivoting) yuvarlama hatalarının büyümesini önler.

💡 Bir lineer sistemi üç adımda çözmek

1. Katsayılar matrisini `A`, sağ tarafı `b` olarak kayan noktalı diziler biçiminde kurun.
2. `x = np.linalg.solve(A, b)` ile çözün.
3. Sağlama yapın: `np.allclose(A @ x, b)` ifadesi `True` olmalı, $r = b - Ax$ **artık** (residual) vektörü sıfıra yakın olmalıdır.

Örnek 7.2 (Üç Bilinmeyenli Bir Sistem).

$$\begin{aligned} 2x + y - 3z &= 5 \\ 3x - 2y + 2z &= 5 \\ 5x - 3y - z &= 16 \end{aligned}$$

sistemini `solve` ile çözünüz ve çözümü sağlayınız.

Çözüm.
Üç adımı sırayla uyguluyoruz: katsayıları ve sağ tarafı kuruyor, `solve` ile çözüyor, sonra sağlama yapıp artığı hesaplıyoruz.

```
import numpy as np

A = np.array([[2.0, 1.0, -3.0],
              [3.0, -2.0, 2.0],
              [5.0, -3.0, -1.0]])
b = np.array([5.0, 5.0, 16.0])

x = np.linalg.solve(A, b)
print(x)
print(np.allclose(A @ x, b))    # sağlama
print(b - A @ x)                # artık vektörü
```

Çıktı:

```
[ 1. -3. -2.]
True
[0. 0. 0.]
```

 CC BY-NC-SA 4.0

Çözüm $(x, y, z) = (1, -3, -2)$'dir. Gerçekten $2 - 3 + 6 = 5$, $3 + 6 - 4 = 5$ ve $5 + 9 + 2 = 16$'dır. Bu sistem Lineer Cebir notlarında Gauss yöntemiyle elle çözülen sistemlerden biridir ve orada da aynı çözüm bulunur. Artığın tam sıfır çıkması bu küçük örneğe özgüdür; genelde 10^{-16} mertebesinde sayılar görürüz. ■

`solve`'un ikinci argümanı bir matris de olabilir. Bu durumda her sütun ayrı bir sağ taraf sayılır ve sonuç, sütunları karşılık gelen çözümler olan bir matristir. LU ayrışımı yalnızca bir kez yapıldığı için bu yol, sistemleri tek tek çözmekten ucuzdur.

```
import numpy as np

A = np.array([[4.0, 1.0],
              [2.0, 3.0]])
B = np.array([[1.0, 0.0, 9.0],
              [0.0, 1.0, 7.0]])   # üç sağ taraf, sütun sütun

X = np.linalg.solve(A, B)
print(X)
```

Çıktı:

```
[[ 0.3 -0.1  2. ]
 [-0.2  0.4  1. ]]
```

İlk iki sağ taraf birim matrisin sütunları olduğundan çözümün ilk iki sütunu A^{-1}'in sütunlarıdır:

$$A^{-1} = \frac{1}{10}\begin{pmatrix} 3 & -1 \\ -2 & 4 \end{pmatrix}.$$

Üçüncü sütun, $(9, 7)$ sağ tarafının çözümü olan $(2, 1)$'dir.
A tekil olduğunda `solve` çoğu zaman `LinAlgError` hatası verir. Bu hatayı Sınıflar, Hata Yakalama ve Dosyalar bölümündeki `try`/`except` yapısıyla yakalayıp sistemi rank ölçütüyle inceleyebiliriz.

Örnek 7.3 (Sonsuz Çözümlü Bir Sistem).

$$\begin{aligned} x + 2y - 3z &= 6 \\ 2x - y + 4z &= 2 \\ 4x + 3y - 2z &= 14 \end{aligned}$$

sistemini `solve` ile çözmeyi deneyiniz; başarısız olursa sistemin çözülebilir olup olmadığına rank ölçütüyle karar veriniz.

Çözüm.
`np.column_stack` katsayılar matrisinin yanına b'yi bir sütun olarak ekler ve genişletilmiş matrisi kurar.

```
import numpy as np

A = np.array([[1.0, 2.0, -3.0],
              [2.0, -1.0, 4.0],
              [4.0, 3.0, -2.0]])
b = np.array([6.0, 2.0, 14.0])

try:
    x = np.linalg.solve(A, b)
except np.linalg.LinAlgError as err:
    print("solve başarısız:", err)

Ab = np.column_stack([A, b])    # genişletilmiş matris [A | b]
print(np.linalg.matrix_rank(A), np.linalg.matrix_rank(Ab))
```

 CC BY-NC-SA 4.0

Çıktı:

```
solve başarısız: Singular matrix
2 2
```

`solve` hata verdi, çünkü katsayılar matrisi tekildir. `matrix_rank` hem A'nın hem de $[A \mid b]$ genişletilmiş matrisinin rankını 2 buldu. Çözülebilirliğin rank ölçütüne göre iki rank eşit olduğundan sistem çözülebilirdir. Homojen sistemin çözüm uzayı $3 - 2 = 1$ boyutludur, yani bir serbest değişken ve sonsuz çok çözüm vardır. Gauss yöntemiyle bulunan genel çözüm $a \in \mathbb{R}$ olmak üzere $(2 - a,\, 2 + 2a,\, a)$'dır; örneğin $a = 1$ için $(1, 4, 1)$ sistemi sağlar: $1 + 8 - 3 = 6$, $2 - 4 + 4 = 2$, $4 + 12 - 2 = 14$. ■

7.3 Ters Matris, Determinant ve Rank

Lineer Cebir'de bir kare matrisin tersini, determinantını ve rankını elle hesaplamayı öğrendik. NumPy'daki karşılıkları `np.linalg.inv`, `np.linalg.det` ve `np.linalg.matrix_rank` fonksiyonlarıdır.

```
import numpy as np

A = np.array([[2.0, 1.0, 1.0],
              [1.0, 3.0, 2.0],
              [1.0, 0.0, 0.0]])

A_inv = np.linalg.inv(A)
print(A_inv)
print(np.linalg.det(A))
print(np.linalg.matrix_rank(A))
print(np.allclose(A @ A_inv, np.eye(3)))
```

Çıktı:

```
[[ 0.  0.  1.]
 [-2.  1.  3.]
 [ 3. -1. -5.]]
-1.0
3
True
```

`inv` ve `det` de LU ayrışımına dayanır: determinant, U'nun köşegen elemanlarının çarpımıdır ve satır değişimlerinin sayısı tekse işaret değiştirir. Bu matrisin determinantı -1 olduğu ve elemanları tamsayı olduğu için tersinin elemanları da tamsayıdır (Lineer Cebir, tersin ek matrisle formülü). Elle hesabın aksine bilgisayar her adımda yuvarlama yapar. Bu küçük örnekte görünür bir hata oluşmadı, ama genel durumda `A @ A_inv` birim matrise yalnızca yaklaşık olarak eşittir; karşılaştırmayı bu yüzden `==` ile değil `np.allclose` ile yapıyoruz.

Örnek 7.4 (Sessizce Yanlış Bir Çözüm).

$$A = \begin{pmatrix} 0{,}1 & 0{,}2 & 0{,}3 \\ 0{,}4 & 0{,}5 & 0{,}6 \\ 0{,}7 & 0{,}8 & 0{,}9 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

olsun. $Ax = b$ sisteminin çözümü olmadığını gösterip `solve`'un bu sisteme verdiği yanıtı inceleyiniz.

Çözüm.
A'nın satırları R_1, R_2, R_3 olsun. $R_1 - 2R_2 + R_3$ birleşiminin bileşenleri

$$0{,}1 - 0{,}8 + 0{,}7 = 0, \qquad 0{,}2 - 1 + 0{,}8 = 0, \qquad 0{,}3 - 1{,}2 + 0{,}9 = 0$$

olduğundan satırlar lineer bağımlıdır ve A tekildir. Aynı birleşimi denklemlere uygularsak sol taraf 0, sağ taraf $1 - 2 \cdot 0 + 0 = 1$ olur. $0 = 1$ çelişkisi sistemin çözümü olmadığını gösterir. Şimdi NumPy'a soralım:

 CC BY-NC-SA 4.0

```python
import numpy as np

A = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6],
              [0.7, 0.8, 0.9]])
b = np.array([1.0, 0.0, 0.0])

x = np.linalg.solve(A, b)       # hata vermiyor!
print(x)
print(A @ x)                    # b'ye hiç benzemiyor
print(np.linalg.det(A))
print(np.linalg.matrix_rank(A))
```

Çıktı:

```
[-4.50359963e+15  9.00719925e+15 -4.50359963e+15]
[ 0.85  0.2  -0.45]
6.661338147750926e-18
2
```

`solve` hata vermedi ve bileşenleri 10^{15} mertebesinde olan bir "çözüm" döndürdü. Sağlama yapınca Ax'in b'ye hiç benzemediği görülüyor. Determinant da 0 değil, $6{,}7 \cdot 10^{-18}$ çıktı. Sebep, 0,1, 0,2, ... sayılarının ikili sistemde tam olarak gösterilememesidir (Nümerik Analiz). Bellekteki matris, tekil matrisin çok küçük bir bozulmasıdır ve tam olarak tekil değildir. LU ayrışımı 0 olması gereken yerde $1{,}1 \cdot 10^{-16}$ büyüklüğünde bir pivot bulur ve ona böler. `matrix_rank` ise doğru yanıtı, 2'yi verdi; bu kararı nasıl verdiğini tekil değer ayrışımı kısmında göreceğiz. ■

> ⚠ Tekilliği determinantla sınamayın
>
> `det(A) == 0` sınaması kayan noktalı sayılarla anlamsızdır. Yukarıdaki tekil matrisin determinantı 0 çıkmadı. Tersine, $0{,}1\,I_{20}$ matrisinin determinantı 10^{-20}'dir, ama bu matrisle sistem çözmek son derece kolaydır. Determinant matrisin ölçeğine bağlıdır ve matrisin tekilliğe ne kadar yakın olduğunu ölçmez. Doğru araçlar `matrix_rank` ve aşağıda tanımlayacağımız koşul sayısıdır.

7.4 Vektör ve Matris Normları

Bir hesabın ne kadar doğru olduğunu söyleyebilmek için vektörlerin ve matrislerin büyüklüğünü tek bir sayıyla ölçmemiz gerekir. Vektör normlarını Analiz 4 notlarında tanımladık: Öklid normu $\| x \|_2$, 1-normu $\| x \|_1 = \sum |\, x_i \,|$ ve maksimum normu $\| x \|_\infty = \max |\, x_i \,|$ (p-normları). Hepsini `np.linalg.norm` hesaplar; ikinci argüman normun türünü seçer.

```python
import numpy as np

x = np.array([3.0, -4.0, 12.0])
print(np.linalg.norm(x))            # Öklid normu (2-normu)
print(np.linalg.norm(x, 1))         # mutlak değerler toplamı
print(np.linalg.norm(x, np.inf))    # en büyük mutlak değer
```

Çıktı:

```
13.0
19.0
12.0
```

 CC BY-NC-SA 4.0

Gerçekten $\| x \|_2 = \sqrt{9+16+144} = 13$, $\| x \|_1 = 3+4+12 = 19$ ve $\| x \|_\infty = 12$'dir. Matrisler için en kullanışlı normlar, bir vektör normundan türetilenlerdir.

Tanım 7.2 (Doğal Matris Normu). $\mathbb{R}^n$ ve $\mathbb{R}^m$ üzerinde aynı türden bir $\| \cdot \|$ vektör normu verilsin (örneğin ikisinde de Öklid normu). $m \times n$ bir A matrisi için ($x \in \mathbb{R}^n$, $Ax \in \mathbb{R}^m$)

$$\| A \| = \max_{x \neq 0} \frac{\| Ax \|}{\| x \|} = \max_{\|x\|=1} \| Ax \|$$

sayısına bu vektör normunun ürettiği **doğal matris normu** denir. $\| \cdot \|_1$, $\| \cdot \|_2$ ve $\| \cdot \|_\infty$ vektör normlarının ürettiği matris normları sırasıyla $\| A \|_1$, $\| A \|_2$ ve $\| A \|_\infty$ ile gösterilir.

Yani $\| A \|$, A'nın bir vektörü en fazla kaç katına uzatabildiğidir. Tanımdan her x için $\| Ax \| \leq \| A \| \| x \|$ eşitsizliği, buradan da $\| AB \| \leq \| A \| \| B \|$ eşitsizliği çıkar. Doğal normların yanında, elemanların kareleri toplamından gelen bir norm da sık kullanılır.

Tanım 7.3 (Frobenius Normu). $m \times n$ bir A matrisinin **Frobenius normu**

$$\| A \|_F = \Big(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 \Big)^{1/2}$$

sayısıdır.

Yani Frobenius normu, matrisi mn bileşenli uzun bir vektör gibi düşünüp aldığımız Öklid normudur. Bir vektör normundan türetilmez, ama her x için $\| Ax \|_2 \leq \| A \|_F \| x \|_2$ eşitsizliğini sağlar. 1- ve sonsuz normlarının ise basit formülleri vardır.

Önerme 7.1 (Bir ve Sonsuz Normlarının Formülleri). *$n \times n$ bir A matrisi için*

$$\| A \|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} | a_{ij} |, \qquad \| A \|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} | a_{ij} |$$

dir. Yani $\| A \|_1$ mutlak değerlerle en büyük sütun toplamı, $\| A \|_\infty$ en büyük satır toplamıdır.

> ***İspat.***
> ***Sonsuz normu.*** *$\| x \|_\infty = 1$ olsun. Her j için $| x_j | \leq 1$ olduğundan her i için*
>
> $$| (Ax)_i | = \Big| \sum_j a_{ij} x_j \Big| \leq \sum_j | a_{ij} | \, | x_j | \leq \sum_j | a_{ij} |$$
>
> *olur; demek ki $\| Ax \|_\infty$ en büyük satır toplamını aşmaz. Eşitliği göstermek için en büyük toplamın bulunduğu satır k olsun ve $a_{kj} \geq 0$ ise $x_j = 1$, değilse $x_j = -1$ seçelim. O zaman $\| x \|_\infty = 1$ ve $(Ax)_k = \sum_j | a_{kj} |$ olur.*
> ***1-normu.*** *Her x için*
>
> $$\| Ax \|_1 = \sum_i \Big| \sum_j a_{ij} x_j \Big| \leq \sum_j | x_j | \sum_i | a_{ij} | \leq \Big(\max_j \sum_i | a_{ij} | \Big) \| x \|_1$$
>
> *dir. En büyük sütun toplamı k. sütundaysa $x = e_k$ için eşitlik sağlanır, çünkü Ae_k, A'nın k. sütunudur.* ■

$\| A \|_2$ için bu kadar basit bir formül yoktur; onu tekil değer ayrışımı kısmında A'nın en büyük tekil değeri olarak tanıyacağız. `np.linalg.norm` matrislerde de ikinci argümanla çalışır. Dikkat: ikinci argüman verilmezse matrisin 2-normu değil Frobenius normu hesaplanır.

 CC BY-NC-SA 4.0

```
import numpy as np

A = np.array([[1.0, -2.0],
              [3.0, 4.0]])
print(np.linalg.norm(A, 1))         # en büyük sütun toplamı
print(np.linalg.norm(A, np.inf))    # en büyük satır toplamı
print(np.linalg.norm(A, 2))         # 2-normu
print(np.linalg.norm(A))            # varsayılan: Frobenius normu
```

Çıktı:

```
6.0
7.0
5.116672736016927
5.477225575051661
```

A'nın mutlak değerlerle sütun toplamları $1+3=4$ ve $2+4=6$, satır toplamları $1+2=3$ ve $3+4=7$'dir. Bu yüzden $\| A \|_1 = 6$ ve $\| A \|_\infty = 7$'dir. Frobenius normu $\| A \|_F = \sqrt{1+4+9+16}$, yani $\sqrt{30} \approx 5{,}4772$'dir.

7.5 Koşul Sayısı

Sessizce yanlış çözüm örneğinde (Örnek 7.4) veride 10^{-17} mertebesindeki yuvarlama hataları, çözümü olmayan bir sistemden bileşenleri 10^{15} mertebesinde olan bir "çözüm" üretti. Bir sistemin verisindeki küçük değişikliklerin çözümü ne kadar değiştirebileceğini ölçen sayı koşul sayısıdır.

Tanım 7.4 (Koşul Sayısı). Regüler bir A kare matrisinin, bir doğal matris normuna göre **koşul sayısı** (condition number)

$$\kappa(A) = \| A \| \, \| A^{-1} \|$$

sayısıdır. A tekil ise $\kappa(A) = \infty$ kabul edilir. NumPy'da `np.linalg.cond(A)` 2-normuna göre koşul sayısını, `np.linalg.cond(A, 1)` ve `np.linalg.cond(A, np.inf)` öteki iki normdakini verir.

Yani koşul sayısı A ile A^{-1}'in büyüklüklerinin çarpımıdır ve hiçbir zaman 1'den küçük olmaz: $1 = \| I \| = \| AA^{-1} \| \leq \| A \| \, \| A^{-1} \|$. Koşul sayısı 1'e yakın matrislere **iyi koşullu**, çok büyük olanlara **kötü koşullu** denir. Koşul sayısının neyi ölçtüğünü şu önerme söyler.

Önerme 7.2 (Sağ Taraftaki Hatanın Büyümesi). *A regüler, $b \neq 0$, $Ax = b$ ve $A(x + \delta x) = b + \delta b$ olsun. O zaman*

$$\frac{\| \delta x \|}{\| x \|} \leq \kappa(A) \frac{\| \delta b \|}{\| b \|}$$

dir.

> ***İspat.***
> *İki eşitliği taraf tarafa çıkarırsak $A\,\delta x = \delta b$, yani $\delta x = A^{-1}\delta b$ olur ve buradan $\| \delta x \| \leq \| A^{-1} \| \, \| \delta b \|$ bulunur. Öte yandan $\| b \| = \| Ax \| \leq \| A \| \, \| x \|$ olduğundan*
>
> $$\frac{1}{\| x \|} \leq \frac{\| A \|}{\| b \|}$$
>
> *dir ($b \neq 0$ olduğundan $x \neq 0$'dır). Bu iki eşitsizliği taraf tarafa çarparsak*
>
> $$\frac{\| \delta x \|}{\| x \|} \leq \| A \| \, \| A^{-1} \| \frac{\| \delta b \|}{\| b \|} = \kappa(A) \frac{\| \delta b \|}{\| b \|}$$
>
> *elde edilir.* ■

 CC BY-NC-SA 4.0

Yani sağ taraftaki göreli hata çözümde en fazla $\kappa(A)$ katına çıkabilir; A'nın kendisindeki küçük hatalar için de benzer bir sınır kanıtlanabilir. Kayan noktalı bir sayının göreli hassasiyeti $\varepsilon \approx 2{,}2 \cdot 10^{-16}$ olduğundan (Python ile İlk Adımlar) veri zaten bu büyüklükte hatalarla saklanır. Bu yüzden pratik kural şudur: $\kappa(A) \approx 10^k$ ise çözümün yaklaşık 16 anlamlı basamağından k tanesi kaybedilir.

Örnek 7.5 (Neredeyse Paralel İki Doğru). $x + y = 2, x - y = 0$ sisteminde ve $x + y = 2, x + 1{,}1\,y = 2{,}1$ sisteminde ikinci denklemin sağ tarafı $0{,}1$ artırıldığında çözümlerin ne kadar değiştiğini karşılaştırınız ve sonucu koşul sayısıyla açıklayınız.

Çözüm.
İki sistemi ve değiştirilmiş sağ tarafları aynı döngüde çözüyoruz. Her satırda önce özgün, sonra değişmiş sistemin çözümü yazılıyor.

```
import numpy as np

A_good = np.array([[1.0, 1.0],
                   [1.0, -1.0]])    # x + y = 2,  x - y = 0
A_bad = np.array([[1.0, 1.0],
                  [1.0, 1.1]])      # x + y = 2,  x + 1.1y = 2.1
b_good = np.array([2.0, 0.0])
b_bad = np.array([2.0, 2.1])
db = np.array([0.0, 0.1])           # ikinci bileşene 0.1 ekleniyor

for A, b in [(A_good, b_good), (A_bad, b_bad)]:
    x = np.linalg.solve(A, b)
    x2 = np.linalg.solve(A, b + db)
    print(x, x2)
```

Çıktı:

```
[1. 1.] [1.05 0.95]
[1. 1.] [0. 2.]
```

İlk sistemde çözüm $(1, 1)$'den $(1{,}05;\, 0{,}95)$'e, yani sağ taraftaki değişiklikle aynı mertebede kaydı. İkinci sistemde aynı değişiklik çözümü $(1, 1)$'den $(0, 2)$'ye taşıdı. İkinci sistemi koşul sayısıyla inceleyelim:

```
import numpy as np

A = np.array([[1.0, 1.0],
              [1.0, 1.1]])
b = np.array([2.0, 2.1])
b2 = np.array([2.0, 2.2])

x = np.linalg.solve(A, b)
x2 = np.linalg.solve(A, b2)

kappa = np.linalg.cond(A)
rel_b = np.linalg.norm(b2 - b) / np.linalg.norm(b)
rel_x = np.linalg.norm(x2 - x) / np.linalg.norm(x)
print(f"koşul sayısı          : {kappa:.4f}")
print(f"b'deki göreli değişim : {rel_b:.4f}")
print(f"x'teki göreli değişim : {rel_x:.4f}")
print(f"önermedeki üst sınır  : {kappa * rel_b:.4f}")
```

Çıktı:

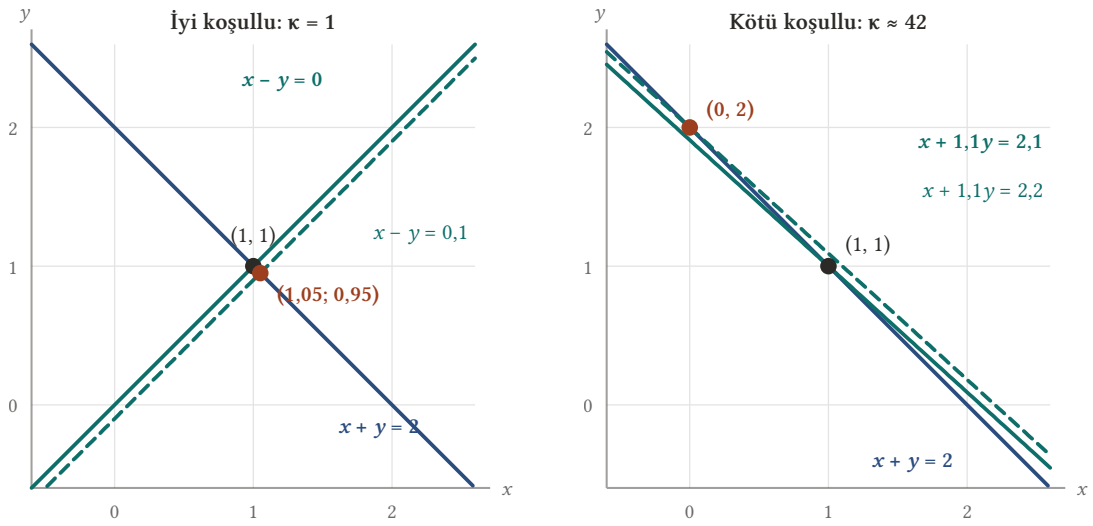
```
koşul sayısı          : 42.0762
b'deki göreli değişim : 0.0345
```

 CC BY-NC-SA 4.0

```
x'teki göreli değişim : 1.0000
önermedeki üst sınır  : 1.4509
```

b'deki %3,4'lük değişiklik x'te %100'lük bir değişikliğe yol açtı. Büyüme katsayısı $1/0{,}0345 \approx 29$'dur ve önermedeki üst sınırı (Önerme 7.2), yani $\kappa(A) \approx 42$ katını aşmaz. İlk sistemin katsayılar matrisinin sütunları dik ve aynı uzunlukta olduğundan koşul sayısı 1'dir. ■

Aşağıdaki şekil iki durumu geometrik olarak gösteriyor. İki doğru dik açıyla kesiştiğinde birini biraz kaydırmak kesişim noktasını da biraz kaydırır. Doğrular neredeyse paralel olduğunda ise kesişim noktası çok hassastır: küçük bir kayma onu doğrular boyunca uzağa taşır.



Şekil 7.2: Her iki sistemde de ikinci denklemin sağ tarafı 0,1 artırıldı: ikinci doğru, kesikli konumuna kayar. Solda doğrular dik açıyla kesiştiği için çözüm (1, 1)'den yalnızca (1,05; 0,95)'e gider. Sağda doğrular neredeyse paraleldir; aynı küçük kayma kesişim noktasını (1, 1)'den (0, 2)'ye taşır.

Determinantın bu hassasiyeti ölçmediğini görmek için $0{,}1\,I_{20}$ matrisine bakalım.

```
import numpy as np

D = 0.1 * np.eye(20)
print(np.linalg.det(D))          # neredeyse sıfır ...
print(np.linalg.cond(D))         # ... ama koşul sayısı en iyi değerde
print(np.linalg.solve(D, np.ones(20))[:4])
```

Çıktı:

```
1.0000000000000135e-20
1.0
[10. 10. 10. 10.]
```

Determinant 10^{-20}'dir, ama koşul sayısı mümkün olan en iyi değer olan 1'dir ve sistem tam olarak çözülür. Şimdi koşul sayısı çok hızlı büyüyen ünlü bir matris ailesine bakalım.

Örnek 7.6 (Hilbert Matrisi). Elemanları $h_{ij} = \dfrac{1}{i+j-1}$ $(i, j = 1, \ldots, n)$ olan $n \times n$ matrise **Hilbert matrisi** denir ve H_n ile gösterilir. $n = 5, 6, \ldots, 12$ için gerçek çözümü $x = (1, 1, \ldots, 1)$ olan $H_n x = b$ sistemini kurup `solve` ile ve `inv(H) @ b` ile çözünüz; iki yöntemin hatalarını ve artıklarını karşılaştırınız.

Çözüm.

Gerçek çözüm birlerden oluşsun diye $b = H_n \mathbf{1}$ alıyoruz. Matrisi broadcasting ile kuruyoruz (Vektörizasyon ve Broadcasting): `i[:, None] + i[None, :]` ifadesi $i + j$ değerlerinden oluşan $n \times n$ diziyi verir. Hataları göreli olarak ölçüyoruz: hata $\| \hat{x} - x \|_2 / \| x \|_2$, artık $\| H_n \hat{x} - b \|_2 / \| b \|_2$'dir; burada $\hat{x}$ hesaplanan çözümdür.

```
import numpy as np


def rel_err(approx, exact):
    """Göreli hata: ||approx - exact|| / ||exact||."""
    return np.linalg.norm(approx - exact) / np.linalg.norm(exact)


print(" n    koşul   hata:solve  hata:inv  artık:solve  artık:inv")
for n in range(5, 13):
    i = np.arange(1, n + 1)
    H = 1.0 / (i[:, None] + i[None, :] - 1)    # Hilbert matrisi
    x_true = np.ones(n)
    b = H @ x_true
    x1 = np.linalg.solve(H, b)
    x2 = np.linalg.inv(H) @ b
    print(f"{n:2d}  {np.linalg.cond(H):8.1e}"
          f"  {rel_err(x1, x_true):9.1e}  {rel_err(x2, x_true):8.1e}"
          f"  {rel_err(H @ x1, b):10.1e}  {rel_err(H @ x2, b):9.1e}")
```

Çıktı:

```
 n    koşul   hata:solve  hata:inv  artık:solve  artık:inv
 5   4.8e+05    1.3e-12   2.7e-11     7.9e-17    2.3e-13
 6   1.5e+07    4.1e-11   9.7e-10     1.5e-16    8.1e-12
 7   4.8e+08    1.8e-09   1.0e-08     6.5e-17    9.2e-10
 8   1.5e+10    6.7e-08   4.8e-07     1.6e-16    2.7e-08
 9   4.9e+11    3.0e-06   8.4e-05     7.9e-17    1.9e-06
10   1.6e+13    1.2e-05   2.4e-03     1.6e-16    3.4e-05
11   5.2e+14    1.1e-03   1.0e-01     6.7e-17    4.6e-04
12   1.6e+16    1.3e-01   2.7e+00     1.1e-16    4.6e-02
```

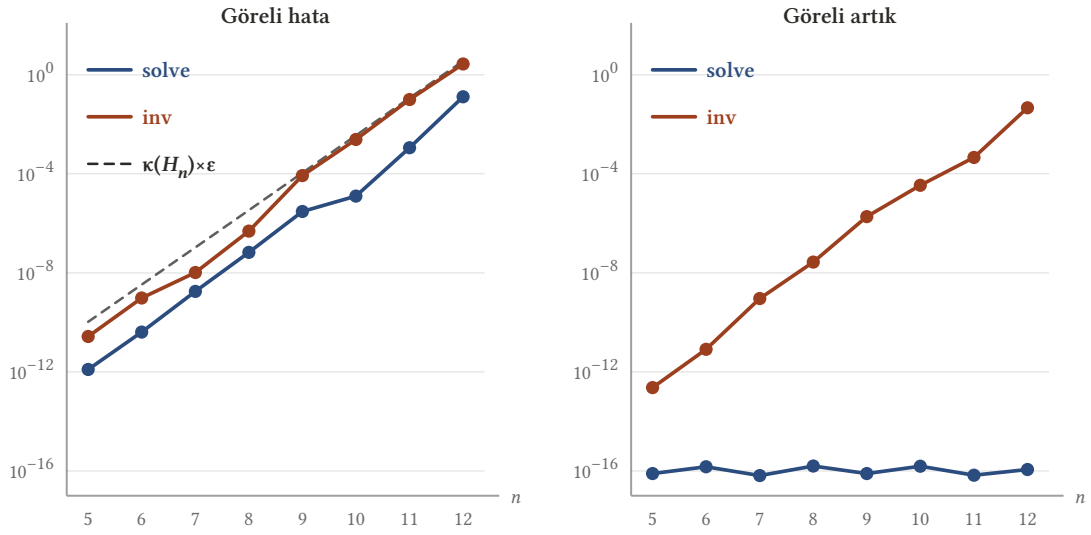
Tablodan üç sonuç okunuyor.

- Koşul sayısı n bir arttıkça yaklaşık 33 katına çıkıyor ve $n = 12$'de 10^{16}'ya, yani $1/\varepsilon$ düzeyine ulaşıyor. Bu noktada koşul sayısının kendisi bile ancak kabaca hesaplanabilir; yüksek duyarlıklı bir hesap gerçek değerin $1{,}7 \cdot 10^{16}$ olduğunu gösterir.
- İki yöntemin de hatası koşul sayısıyla birlikte büyüyor ve kabaca $\kappa(H_n)\,\varepsilon$ düzeyinde kalıyor; önermenin (Önerme 7.2) söylediği budur. `inv` ile bulunan çözümün hatası her satırda daha büyük.
- Asıl fark artıkta. `solve`'un artığı her n için 10^{-16} civarında kalırken `inv`'inki koşul sayısıyla büyüyor; $n = 12$'de `inv` ile bulunan çözüm denklemi ancak $\%5$ civarında bir hatayla sağlıyor. ■

7.6 Neden Ters Matrisle Çözmüyoruz?

Matematikte $Ax = b$'nin çözümünü $x = A^{-1}b$ diye yazmak doğaldır ve doğrudur. Kodda bunu `np.linalg.inv(A) @ b` diye yazmak ise hem daha az doğru hem de daha yavaştır. Hilbert örneğinin (Örnek 7.6) sonuçları aşağıdaki şekilde özetleniyor.

 CC BY-NC-SA 4.0



Şekil 7.3: Hilbert sistemleri $H_n x = b$ için `solve` ve `inv` ile bulunan çözümler (dikey eksen logaritmik). Solda göreli hata: ikisi de koşul sayısıyla büyür; kesikli çizgi $\kappa(H_n)\cdot\varepsilon$ kaba tahminidir. Sağda göreli artık $\|H_n x - b\|/\|b\|$: `solve`'un artığı 10^{-16} düzeyinde kalır, `inv`'inki koşul sayısıyla birlikte büyür.

Artıktaki farkın sebebi şudur. `solve`'un kullandığı kısmi pivotlamalı LU yöntemi pratikte **geriye doğru kararlıdır** (backward stable): bulduğu $\hat{x}$, A'ya çok yakın bir $A + \delta A$ matrisi için ($\| \delta A \| / \| A \|$ oranı ε mertebesinde) sistemin tam çözümüdür. Bu yüzden $b - A\hat{x} = \delta A\, \hat{x}$ artığı her zaman ε mertebesindedir. Hata ise bu küçük bozulmanın koşul sayısıyla büyümesinden gelir ve hiçbir yöntem onu ortadan kaldıramaz. `inv` ile hesaplanan ters matris de yaklaşık olarak doğrudur, ama $\hat{x} = \widehat{A^{-1}}\, b$ çarpımı bu güvenceyi taşımaz: artığı koşul sayısıyla büyür.

Hız farkını görmek için rastgele bir 2000×2000 sistemi iki yolla çözelim. `rng.standard_normal((n, n))` elemanları standart normal dağılımdan, yani ortalaması 0 ve standart sapması 1 olan normal dağılımdan gelen $n \times n$ bir dizi üretir. `time.perf_counter()` saniye cinsinden bir saat değeri verir; her yöntemi beş kez çalıştırıp en kısa süreyi alıyoruz.

```
import time

import numpy as np

rng = np.random.default_rng(1)
n = 2000
A = rng.standard_normal((n, n))
b = rng.standard_normal(n)


def best_time(f, repeat=5):
    """f'yi birkaç kez çalıştırıp en kısa süreyi saniye cinsinden verir."""
    times = []
    for _ in range(repeat):
        t0 = time.perf_counter()
        f()
        times.append(time.perf_counter() - t0)
    return min(times)


t_solve = best_time(lambda: np.linalg.solve(A, b))
t_inv = best_time(lambda: np.linalg.inv(A) @ b)
print(f"solve     : {t_solve:.1g} s")
```

```
print(f"inv ve @  : {t_inv:.1g} s")
print(f"oran      : yaklaşık {round(t_inv / t_solve)}")
```

Çıktı:

```
solve     : 0.1 s
inv ve @  : 0.4 s
oran      : yaklaşık 4
```

Bu süreler bu bilgisayarda, tek iş parçacığıyla ölçüldü; başka bir makinede farklı sayılar çıkar. Oran ise kabaca sabittir: LU ayrışımı yaklaşık $\frac{2}{3}n^3$, tersi hesaplamak yaklaşık $2n^3$ işlem gerektirir; yani `inv` yaklaşık üç kat iş yapar. Ölçülen oranın bundan biraz büyük çıkması, süreyi yalnızca işlem sayısının değil, belleğe erişim gibi uygulama ayrıntılarının da belirlemesindendir.

> ⚠ Ters matris yerine solve
>
> Kodda $A^{-1}b$ görürseniz onu `np.linalg.solve(A, b)` olarak yazın. Ters matris yalnızca matrisin kendisi gerçekten gerektiğinde (örneğin elemanları yorumlanacaksa) hesaplanır. Aynı A ile birçok sistem çözülecekse sağ tarafları bir matrisin sütunlarına koyup `solve(A, B)` çağırın ya da SciPy'daki `scipy.linalg.lu_factor` ile ayrışımı bir kez hesaplayıp `scipy.linalg.lu_solve` ile tekrar tekrar kullanın.

7.7 Özdeğerler ve Özvektörler

Özdeğer ve özvektör kavramlarını Lineer Cebir'den biliyoruz: $v \neq 0$ ve $Av = \lambda v$. Elle hesapta önce karakteristik polinomun köklerini buluruz. Sayısal hesapta bu yol kullanılmaz, çünkü bir polinomun kökleri katsayılardaki küçük hatalara çok duyarlıdır. LAPACK bunun yerine, matrisi benzerlik dönüşümleriyle adım adım üçgensel biçime yaklaştıran **QR algoritmasını** kullanır. NumPy'daki karşılığı `np.linalg.eig`'dir.

```
import numpy as np

A = np.array([[2.0, 1.0],
              [1.0, 2.0]])
w, V = np.linalg.eig(A)
print(w)                          # özdeğerler
print(V)                          # sütunlar: birim özvektörler
print(np.allclose(A @ V, V * w))  # her i için A v_i = w_i v_i
print(np.real_if_close(w))        # sanal kısımlar sıfırsa at
```

Çıktı:

```
[3.+0.j 1.+0.j]
[[ 0.70710678+0.j -0.70710678+0.j]
 [ 0.70710678+0.j  0.70710678+0.j]]
True
[3. 1.]
```

`eig` iki dizi döndürür: `w[i]` özdeğerine `V[:, i]` **sütunu** karşılık gelir, satırı değil. Özvektörler birim uzunluktadır, işaretleri keyfidir ve özdeğerler belli bir sıraya dizilmez. `V * w` çarpımı broadcasting ile V'nin i. sütununu `w[i]` ile çarpar; böylece bütün $Av_i = \lambda_i v_i$ eşitlikleri tek satırda sınanır.
Çıktıdaki `+0.j` kısımları sonucun karmaşık türde (`complex128`) geldiğini gösteriyor. Eski NumPy sürümleri, sanal kısımların hepsi sıfır olduğunda gerçel türde bir dizi döndürürdü. Kullandığımız NumPy 2.5 sürümünde ise `eig` sonucu, özdeğerlerin hepsi gerçel olsa bile her zaman karmaşık türde verir; NumPy belgelerindeki `eig` örnekleri de bunu gösterir. Sanal kısımlar sıfırsa `np.real_if_close` diziyi gerçel türe çevirir; kodun her sürümde aynı sonucu vermesi için bundan sonra bu adımı ekleyeceğiz.

 CC BY-NC-SA 4.0

Özdeğer hesabını üç adımda doğrulamak

1. Özdeğer `w[i]` ise ona ait özvektör `V[:, i]` sütunudur.
2. `np.allclose(A @ V, V * w)` ile bütün $Av_i = \lambda_i v_i$ eşitliklerini sınayın.
3. Köşegenleştirme için `np.linalg.cond(V)` değerine bakın: makul büyüklükteyse V regülerdir ve $V^{-1}AV$ köşegendir.

Örnek 7.7 (Bir Matrisin Köşegenleştirilmesi).

$$A = \begin{pmatrix} 1 & -3 & 3 \\ 3 & -5 & 3 \\ 6 & -6 & 4 \end{pmatrix}$$

matrisinin özdeğerlerini ve özvektörlerini `eig` ile bulup $V^{-1}AV$ matrisinin köşegen olduğunu doğrulayınız.

Çözüm.
Üç adımı uyguluyoruz. $D = V^{-1}AV$ matrisini ters almadan, $VD = AV$ sistemini `solve` ile çözerek hesaplıyoruz.

```
import numpy as np

A = np.array([[1.0, -3.0, 3.0],
              [3.0, -5.0, 3.0],
              [6.0, -6.0, 4.0]])
w, V = np.linalg.eig(A)
w, V = np.real_if_close(w), np.real_if_close(V)
print(w)
print(np.allclose(A @ V, V * w))     # 2. adım: A v_i = w_i v_i
print(np.linalg.cond(V))             # 3. adım: V regüler mi?

D = np.linalg.solve(V, A @ V)        # V^{-1} A V, ters almadan
print(np.allclose(D, np.diag(w)))
```

Çıktı:

```
[ 4. -2. -2.]
True
4.300625249463308
True
```

Özdeğerler 4, -2 ve -2'dir; -2 iki katlıdır. V'nin koşul sayısı yaklaşık $4{,}3$ olduğundan sütunları rahatça bağımsızdır ve $V^{-1}AV = \operatorname{diag}(4, -2, -2)$ eşitliği sağlanır. Bu matris Lineer Cebir notlarında elle köşegenleştirilmişti. Oradaki P matrisinin sütunları buradaki V'nin sütunlarından farklıdır, çünkü iki katlı özdeğerin öz uzayı iki boyutludur ve bu uzayda sonsuz çok taban seçilebilir. ■

Örnek 7.8 (Köşegenleştirilemeyen Bir Matris).

$$B = \begin{pmatrix} -3 & 1 & -1 \\ -7 & 5 & -1 \\ -6 & 6 & -2 \end{pmatrix}$$

matrisinin karakteristik polinomu bir önceki örnekteki A'nınkiyle aynıdır, ama B köşegenleştirilemez. `eig`'in bu matrise verdiği yanıtı inceleyiniz.

Çözüm.
Özdeğerleri, özvektörleri ve V'nin koşul sayısını yazdıralım:

 CC BY-NC-SA 4.0

```
import numpy as np

B = np.array([[-3.0, 1.0, -1.0],
              [-7.0, 5.0, -1.0],
              [-6.0, 6.0, -2.0]])
w, V = np.linalg.eig(B)
w, V = np.real_if_close(w), np.real_if_close(V)
print(w)
print(V)
print(np.linalg.cond(V))
```

Çıktı:

```
[ 4.         -1.99999996 -2.00000004]
[[ 8.80956881e-17 -7.07106781e-01  7.07106781e-01]
 [-7.07106781e-01 -7.07106781e-01  7.07106781e-01]
 [-7.07106781e-01  2.62505234e-08  2.62505208e-08]]
71765775.70757502
```

Tam aritmetikte özdeğerler yine 4, -2 ve -2'dir, ama -2'nin öz uzayı tek boyutludur (Lineer Cebir). NumPy'ın sonucunda iki şey dikkat çekiyor.

- İki katlı özdeğer $-1{,}99999996$ ve $-2{,}00000004$ olarak, yani yalnızca 8 basamak doğrulukla bulundu. Köşegenleştirilemeyen bir matrisin katlı özdeğeri, verideki ε büyüklüğündeki hataları $\sqrt{\varepsilon} \approx 10^{-8}$ büyüklüğüne çıkarır.
- V'nin son iki sütunu neredeyse aynı doğrultudadır: ikisi de $(1, 1, 0)$ doğrultusuna çok yakındır ve biri ötekinin yaklaşık -1 katıdır. Bu yüzden V'nin koşul sayısı $7 \cdot 10^{7}$ civarındadır.

Sayısal hesapta köşegenleştirilemezliği, V'nin tam olarak tekil çıkmasından değil, koşul sayısının çok büyük olmasından anlarız. ■

Gerçel bir matrisin özdeğerleri karmaşık da olabilir. Lineer Cebir notlarındaki $\begin{pmatrix} 1 & -1 \\ 2 & -1 \end{pmatrix}$ matrisinin karakteristik polinomu $t^2 + 1$'dir ve özdeğerleri $\pm i$'dir. Python'da sanal birim `1j` ile yazılır.

```
import numpy as np

B = np.array([[1.0, -1.0],
              [2.0, -1.0]])
w, V = np.linalg.eig(B)
print(w)
print(V[:, 0])
print(np.allclose(B @ V, V * w))
```

Çıktı:

```
[-9.71445147e-17+1.j -9.71445147e-17-1.j]
[0.40824829+0.40824829j 0.81649658+0.j        ]
True
```

Gerçel kısımlardaki $-9{,}7 \cdot 10^{-17}$, tam olarak 0 olması gereken bir sayının yuvarlama hatasıdır. i özdeğerine ait özvektör $\left(\frac{1+i}{2}, 1\right)$ vektörünün birim uzunluğa ölçeklenmiş hâlidir: bu vektörün uzunluğunun karesi $\frac{1}{2} + 1 = \frac{3}{2}$ olduğundan ölçek çarpanı $\sqrt{2/3} \approx 0{,}8165$'tir.

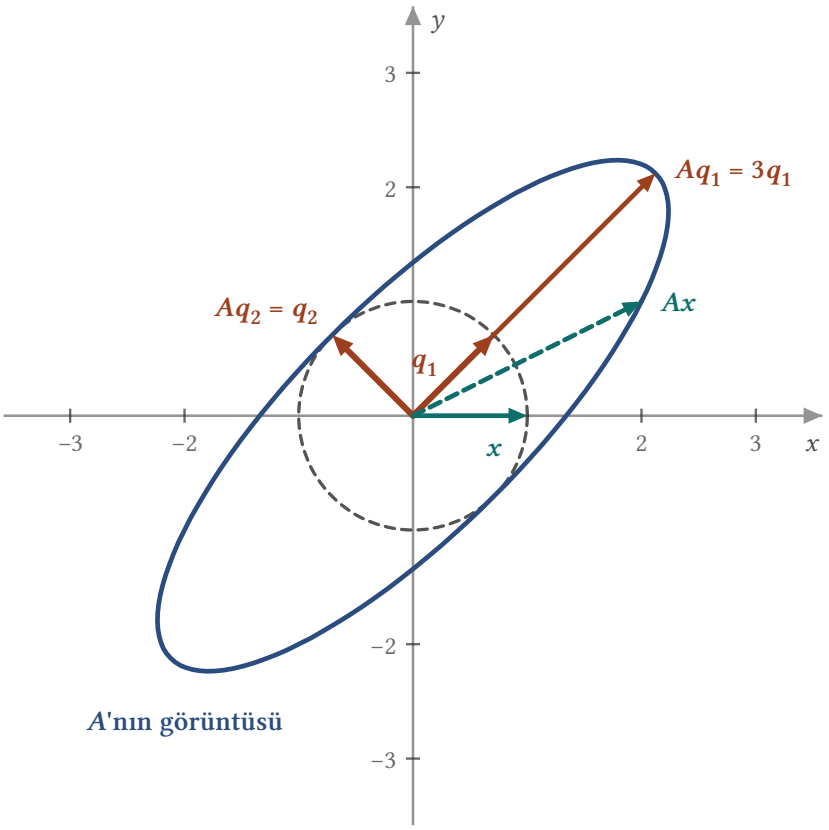
7.8 Simetrik Matrisler

Uygulamada en sık karşılaşılan matrisler simetriktir: kuadratik formların matrisleri, ikinci türev (Hesse) matrisleri, kovaryans matrisleri. Simetrik matrislerin özdeğer problemi çok daha iyi huyludur.

 CC BY-NC-SA 4.0

Teorem 7.1 (Simetrik Matrisler İçin Spektral Teorem). *A gerçel ve simetrik ($A^T = A$) bir $n \times n$ matris olsun. O zaman A'nın bütün özdeğerleri gerçeldir ve sütunları A'nın özvektörlerinden oluşan, $Q^TQ = I$ eşitliğini sağlayan bir Q matrisi vardır:*

$$A = Q\Lambda Q^T, \qquad \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n).$$



Şekil 7.4: A = *[[2, 1], [1, 2]] birim çemberi (kesikli) elipse götürür. Elipsin eksenleri* A*'nın dik özvektörleri boyuncadır:* q_1 = *(1, 1)/√2 doğrultusu 3 kat uzar,* q_2 = *(−1, 1)/√2 doğrultusu olduğu gibi kalır. Özvektör olmayan* x = *(1, 0) ise* Ax = *(2, 1)'e gider ve doğrultusu değişir.*

Yani simetrik bir matris, birbirine dik n doğrultunun her birini kendi özdeğeri kadar uzatır ya da kısaltır; şekil bunu $\left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ matrisi için gösteriyor. Teoremin ispatı iç çarpım uzaylarının konusudur; burada sonuçlarını kullanacağız.

Simetrik matrisler için `np.linalg.eigh` kullanılır. `eigh` matrisin yalnızca alt üçgenini okur, `eig`'den daha hızlıdır, özdeğerleri gerçel türde ve küçükten büyüğe sıralı döndürür ve özvektörlerin dik olmasını garanti eder. Matris simetrik değilse `eigh` uyarı vermeden yanlış sonuç verir; bu yüzden yalnızca simetrik matrislerde kullanılmalıdır.

Örnek olarak, ikinci türevin sonlu farklarla yaklaşımında ortaya çıkan matrisi alalım. $x_0 < x_1 < \ldots < x_{n+1}$ eşit aralıklı noktalar ve $h = x_{i+1} - x_i$ ise iç noktalarda

$$f''(x_i) \approx \frac{f(x_{i-1}) - 2f(x_i) + f(x_{i+1})}{h^2}$$

yaklaşımı kullanılır (sonlu fark yaklaşımlarını Sayısal Türev ve İntegral bölümünde inceleyeceğiz). Uçlarda $f(x_0) = f(x_{n+1}) = 0$ ise bu yaklaşımları $i = 1, \ldots, n$ için alt alta yazmak, iç noktalardaki değerlerin vektörünü $-L/h^2$ matrisiyle çarpmak demektir. L'nin köşegeninde 2, köşegenin hemen üstünde ve altında -1, öteki yerlerinde 0 bulunur. $n = 3$ için:

 CC BY-NC-SA 4.0

```python
import numpy as np

L = np.array([[2.0, -1.0, 0.0],
              [-1.0, 2.0, -1.0],
              [0.0, -1.0, 2.0]])
w, Q = np.linalg.eigh(L)
print(w)                                    # küçükten büyüğe, gerçel
print(2 - np.sqrt(2), 2 + np.sqrt(2))       # beklenen uç değerler
print(np.allclose(Q.T @ Q, np.eye(3)))      # Q ortogonal
print(np.allclose(Q @ np.diag(w) @ Q.T, L))   # L = Q Λ Q^T
```

Çıktı:

```
[0.58578644 2.         3.41421356]
0.5857864376269049 3.414213562373095
True
True
```

Bu matrisin özdeğerleri $2-\sqrt{2}$, 2 ve $2+\sqrt{2}$'dir; `eigh` bunları sırayla verdi. $Q^TQ = I$ ve $L = Q\Lambda Q^T$ eşitlikleri de sağlanıyor.

7.9 En Küçük Kareler

Denklem sayısı bilinmeyen sayısından fazla olan bir $Ax = b$ sisteminin (A $m \times n$ ve $m > n$) genellikle çözümü yoktur. Ölçüm verisine bir doğru ya da eğri uydururken karşılaştığımız durum budur. Nümerik Analiz notlarındaki en küçük kareler yöntemi, denklemleri tam sağlamaya çalışmak yerine hataların kareleri toplamını en küçük yapar.

Tanım 7.5 (En Küçük Kareler Çözümü). A bir $m \times n$ matris ve $b \in \mathbb{R}^m$ olsun. $\| Ax - b \|^2$ değerini en küçük yapan bir $\hat{x} \in \mathbb{R}^n$ vektörüne $Ax = b$ sisteminin **en küçük kareler çözümü** (least squares solution) denir.

Yani $\hat{x}$, $r = b - A\hat{x}$ artık vektörünün bileşenlerinin kareleri toplamını en küçük yapar. A'nın sütunları lineer bağımsızsa (rank $A = n$) $\hat{x}$ tektir ve $A^TA\hat{x} = A^Tb$ **normal denklemlerinin** çözümüdür. Doğru uydururken kurulan normal denklemler bunun $n = 2$ hâlidir. NumPy'da `np.linalg.lstsq(A, b)` dört şey döndürür: çözüm, artık kareleri toplamı, A'nın rankı ve A'nın tekil değerleri.

Örnek 7.9 (Dört Noktaya Doğru Uydurma). $(-1;\,1)$, $(-0{,}1;\,1{,}099)$, $(0{,}2;\,0{,}808)$ ve $(1;\,1)$ noktalarına en küçük kareler anlamında en iyi uyan $y = ax + b$ doğrusunu `lstsq` ile bulunuz.

Çözüm.
Her nokta bir $ax_i + b = y_i$ denklemi verir. Bilinmeyenler a ve b olduğundan katsayılar matrisinin sütunları x_i değerleri ve birlerdir:

$$\begin{pmatrix} -1 & 1 \\ -0{,}1 & 1 \\ 0{,}2 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \approx \begin{pmatrix} 1 \\ 1{,}099 \\ 0{,}808 \\ 1 \end{pmatrix}.$$

```python
import numpy as np

x = np.array([-1.0, -0.1, 0.2, 1.0])
y = np.array([1.0, 1.099, 0.808, 1.0])

A = np.column_stack([x, np.ones_like(x)])   # sütunlar: x ve 1
coef, rss, rank, sv = np.linalg.lstsq(A, y)
a, b = coef
print(f"a = {a:.9f}, b = {b:.9f}")
```

 CC BY-NC-SA 4.0

```
print("hata kareleri toplamı:", rss)
print("rank:", rank)
print("tekil değerler:", sv)
```

Çıktı:

```
a = -0.022454212, b = 0.977311355
hata kareleri toplamı: [0.04347042]
rank: 2
tekil değerler: [2.00127829 1.42999483]
```

Doğru $y \approx -0{,}022454x + 0{,}977311$'dir. Bu veri Nümerik Analiz notlarında normal denklemlerle elle çözülmüştü; orada bulunan $a = -0{,}022454212$ ve $b = 0{,}977311355$ değerleriyle dokuz basamak uyuşuyor. Hata kareleri toplamı da oradaki $E \approx 0{,}0435$ değeridir. ■

Örnek 7.10 (Gürültülü Veriye Parabol Uydurma). $[0, 4]$ aralığında eşit aralıklı 15 noktada $y = 1 + 2t - 0{,}5t^2$ değerlerine standart sapması $0{,}3$ olan normal dağılımlı gürültü eklenerek bir veri üretiliyor. Bu veriye $y = c_0 + c_1 t + c_2 t^2$ parabolünü en küçük kareler yöntemiyle uydurunuz.

Çözüm.
`rng.normal(0, 0.3, t.size)` ortalaması 0, standart sapması $0{,}3$ olan normal dağılımdan `t.size` tane sayı üretir (Rastgele Sayılar ve Monte Carlo Yöntemleri); tohum sabit olduğu için her çalıştırmada aynı veri çıkar. Model katsayılara göre lineerdir, bu yüzden 1, t, t^2 sütunlarından oluşan 15×3 bir M matrisiyle $Mc \approx y$ sistemini kuruyoruz.

```
import numpy as np

rng = np.random.default_rng(7)
t = np.linspace(0, 4, 15)
y = 1 + 2 * t - 0.5 * t**2 + rng.normal(0, 0.3, t.size)

M = np.column_stack([np.ones_like(t), t, t**2])   # sütunlar: 1, t, t^2
c, rss, rank, sv = np.linalg.lstsq(M, y)
print("katsayılar:", c)
r = y - M @ c                                     # artıklar
print("artık normu:", np.linalg.norm(r))
print("koşul sayıları:", np.linalg.cond(M), np.linalg.cond(M.T @ M))
```

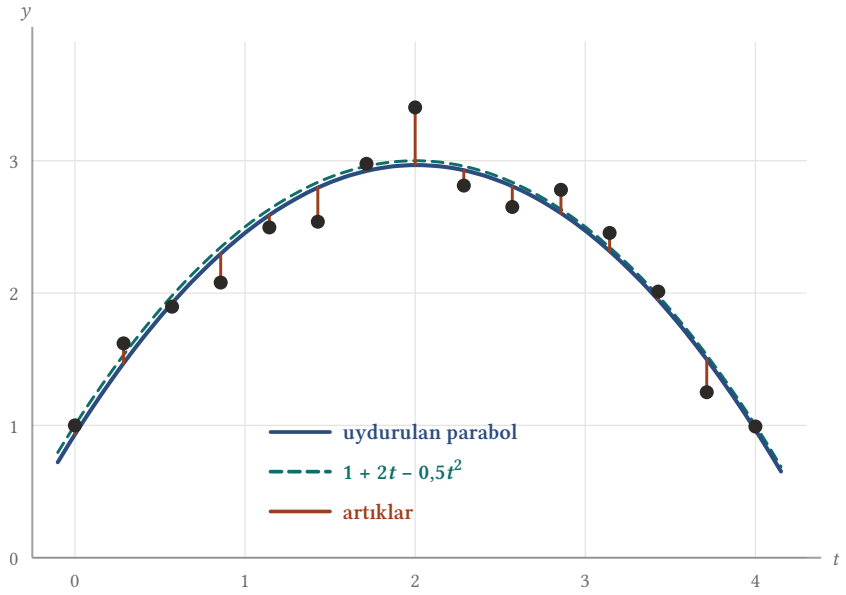
Çıktı:

```
katsayılar: [ 0.92977395  2.02839706 -0.50492187]
artık normu: 0.7047186321326601
koşul sayıları: 31.06032228313357 964.7436203321174
```

Uydurulan parabol $y \approx 0{,}930 + 2{,}028t - 0{,}505t^2$'dir; gürültüye rağmen katsayılar verinin üretildiği 1, 2 ve $-0{,}5$ değerlerine yakın çıktı. Son satır önemli bir gözlem içeriyor: $M^T M$'nin koşul sayısı M'ninkinin karesidir, $31{,}06^2 \approx 964{,}7$. ■

Uydurulan parabol, verinin üretildiği eğri ve artıklar aşağıdaki şekilde görülüyor. Bu tür grafikleri bir sonraki bölümde Matplotlib ile çizeceğiz.

 CC BY-NC-SA 4.0



Şekil 7.5: On beş gürültülü veri noktası, `lstsq` ile uydurulan parabol $y \approx 0{,}930 + 2{,}028t - 0{,}505t^2$ ve verinin üretildiği $1 + 2t - 0{,}5t^2$ eğrisi (kesikli). Dikey parçalar artıklardır; `lstsq` bunların kareleri toplamını en küçük yapar.

> ⚠ Normal denklemleri doğrudan çözmeyin
>
> $A^TA\hat{x} = A^Tb$ sistemini `solve` ile çözmek kolay görünür, ama 2-normunda $\kappa(A^TA) = \kappa(A)^2$ olduğundan kaybedilen basamak sayısı iki katına çıkar (Alıştırma 7.9). `lstsq` normal denklemleri kurmaz, A'nın tekil değer ayrışımını kullanır. Polinom uydurmanın hazır araçlarını Polinomlar, İnterpolasyon ve Eğri Uydurma bölümünde göreceğiz.

7.10 Tekil Değer Ayrışımı

Bölüm boyunca birkaç kez tekil değerlere gönderme yaptık: `matrix_rank`, 2-normu, `cond` ve `lstsq` hep onlara dayanır. Şimdi bu ayrışımı tanıyalım.

Tanım 7.6 (Tekil Değer Ayrışımı). A bir $m \times n$ gerçel matris ve $p = \min(m, n)$ olsun. U $(m \times m)$ ve V $(n \times n)$ matrisleri $U^TU = I$ ve $V^TV = I$ eşitliklerini sağlasın; Σ ise köşegeninde $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$ sayıları, öteki yerlerinde 0 bulunan $m \times n$ bir matris olsun.

$$A = U\Sigma V^T$$

yazılışına A'nın **tekil değer ayrışımı** (singular value decomposition, SVD), σ_i sayılarına A'nın **tekil değerleri**, U'nun ve V'nin u_i ve v_i sütunlarına **sol** ve **sağ tekil vektörler** denir.

Yani her $i \leq p$ için $Av_i = \sigma_i u_i$'dir: A, birbirine dik v_i doğrultularını birbirine dik u_i doğrultularına gönderir ve i. doğrultuyu σ_i katına uzatır. Spektral teoremden farkı, giriş ve çıkış için iki ayrı dik taban kullanılmasıdır; bu sayede ayrışım her matris için, kare olmayanlar için bile vardır. NumPy'da `np.linalg.svd(A)` üç dizi döndürür: `U`, tekil değerleri büyükten küçüğe içeren tek eksenli `s` dizisi ve V'nin kendisi değil, transpozu olan `Vt`.

```
import numpy as np

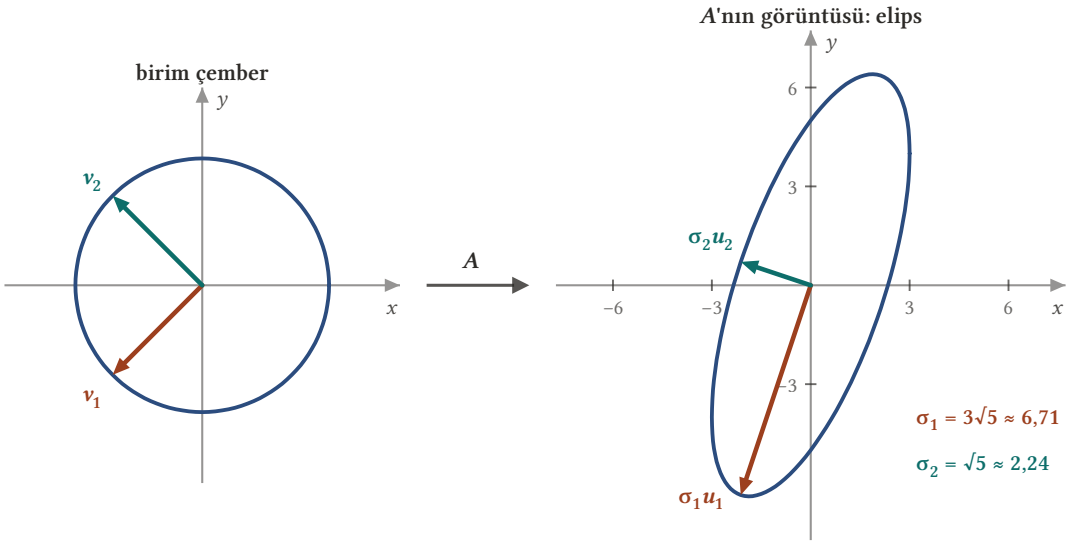
A = np.array([[3.0, 0.0],
              [4.0, 5.0]])
```

```
U, s, Vt = np.linalg.svd(A)
print(U)
print(s)                    # tekil değerler, büyükten küçüğe
print(Vt)                   # dikkat: V değil, V^T
print(np.allclose(U @ np.diag(s) @ Vt, A))
print(np.sqrt(45), np.sqrt(5))
```

Çıktı:

```
[[-0.31622777 -0.9486833 ]
 [-0.9486833   0.31622777]]
[6.70820393 2.23606798]
[[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]
True
6.708203932499369 2.23606797749979
```

$A^TA = \begin{pmatrix} 25 & 20 \\ 20 & 25 \end{pmatrix}$ matrisinin özdeğerleri 45 ve 5'tir ve tekil değerler bunların karekökleridir: $\sigma_1 = 3\sqrt{5}$, $\sigma_2 = \sqrt{5}$. Sağ tekil vektörler $v_1 = -(1,1)/\sqrt{2}$ ve $v_2 = (-1,1)/\sqrt{2}$, sol tekil vektörler $u_1 = -(1,3)/\sqrt{10}$ ve $u_2 = (-3,1)/\sqrt{10}$'dur. Örneğin $Av_1 = -(3,9)/\sqrt{2} = \sigma_1 u_1$'dir. Şekil ayrışımın geometrisini gösteriyor: A birim çemberi bir elipse götürür ve elipsin yarı eksenleri $\sigma_1 u_1$ ile $\sigma_2 u_2$'dir.



Şekil 7.6: A = [[3, 0], [4, 5]] matrisinin tekil değer ayrışımının geometrisi. Birim çemberdeki dik v_1, v_2 vektörleri, elipsin yarı eksenleri olan $\sigma_1 u_1$ ve $\sigma_2 u_2$ vektörlerine gider. Vektörlerin işaretleri `svd`'nin döndürdüğü gibidir; işaret seçimi keyfidir.

Önerme 7.3 (Tekil Değerlerin Söyledikleri). *Her gerçel A matrisinin bir tekil değer ayrışımı vardır ve tekil değerleri tek türlü belirlidir: $\sigma_1^2, \ldots, \sigma_p^2$ sayıları A^TA'nın özdeğerleridir (varsa A^TA'nın öteki özdeğerleri 0'dır). Ayrıca şunlar doğrudur.*
(i) A'nın rankı, sıfırdan farklı tekil değerlerinin sayısıdır.
(ii) $\| A \|_2 = \sigma_1$'dir.
(iii) A kare ve regüler ise $\kappa(A) = \sigma_1/\sigma_n$'dir (2-normunda).

Kare olmayan bir matrisin tersi yoktur, ama (iii)'teki oran onun için de anlamlıdır. Sütunları lineer bağımsız $m \times n$ bir A matrisinin $(m > n)$ 2-normundaki koşul sayısı $\kappa(A) = \sigma_1/\sigma_n$ olarak tanımlanır ve `np.linalg.cond` kare olmayan matrislerde bu oranı hesaplar. En küçük kareler kısmında M için

yazdırdığımız koşul sayısı budur. M^TM'nin özdeğerleri σ_i^2 olduğundan $\kappa(M^TM) = \sigma_1^2/\sigma_n^2 = \kappa(M)^2$ eşitliği de buradan çıkar.
Sessizce yanlış çözüm örneğindeki (Örnek 7.4) matrise tekil değerleriyle bakalım. `compute_uv=False` yalnızca tekil değerleri hesaplatır; `np.finfo(float).eps` makine epsilonunu verir.

```
import numpy as np

A = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6],
              [0.7, 0.8, 0.9]])
s = np.linalg.svd(A, compute_uv=False)     # yalnız tekil değerler
print(s)

tol = s.max() * max(A.shape) * np.finfo(float).eps
print(tol, np.sum(s > tol), np.linalg.matrix_rank(A))
print(np.isclose(np.linalg.norm(A, 2), s[0]))
print(s[0] / s[-1], np.linalg.cond(A))
```

Çıktı:

```
[1.68481034e+00 1.06836951e-01 3.65063904e-17]
1.1223091358001955e-15 2 2
True
4.615110719530146e+16 4.615110719530146e+16
```

Üçüncü tekil değer $3{,}7 \cdot 10^{-17}$'dir: tam aritmetikte 0 olması gereken bir sayının yuvarlama hatası. `matrix_rank`, $\sigma_1 \cdot \max(m,n) \cdot \varepsilon \approx 1{,}1 \cdot 10^{-15}$ eşiğinden büyük tekil değerleri sayar ve rankı 2 bulur. 2-normu en büyük tekil değere, koşul sayısı $\sigma_1/\sigma_3 \approx 4{,}6 \cdot 10^{16}$ oranına eşittir. Bu koşul sayısı $1/\varepsilon$'dan büyük olduğundan matris sayısal olarak tekildir ve `solve`'un ürettiği sayılar anlamsızdır.
Tekil değer ayrışımı A'yı rankı 1 olan matrislerin toplamı olarak da yazar:

$$A = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \ldots + \sigma_p u_p v_p^T.$$

Tekil değerler büyükten küçüğe sıralı olduğundan bu toplamın ilk terimleri en önemli kısımdır.

Teorem 7.2 (En İyi Düşük Ranklı Yaklaşım). *$A = U\Sigma V^T$ ve $k < \operatorname{rank} A$ olsun.*

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

matrisi, rankı en fazla k olan bütün B matrisleri arasında $\| A - B \|_2$ değerini en küçük yapar ve $\| A - A_k \|_2 = \sigma_{k+1}$'dir.

Bu sonuç Eckart–Young teoremi olarak bilinir. Yani A'yı k terimle yaklaşık yazmanın en iyi yolu ayrışımın ilk k terimini almaktır ve yapılan hata, atılan ilk tekil değerdir. $m \times n$ bir matrisin A_k yaklaşımını saklamak için mn yerine $k(m+n+1)$ sayı yeter.

Örnek 7.11 (Bir Görüntünün Düşük Ranklı Yaklaşımı). 32×32 bir matrisin elemanlarını $[0,1]$ aralığındaki gri tonlar olarak düşünelim (0 zemin rengi, 1 en yoğun ton). Bir halka ve onu kesen çapraz bir şeritten oluşan görüntünün rank 2, 6 ve 12 yaklaşımlarını hesaplayıp hataları tekil değerlerle karşılaştırınız.

> **Çözüm.**
> Izgarayı broadcasting ile kuruyoruz: `X` sütunun, `Y` satırın konumunu tutar. Halka ile şeridin kesiştiği hücrelerde toplam 1'i aşar; `np.clip(z, 0, 1)` bir `z` dizisinin 0'dan küçük elemanlarını 0'a, 1'den büyük elemanlarını 1'e eşitler. `U[:, :k] * s[:k]` çarpımı U'nun ilk k sütununu karşılık gelen tekil değerlerle çarpar; bunu `Vt[:k]` ile çarpmak ayrışımın ilk k teriminin toplamını verir.
>
> ```
> import numpy as np
> ```

 CC BY-NC-SA 4.0

```
n = 32
g = np.linspace(-1, 1, n)
X, Y = g[None, :], g[:, None]                  # broadcasting ile ızgara
R = np.sqrt(X**2 + Y**2)
ring = np.exp(-((R - 0.6) / 0.12) ** 2)        # bir halka
band = (np.abs(X - Y) < 0.15).astype(float)    # çapraz bir şerit
img = np.clip(ring + band, 0, 1)               # [0, 1] aralığına kırp

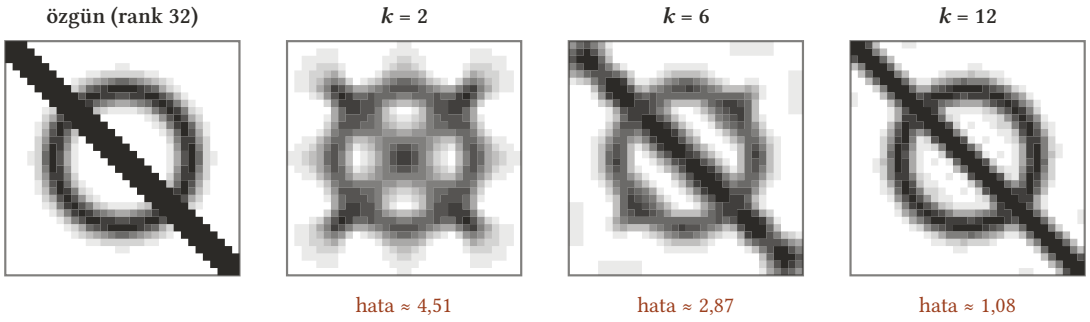
U, s, Vt = np.linalg.svd(img)
print("rank:", np.linalg.matrix_rank(img))
print(np.round(s[:6], 3))
for k in (2, 6, 12):
    img_k = (U[:, :k] * s[:k]) @ Vt[:k]        # ilk k terimin toplamı
    err = np.linalg.norm(img - img_k, 2)
    print(k, f"{err:.6f}", f"{s[k]:.6f}", k * (2 * n + 1))
```

Çıktı:

```
rank: 32
[12.502  5.364  4.514  3.85   3.834  3.251]
2 4.514021 4.514021 130
6 2.872318 2.872318 390
12 1.077171 1.077171 780
```

Keskin kenarlı çapraz şerit tek başına neredeyse tam ranklıdır; bu yüzden görüntünün rankı da tamdır (32). Her satırda hata, Eckart–Young teoreminin (Teorem 7.2) söylediği gibi atılan ilk tekil değer olan σ_{k+1}'e eşit çıktı. Son sütun yaklaşımı saklamak için gereken sayı adedidir: 12 terim için 780 sayı, özgün görüntünün 1024 sayısından azdır. ■

Yaklaşımları görüntü olarak çizdiğimizde rank 2'nin yalnızca kaba bir desen verdiği, rank 12'nin ise halkayı ve şeridi tanınır biçimde yakaladığı görülüyor. Çapraz şerit gibi satırlara ve sütunlara paralel olmayan ayrıntılar çok terim gerektirir.



Şekil 7.7: 32 × 32 boyutlu bir matris gri tonlu bir görüntü olarak (renk ne kadar yoğunsa değer o kadar büyük) ve onun `svd` ile bulunan rank 2, 6 ve 12 yaklaşımları. Alttaki sayılar 2-normundaki hatalardır ve sırasıyla σ_3, σ_7, σ_{13} tekil değerlerine eşittir. Çizimde 0'ın altına ya da 1'in üstüne taşan değerler 0 ve 1'e kırpıldı.

7.11 Alıştırmalar

Aşağıdaki alıştırmaları önce kendiniz çözmeyi deneyin; çözümlerdeki kodlar NumPy 2.5 ile çalıştırılmış ve çıktıları olduğu gibi verilmiştir.

 CC BY-NC-SA 4.0

Alıştırma 7.1 (Cramer Kuralının Kodu). Cramer kuralını uygulayan bir `cramer(A, b)` fonksiyonu yazınız ve Üç Bilinmeyenli Bir Sistem örneğindeki (Örnek 7.2) sistemde sonucunu `solve` ile karşılaştırınız.

Çözüm.
Cramer kuralına göre $x_j = \det(A_j)/\det(A)$'dır; A_j, A'nın j. sütununun yerine b konarak elde edilir. `A.copy()` ile kopya almak önemlidir; `Aj = A` yazsaydık `Aj` yeni bir dizi değil, aynı dizinin ikinci adı olurdu ve `A`'nın kendisi değişirdi (NumPy Dizileri).

```
import numpy as np


def cramer(A, b):
    """Ax = b sistemini Cramer kuralıyla çözer (A kare ve regüler)."""
    d = np.linalg.det(A)
    x = np.empty(len(b))
    for j in range(len(b)):
        Aj = A.copy()
        Aj[:, j] = b                # j. sütunun yerine b konur
        x[j] = np.linalg.det(Aj) / d
    return x


A = np.array([[2.0, 1.0, -3.0],
              [3.0, -2.0, 2.0],
              [5.0, -3.0, -1.0]])
b = np.array([5.0, 5.0, 16.0])
print(cramer(A, b))
print(np.linalg.solve(A, b))
print(np.allclose(cramer(A, b), np.linalg.solve(A, b)))
```

Çıktı:

```
[ 1. -3. -2.]
[ 1. -3. -2.]
True
```

İki yöntem aynı sonucu veriyor. Ama Cramer kuralı $n+1$ determinant hesaplar; her biri yaklaşık $\frac{2}{3}n^3$ işlem tuttuğundan toplam maliyet n^4 mertebesindedir. Bu yüzden yöntem kuramsal olarak değerlidir, ama büyük sistemlerin çözümünde kullanılmaz. ■

Alıştırma 7.2 (Dört Denklemli Bir Sistem).

$$\begin{aligned} x + 2y + 2z &= 2 \\ 3x - 2y - z &= 5 \\ 2x - 5y + 3z &= -4 \\ x + 4y + 6z &= 0 \end{aligned}$$

sisteminin çözülebilir olduğunu rank ölçütüyle gösterip çözümünü `lstsq` ile bulunuz.

Çözüm.
Katsayılar matrisi 4×3 olduğundan `solve` kullanılamaz, çünkü `solve` kare matris ister. Önce rankları karşılaştırıp sonra `lstsq`'yu çalıştırıyoruz.

```
import numpy as np

A = np.array([[1.0, 2.0, 2.0],
              [3.0, -2.0, -1.0],
```

 CC BY-NC-SA 4.0

```
              [2.0, -5.0, 3.0],
              [1.0, 4.0, 6.0]])
b = np.array([2.0, 5.0, -4.0, 0.0])

Ab = np.column_stack([A, b])
print(np.linalg.matrix_rank(A), np.linalg.matrix_rank(Ab))

x, rss, rank, sv = np.linalg.lstsq(A, b)
print(x)
print(np.linalg.norm(A @ x - b))
```

Çıktı:

```
3 3
[ 2.  1. -1.]
6.556898606395034e-15
```

$\operatorname{rank} A = \operatorname{rank}[A \mid b] = 3$ olduğundan sistem çözülebilirdir; rank bilinmeyen sayısına eşit olduğundan çözüm tektir. Çözülebilir bir sistemde en küçük kareler çözümü gerçek çözümdür: artık normu 10^{-15} mertebesindedir, yani denklemler yuvarlama hatası dışında tam sağlanır. Çözüm $(x, y, z) = (2, 1, -1)$'dir: $2 + 2 - 2 = 2$, $6 - 2 + 1 = 5$, $4 - 5 - 3 = -4$ ve $2 + 4 - 6 = 0$. ■

Alıştırma 7.3 (Normlar Arasındaki İki Eşitsizlik).

$$A = \begin{pmatrix} 1 & -2 & 3 \\ 0 & 4 & -1 \\ 2 & 1 & 5 \end{pmatrix}$$

matrisinin $\|A\|_1$, $\|A\|_\infty$, $\|A\|_2$ ve $\|A\|_F$ normlarını hesaplayıp $\|A\|_2 \le \|A\|_F$ ve $\|A\|_2 \le \sqrt{\|A\|_1 \|A\|_\infty}$ eşitsizliklerinin sağlandığını doğrulayınız.

Çözüm.
Dört normu hesaplayıp iki eşitsizliği sınıyoruz.

```
import numpy as np

A = np.array([[1.0, -2.0, 3.0],
              [0.0, 4.0, -1.0],
              [2.0, 1.0, 5.0]])
n1 = np.linalg.norm(A, 1)
n_inf = np.linalg.norm(A, np.inf)
n2 = np.linalg.norm(A, 2)
n_fro = np.linalg.norm(A, "fro")
print(n1, n_inf, n2, n_fro)
print(n2 <= n_fro, n2 <= np.sqrt(n1 * n_inf))
```

Çıktı:

```
9.0 8.0 6.399487517369997 7.810249675906654
True True
```

Mutlak değerlerle sütun toplamları 3, 7, 9 ve satır toplamları 6, 5, 8 olduğundan $\|A\|_1 = 9$ ve $\|A\|_\infty = 8$'dir. $\|A\|_F = \sqrt{61} \approx 7{,}81$ ve $\sqrt{9 \cdot 8} = \sqrt{72} \approx 8{,}49$ değerleri $\|A\|_2 \approx 6{,}40$'tan büyüktür.
İki eşitsizlik her kare matris için doğrudur. Birincisi tekil değerlerden gelir: Frobenius normu dik matrislerle çarpınca değişmediğinden

 CC BY-NC-SA 4.0

$$\| A \|_F^2 = \| \Sigma \|_F^2 = \sigma_1^2 + \ldots + \sigma_n^2 \geq \sigma_1^2 = \| A \|_2^2$$

dir. İkincisi için $\| A \|_2^2$'nin A^TA'nın en büyük özdeğeri olduğunu kullanalım. $Mv = \lambda v$ ve $v \neq 0$ ise $|\lambda| \, \| v \|_1 = \| Mv \|_1 \leq \| M \|_1 \| v \|_1$ olduğundan her özdeğerin mutlak değeri $\| M \|_1$'i aşmaz. $M = A^TA$ için bu

$$\| A \|_2^2 \leq \| A^TA \|_1 \leq \| A^T \|_1 \| A \|_1 = \| A \|_\infty \| A \|_1$$

verir; son adımda A^T'nin sütunlarının A'nın satırları olduğunu kullandık. ■

Alıştırma 7.4 (Vandermonde Matrislerinin Koşul Sayısı). $[0, 1]$ aralığında eşit aralıklı n nokta $x_1, \ldots, x_n$ için (i, j) elemanı x_i^{j-1} olan $n \times n$ Vandermonde matrisinin koşul sayısını $n = 4, 8, 12, 16, 20$ için hesaplayıp her durumda kaybedilmesi beklenen anlamlı basamak sayısını tahmin ediniz.

Çözüm.
`np.vander(x, increasing=True)` sütunları $1, x, x^2, \ldots$ olan Vandermonde matrisini kurar (NumPy Dizileri). Kaybedilen basamak sayısını $\log_{10} \kappa$ ile tahmin ediyoruz.

```
import numpy as np

for n in (4, 8, 12, 16, 20):
    x = np.linspace(0, 1, n)
    V = np.vander(x, increasing=True)    # sütunlar: 1, x, x^2, ...
    kappa = np.linalg.cond(V)
    print(f"n = {n:2d}   koşul = {kappa:8.2e}   "
          f"kaybedilen basamak ≈ {np.log10(kappa):4.1f}")
```

Çıktı:

```
n =  4   koşul = 9.89e+01   kaybedilen basamak ≈  2.0
n =  8   koşul = 2.68e+05   kaybedilen basamak ≈  5.4
n = 12   koşul = 8.83e+08   kaybedilen basamak ≈  8.9
n = 16   koşul = 3.12e+12   kaybedilen basamak ≈ 12.5
n = 20   koşul = 1.15e+16   kaybedilen basamak ≈ 16.1
```

Koşul sayısı n ile üstel olarak büyüyor: $n = 8$'de yaklaşık 5, $n = 12$'de yaklaşık 9 basamak kaybedilir ve $n = 20$'de koşul sayısı 10^{16}'yı geçer; bu matrisle kurulan bir sistemin çözümünde hiçbir basamağa güvenilemez. Bu matris, verilen noktalardan geçen polinomun katsayılarını bulmak için kurulan sistemin matrisidir. Bu yüzden interpolasyon polinomu pratikte bu sistem çözülerek değil, Lagrange ya da Newton biçimiyle hesaplanır. ■

Alıştırma 7.5 (Özdeğerlerle Fibonacci Sayıları). Fibonacci örneğindeki (Örnek 7.1) Q matrisini $Q = V\Lambda V^{-1}$ biçiminde köşegenleştirip F_n'yi $Q^n = V\Lambda^n V^{-1}$ formülüyle hesaplayınız ve sonucu $n = 10, 40, 70, 80$ için tam değerle karşılaştırınız.

Çözüm.
Q'nun karakteristik polinomu $t^2 - t - 1$'dir; özdeğerleri $\varphi = \frac{1+\sqrt{5}}{2}$ ve $-1/\varphi = \frac{1-\sqrt{5}}{2}$'dir. Λ köşegen olduğundan Λ^n'yi hesaplamak köşegen elemanların n. kuvvetini almaktır. Q simetrik olduğundan `eigh` kullanıyoruz: döndürdüğü V ortogonaldir, yani $V^{-1} = V^T$'dir ve ters matris hesaplamaya gerek kalmaz. `eigh` özdeğerleri küçükten büyüğe sıraladığı için önce $-1/\varphi$ gelir. Tam değerleri `dtype=object` ile tamsayı matris kuvvetinden alıyoruz.

```
import numpy as np

Q = np.array([[1.0, 1.0],
              [1.0, 0.0]])
w, V = np.linalg.eigh(Q)
print(w)                                        # -1/φ ve φ

Q_obj = np.array([[1, 1], [1, 0]], dtype=object)
for n in (10, 40, 70, 80):
    Qn = V @ np.diag(w**n) @ V.T                # Q^n = V Λ^n V^T
    exact = np.linalg.matrix_power(Q_obj, n)[0, 1]
    approx = round(Qn[0, 1])
    print(n, approx, exact, approx - exact)
```

Çıktı:

```
[-0.61803399  1.61803399]
10 55 55 0
40 102334155 102334155 0
70 190392490709135 190392490709135 0
80 23416728348467740 23416728348467685 55
```

$n = 70$'e kadar sonuç tam doğru. $F_{80} = 23\,416\,728\,348\,467\,685$ ise 17 basamaklı bir sayıdır; `float64` yalnızca 15 ile 16 arası anlamlı basamak taşıyabildiğinden son basamaklar yanlış çıkar ve fark 55 olur. Bu hesabın ifade ettiği formül $F_n = \left(\varphi^n - (-1/\varphi)^n \right) / \sqrt{5}$ Binet formülüdür. Matematikte tam olan bu formül, kayan noktalı sayılarla büyük n için tam sonuç veremez. ■

Alıştırma 7.6 (Kuvvet Yöntemi).

$$A = \begin{pmatrix} 4 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

matrisinin mutlak değerce en büyük özdeğerini **kuvvet yöntemiyle** yaklaşık olarak bulup `eigh` ile karşılaştırınız: $x_0 = (1, 1, 1)$ vektöründen başlayıp $x_{k+1} = Ax_k / \| Ax_k \|_2$ adımlarını tekrarlayınız ve özdeğer tahmini olarak $\lambda_k = x_k^T A x_k$ Rayleigh bölümünü alınız.

Çözüm.
A simetrik olduğundan özvektörleri dik bir q_1, q_2, q_3 tabanı oluşturur. $x_0 = c_1 q_1 + c_2 q_2 + c_3 q_3$ ise

$$A^k x_0 = c_1 \lambda_1^k q_1 + c_2 \lambda_2^k q_2 + c_3 \lambda_3^k q_3$$

olur. $|\lambda_1|$ en büyükse ve $c_1 \neq 0$ ise ilk terim baskın çıkar, yani x_k birim vektörü $\pm q_1$'e yaklaşır. Her adımda normalleştirmek sayıların taşmasını önler.

```
import numpy as np

A = np.array([[4.0, 1.0, 0.0],
              [1.0, 3.0, 1.0],
              [0.0, 1.0, 2.0]])
x = np.array([1.0, 1.0, 1.0])
for k in range(1, 31):
    y = A @ x
    x = y / np.linalg.norm(y)       # her adımda birim vektöre indir
    lam = x @ A @ x                 # Rayleigh bölümü
    if k % 5 == 0:
```

 CC BY-NC-SA 4.0

```
        print(f"{k:2d}  {lam:.12f}")

w, Q = np.linalg.eigh(A)
print("eigh:", w)
```

Çıktı:

```
 5  4.729619855519
10  4.732025279744
15  4.732050539813
20  4.732050804760
25  4.732050807539
30  4.732050807569
eigh: [1.26794919 3.         4.73205081]
```

En büyük özdeğer $3+\sqrt{3}\approx 4{,}732050808$'dir ve 30 adımda on iki basamak doğru bulundu. Yakınsama hızını $|\,\lambda_2/\lambda_1\,| = 3/4{,}732 \approx 0{,}634$ oranı belirler: Rayleigh bölümünün hatası her adımda yaklaşık bu oranın karesiyle, $0{,}40$ ile çarpılır. Tabloda beş adımda hata yaklaşık $0{,}40^5 \approx 0{,}01$ katına iniyor. İki büyük özdeğer birbirine yakın olsaydı yöntem çok yavaşlardı. ■

Alıştırma 7.7 (Bir Markov Zincirinin Daimi Durumu). Geçiş matrisi

$$P = \begin{pmatrix} 0{,}5 & 0{,}5 \\ 0{,}1 & 0{,}9 \end{pmatrix}$$

olan Markov zincirinin daimi durum olasılığını, yani $\pi P = \pi$ ve $\pi_1 + \pi_2 = 1$ koşullarını sağlayan π satır vektörünü `eig` ile bulunuz.

Çözüm.
$\pi P = \pi$ eşitliğinin transpozunu alırsak $P^T \pi^T = \pi^T$ olur: π^T, P^T'nin 1 özdeğerine ait bir özvektörüdür. `eig` birim uzunlukta bir özvektör döndürür; onu bileşenleri toplamı 1 olacak biçimde ölçekliyoruz. `np.argmin(np.abs(w - 1))`, 1'e en yakın özdeğerin indeksini verir.

```
import numpy as np

P = np.array([[0.5, 0.5],
              [0.1, 0.9]])
w, V = np.linalg.eig(P.T)          # πP = π  ⟺  P^T π^T = π^T
w, V = np.real_if_close(w), np.real_if_close(V)
print(w)
i = np.argmin(np.abs(w - 1))       # özdeğeri 1 olan sütun
pi = V[:, i] / V[:, i].sum()       # bileşenler toplamı 1 olsun
print(pi)
print(np.allclose(pi @ P, pi))
```

Çıktı:

```
[0.4 1. ]
[0.16666667 0.83333333]
True
```

P^T'nin özdeğerleri $0{,}4$ ve 1'dir. Daimi durum $\pi = (1/6,\, 5/6)$, yani yaklaşık $(0{,}1667;\, 0{,}8333)$'tür; bu, Raslantı Süreçleri notlarında denklem sistemiyle bulunan sonuçla aynıdır. Zincir uzun vadede zamanın altıda birini birinci, altıda beşini ikinci durumda geçirir. ■

 CC BY-NC-SA 4.0

Alıştırma 7.8 (Bir Elipsin Asal Eksenleri). $5x^2 + 4xy + 2y^2 = 1$ elipsinin asal eksen doğrultularını ve yarı eksen uzunluklarını, denklemi $\mathbf{u} = (x, y)^T$ olmak üzere $\mathbf{u}^T S\,\mathbf{u} = 1$ biçiminde yazıp S simetrik matrisinin özdeğer ayrışımıyla bulunuz.

Çözüm.
$\mathbf{u}^T S\,\mathbf{u} = s_{11}x^2 + 2s_{12}xy + s_{22}y^2$ olduğundan karma terimin katsayısı 4 köşegen dışına ikiye bölünerek yazılır:

$$S = \begin{pmatrix} 5 & 2 \\ 2 & 2 \end{pmatrix}.$$

Spektral teoremle (Teorem 7.1) $S = Q\Lambda Q^T$ yazıp $\mathbf{u}' = Q^T\mathbf{u}$ yeni koordinatlarına geçersek denklem $\lambda_1 x'^2 + \lambda_2 y'^2 = 1$ olur; yarı eksenler $1/\sqrt{\lambda_1}$ ve $1/\sqrt{\lambda_2}$'dir.

```
import numpy as np

S = np.array([[5.0, 2.0],
              [2.0, 2.0]])     # 5x^2 + 4xy + 2y^2 = [x y] S [x y]^T
w, Q = np.linalg.eigh(S)
print(w)
print(Q)
print(1 / np.sqrt(w))          # yarı eksen uzunlukları
theta = np.degrees(np.arctan(Q[1, 1] / Q[0, 1]))
print(theta)                   # λ = 6 özvektörünün x ekseniyle açısı
```

Çıktı:

```
[1. 6.]
[[ 0.4472136  -0.89442719]
 [-0.89442719 -0.4472136 ]]
[1.         0.40824829]
26.56505117707799
```

Özdeğerler 1 ve 6'dır, dolayısıyla yeni koordinatlarda denklem $x'^2 + 6y'^2 = 1$'dir. Büyük yarı eksen 1 uzunluğundadır ve $\lambda = 1$'in özvektörü $(1, -2)/\sqrt{5}$ doğrultusundadır. Küçük yarı eksen $1/\sqrt{6} \approx 0{,}408$ uzunluğundadır ve $\lambda = 6$'nın özvektörü $(2, 1)/\sqrt{5}$ doğrultusundadır (`eigh` işaretleri keyfi seçer). $(2, 1)$ doğrultusunun x ekseniyle yaptığı açı $\arctan\frac{1}{2} \approx 26{,}57°$'dir. Bu, Analitik Geometri notlarındaki $\tan 2\theta = 2a_{12}/(a_{11} - a_{22}) = 4/3$ formülünün verdiği dönme açısıdır. ■

Alıştırma 7.9 (Normal Denklemlerin Kaybettirdiği Basamaklar). $[0, 1]$ aralığında eşit aralıklı 30 noktada $p(t) = 1 + t + t^2 + \ldots + t^9$ polinomunun değerleri veriliyor. Bu veriye dokuzuncu dereceden bir polinomu hem `lstsq` ile hem de normal denklemleri `solve` ile çözerek uydurunuz ve bulunan katsayıların en büyük hatalarını karşılaştırınız.

Çözüm.
Veri tam olarak bir polinomdan geldiği için doğru katsayıların hepsi 1'dir; hatayı $\max_k |\, c_k - 1\,|$ ile ölçüyoruz. Katsayılar matrisi M, sütunları $1, t, \ldots, t^9$ olan 30×10 bir Vandermonde matrisidir.

```
import numpy as np

t = np.linspace(0, 1, 30)
M = np.vander(t, 10, increasing=True)   # sütunlar: 1, t, ..., t^9
c_true = np.ones(10)
y = M @ c_true                  # tam olarak bir polinomun değerleri

c_lstsq = np.linalg.lstsq(M, y)[0]
c_normal = np.linalg.solve(M.T @ M, M.T @ y)
```

 CC BY-NC-SA 4.0

```
print(f"koşul(M)       = {np.linalg.cond(M):.1e}")
print(f"koşul(M^T M)   = {np.linalg.cond(M.T @ M):.1e}")
print(f"lstsq hatası        : {np.max(np.abs(c_lstsq - c_true)):.1e}")
print(f"normal denk. hatası : {np.max(np.abs(c_normal - c_true)):.1e}")
```

Çıktı:

```
koşul(M)       = 3.5e+06
koşul(M^T M)   = 1.2e+13
lstsq hatası        : 9.2e-11
normal denk. hatası : 9.3e-04
```

M'nin koşul sayısı $3{,}5 \cdot 10^{6}$, $M^{T}M$'ninki onun karesi olan $1{,}2 \cdot 10^{13}$'tür. `lstsq` katsayıları 10^{-10} mertebesinde bir hatayla bulurken normal denklemlerin hatası 10^{-3} mertebesine çıktı: aşağı yukarı yedi basamak fazladan kaybedildi. ■

Alıştırma 7.10 (Tekil Değer Ayrışımıyla Çekirdek).

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

matrisinin çekirdeğini, yani $Mv = 0$ denkleminin çözüm uzayını tekil değer ayrışımı yardımıyla bulunuz.

Çözüm.
$M = U\Sigma V^{T}$ ise $Mv_i = \sigma_i u_i$'dir. Bu yüzden sıfır olan (ya da sayısal olarak sıfır sayılan) tekil değerlere ait sağ tekil vektörler çekirdeği gerer. Tekil değerler büyükten küçüğe sıralı olduğundan aranan vektör `Vt`'nin son satırıdır.

```
import numpy as np

M = np.array([[1.0, 2.0, 3.0],
              [4.0, 5.0, 6.0],
              [7.0, 8.0, 9.0]])
U, s, Vt = np.linalg.svd(M)
print(s)
v = Vt[-1]                  # en küçük tekil değere ait sağ tekil vektör
print(v / v[0])             # ilk bileşeni 1 olacak biçimde ölçekle
print(np.linalg.norm(M @ v))
```

Çıktı:

```
[1.68481034e+01 1.06836951e+00 3.33475287e-16]
[ 1. -2.  1.]
5.20740757162067e-16
```

Üçüncü tekil değer $3{,}3 \cdot 10^{-16}$, yani sayısal olarak sıfırdır; öteki ikisi açıkça sıfırdan farklı olduğundan $\operatorname{rank} M = 2$ ve çekirdek bir boyutludur. Son sağ tekil vektör, ilk bileşeni 1 olacak biçimde ölçeklenince $(1, -2, 1)$ çıkıyor. Gerçekten $1 - 4 + 3 = 0$, $4 - 10 + 6 = 0$ ve $7 - 16 + 9 = 0$'dır; çekirdek $\{t\,(1, -2, 1) : t \in \mathbb{R}\}$ doğrusudur. ■

Alıştırma 7.11 (En İyi Rank Bir Yaklaşım).

$$A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}$$

 CC BY-NC-SA 4.0

matrisine 2-normunda en yakın olan, rankı 1 olan A_1 matrisini bulup $\| A - A_1 \|_2$ değerini hesaplayınız.

Çözüm.
Eckart–Young teoremine (Teorem 7.2) göre aranan matris $A_1 = \sigma_1 u_1 v_1^T$'dir ve hata σ_2'dir. `np.outer(u, v)` iki vektörün dış çarpımı olan uv^T matrisini verir.

```
import numpy as np

A = np.array([[3.0, 2.0, 2.0],
              [2.0, 3.0, -2.0]])
U, s, Vt = np.linalg.svd(A)
print(s)
A1 = s[0] * np.outer(U[:, 0], Vt[0])    # σ1 u1 v1^T
print(np.round(A1, 12))
print(np.linalg.norm(A - A1, 2))
```

Çıktı:

```
[5. 3.]
[[ 2.5  2.5 -0. ]
 [ 2.5  2.5 -0. ]]
2.9999999999999996
```

Tekil değerler 5 ve 3'tür; gerçekten $AA^T = \begin{pmatrix} 17 & 8 \\ 8 & 17 \end{pmatrix}$ matrisinin özdeğerleri 25 ve 9'dur. $u_1 = (1,1)/\sqrt{2}$ ve $v_1 = (1,1,0)/\sqrt{2}$ olduğundan (işaretler birlikte değişebilir)

$$A_1 = 5 \cdot \frac{1}{2}\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2{,}5 & 2{,}5 & 0 \\ 2{,}5 & 2{,}5 & 0 \end{pmatrix}$$

bulunur. Çıktıdaki `-0.`, -10^{-16} gibi çok küçük negatif bir sayının yuvarlanmasından doğan işaretli sıfırdır; IEEE 754'te -0 ile 0 eşittir. Hata $\| A - A_1 \|_2 = 3 = \sigma_2$'dir; çıktıdaki $2{,}9999999999999996$ yine yuvarlama hatasıdır. ■

Bu bölümde NumPy'ın lineer cebir araçlarını ve onları güvenle kullanmanın kurallarını gördük: sistemleri `inv` ile değil `solve` ile çözmek, tekilliği determinantla değil rank ve koşul sayısıyla sınamak, simetrik matrislerde `eigh`'i seçmek, aşırı belirli sistemlerde normal denklemler yerine `lstsq`'ya başvurmak. Sonuçları bu bölümde sık sık şekillerle gösterdik; bir sonraki bölümde, Matplotlib ile Grafik Çizimi bölümünde, bu tür grafikleri kendimiz çizmeyi öğreneceğiz.

 CC BY-NC-SA 4.0

8. Matplotlib ile Grafik Çizimi

Bir fonksiyonun grafiğine bakmak, çoğu zaman bir sayfa hesabın söylediğini tek bakışta söyler: fonksiyon nerede artıyor, nerede sıçrıyor, bir yaklaşım hangi aralıkta iyi, bir dizi ne hızla yakınsıyor. Önceki bölümlerde NumPy ile bu sayıları üretmeyi öğrendik; bu bölümde onları görmeyi öğreneceğiz.
Python'da bilimsel çizimin standart aracı **Matplotlib** kütüphanesidir. Anaconda ve Colab ile hazır gelir; değilse `pip install matplotlib` komutuyla kurulur. Kütüphanenin iki kullanım biçimi vardır: kısa komutlarla "o anki" grafiğe çizen `pyplot` arayüzü ve çizimin her parçasını bir nesne olarak tutan nesne yönelimli arayüz. Biz baştan itibaren ikincisini kullanacağız, çünkü bir sayfaya birden çok grafik koyduğumuzda hangi komutun neyi değiştirdiği ancak böyle açık kalır. Önce tek bir eğri çizip grafiğin parçalarını tanıyacağız; sonra alt grafiklere, logaritmik eksenlere, parametrik ve kutupsal eğrilere, saçılım grafiklerine ve histogramlara, kontur çizimlerine ve üç boyutlu yüzeylere geçeceğiz.
Bu bölümdeki kodlar grafikleri `fig.savefig(...)` ile dosyaya yazar ve çizdikleri veriden birkaç sayıyı ekrana basar. Kodu Jupyter ya da Colab'de çalıştırırsanız grafik hücrenin altında kendiliğinden görünür; bir `.py` betiği olarak çalıştırıyorsanız grafiği bir pencerede görmek için en sona `plt.show()` satırını ekleyin. Sayfadaki şekiller, kodun ürettiği verinin aynısından sitenin temasına uygun renklerle yeniden çizilmiştir; bu yüzden renkleri Matplotlib'in varsayılan renklerinden farklıdır.

8.1 Figure ve Axes

Matplotlib'de her çizim iki katlı bir yapıya oturur: bir tuval ve tuvalin üzerindeki koordinat sistemleri.

Tanım 8.1 (Figure ve Axes). **Figure**, bir çizimin bütün tuvalidir; boyutu, çözünürlüğü ve üzerindeki her şey ona aittir. **Axes** ise bir Figure içindeki tek bir koordinat sistemidir; kendi yatay ve dikey eksenleri, tik işaretleri, eksen adları, başlığı ve lejantı vardır. Bir Figure bir ya da daha çok Axes taşır.

Yani Figure bir kâğıt, Axes o kâğıda çizilmiş bir koordinat sistemidir. Eğri çizmek, eksenlere ad koymak, sınırları ayarlamak gibi işlerin hepsi bir Axes nesnesinin metotlarıyla yapılır: `ax.plot`, `ax.set_xlabel`, `ax.set_xlim` gibi. Figure'a yalnız bütünle ilgili işler kalır: `fig.savefig` ile kaydetmek ya da `fig.suptitle` ile bütün çizime ortak bir başlık koymak gibi. Adın çoğul görünmesi yanıltmasın: bir Axes tek bir koordinat sistemidir, tek tek eksenlere ise Matplotlib *Axis* der.
İkisini birlikte üreten komut `plt.subplots()`'tur; burada `plt`, `matplotlib.pyplot` modülünün geleneksel kısaltmasıdır. `fig, ax = plt.subplots()` satırı bir Figure ile içinde tek bir Axes oluşturur ve döndürdüğü iki nesneyi iki değişkene açar. İnternetteki örneklerde sık görülen `plt.plot(x, y)` ya da `plt.xlabel("x")` gibi komutlar ise o anda etkin olan Axes'e çizer. Tek grafiklik kısa betiklerde bu biçim işe yarar, ama birden çok Axes olduğunda hangisinin etkin olduğunu izlemek zorlaşır; bu derste her zaman `ax` üzerinden çalışacağız.

Dört adımda bir grafik

1. Çizilecek veriyi NumPy dizileri olarak hazırla: `x = np.linspace(a, b, n)` ve `y = f(x)`.
2. `fig, ax = plt.subplots()` ile bir Figure ve bir Axes oluştur.
3. `ax.plot(x, y)` ile çiz; eksen adlarını, başlığı ve lejantı ekle.
4. `fig.savefig("ad.png")` ile kaydet ya da `plt.show()` ile ekranda göster.

Örnek 8.1 (Sinüs ve Kosinüs Aynı Eksende). $\sin x$ ve $\cos x$ fonksiyonlarını $[0, 2\pi]$ aralığında aynı Axes üzerinde, eğrileri bir lejantla adlandırarak çiziniz.

Çözüm.
Tarifin dört adımını sırayla uyguluyoruz.

 CC BY-NC-SA 4.0

```python
import numpy as np
import matplotlib.pyplot as plt

# 1. Veri: [0, 2π] aralığında eşit aralıklı 200 nokta
x = np.linspace(0, 2 * np.pi, 200)

# 2. Figure ve Axes
fig, ax = plt.subplots(figsize=(6, 4))

# 3. Çizim ve etiketler
ax.plot(x, np.sin(x), label=r"$\sin x$")
ax.plot(x, np.cos(x), linestyle="--", label=r"$\cos x$")
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_title("Sinüs ve kosinüs")
ax.legend()
ax.grid(True)

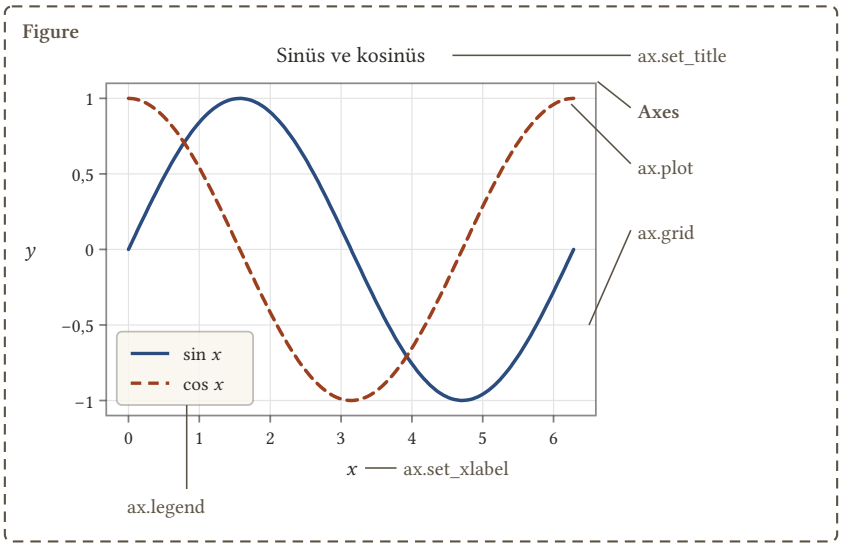
# 4. Kaydetme
fig.savefig("sinus_kosinus.png", dpi=150)

print(type(fig).__name__, type(ax).__name__)
print("Çizilen eğri sayısı:", len(ax.lines))
print("Piksel boyutu:", fig.get_size_inches() * 150)
```

Çıktı:

```
Figure Axes
Çizilen eğri sayısı: 2
Piksel boyutu: [900. 600.]
```

`print` satırları neyin oluştuğunu doğruluyor: `plt.subplots` bir `Figure` ve bir `Axes` döndürdü, `ax.lines` listesinde iki eğri var. Figure 6 × 4 inç boyutunda ve inç başına 150 nokta (`dpi=150`) ile kaydedildiği için PNG dosyası 900 × 600 pikseldir. ■



Şekil 8.1: İlk örnekteki kodun çizdiği grafik. Kesikli çerçeve bütün tuvali, yani *Figure* nesnesini; içteki çerçeve tek koordinat sistemini, yani *Axes* nesnesini gösterir. Kenardaki etiketler her parçayı hangi metodun oluşturduğunu söyler.

 CC BY-NC-SA 4.0

Koddaki her satırın bir görevi var:

- `np.linspace(0, 2 * np.pi, 200)` aralığı 200 eşit aralıklı noktayla örnekler. Matplotlib bu noktaları sırayla doğru parçalarıyla birleştirir; ekranda düzgün görünen eğri aslında 199 parçalı bir kırık çizgidir.
- `figsize=(6, 4)` Figure'ın boyutunu inç cinsinden verir.
- `ax.plot(x, y)` bir eğri çizer. Aynı Axes'e yapılan her `plot` çağrısı yeni bir eğri ekler ve Matplotlib her birine sıradaki rengi verir. `label=` lejantta görünecek adı, `linestyle="--"` kesikli çizgiyi seçer.
- `r"$\sin x$"` gibi dolar işaretleri arasındaki metinler, Matplotlib'in kendi TeX dizgicisiyle (mathtext) matematik olarak dizilir; bunun için bilgisayarda LaTeX kurulu olması gerekmez. Dizgideki ters bölü bir kaçış dizisi sanılmasın diye dizgiyi `r` önekiyle ham dizgi (raw string) olarak yazarız.
- `ax.legend()` etiketli eğrilerden lejantı kurar, `ax.grid(True)` ızgara çizgilerini açar.

8.2 Çizgiler, Renkler ve Açıklamalar

İlk grafikte renkleri, çizgi kalınlıklarını ve eksen sınırlarını Matplotlib kendisi seçti. Bu seçimlerin her biri `plot` çağrısının anahtar kelimeli argümanlarıyla ya da Axes'in metotlarıyla değiştirilebilir. En sık kullanılan argümanlar şunlardır:

Argüman	Örnek değerler	Anlamı
`color`	`"C0"`, `"C3"`, `"k"`, `"tab:red"`	Renk; `"C0"`–`"C9"` varsayılan renk sırası, `"k"` siyah
`linestyle`	`"-"`, `"--"`, `":"`, `"-."`	Düz, kesikli, noktalı ve kesik-noktalı çizgi
`linewidth`	`1`, `2.5`	Çizgi kalınlığı (punto)
`marker`	`"o"`, `"s"`, `"^"`, `"x"`, `"."`	Her veri noktasına konan işaret
`markersize`	`4`, `7`	İşaretin büyüklüğü

Kısa yol olarak üçüncü konumsal argümana bir biçim dizgisi yazılabilir: `ax.plot(x, y, "ro--")` kırmızı (`r`), daire işaretli (`o`), kesikli (`--`) bir çizgi çizer. Biçim dizgisinde çizgi stili yoksa, örneğin `"o"`, noktalar birleştirilmez; yalnız işaretler çizilir.
Grafiğin geri kalanını Axes'in metotları ayarlar:

- `ax.set_xlim(a, b)` ve `ax.set_ylim(c, d)` eksen sınırlarını sabitler. Sınırlar verilmezse Matplotlib onları veriden seçer ve iki yana veri aralığının %5'i kadar pay bırakır.
- `ax.set_aspect("equal")` iki eksende birim uzunluğu eşitler; çember çember, kare kare görünür.
- `ax.axhline(c)` ve `ax.axvline(c)` sırasıyla $y = c$ ve $x = c$ doğrularını çizer. Koordinat eksenlerini göstermek için kullanışlıdır.
- `ax.annotate(metin, xy=(x0, y0), xytext=(x1, y1), arrowprops=dict(arrowstyle="->"))` metni (x_1, y_1) noktasına yazar ve oradan (x_0, y_0) noktasına bir ok çizer. Ok gerekmiyorsa `ax.text(x, y, metin)` yeterlidir.
- `ax.fill_between(x, y1, y2)` iki eğrinin arasını boyar; `where=` argümanına bir boolean dizi verilirse yalnız koşulun sağlandığı yerler boyanır.

Örnek 8.2 (Yerel Ekstremumları İşaretlemek). $f(x) = x^3 - 3x$ fonksiyonunun $[-2{,}5;\, 2{,}5]$ aralığındaki grafiğini, kritik noktaları işaretlenmiş ve yerel ekstremumları okla gösterilmiş olarak çiziniz.

Çözüm.
$f'(x) = 3x^2 - 3 = 3(x-1)(x+1)$ olduğundan kritik noktalar $x = \pm 1$'dir. $f''(x) = 6x$ olduğundan $f''(-1) < 0$ ve $f''(1) > 0$'dır; $x = -1$ yerel maksimum, $x = 1$ yerel minimum noktasıdır. Kritik noktaları ayrı bir `plot` çağrısında `"o"` işaretiyle çiziyor, okları `annotate` ile koyuyoruz.

```
import numpy as np
import matplotlib.pyplot as plt


def f(x):
    return x**3 - 3 * x
```

```
x = np.linspace(-2.5, 2.5, 300)
crit = np.array([-1.0, 1.0])      # f'(x) = 3x^2 - 3 = 0 olan noktalar

fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(x, f(x), color="C0", linewidth=2, label="$f(x) = x^3 - 3x$")
ax.plot(crit, f(crit), "o", color="C3", markersize=7,
        label="kritik noktalar")
ax.axhline(0, color="k", linewidth=0.8)
ax.axvline(0, color="k", linewidth=0.8)
ax.annotate("yerel maksimum", xy=(-1, 2), xytext=(-2.4, 3.6),
            arrowprops=dict(arrowstyle="->"))
ax.annotate("yerel minimum", xy=(1, -2), xytext=(-0.6, -4.2),
            arrowprops=dict(arrowstyle="->"))
ax.set_ylim(-5, 5)
ax.legend(loc="lower right")
fig.savefig("ekstremum.png", dpi=150)

for c in crit:
    print(f"x = {c:+.0f} noktasında f(x) = {f(c):+.0f}")
```

Çıktı:

```
x = -1 noktasında f(x) = +2
x = +1 noktasında f(x) = -2
```

`f` fonksiyonu NumPy işlemleriyle yazıldığı için hem tek bir sayıya hem bir diziye uygulanabilir: `f(x)` 300 değerlik eğriyi, `f(crit)` iki kritik değeri verir. Çıktı $f(-1) = 2$ ve $f(1) = -2$ hesabıyla uyuşur. `set_ylim(-5, 5)` olmasaydı dikey eksen, uç noktalardaki $f(\pm 2{,}5) = \pm 8{,}125$ değerlerine göre seçilir ve ekstremumlar küçük kalırdı. ■

8.3 Alt Grafikler ve Logaritmik Eksenler

Birbiriyle karşılaştırılacak iki grafiği aynı Figure'a yan yana koymak çoğu zaman en açık anlatımdır. `plt.subplots(nrows, ncols)` çağrısı `nrows` satır ve `ncols` sütunluk bir Axes ızgarası kurar ve Axes nesnelerini bir NumPy dizisi olarak döndürür. Tek satır ya da tek sütun varsa dizi tek boyutludur (`axs[0]`, `axs[1]`, ...); yoksa iki boyutludur (`axs[1, 0]` ikinci satırın ilk Axes'i). Bütün Axes'leri tek bir döngüyle dolaşmak için `axs.flat` kullanılır. Yararlı üç argüman daha var: `sharex=True` ve `sharey=True` Axes'lere ortak eksen sınırları verir, `layout="constrained"` ise başlıkların ve eksen adlarının birbirinin üstüne binmemesi için Axes'lerin yerini kendiliğinden ayarlar.
Yan yana grafiklerin sık kullanıldığı bir yer, bir yaklaşımı ve onun hatasını birlikte göstermektir. Hata ise çoğu zaman birçok mertebeye yayılır; onu okunur kılan logaritmik eksendir.

Tanım 8.2 (Logaritmik Eksen). Bir eksen **logaritmik ölçekli** ise üzerindeki $v > 0$ değerinin konumu $\log_{10} v$ ile orantılıdır. Yalnız dikey ekseni logaritmik ölçekli olan grafiğe **yarı logaritmik grafik**, iki ekseni de logaritmik ölçekli olan grafiğe **log-log grafik** denir.

Yani logaritmik eksende 1 ile 10, 10 ile 100 ve 100 ile 1000 arasındaki uzaklıklar eşittir; çok farklı büyüklükteki sayıları aynı resme sığar. İki özel durum her yerde karşımıza çıkar:

- $y = Ca^x$ ise $\log y = \log C + x \log a$ olur; üstel bir fonksiyon yarı logaritmik grafikte bir doğrudur.
- $y = Cx^p$ ise $\log y = \log C + p \log x$ olur; bir kuvvet fonksiyonu log-log grafikte eğimi p olan bir doğrudur. Örneğin bir yöntemin hatası h^2 ile orantılı azalıyorsa log-log grafikte eğimi 2 olan bir doğru görürüz.

Matplotlib'de `ax.semilogy(x, y)` yarı logaritmik, `ax.loglog(x, y)` log-log grafik çizer; var olan bir Axes'in ölçeği `ax.set_yscale("log")` ile de değiştirilebilir. Sıfırın ve negatif sayıların logaritması olmadığından bu değerler logaritmik eksende gösterilemez. Matplotlib onları eksenin çok altına yerleştirir; bu

 CC BY-NC-SA 4.0

yüzden böyle bir noktaya giden çizgi dik bir şekilde aşağı inip grafikten çıkar. `ax.set_yscale("log", nonpositive="mask")` yazılırsa bu noktalar atlanır ve çizgi orada kesilir.

Örnek 8.3 (Sinüsün Taylor Polinomları ve Hataları). $\sin x$ fonksiyonunu ve 0 etrafındaki T_1, T_3, T_5, T_7 Taylor polinomlarını $[-2\pi, 2\pi]$ aralığında bir Axes'te, $|\sin x - T_n(x)|$ hatalarını ise yanındaki ikinci Axes'te yarı logaritmik ölçekte çiziniz.

Çözüm.
$\sin x$'in 0 etrafındaki Taylor polinomu yalnız tek dereceli terimlerden oluşur (bkz. Analiz 2). n tek olmak üzere

$$T_n(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \pm \frac{x^n}{n!} = \sum_{j=0}^{(n-1)/2} (-1)^j \frac{x^{2j+1}}{(2j+1)!}$$

olur. Bu toplamı bir fonksiyona yazıyor, `plt.subplots(1, 2)` ile yan yana iki Axes açıyoruz. Fonksiyondaki `k`, tek dereceleri $1, 3, \ldots, n$ sırasıyla dolaşır; $k = 2j+1$ için işaret $(-1)^{(k-1)/2} = (-1)^j$'dir.

```
import math

import numpy as np
import matplotlib.pyplot as plt


def taylor_sin(x, n):
    """sin x'in 0 etrafındaki n. dereceden Taylor polinomu."""
    total = np.zeros_like(x)
    for k in range(1, n + 1, 2):
        sign = (-1) ** ((k - 1) // 2)
        total += sign * x**k / math.factorial(k)
    return total


x = np.linspace(-2 * np.pi, 2 * np.pi, 400)
degrees = (1, 3, 5, 7)

fig, axs = plt.subplots(1, 2, figsize=(10, 4), layout="constrained")

axs[0].plot(x, np.sin(x), "k", linewidth=2.5, label=r"$\sin x$")
for n in degrees:
    axs[0].plot(x, taylor_sin(x, n), label=f"$T_{n}$")
axs[0].set_ylim(-3, 3)
axs[0].set_title("Taylor polinomları")
axs[0].legend()

for n in degrees:
    error = np.abs(np.sin(x) - taylor_sin(x, n))
    axs[1].semilogy(x, error, label=f"$n = {n}$")
axs[1].set_ylim(1e-16, 1e3)
axs[1].set_title(r"Hata $|\sin x - T_n(x)|$")
axs[1].legend()

fig.savefig("taylor_sinus.png", dpi=150)

print(type(axs).__name__, axs.shape)
for n in degrees:
    e = abs(np.sin(1.0) - taylor_sin(np.array([1.0]), n)[0])
    print(f"n = {n}: x = 1 noktasındaki hata {e:.2e}")
```

Çıktı:

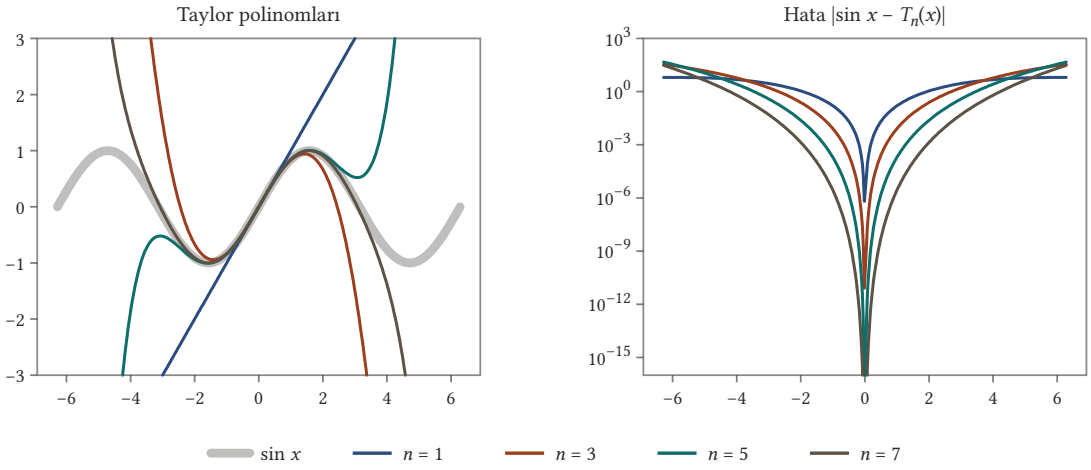
 CC BY-NC-SA 4.0

```
ndarray (2,)
n = 1: x = 1 noktasındaki hata 1.59e-01
n = 3: x = 1 noktasındaki hata 8.14e-03
n = 5: x = 1 noktasındaki hata 1.96e-04
n = 7: x = 1 noktasındaki hata 2.73e-06
```

`axs`, iki Axes tutan tek boyutlu bir NumPy dizisidir; `axs[0]` soldaki, `axs[1]` sağdaki koordinat sistemidir. Polinomlar büyük $|\,x\,|$ için hızla büyüdüğünden sol Axes'in dikey sınırlarını `set_ylim(-3, 3)` ile sabitledik; aksi hâlde Matplotlib ölçeği polinomların uçlarda 46'ya varan değerlerine göre seçer ve sinüs dümdüz görünürdü. Sağ Axes'te de alt sınırı 10^{-16}'ya sabitledik, çünkü $x = 0$ yakınında hata, çift duyarlıklı sayıların yuvarlama düzeyine iner.
Çıktıdaki hatalar her adımda sırasıyla yaklaşık 19, 42 ve 72 kat küçülür. Bu hızı Lagrange kalanlı Taylor teoremi açıklar (bkz. Analiz 2). n tek olduğundan $n+1$ çifttir. Sinüsün çift mertebeden türevleri $\pm\sin$ olduğu için T_{n+1}'e eklenen x^{n+1} teriminin katsayısı $\sin^{(n+1)}(0)/(n+1)! = 0$'dır; yani $T_n = T_{n+1}$'dir. Teoremi T_{n+1}'e uygularsak 0 ile x arasındaki bir c için

$$\sin x - T_n(x) = \frac{\sin^{(n+2)}(c)}{(n+2)!}\,x^{n+2}$$

olur. Sinüsün her türevi $\pm\sin$ ya da $\pm\cos$ olduğundan $|\sin^{(n+2)}(c)\,| \le 1$'dir; bu yüzden $x = 1$ noktasındaki hata $1/(n+2)!$ ile sınırlıdır. Bu sınırlar $1/3! \approx 0{,}167$, $1/5! \approx 0{,}0083$, $1/7! \approx 1{,}98 \cdot 10^{-4}$ ve $1/9! \approx 2{,}76 \cdot 10^{-6}$'dır ve gerçek hatalar bunların hemen altında kalır. ■



Şekil 8.2: Alt grafik örneğindeki kodun çizdiği iki Axes. Solda kalın açık renkli eğri sin x, ince eğriler T_1, T_3, T_5, T_7 Taylor polinomlarıdır; derece arttıkça polinom sinüse daha geniş bir aralıkta yapışır. Sağda aynı renklerle $|\sin x - T_n(x)|$ hataları logaritmik eksendedir: 0 yakınında hata 10^{-16} düzeyine iner, uçlarda 10^{2} düzeyine çıkar.

Yan yana grafikler, bir çizimin yanlış olduğunu fark etmek için de iyi bir araçtır. Matplotlib noktaları sırayla birleştirdiği için, fonksiyonun süreksiz olduğu bir yerde ne olduğunu kendisi bilemez.

Örnek 8.4 (Asimptotları Olan Bir Grafik). $\tan x$ fonksiyonunu $[-\pi, \pi]$ aralığında 1000 noktayla, asimptotlarda düzgün görünecek biçimde çiziniz.

Çözüm.
Önce fonksiyonu olduğu gibi çiziyor, sonra $|\,y\,| > 10$ olan değerleri `np.nan` (Not a Number, "sayı değil") ile değiştirip yeniden çiziyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-np.pi, np.pi, 1000)
y = np.tan(x)

fig, axs = plt.subplots(1, 2, figsize=(10, 4), layout="constrained")

axs[0].plot(x, y)
axs[0].set_title("Düzeltilmemiş")

y_cut = np.where(np.abs(y) > 10, np.nan, y)
axs[1].plot(x, y_cut)
axs[1].set_ylim(-10, 10)
axs[1].set_title(r"$|y| > 10$ olan noktalar NaN")

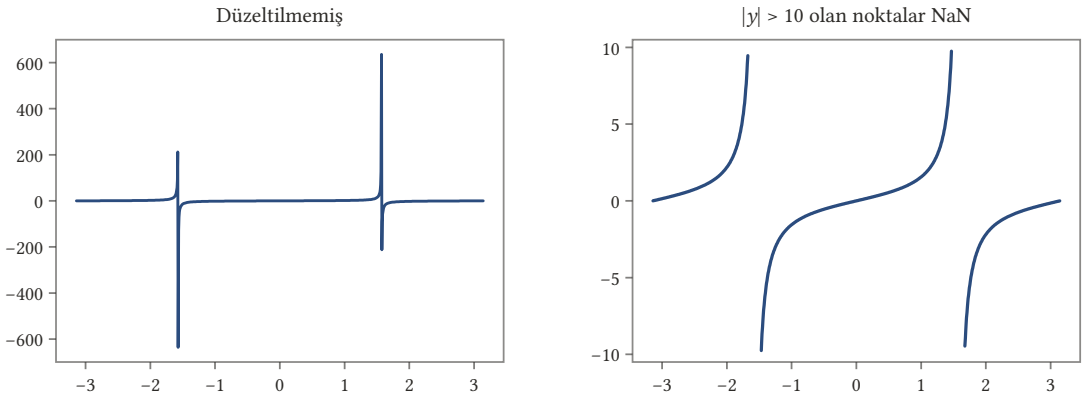
fig.savefig("tanjant.png", dpi=150)

print(f"En büyük |tan x| değeri: {np.max(np.abs(y)):.1f}")
print("NaN yapılan nokta sayısı:", np.isnan(y_cut).sum())
```

Çıktı:

```
En büyük |tan x| değeri: 636.0
NaN yapılan nokta sayısı: 64
```

Soldaki grafikte iki kusur var. Örnek noktalardan biri $\pi/2$'ye çok yakın düştüğü için $|\tan x|$ 636'ya ulaşmış ve Matplotlib dikey ölçeği bu değere göre seçmiştir; dalların biçimi kaybolmuş, her dal yatay bir çizgi gibi görünür. Ayrıca asimptotun solundaki büyük pozitif değerle sağındaki büyük negatif değer bir doğru parçasıyla birleştirilmiş, grafikte fonksiyona ait olmayan dikey çizgiler belirmiştir. Matplotlib NaN olan noktaları atlar ve çizgiyi orada keser: sağdaki grafikte 64 nokta NaN yapıldığı için dallar birbirinden ayrılır, `set_ylim(-10, 10)` de ölçeği okunur bir aralığa sabitler. ■



Şekil 8.3: tan x örneğindeki kodun çizdiği iki Axes. Solda Matplotlib, asimptotun iki yanındaki noktaları dikey çizgilerle birleştirmiş ve ölçeği ±600 düzeyine çıkarmıştır; dalların biçimi kaybolur. Sağda $|y| > 10$ olan değerler NaN yapıldığı için çizgi asimptotlarda kopar.

 CC BY-NC-SA 4.0

> ⚠ Noktalar sırayla birleştirilir
>
> `ax.plot` verilen noktaları sırayla doğru parçalarıyla birleştirir; fonksiyonun bu noktalar arasında ne yaptığını bilmez. Bu yüzden süreksizlik noktalarında dikey çizgiler belirir ve nokta sayısı azsa hızlı salınımlar kaybolur. Süreksizliklerde değerleri `np.nan` yapın, salınan fonksiyonlarda nokta sayısını artırın ve kuşkulu bir grafiği daha sık örnekleyerek yeniden çizin.

8.4 Parametrik ve Kutupsal Eğriler

Bir eğri her zaman $y = f(x)$ biçiminde verilmez; düzlemde dolanan ve kendini kesen eğriler bir parametreyle ya da kutupsal koordinatlarla daha doğal yazılır.
$x = x(t)$, $y = y(t)$, $a \le t \le b$ parametrik eğrisini çizmek için önce `t = np.linspace(a, b, n)` dizisini kurar, sonra iki koordinat dizisini `ax.plot(x(t), y(t))` ile çizeriz; `plot` için önemli olan yalnız noktaların sırasıdır. Böyle eğrilerde `ax.set_aspect("equal")` neredeyse her zaman gerekir, yoksa eğri bir yönde basık görünür.
Kutupsal bir eğri için Axes'i `projection="polar"` ile kutupsal yaparız. Böyle bir Axes'te `ax.plot(theta, r)` çağrısının ilk argümanı açı (radyan), ikincisi yarıçaptır. Bir Figure'a farklı türde Axes'ler koymak için Figure'ı önce `plt.figure()` ile boş kurar, Axes'leri `fig.add_subplot(nrows, ncols, index)` ile tek tek ekleriz; `index` soldan sağa, yukarıdan aşağı 1'den başlayarak sayılır.

Örnek 8.5 (Bir Lissajous Eğrisi ve Kardioid). $x = \sin 3t$, $y = \sin 2t$ $(0 \le t \le 2\pi)$ parametrik eğrisini ve $r = 1 + \cos\theta$ kardioidini aynı Figure'da yan yana çiziniz.

Çözüm.
Sağdaki Axes kutupsal olacağı için Figure'ı boş kuruyor, iki Axes'i `fig.add_subplot` ile ekliyoruz. Kodun sonunda kardioidin sınırladığı bölgenin alanını da yaklaşık hesaplıyoruz. Kutupsal bir bölgenin alanı

$$A = \frac{1}{2}\int_0^{2\pi} r(\theta)^2\, d\theta$$

formülüyle verilir (bkz. Analiz 2). İntegrali, θ ızgarasının sol uç noktalarındaki değerlerle kurulan bir Riemann toplamıyla yaklaşık hesaplıyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, 2 * np.pi, 400)

fig = plt.figure(figsize=(10, 4.5), layout="constrained")

# Sol: parametrik eğri, eşit ölçekli eksende
ax1 = fig.add_subplot(1, 2, 1)
ax1.plot(np.sin(3 * t), np.sin(2 * t))
ax1.set_aspect("equal")
ax1.set_title(r"$x = \sin 3t,\ y = \sin 2t$")

# Sağ: kutupsal eksen; plot(θ, r) sırasıyla
ax2 = fig.add_subplot(1, 2, 2, projection="polar")
r = 1 + np.cos(t)
ax2.plot(t, r)
ax2.set_rticks([0.5, 1, 1.5, 2])
ax2.set_rlabel_position(157.5)
ax2.set_title(r"$r = 1 + \cos\theta$")

fig.savefig("parametrik_kutupsal.png", dpi=150)

# Kardioidin alanı (1/2)∫ r^2 dθ için bir Riemann toplamı
```

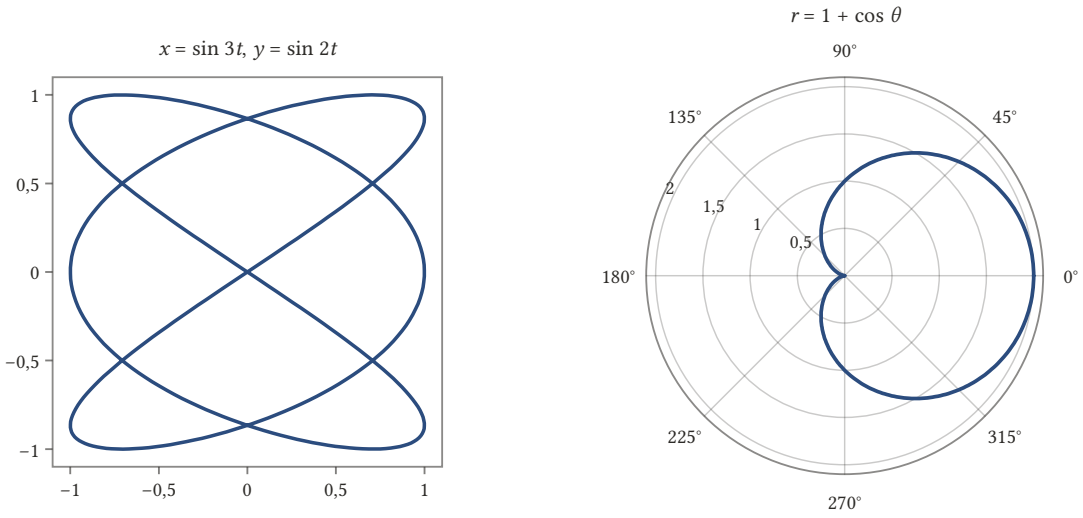
 CC BY-NC-SA 4.0

```
dtheta = t[1] - t[0]
area = 0.5 * np.sum(r[:-1] ** 2) * dtheta
print(type(ax2).__name__)
print(f"Riemann toplamı: {area:.6f}")
print(f"3π/2           : {3 * np.pi / 2:.6f}")
```

Çıktı:

```
PolarAxes
Riemann toplamı: 4.712389
3π/2           : 4.712389
```

Kardioidin alanı $\frac{3\pi}{2} \approx 4{,}712389$'dur (bkz. Analiz 2); Riemann toplamı altı ondalık basamağa kadar bu değeri verir. Toplamı `r[:-1]` ile kurduk, çünkü `linspace` dizisinin son elemanı olan 2π, ilk eleman 0 ile aynı açıdır. `set_rticks` yarıçap çemberlerinin yerlerini, `set_rlabel_position(157.5)` ise yarıçap etiketlerinin yazılacağı açıyı derece cinsinden seçer; etiketleri eğriden uzak tutmak için 157,5°'yi seçtik. ■



Şekil 8.4: Parametrik ve kutupsal örnekteki kodun çizdiği iki Axes. Solda eşit ölçekli eksende $x = \sin 3t$, $y = \sin 2t$ Lissajous eğrisi; sağda kutupsal eksende $r = 1 + \cos\theta$ kardioidi. Kutupsal eksende açılar derece, yarıçap çemberleri 0,5 aralıklıdır.

> ⚠ **Kutupsal eksende negatif yarıçap**
>
> Kutupsal koordinatlarda $r < 0$ iken (r, θ) noktası, $(|\,r\,|, \theta + \pi)$ noktasıdır. Matplotlib'in kutupsal Axes'i bu kuralı uygulamaz: negatif yarıçaplar varsa merkeze $r = 0$'ı değil, eksenin en küçük r değerini koyar ve eğriyi biçimini bozarak çizer. Yarıçap negatif değer alabiliyorsa eğriyi $x = r\cos\theta$, $y = r\sin\theta$ dönüşümüyle sıradan bir Axes'te çizin.

8.5 Saçılım Grafikleri ve Histogramlar

Şimdiye kadar çizdiğimiz her şey, birbirine doğru parçalarıyla bağlanan noktalardı. Bazı verilerde ise noktalar arasında bir sıra yoktur; önemli olan tek tek noktaların nerede durduğu ya da sayı doğrusuna nasıl yayıldığıdır.

Tek tek noktaları `ax.scatter(x, y)` çizer. `s=` işaretin alanını (punto kare), `marker=` biçimini, `alpha=` saydamlığını, `c=` rengini verir. `c=` bir sayı dizisi olarak verilirse her nokta kendi değerine göre bir renk skalasından (`cmap=`) boyanır ve `fig.colorbar(...)` bu skalayı grafiğin yanına çizer. Bütün noktaları aynı biçimde çizmek için `ax.plot(x, y, "o")` de yeterlidir; `scatter`, boyut ya da renk noktadan noktaya değiştiğinde gereklidir.

Bir saçılım grafiğinin açığa çıkardığı bir düzeni görelim. Rastgele bir matrisin özdeğerleri nerede durur? Tek tek bakıldığında düzensiz görünen bu karmaşık sayılar, büyük matrislerde şaşırtıcı bir düzene uyar ve bunu ancak çizince fark ederiz. Özdeğerleri, NumPy ile Lineer Cebir bölümündeki `np.linalg.eig`'in yalnız özdeğerleri döndüren biçimi olan `np.linalg.eigvals` ile hesaplıyoruz. Rastgele sayıları `np.random.default_rng` üreteci verir; onu Rastgele Sayılar ve Monte Carlo Yöntemleri bölümünde ayrıntılı göreceğiz. Üretece sabit bir tohum (seed) verdiğimiz için kod her çalıştırmada aynı sayıları üretir.

Örnek 8.6 (Rastgele Bir Matrisin Özdeğerleri). Girdileri bağımsız standart normal sayıların $\sqrt{300}$'e bölümü olan 300×300'lük bir A matrisinin özdeğerlerini, reel özdeğerler ayrı bir işaretle gösterilmek üzere karmaşık düzlemde bir saçılım grafiğiyle çiziniz.

Çözüm.
Reel özdeğerleri, sanal kısmının mutlak değeri çok küçük olanlar olarak bir boolean maskeyle ayırıyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(2024)
n = 300
A = rng.standard_normal((n, n)) / np.sqrt(n)
lam = np.linalg.eigvals(A)
is_real = np.abs(lam.imag) < 1e-12

t = np.linspace(0, 2 * np.pi, 400)
fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(np.cos(t), np.sin(t), "k--", linewidth=1, label="birim çember")
ax.scatter(lam[~is_real].real, lam[~is_real].imag, s=12,
           label="reel olmayan")
ax.scatter(lam[is_real].real, lam[is_real].imag, s=30, marker="s",
           label="reel")
ax.set_aspect("equal")
ax.set_xlim(-1.5, 1.5)
ax.set_ylim(-1.5, 1.5)
ax.set_xlabel(r"Re $\lambda$")
ax.set_ylabel(r"Im $\lambda$")
ax.legend(loc="upper right")
fig.savefig("ozdegerler.png", dpi=150)

print("Özdeğer sayısı:", lam.size)
print("Reel özdeğer sayısı:", is_real.sum())
print(f"En büyük |λ|: {np.abs(lam).max():.4f}")
print(f"|λ| ≤ 1 olanların oranı: {np.mean(np.abs(lam) <= 1):.3f}")
```

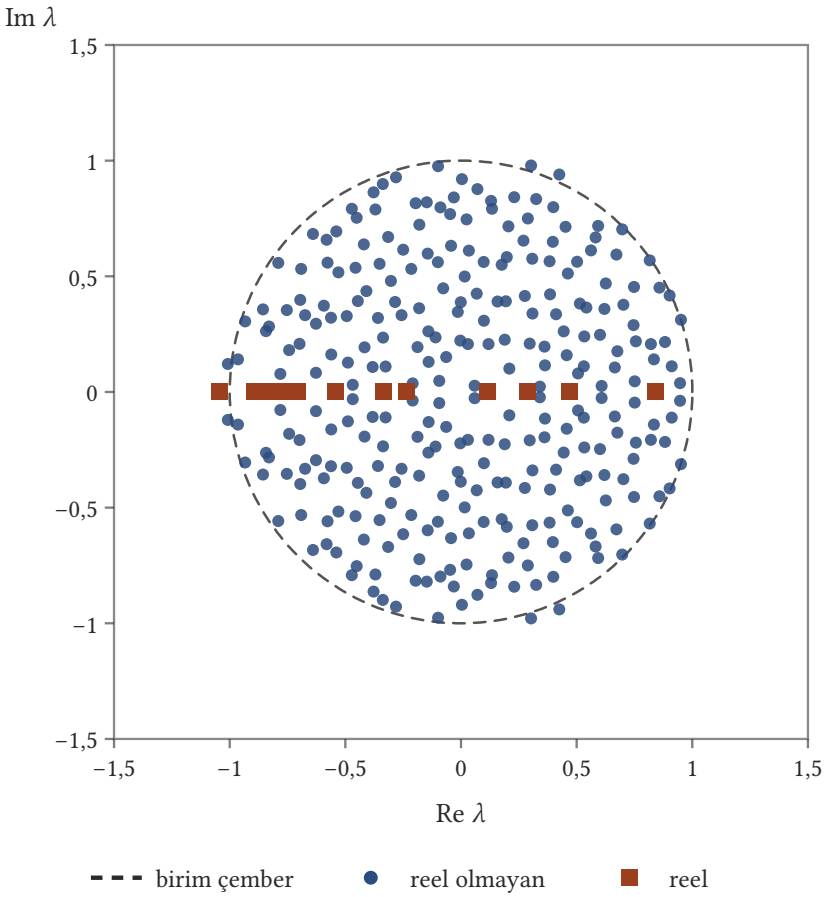
Çıktı:

```
Özdeğer sayısı: 300
Reel özdeğer sayısı: 12
En büyük |λ|: 1.0453
|λ| ≤ 1 olanların oranı: 0.970
```

`is_real` bir boolean dizidir; `lam[is_real]` reel özdeğerleri, `lam[~is_real]` geri kalanları seçer (`~` maskeyi tersine çevirir). İki ayrı `scatter` çağrısı lejantta iki ayrı girdi verir. Resim belirgin bir düzen gösteriyor: özdeğerler birim diske neredeyse düzgün dağılmış, %97'si birim çemberin içinde kalmıştır.

 CC BY-NC-SA 4.0

A reel bir matris olduğundan karakteristik polinomunun katsayıları reeldir; bu yüzden reel olmayan özdeğerler eşlenik çiftler hâlinde gelir ve resim reel eksene göre simetriktir. Gördüğümüz düzen, rastgele matris kuramının **dairesel yasa** (circular law) adlı sonucunun $n = 300$ için bir görüntüsüdür: $n \to \infty$ iken böyle bir matrisin özdeğerleri birim diskte düzgün dağılıma yaklaşır. ■



Şekil 8.5: Saçılım örneğindeki kodun çizdiği grafik: girdileri bağımsız standart normal sayıların √300'e bölümü olan 300 × 300'lük bir matrisin 300 özdeğeri. Daireler reel olmayan, kareler 12 reel özdeğerdir; noktalar birim diske neredeyse düzgün dağılır.

Noktaların tek tek konumu yerine sayı doğrusuna nasıl yayıldıklarını görmek istediğimizde histogram çizeriz.

Tanım 8.3 (Histogram). $x_1, \ldots, x_N$ gerçel sayıları ve $b_0 < b_1 < \cdots < b_m$ kutu sınırları, verilerin hepsi $[b_0, b_m]$ aralığında kalacak biçimde verilsin. j-inci kutuya, yani $[b_{j-1}, b_j)$ aralığına düşen veri sayısı n_j olsun; son kutu $[b_{m-1}, b_m]$ kapalı aralığıdır. Her kutunun üstüne yüksekliği n_j olan bir dikdörtgen dikilerek elde edilen grafiğe **histogram** denir. Yükseklik olarak

$$\frac{n_j}{N\,(b_j - b_{j-1})}$$

alınırsa dikdörtgenlerin alanları toplamı 1 olur; bu grafiğe **yoğunluk ölçeğinde histogram** denir.

Yani histogram, verinin sayı doğrusuna nasıl yayıldığının resmidir. Yoğunluk ölçeğinde bir dikdörtgenin alanı, verinin o kutuya düşen oranıdır; bu yüzden böyle bir histogram bir olasılık yoğunluk fonksiyonuyla aynı eksende karşılaştırılabilir (bkz. Olasılık Teorisi). Matplotlib'de `counts, edges, patches = ax.hist(data, bins=40, density=True)` çağrısı histogramı çizer; kutu yüksekliklerini (`counts`), kutu

 CC BY-NC-SA 4.0

sınırlarını (`edges`) ve çizilen dikdörtgenleri (`patches`) döndürür. `bins` bir tam sayıysa veri aralığı o kadar eşit kutuya bölünür, bir diziyse kutu sınırları doğrudan o dizi olur.

Örnek 8.7 (Simetrik Rastgele Bir Matrisin Özdeğer Histogramı). G, girdileri bağımsız standart normal sayılar olan 1000×1000'lik bir matris olsun. $H = (G + G^T)/\sqrt{2000}$ simetrik matrisinin özdeğerlerinin 40 kutulu, yoğunluk ölçeğinde histogramını

$$\rho(x) = \frac{\sqrt{4 - x^2}}{2\pi}, \qquad -2 \le x \le 2$$

yarım çember yoğunluğuyla aynı eksende çiziniz.

Çözüm.
H simetrik olduğundan özdeğerleri reeldir. Onları `np.linalg.eigh`'in yalnız özdeğerleri döndüren biçimi olan `np.linalg.eigvalsh` ile hesaplıyoruz; bu fonksiyon özdeğerleri artan sırada verir.

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(7)
n = 1000
G = rng.standard_normal((n, n))
H = (G + G.T) / np.sqrt(2 * n)          # simetrik rastgele matris
lam = np.linalg.eigvalsh(H)             # artan sırada reel özdeğerler

x = np.linspace(-2, 2, 400)
semicircle = np.sqrt(4 - x**2) / (2 * np.pi)

fig, ax = plt.subplots(figsize=(6, 4))
counts, edges, _ = ax.hist(lam, bins=40, density=True, alpha=0.6,
                           label="özdeğerler")
ax.plot(x, semicircle, "k", linewidth=2, label="yarım çember")
ax.set_xlabel(r"$\lambda$")
ax.set_ylim(0, 0.45)
ax.legend()
fig.savefig("yarim_cember.png", dpi=150)

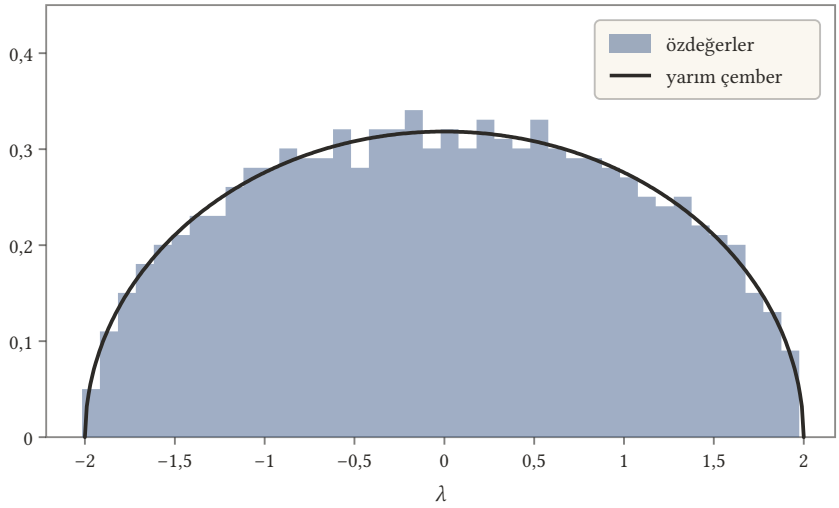
width = edges[1] - edges[0]
print(f"En küçük ve en büyük özdeğer: {lam[0]:.3f}, {lam[-1]:.3f}")
print(f"Kutu genişliği: {width:.4f}")
print(f"Dikdörtgenlerin alanları toplamı: {np.sum(counts * width):.6f}")
print(f"En yüksek kutu: {counts.max():.3f}, "
      f"yarım çemberin tepesi: {1 / np.pi:.3f}")
```

Çıktı:

```
En küçük ve en büyük özdeğer: -2.016, 1.975
Kutu genişliği: 0.0998
Dikdörtgenlerin alanları toplamı: 1.000000
En yüksek kutu: 0.341, yarım çemberin tepesi: 0.318
```

`ax.hist` histogramı çizerken kutu yüksekliklerini de döndürdü. Kutular eşit genişlikte olduğundan alanların toplamı `np.sum(counts * width)` ile bulunur ve çıktıdaki gibi 1'dir. Özdeğerler $[-2, 2]$ aralığına yayılmış ve histogram yarım çemberi yakından izliyor. Bu, Wigner'in **yarım çember yasası** (semicircle law) denen sonucunun bir görüntüsüdür: $n \to \infty$ iken böyle matrislerin özdeğer dağılımı ρ yoğunluğuna yaklaşır. ρ gerçekten bir olasılık yoğunluğudur: $\sqrt{4 - x^2}$ eğrisinin altı yarıçapı 2 olan yarım dairedir, alanı 2π'dir ve 2π'ye bölününce 1 kalır. Yoğunluğun tepe değeri $\rho(0) = 1/\pi \approx 0{,}318$'dir; en yüksek kutunun $0{,}341$ çıkması rastgele dalgalanmadır. ■

 CC BY-NC-SA 4.0



Şekil 8.6: Histogram örneğindeki kodun çizdiği grafik: 1000 × 1000'lik simetrik rastgele bir matrisin özdeğerleri 40 kutuda, yoğunluk ölçeğinde. Siyah eğri $\sqrt{(4-x^2)}/(2\pi)$ yarım çember yoğunluğudur; dikdörtgenler onu yakından izler.

8.6 Kontur Çizimleri

İki değişkenli bir $f(x, y)$ fonksiyonunu düzlemde görmenin en yaygın yolu, haritalardaki eş yükselti eğrileri gibi, f'nin sabit bir değer aldığı eğrileri çizmektir. Bu eğriler f'nin seviye kümeleridir (bkz. Diferansiyel Geometri) ve Matplotlib'de onları `contour` çizer. Kontur çizmek için f'nin değerlerine dikdörtgen bir ızgaranın her noktasında ihtiyaç vardır; ızgarayı NumPy'ın `meshgrid` fonksiyonu kurar.

Tanım 8.4 (Izgara Dizileri). x, m elemanlı ve y, n elemanlı iki dizi olsun. `X, Y = np.meshgrid(x, y)` çağrısı $n \times m$ biçimli iki dizi döndürür:

$$X_{ij} = x_j, \qquad Y_{ij} = y_i.$$

(X_{ij}, Y_{ij}) çiftleri, x ve y değerlerinden kurulabilecek bütün noktaları, yani dikdörtgen bir ızgarayı dolaşır.

Yani `X`'in her satırı x'in bir kopyası, `Y`'nin her sütunu y'nin bir kopyasıdır. Satır indisi y'yi, sütun indisi x'i sayar; matris kâğıda yazıldığında sütunların soldan sağa dizilmesi x ekseniyle uyumludur. Küçük bir örnek:

```
import numpy as np

x = np.array([0, 1, 2])
y = np.array([10, 20])
X, Y = np.meshgrid(x, y)
print(X)
print(Y)
print(X + Y)
```

Çıktı:

```
[[0 1 2]
 [0 1 2]]
[[10 10 10]
 [20 20 20]]
[[10 11 12]
 [20 21 22]]
```

 CC BY-NC-SA 4.0

Bu iki diziyle `Z = f(X, Y)` yazmak, f'yi ızgaranın her noktasında tek bir vektörel işlemle hesaplar. Aynı sonuca Vektörizasyon ve Broadcasting bölümündeki kurallarla `x[np.newaxis, :] + y[:, np.newaxis]` biçiminde de ulaşılabilir; `meshgrid` bunu açık ve okunur kılar.
Kontur komutlarının `levels` argümanı bir tam sayıysa Matplotlib kabaca o sayıda, yuvarlak değerli seviyeleri kendisi seçer (seviye sayısı istenenden biraz farklı çıkabilir), bir diziyse tam olarak o seviyeleri kullanır. `contour` çizgileri, `contourf` ise ardışık iki seviye arasındaki bantları boyar.

Dört adımda kontur grafiği

1. Eksenlerdeki örnek noktaları kur: `x = np.linspace(a, b, m)` ve `y = np.linspace(c, d, n)`.
2. Izgarayı kur: `X, Y = np.meshgrid(x, y)`.
3. Fonksiyonu ızgaranın her noktasında hesapla: `Z = f(X, Y)`.
4. Seviye eğrilerini `cs = ax.contour(X, Y, Z, levels=...)` ile ya da boyalı bantları `cf = ax.contourf(X, Y, Z, levels=...)` ile çiz; `ax.clabel(cs)` seviyelerin değerlerini çizgilere yazar, `fig.colorbar(cf, ax=ax)` renk skalasını ekler.

Örnek 8.8 (Bir Tepe ve Bir Çukurun Kontur Grafikleri). $f(x,y) = x\,e^{-x^2-y^2}$ fonksiyonunun $[-2,2]\times[-2,2]$ karesindeki $-0{,}45;\ -0{,}35;\ \ldots;\ 0{,}45$ seviyelerini önce çizgilerle, sonra boyanmış bantlarla çiziniz.

Çözüm.
Grafiği yorumlayabilmek için önce kritik noktaları bulalım:

$$f_x = (1-2x^2)\,e^{-x^2-y^2}, \qquad f_y = -2xy\,e^{-x^2-y^2}.$$

$f_x = 0$ denkleminden $x = \pm 1/\sqrt{2}$ bulunur; bu durumda $f_y = 0$ denklemi $y = 0$ verir. f sonsuzda 0'a gider ve hem pozitif hem negatif değerler alır; bu yüzden en büyük değerini $(1/\sqrt{2},0)$, en küçük değerini $(-1/\sqrt{2},0)$ noktasında alır:

$$f\left(\pm\frac{1}{\sqrt{2}},0\right) = \pm\frac{e^{-1/2}}{\sqrt{2}} = \pm\frac{1}{\sqrt{2e}} \approx \pm 0{,}4289.$$

Kodun sonunda ızgaradaki en büyük değeri bu sayıyla karşılaştırıyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2, 2, 200)
y = np.linspace(-2, 2, 200)
X, Y = np.meshgrid(x, y)
Z = X * np.exp(-X**2 - Y**2)
levels = np.linspace(-0.45, 0.45, 10)

fig, axs = plt.subplots(1, 2, figsize=(10, 4.2), layout="constrained")

cs = axs[0].contour(X, Y, Z, levels=levels, colors="k")
axs[0].clabel(cs, fontsize=8)
axs[0].set_title("contour")

cf = axs[1].contourf(X, Y, Z, levels=levels, cmap="RdBu_r")
fig.colorbar(cf, ax=axs[1])
axs[1].set_title("contourf")

for ax in axs:
    ax.set_aspect("equal")
fig.savefig("kontur.png", dpi=150)

i, j = np.unravel_index(np.argmax(Z), Z.shape)
print("Seviyeler:", levels.round(2))
```

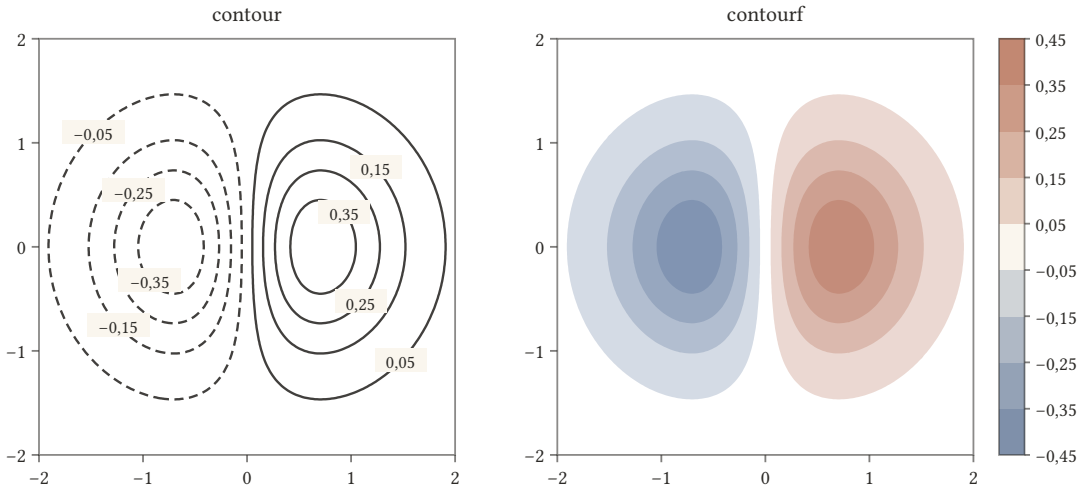
 CC BY-NC-SA 4.0

```
print(f"Izgaradaki en büyük değer: {Z[i, j]:.4f}, "
      f"(x, y) = ({X[i, j]:.3f}, {Y[i, j]:.3f})")
print(f"Gerçek maksimum: {1 / np.sqrt(2 * np.e):.4f}, "
      f"(x, y) = ({1 / np.sqrt(2):.3f}, 0)")
```

Çıktı:

```
Seviyeler: [-0.45 -0.35 -0.25 -0.15 -0.05  0.05  0.15  0.25  0.35  0.45]
Izgaradaki en büyük değer: 0.4288, (x, y) = (0.714, -0.010)
Gerçek maksimum: 0.4289, (x, y) = (0.707, 0)
```

`np.argmax(Z)` iki boyutlu diziyi düzleştirip en büyük elemanın sıra numarasını verir; `np.unravel_index` bu sıra numarasını `(satır, sütun)` indis çiftine çevirir. Izgara aralığı $4/199 \approx 0{,}02$ olduğundan ızgaradaki en büyük değer gerçek maksimumdan ancak dördüncü ondalık basamakta ayrılır ve maksimum noktasına en yakın ızgara noktasında alınır. Soldaki grafikte negatif seviyeler kesiklidir: `colors="k"` ile tek renk seçildiğinde Matplotlib negatif seviyeleri kendiliğinden kesikli çizer. En büyük değer $0{,}4289 < 0{,}45$ olduğundan $\pm 0{,}45$ seviyeleri grafikte hiç görünmez; renk çubuğu yine de bütün bantları gösterir. ■



Şekil 8.7: Kontur örneğindeki kodun çizdiği iki Axes: $z = xe^{-x^2-y^2}$ fonksiyonunun −0,45, −0,35, ..., 0,45 seviyeleri. Solda `contour` çizgileri (negatif seviyeler kesikli, değerler çizginin üstünde), sağda `contourf` ile boyanmış bantlar ve renk çubuğu. Sağdaki tepe (0,707; 0) noktasında maksimum, soldaki çukur (−0,707; 0) noktasında minimumdur.

Kontur çizimi, kapalı biçimde verilmiş bir $F(x, y) = 0$ eğrisini çizmenin de en kolay yoludur: tek seviyesi 0 olan bir kontur grafiği çizmek yeter. Çizilen eğrinin noktalarına da ulaşabiliriz: `cs.get_paths()` her seviye için bir yol (`Path`) nesnesi döndürür, bu nesnenin `vertices` özelliği de noktaları $N \times 2$ biçimli bir dizi olarak verir.

Örnek 8.9 (Descartes Yaprağı). $x^3 + y^3 = 3xy$ eğrisini (Descartes yaprağı) kontur çizimiyle çizerek eğrinin döngüsü üzerinde $x + y$ toplamını en büyük yapan noktayı bulunuz.

Çözüm.
$F(x, y) = x^3 + y^3 - 3xy$ dersek eğri, F'nin 0 seviye kümesidir. Döngü birinci bölgededir; döngünün noktalarını, kontur çizgisinin $x > 0$ ve $y > 0$ olan noktaları olarak bir maskeyle seçiyoruz.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.linspace(-3, 3, 601)
X, Y = np.meshgrid(x, x)
F = X**3 + Y**3 - 3 * X * Y

fig, ax = plt.subplots(figsize=(5, 5))
cs = ax.contour(X, Y, F, levels=[0], colors="C0")
ax.plot([-3, 3], [2, -4], "k:", linewidth=1)    # asimptot y = -x - 1
ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)
ax.set_aspect("equal")
fig.savefig("descartes_yapragi.png", dpi=150)

path = cs.get_paths()[0]
pts = path.vertices
print("Eğri üzerindeki nokta sayısı:", len(pts))
loop = pts[(pts[:, 0] > 0) & (pts[:, 1] > 0)]
k = np.argmax(loop.sum(axis=1))
print(f"Döngünün x + y'yi en büyük yapan noktası: "
      f"({loop[k, 0]:.3f}, {loop[k, 1]:.3f})")
residual = np.abs(pts[:, 0]**3 + pts[:, 1]**3 - 3 * pts[:, 0] * pts[:, 1])
print(f"En büyük |F| değeri: {residual.max():.1e}")
```

Çıktı:

```
Eğri üzerindeki nokta sayısı: 1642
Döngünün x + y'yi en büyük yapan noktası: (1.500, 1.500)
En büyük |F| değeri: 2.1e-04
```

Bulduğumuz noktayı Lagrange çarpanları yöntemiyle doğrulayalım (bkz. Analiz 4). $x+y$'nin $F=0$ üzerindeki ekstremumlarında ∇F, $(1,1)$ vektörüne paraleldir; yani $F_x = F_y$ olur:

$$3x^2 - 3y = 3y^2 - 3x \Leftrightarrow (x-y)(x+y+1) = 0.$$

Döngü üzerinde $x, y > 0$ olduğundan $x+y+1 > 0$'dır, dolayısıyla $x = y$ olur. Bunu $F=0$'da yerine yazınca $2x^3 = 3x^2$, yani $x = 3/2$ bulunur. Kodun bulduğu $(1{,}5;\,1{,}5)$ noktası budur. Kontur noktaları ızgara hücrelerinde doğrusal interpolasyonla hesaplandığından F bu noktalarda tam sıfır değildir; çıktıdaki en büyük $|F|$ değeri $2{,}1 \cdot 10^{-4}$'tür. Grafiğe eğrinin $y = -x - 1$ asimptotunu da noktalı çizgiyle ekledik. ■

8.7 Üç Boyutlu Yüzeyler

İki değişkenli bir fonksiyonun grafiği uzayda bir yüzeydir (bkz. Analiz 4). Matplotlib bir Axes'i `projection="3d"` ile üç boyutlu yapar. Böyle bir Axes'te `plot_surface(X, Y, Z)` yüzeyi, `plot_wireframe(X, Y, Z)` yalnız tel kafesini, `plot(x, y, z)` ise uzayda bir eğriyi çizer. Bakış doğrultusunu `view_init(elev=..., azim=...)` seçer: `elev` bakış doğrultusunun xy-düzlemiyle yaptığı açı, `azim` ise bu doğrultunun xy-düzlemindeki izdüşümünün x ekseniyle yaptığı açıdır; ikisi de derece cinsindendir. `plt.show()` ile açılan pencerede üç boyutlu bir Axes fareyle döndürülebilir.

Örnek 8.10 (Bir Tepe ve Bir Çukurun Yüzeyi). Kontur örneğindeki $f(x,y) = x\,e^{-x^2-y^2}$ fonksiyonunun $[-2,2] \times [-2,2]$ üzerindeki grafiğini 21×21'lik bir ızgarayla yüzey olarak çiziniz.

Çözüm.
Kontur örneğindeki ızgarayı daha seyrek kuruyor, Axes'i `projection="3d"` ile ekliyoruz.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.linspace(-2, 2, 21)
y = np.linspace(-2, 2, 21)
X, Y = np.meshgrid(x, y)
Z = X * np.exp(-X**2 - Y**2)

fig = plt.figure(figsize=(6, 5))
ax = fig.add_subplot(projection="3d")
ax.plot_surface(X, Y, Z, cmap="RdBu_r", edgecolor="k", linewidth=0.3)
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_zlabel("$z$")
ax.view_init(elev=25, azim=-60)
fig.savefig("yuzey.png", dpi=150)

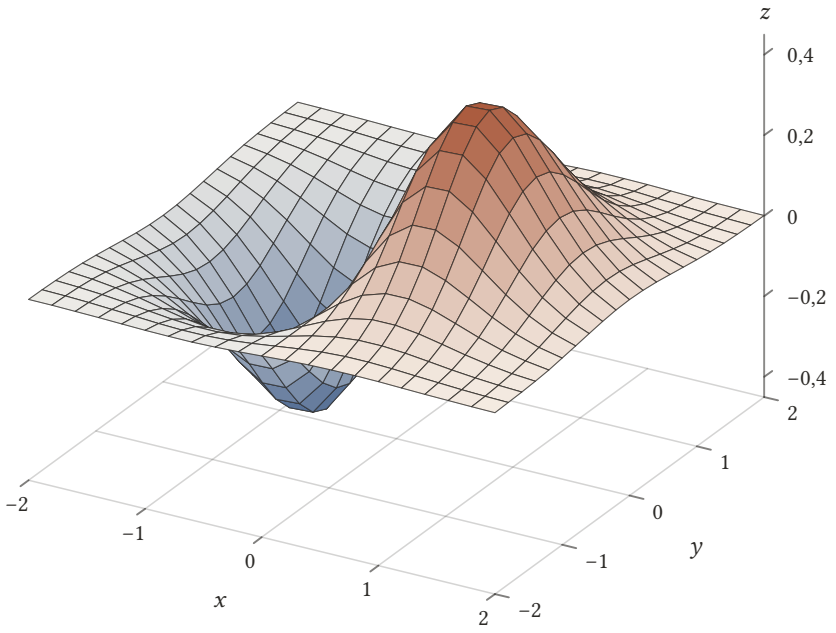
print(type(ax).__name__)
print("Izgara:", Z.shape, "->", (Z.shape[0] - 1) * (Z.shape[1] - 1),
      "dörtgen yüzey parçası")
print(f"z aralığı: [{Z.min():.4f}, {Z.max():.4f}]")
```

Çıktı:

```
Axes3D
Izgara: (21, 21) -> 400 dörtgen yüzey parçası
z aralığı: [-0.4218, 0.4218]
```

`plot_surface` ızgaranın her hücresini bir dörtgenle doldurur; 21×21 noktalı ızgarada $20 \cdot 20 = 400$ dörtgen vardır. `cmap="RdBu_r"` dörtgenleri yüksekliklerine göre pozitifte kırmızı, negatifte mavi tonlarla boyar; `edgecolor` ve `linewidth` dörtgenlerin kenarlarını çizer. Izgara aralığı $0{,}2$ olduğundan ızgara $x = 1/\sqrt{2}$ noktasından geçmez: ızgaradaki en büyük değer $0{,}4218$, gerçek maksimum ise $0{,}4289$'dur ve tepe şekilde hafifçe düzleşmiş görünür. Daha sık bir ızgara daha düzgün bir yüzey verir ama çizimi yavaşlatır. ■

 CC BY-NC-SA 4.0



Şekil 8.8: Yüzey örneğindeki kodun çizdiği grafik: $z = xe^{-x^2-y^2}$ yüzeyi 21 × 21'lik ızgaranın 400 dörtgeniyle, yükseklik pozitifken kırmızı, negatifken mavi tonlarla. Ön sağda tepe, arka solda çukur görünür; kaba ızgara yüzünden tepe hafifçe düzleşmiştir.

8.8 Grafikleri Kaydetmek

Bir grafiği bir rapora, sunuma ya da LaTeX belgesine koymak için dosyaya kaydederiz; bunun tek komutu `fig.savefig`'tir. Dosya biçimi uzantıdan anlaşılır:

- `.png` bir piksel resmidir (raster). Piksel boyutu, Figure'ın inç cinsinden boyutu ile `dpi` (inç başına nokta) değerinin çarpımıdır.
- `.pdf` ve `.svg` vektör biçimleridir: çizgileri ve yazıları geometrik nesneler olarak saklar, istenildiği kadar büyütülünce bile bozulmaz. LaTeX belgeleri için PDF en uygun seçimdir.

`bbox_inches="tight"` resmin kenarlarındaki boşluğu kırpar, `transparent=True` arka planı saydam yapar.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 1, 100)
fig, ax = plt.subplots(figsize=(4, 3))
ax.plot(x, x**2)

for dpi in (100, 200):
    name = f"parabol_{dpi}.png"
    fig.savefig(name, dpi=dpi)
    image = plt.imread(name)
    print(f"dpi = {dpi}: {image.shape[1]} x {image.shape[0]} piksel")

fig.savefig("parabol.pdf")
fig.savefig("parabol.svg")
print("Açık figür sayısı:", len(plt.get_fignums()))
plt.close(fig)
print("Kapattıktan sonra:", len(plt.get_fignums()))
```

Çıktı:

 CC BY-NC-SA 4.0

```
dpi = 100: 400 x 300 piksel
dpi = 200: 800 x 600 piksel
Açık figür sayısı: 1
Kapattıktan sonra: 0
```

4×3 inçlik Figure, 100 dpi ile 400×300, 200 dpi ile 800×600 piksel olarak kaydedilmiştir. `plt.imread` bir PNG dosyasını (yükseklik, genişlik, renk kanalı) biçimli bir diziye okur; genişliğin `shape[1]`, yüksekliğin `shape[0]` olmasının nedeni budur. `plt.get_fignums()` açık Figure'ların numaralarını listeler. `pyplot`, `plt.subplots` ile oluşturulan her Figure'ı biz kapatana kadar bellekte tutar; çok sayıda grafik üreten döngülerde her Figure kaydedildikten sonra `plt.close(fig)` ile kapatılmalıdır. Betiklerde `plt.show()` penceresi kapanınca `pyplot` o Figure'ı da kapatır; bu yüzden önce `savefig`, sonra `show` çağrılır.

8.9 Alıştırmalar

Alıştırma 8.1 (Az Noktayla Çizilen Bir Salınım). $\sin 20x$ fonksiyonunu $[0, \pi]$ aralığında önce 21, sonra 1000 eşit aralıklı noktayla çiziniz. 21 noktalı grafiğin neden yanlış olduğunu, çizilen $|\,y\,|$ değerlerinin en büyüğüne bakarak açıklayınız.

Çözüm.
İki grafiği ortak dikey eksenli (`sharey=True`) iki Axes'te çiziyor, her birinde çizilen $|\,y\,|$ değerlerinin en büyüğünü yazdırıyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True,
                        layout="constrained")
for ax, n in zip(axs, (21, 1000)):
    x = np.linspace(0, np.pi, n)
    y = np.sin(20 * x)
    ax.plot(x, y)
    ax.set_title(f"{n} nokta")
    print(f"n = {n:4d}: en büyük |y| = {np.max(np.abs(y)):.2e}")
fig.savefig("ornekleme.png", dpi=150)
```

Çıktı:

```
n =   21: en büyük |y| = 5.88e-15
n = 1000: en büyük |y| = 1.00e+00
```

21 noktalı ızgarada örnek noktaları $x_k = k\pi/20$'dir ($k = 0, 1, \ldots, 20$) ve bu noktalarda $\sin 20x_k = \sin k\pi = 0$ olur. Yani örnek noktaların hepsi fonksiyonun sıfırlarına denk gelmiştir; çıktıdaki $5{,}88 \cdot 10^{-15}$ yalnız yuvarlama hatasıdır ve grafik yatay bir doğruya dönüşür. Fonksiyonun periyodu $2\pi/20 = \pi/10$ olduğundan $[0, \pi]$ aralığında 10 tam periyot vardır; bir periyodu düzgün göstermek için periyot başına en az birkaç düzine nokta gerekir. 1000 noktalı ızgarada her periyoda yaklaşık 100 nokta düşer ve en büyük değer beklendiği gibi 1'e çok yakındır. ■

Alıştırma 8.2 (İki Eğri Arasındaki Bölge). $[0, 2\pi]$ aralığında $\sin x \geq \cos x$ olan bölgeyi `fill_between` ile tarayan bir grafik çiziniz ve aynı boolean maskeyle kurulan bir Riemann toplamıyla bu bölgenin alanını yaklaşık hesaplayınız.

Çözüm.
$\sin x \geq \cos x$ eşitsizliği $[0, 2\pi]$ aralığında tam olarak $[\pi/4, 5\pi/4]$ üzerinde sağlanır. Bölgenin alanı

$$\int_{\pi/4}^{5\pi/4} (\sin x - \cos x)\, dx = \Big[-\cos x - \sin x \Big]_{\pi/4}^{5\pi/4} = \sqrt{2} - \left(-\sqrt{2}\right) = 2\sqrt{2}$$

olur. `fill_between` çağrısına `where=s >= c` maskesini veriyor, Riemann toplamını aynı maskeyle seçilen noktalar üzerinden kuruyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 2001)
s, c = np.sin(x), np.cos(x)

fig, ax = plt.subplots()
ax.plot(x, s, label=r"$\sin x$")
ax.plot(x, c, label=r"$\cos x$")
ax.fill_between(x, s, c, where=s >= c, alpha=0.3,
                label=r"$\sin x \geq \cos x$")
ax.legend()
fig.savefig("sin_cos_alan.png", dpi=150)

dx = x[1] - x[0]
inside = s >= c
area = np.sum((s - c)[inside]) * dx
print(f"Riemann toplamı: {area:.6f}")
print(f"2√2           : {2 * np.sqrt(2):.6f}")
```

Çıktı:

```
Riemann toplamı: 2.828425
2√2           : 2.828427
```

Riemann toplamı $2\sqrt{2} \approx 2{,}828427$ değerinden yalnız $2 \cdot 10^{-6}$ kadar ayrılır. `where=` argümanı aynı uzunlukta bir boolean dizi bekler; `fill_between` yalnız `True` olan ardışık noktaların arasını boyar. ■

Alıştırma 8.3 (Chebyshev Polinomları Bir Izgarada). $k = 1, 2, 3, 4$ için $T_k(x) = \cos(k \arccos x)$ Chebyshev polinomlarını $[-1, 1]$ aralığında 2×2'lik bir alt grafik ızgarasında çiziniz ve her polinomun örnek noktalar arasında kaç kez işaret değiştirdiğini sayınız.

Çözüm.
2×2'lik `axs` dizisini `axs.flat` ile tek bir döngüde dolaşıyoruz; `enumerate(..., start=1)` sayacı k'yi 1'den başlatır. İki komşu örnek noktadaki değerlerin çarpımı negatifse aralarında bir işaret değişimi vardır.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1, 1, 401)
fig, axs = plt.subplots(2, 2, figsize=(8, 6), sharex=True, sharey=True,
                        layout="constrained")
print("axs.shape =", axs.shape)
for k, ax in enumerate(axs.flat, start=1):
    T = np.cos(k * np.arccos(x))
    ax.plot(x, T)
    ax.axhline(0, color="k", linewidth=0.8)
    ax.set_title(f"$T_{k}(x)$")
    sign_changes = np.sum(T[:-1] * T[1:] < 0)
    print(f"k = {k}: işaret değişimi sayısı {sign_changes}")
fig.savefig("chebyshev.png", dpi=150)
```

Çıktı:

```
axs.shape = (2, 2)
k = 1: işaret değişimi sayısı 1
k = 2: işaret değişimi sayısı 2
k = 3: işaret değişimi sayısı 3
k = 4: işaret değişimi sayısı 4
```

$T_k(x) = 0$ eşitliği $k \arccos x = (2j-1)\pi/2$ demektir; yani T_k, $(-1, 1)$ aralığında

$$x_j = \cos \frac{(2j-1)\pi}{2k}, \qquad j = 1, \ldots, k$$

noktalarında sıfır olur ve her birinde işaret değiştirir. Sayım tam olarak k işaret değişimi buldu. `sharex=True` ve `sharey=True` dört Axes'e aynı eksen sınırlarını verdiği için grafikler doğrudan karşılaştırılabilir. ■

Alıştırma 8.4 (Üstel Fonksiyon Polinomu Geçer). $n = 1, 2, \ldots, 70$ için 2^n ve n^{10} değerlerini yarı logaritmik bir grafikte çiziniz. $2^n > n^{10}$ eşitsizliği hangi n değerinden başlayarak hep sağlanır?

Çözüm.
$2^{70} \approx 1{,}2 \cdot 10^{21}$ sayısı NumPy'ın 64 bitlik tam sayılarına sığmaz ve `2**n` tam sayı dizisi sessizce taşar. Bu yüzden hesabı `2.0**n` ve `n.astype(float)` ile ondalıklı sayılarla yapıyoruz. Yarı logaritmik eksende 2^n bir doğru, n^{10} ise giderek yatıklaşan bir eğri olarak görünür.

```
import numpy as np
import matplotlib.pyplot as plt

n = np.arange(1, 71)
exp_values = 2.0**n
poly_values = n.astype(float) ** 10

fig, ax = plt.subplots()
ax.semilogy(n, exp_values, label="$2^n$")
ax.semilogy(n, poly_values, label="$n^{10}$")
ax.set_xlabel("$n$")
ax.legend()
fig.savefig("ussel_polinom.png", dpi=150)

wins = exp_values > poly_values
print("2^n > n^10 olan n değerleri:")
print(n[wins])
last = n[~wins].max()
print(f"n = {last}: 2^n = {2.0**last:.3e}, n^10 = {float(last)**10:.3e}")
```

Çıktı:

```
2^n > n^10 olan n değerleri:
[ 1 59 60 61 62 63 64 65 66 67 68 69 70]
n = 58: 2^n = 2.882e+17, n^10 = 4.308e+17
```

$n = 1$ için $2 > 1$ olur, $n = 2, \ldots, 58$ için n^{10} daha büyüktür ve $n = 59$'dan itibaren 2^n öne geçer. Grafik yalnız $n \le 70$'e kadar bakar; geçişin kalıcı olduğunu görmek için

$$g(n) = n \ln 2 - 10 \ln n$$

fonksiyonuna bakalım. $g'(n) = \ln 2 - 10/n$ türevi $n > 10/\ln 2 \approx 14{,}4$ için pozitiftir, yani g bu bölgede artandır. $g(58) \approx -0{,}40 < 0$ ve $g(59) \approx 0{,}12 > 0$ olduğundan her $n \ge 59$ için $g(n) > 0$, yani $2^n > n^{10}$ olur. Yanıt $n = 59$'dur. ■

 CC BY-NC-SA 4.0

Alıştırma 8.5 (Bir Serinin Yakınsama Hızı). $S_n = \sum_{k=1}^{n} 1/k^2$ kısmi toplamları $\pi^2/6$ değerine yakınsar. $n = 1, \ldots, 10\,000$ için $\pi^2/6 - S_n$ hatasını log-log grafikte çiziniz ve grafiğin eğimini tahmin ediniz.

Çözüm.
Bütün kısmi toplamları `np.cumsum` (birikimli toplam) tek seferde verir. Karşılaştırma için $1/n$ doğrusunu da çiziyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

N = 10_000
n = np.arange(1, N + 1)
partial = np.cumsum(1 / n**2)          # S_1, S_2, ..., S_N
error = np.pi**2 / 6 - partial

fig, ax = plt.subplots()
ax.loglog(n, error, label=r"$\pi^2/6 - S_n$")
ax.loglog(n, 1 / n, "--", label="$1/n$")
ax.set_xlabel("$n$")
ax.legend()
fig.savefig("yakinsama_hizi.png", dpi=150)

for m in (10, 100, 1000, 10000):
    e = error[m - 1]
    print(f"n = {m:5d}: hata = {e:.3e}, n * hata = {m * e:.4f}")
slope = ((np.log(error[-1]) - np.log(error[9]))
         / (np.log(N) - np.log(10)))
print(f"Eğim tahmini: {slope:.4f}")
```

Çıktı:

```
n =    10: hata = 9.517e-02, n * hata = 0.9517
n =   100: hata = 9.950e-03, n * hata = 0.9950
n =  1000: hata = 9.995e-04, n * hata = 0.9995
n = 10000: hata = 1.000e-04, n * hata = 1.0000
Eğim tahmini: -0.9928
```

İki noktadan geçen doğrunun log-log eğimi

$$\frac{\log E_{10\,000} - \log E_{10}}{\log 10\,000 - \log 10}$$

ile tahmin edilir ($E_n = \pi^2/6 - S_n$) ve $-0{,}99$ çıkar: hata kabaca $1/n$ gibi azalır. Bunu integralle karşılaştırma da gösterir. $1/x^2$ azalan olduğundan

$$\frac{\pi^2}{6} - S_n = \sum_{k=n+1}^{\infty} \frac{1}{k^2},$$

$$\frac{1}{n+1} = \int_{n+1}^{\infty} \frac{dx}{x^2} < \sum_{k=n+1}^{\infty} \frac{1}{k^2} < \int_{n}^{\infty} \frac{dx}{x^2} = \frac{1}{n}$$

olur. Çıktıdaki `n * hata` sütununun 1'e yaklaşması da bunu söyler. Hata $1/n$ hızında azaldığı için 10^{-6} mertebesinde bir hata bu yolla yaklaşık bir milyon terim ister. ■

Alıştırma 8.6 (Üç Yapraklı Gül). $r = \cos 3\theta$ $(0 \le \theta \le 2\pi)$ gülünü önce doğrudan kutupsal bir Axes'te, sonra $x = r\cos\theta$, $y = r\sin\theta$ dönüşümüyle sıradan bir Axes'te çiziniz. İki grafik neden farklıdır?

Çözüm.

 CC BY-NC-SA 4.0

İki grafiği yan yana çiziyor, kutupsal Axes'in merkezine hangi yarıçap değerini koyduğunu `get_rorigin` ile soruyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

theta = np.linspace(0, 2 * np.pi, 600)
r = np.cos(3 * theta)

fig = plt.figure(figsize=(10, 4.5), layout="constrained")
ax1 = fig.add_subplot(1, 2, 1, projection="polar")
ax1.plot(theta, r)
ax1.set_title("Kutupsal eksen, negatif r")

ax2 = fig.add_subplot(1, 2, 2)
ax2.plot(r * np.cos(theta), r * np.sin(theta))
ax2.set_aspect("equal")
ax2.set_title(r"$x = r\cos\theta,\ y = r\sin\theta$")
fig.savefig("gul.png", dpi=150)

print(f"Kutupsal eksenin merkezindeki r değeri: {ax1.get_rorigin():.2f}")
print("r < 0 olan örnek sayısı:", np.sum(r < 0), "/", r.size)
```

Çıktı:

```
Kutupsal eksenin merkezindeki r değeri: -1.10
r < 0 olan örnek sayısı: 300 / 600
```

Örneklerin yarısında $r < 0$'dır. Kutupsal koordinatlarda böyle bir (r, θ) noktası $(\,|\,r\,|, \theta + \pi)$ noktasıdır, ama Matplotlib'in kutupsal Axes'i yarıçap eksenini yaklaşık $[-1{,}1; 1{,}1]$ aralığına ölçekler ve merkeze $r = -1{,}10$ değerini koyar. Bir noktanın merkeze uzaklığı $r + 1{,}10$ olarak çizildiğinden soldaki eğri aslında $\rho = \cos 3\theta + 1{,}10$ eğrisidir: yaprakları şişkin ve merkezden geçmeyen başka bir eğri. Sağdaki grafik noktaları doğru yerlere koyar: her yaprağı merkezden geçen üç yapraklı gül. $\cos(3\theta + 3\pi) = -\cos 3\theta$, yani $r(\theta + \pi) = -r(\theta)$ olduğundan θ ile $\theta + \pi$ aynı noktayı verir; gülün tamamı $\theta \in [0, \pi]$ ile çizilir ve $[\pi, 2\pi]$ aralığı aynı eğriyi ikinci kez dolaşır. Kutupsal Axes'te kalmak istenirse `ax1.plot(np.where(r < 0, theta + np.pi, theta), np.abs(r))` çağrısı noktaları kuralına uygun olarak yerleştirir. ■

Alıştırma 8.7 (Collatz Adımlarının Saçılım Grafiği). n çiftse $n \mapsto n/2$, tekse $n \mapsto 3n + 1$ kuralıyla n'den 1'e ulaşmak için gereken adım sayısını $n = 1, \ldots, 1000$ için hesaplayıp bir saçılım grafiğiyle gösteriniz. Bu aralıkta en çok adım gerektiren n hangisidir?

Çözüm.
Adımları Kontrol Yapıları ve Fonksiyonlar bölümündeki gibi bir `while` döngüsüyle sayıyoruz. Bu hesap kolayca vektörelleştirilemez, çünkü her n'nin yolu farklı uzunluktadır; değerleri bir list comprehension ile toplayıp diziye çeviriyoruz. `int(k)` NumPy tam sayısını Python tam sayısına çevirir; döngü böylece daha hızlı çalışır ve taşma riski taşımaz. Noktaları birleştirmek anlamsız bir zikzak verir; bu yüzden `scatter` kullanıyoruz.

```
import numpy as np
import matplotlib.pyplot as plt


def collatz_steps(n):
    """n'den 1'e ulaşana kadar geçen Collatz adımı sayısı."""
    steps = 0
    while n != 1:
```

```
        n = n // 2 if n % 2 == 0 else 3 * n + 1
        steps += 1
    return steps


n = np.arange(1, 1001)
steps = np.array([collatz_steps(int(k)) for k in n])

fig, ax = plt.subplots()
ax.scatter(n, steps, s=4)
ax.set_xlabel("$n$")
ax.set_ylabel("adım sayısı")
fig.savefig("collatz.png", dpi=150)

i = np.argmax(steps)
print(f"En uzun yol: n = {n[i]}, {steps[i]} adım")
print(f"Ortalama adım sayısı: {steps.mean():.2f}")
```

Çıktı:

```
En uzun yol: n = 871, 178 adım
Ortalama adım sayısı: 59.54
```

1000'e kadar en uzun yol $n = 871$ içindir ve 178 adım sürer. Grafikte noktalar dağınık değil, bantlar hâlinde kümelenir: yakın sayıların yolları çoğu zaman birkaç adım sonra birleşir ve aynı kuyruğu paylaşır. ■

Alıştırma 8.8 (İki Zarın Toplamının Histogramı). İki zar 10 000 kez atılıyor (`rng = np.random.default_rng(42)`). Toplamların yoğunluk ölçeğinde histogramını, her kutu tek bir tam sayı içerecek biçimde çizerek göreli sıklıkları gerçek olasılıklarla karşılaştırınız.

Çözüm.
Toplam S, 2 ile 12 arasında bir tam sayıdır ve $P(S = s) = \dfrac{6 - |\,s - 7\,|}{36}$'dır. Kutu sınırlarını tam sayıların ortasına, 1,5; 2,5; …; 12,5 noktalarına koyarsak her kutu tek bir tam sayı içerir. Kutu genişliği 1 olduğundan yoğunluk ölçeğindeki yükseklik doğrudan göreli sıklıktır. `rng.integers(1, 7, ...)` çağrısı 1, …, 6 arasında tam sayılar üretir; üst sınır dahil değildir. `ax.hist`'e verilen `rwidth=0.9` çubukları kutu genişliğinin %90'ı kadar çizip aralarında boşluk bırakır; `ax.set_xticks(range(2, 13))` yatay eksendeki işaretleri $2, 3, \ldots, 12$ tam sayılarına koyar.

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(42)
N = 10_000
rolls = rng.integers(1, 7, size=(N, 2))     # her satır iki zar
s = rolls.sum(axis=1)

fig, ax = plt.subplots()
counts, edges, _ = ax.hist(s, bins=np.arange(1.5, 13.5), density=True,
                           rwidth=0.9)
ax.set_xticks(range(2, 13))
fig.savefig("zar_toplami.png", dpi=150)

print("Kutu kenarları:", edges)
print(" s  göreli sıklık  olasılık")
for value, c in zip(range(2, 13), counts):
```

 CC BY-NC-SA 4.0

```
    p = (6 - abs(value - 7)) / 36
    print(f"{value:2d}  {c:13.4f}  {p:8.4f}")
```

Çıktı:

```
Kutu kenarları: [ 1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5 11.5 12.5]
 s  göreli sıklık  olasılık
 2         0.0279    0.0278
 3         0.0575    0.0556
 4         0.0906    0.0833
 5         0.1079    0.1111
 6         0.1408    0.1389
 7         0.1645    0.1667
 8         0.1351    0.1389
 9         0.1109    0.1111
10         0.0858    0.0833
11         0.0514    0.0556
12         0.0276    0.0278
```

En büyük fark $s = 4$ için yaklaşık $0{,}007$'dir; 10 000 atışta bu büyüklükte dalgalanmalar beklenir ve atış sayısı arttıkça küçülür. Kutu sınırlarını tam sayılara koymak bir tuzaktır: `bins=10` yazsaydık sınırlar $2, 3, \ldots, 12$ olurdu ve son kutu $[11, 12]$ kapalı olduğu için 11 ile 12 aynı kutuya düşerdi. ■

Alıştırma 8.9 (Himmelblau Fonksiyonunun Minimumları). Himmelblau fonksiyonu

$$F(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

olarak tanımlanır. Bu fonksiyonun $[-5, 5] \times [-5, 5]$ karesindeki kontur grafiğini logaritmik aralıklı seviyelerle çiziniz ve her açık çeyrek düzlemde F'nin ızgaradaki en küçük değerini alan noktayı grafikte işaretleyiniz.

Çözüm.
$F \geq 0$'dır; değerleri minimumların yakınında 0'a inerken köşelerde 250 ile 890 arasına çıkar. Eşit aralıklı seviyeler minimumların çevresini boş bırakacağından seviyeleri `np.logspace(-1, 3, 13)` ile 10^{-1} ile 10^{3} arasında logaritmik aralıklı seçiyoruz. Her çeyrekte en küçük değeri bulmak için çeyreğin dışındaki noktaları `np.where` ile $+\infty$ yapıp `argmin` alıyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 401)
X, Y = np.meshgrid(x, x)
F = (X**2 + Y - 11) ** 2 + (X + Y**2 - 7) ** 2

fig, ax = plt.subplots(figsize=(5, 5))
ax.contour(X, Y, F, levels=np.logspace(-1, 3, 13), cmap="viridis")
ax.set_aspect("equal")

for sx in (1, -1):
    for sy in (1, -1):
        quadrant = (sx * X > 0) & (sy * Y > 0)
        masked = np.where(quadrant, F, np.inf)
        i, j = np.unravel_index(np.argmin(masked), F.shape)
        ax.plot(X[i, j], Y[i, j], "r*", markersize=10)
        print(f"({X[i, j]:6.3f}, {Y[i, j]:6.3f}): F = {F[i, j]:.4f}")
fig.savefig("himmelblau.png", dpi=150)
```

 CC BY-NC-SA 4.0

Çıktı:

```
( 3.000,  2.000): F = 0.0000
( 3.575, -1.850): F = 0.0048
(-2.800,  3.125): F = 0.0024
(-3.775, -3.275): F = 0.0030
```

$(3, 2)$ bir ızgara noktasıdır ve orada $F = 0$ olur: $9 + 2 - 11 = 0$ ve $3 + 4 - 7 = 0$. Fonksiyonun öteki üç minimumu da $F = 0$ değerini alır, ama ızgara noktalarına denk gelmez; ızgara aralığı $0{,}025$ olduğundan bulduğumuz noktalar gerçek minimumlara $0{,}01$'den daha yakındır ve F değerleri sıfıra yakın ama sıfır değildir. Bu noktaları tam olarak bulmayı Kök Bulma ve Optimizasyon bölümünde göreceğiz. ■

Alıştırma 8.10 (Bir Helisin Uzunluğu). $x = \cos t$, $y = \sin t$, $z = t/(2\pi)$ $(0 \le t \le 6\pi)$ helisini üç boyutlu bir Axes'te çiziniz ve 1000 noktalı bir kırık çizgiyle bulunan uzunluğu tam uzunlukla karşılaştırınız.

Çözüm.
Hız vektörü $(-\sin t, \cos t, 1/(2\pi))$'dir ve büyüklüğü $\sqrt{1 + 1/(4\pi^2)}$ sabittir. Bu yüzden tam uzunluk

$$L = \int_0^{6\pi} \sqrt{1 + \frac{1}{4\pi^2}}\, dt = 6\pi \cdot \frac{\sqrt{4\pi^2 + 1}}{2\pi} = 3\sqrt{4\pi^2 + 1} \approx 19{,}086795$$

olur. Kırık çizginin uzunluğu ardışık noktalar arasındaki uzaklıkların toplamıdır; `np.diff` ardışık farkları tek seferde verir.

```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, 6 * np.pi, 1000)
x, y, z = np.cos(t), np.sin(t), t / (2 * np.pi)

fig = plt.figure()
ax = fig.add_subplot(projection="3d")
ax.plot(x, y, z)
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_zlabel("$z$")
fig.savefig("helis.png", dpi=150)

dx, dy, dz = np.diff(x), np.diff(y), np.diff(z)
length = np.sum(np.sqrt(dx**2 + dy**2 + dz**2))
exact = 3 * np.sqrt(4 * np.pi**2 + 1)
print(f"Kırık çizginin uzunluğu: {length:.6f}")
print(f"Tam yay uzunluğu       : {exact:.6f}")
```

Çıktı:

```
Kırık çizginin uzunluğu: 19.086519
Tam yay uzunluğu       : 19.086795
```

Kırık çizginin köşeleri eğrinin üzerindedir ve iki nokta arasındaki en kısa yol doğru parçası olduğundan kırık çizgi eğriden biraz kısadır; fark yaklaşık $2{,}8 \cdot 10^{-4}$'tür. ■

 CC BY-NC-SA 4.0

Alıştırma 8.11 (Bir Döngüde Grafik Kaydetmek). $n = 1, 2, 3$ için $y = x^n$ fonksiyonunun $[0, 1]$ üzerindeki grafiğini ayrı birer dosyaya kaydeden ve iş bittiğinde bellekte açık Figure bırakmayan bir döngü yazınız.

Çözüm.
Her turda yeni bir Figure açıyor, kaydettikten hemen sonra `plt.close(fig)` ile kapatıyoruz. Dosya adlarını bir f-string ile kuruyoruz.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 1, 200)
names = []
for n in range(1, 4):
    fig, ax = plt.subplots(figsize=(4, 3))
    ax.plot(x, x**n)
    ax.set_title(f"$y = x^{n}$")
    name = f"kuvvet_{n}.png"
    fig.savefig(name, dpi=100, bbox_inches="tight")
    plt.close(fig)
    names.append(name)

print("Kaydedilen dosyalar:", names)
print("Açık figür sayısı:", len(plt.get_fignums()))
```

Çıktı:

```
Kaydedilen dosyalar: ['kuvvet_1.png', 'kuvvet_2.png', 'kuvvet_3.png']
Açık figür sayısı: 0
```

`plt.close(fig)` satırı olmasaydı döngü üç Figure'ı bellekte bırakırdı ve son satır 3 basardı. Yüzlerce grafik üreten döngülerde bu bellek tüketir; Matplotlib 20'den çok Figure açık kaldığında bir uyarı verir. `bbox_inches="tight"` ise kaydedilen resmin kenarlarındaki boşluğu kırpar. ■

Bu bölümde sayıları görünür kılmayı öğrendik: tek bir eğriden alt grafiklere, logaritmik eksenlerden kontur çizimlerine ve yüzeylere kadar. Sonraki bölümlerde yöntemlerin davranışını hep bu araçlarla çizeceğiz. Önce hesabın sembolik yüzüne geçiyoruz: SymPy ile Sembolik Hesap bölümünde türev, integral ve limitleri sayılarla değil formüllerle hesaplayan bir kütüphane tanıyacak, sembolik ifadeleri NumPy fonksiyonlarına çevirip bu bölümdeki gibi çizmenin yolunu da göreceğiz.

 CC BY-NC-SA 4.0

9. SymPy ile Sembolik Hesap

Şimdiye kadar bilgisayara hep sayılarla hesap yaptırdık. Bu sayılar yaklaşıktır: Python ile İlk Adımlar bölümünde $0{,}1 + 0{,}2$ toplamının tam olarak $0{,}3$ çıkmadığını gördük ve NumPy de aynı 64 bitlik sayılarla çalışır. Oysa matematik yaparken çoğu zaman bir sayı değil, bir formül isteriz. x^x fonksiyonunun türevini, $\int xe^x\,dx$ integralini ya da $\sin x/x$ oranının sıfırdaki limitini kâğıt üzerinde harflerle ve kesin olarak hesaplarız.
Bu tür hesapları yapan programlara **bilgisayar cebir sistemi** (computer algebra system) denir. SymPy, tamamen Python ile yazılmış, açık kaynaklı bir bilgisayar cebir sistemidir. Ayrı bir program değil, `import` ile yüklenen sıradan bir kütüphanedir. Bu bölümde SymPy ile sembol tanımlamayı, ifadeleri açıp çarpanlarına ayırmayı, limit, türev, integral ve seri hesaplamayı, denklem ve diferansiyel denklem çözmeyi ve matrislerle kesin hesap yapmayı öğreneceğiz. En sonda da sembolik sonuçları `lambdify` ile NumPy'ye aktarıp sayısal hesapta kullanacağız.
SymPy `pip install sympy` komutuyla kurulur; Colab gibi çevrim içi Jupyter ortamlarında hazır gelir. Bütün kodlarda kütüphaneyi `import sympy as sp` kısaltmasıyla yükleyeceğiz. Analiz ve Lineer Cebir derslerinde elle yaptığınız hesapları SymPy ile denetleyebilirsiniz. Yine de SymPy'nin yanıtını doğru yorumlamak, arkasındaki matematiği bilmeyi gerektirir. Bu yüzden bölüm boyunca her hesabı ilgili ders notuna bağlayacağız.

9.1 Sayısal ve Sembolik Hesap

İlk iş, sembolik hesabın sayısal hesaptan nerede ayrıldığını görmek.

Tanım 9.1 (Sembolik Hesap). Sayıların ve değişkenlerin yaklaşık değerleriyle değil, kesin matematiksel ifadeler olarak işlendiği hesaba **sembolik hesap** denir. Sembolik hesabın sonucu bir ifadedir: $\sqrt{8}$ yerine $2\sqrt{2}$, $\frac{1}{3} + \frac{1}{6}$ yerine $\frac{1}{2}$, $(\sin x)'$ yerine $\cos x$.

Yani sembolik hesap, kâğıt kalemle yaptığımız hesabın bilgisayardaki karşılığıdır. Sayısal hesapta $\sqrt{2}$, bellekte sonlu bir ikili açılımla tutulur (ekranda $1{,}4142135623730951$ olarak görünür) ve her işlemde küçük bir yuvarlama hatası birikir (bkz. Nümerik Analiz). Sembolik hesapta ise $\sqrt{2}$, karesi tam olarak 2 olan sayı olarak kalır.
Farkı aynı hesabı iki kütüphaneyle yaparak görelim:

```
import math

import sympy as sp

print(math.sqrt(8))
print(sp.sqrt(8))
print(math.sqrt(2) ** 2)
print(sp.sqrt(2) ** 2)
```

Çıktı:

```
2.8284271247461903
2*sqrt(2)
2.0000000000000004
2
```

`math.sqrt` bir `float` döndürür. `sp.sqrt` ise $\sqrt{8} = 2\sqrt{2}$ sadeleştirmesini yapar ve sonucu kesin biçimde saklar; bu yüzden $\sqrt{2}$'nin karesi tam olarak 2 çıkar. Kesin bir ifadenin ondalık açılımını istediğimiz anda, istediğimiz basamak sayısıyla `evalf` metodu ya da `sp.N` fonksiyonu verir. Kesirler için `sp.Rational` kullanılır:

 CC BY-NC-SA 4.0

```
import sympy as sp

a = sp.Rational(1, 3) + sp.Rational(1, 6)
print(a)
print(sp.pi)
print(sp.pi.evalf(30))
print(sp.N(sp.sqrt(2), 50))
print(float(sp.sqrt(2)))
```

Çıktı:

```
1/2
pi
3.14159265358979323846264338328
1.4142135623730950488016887242096980785696718753769
1.4142135623730951
```

`sp.pi` sembolik π sayısıdır; `evalf(30)` onun 30 anlamlı basamağa doğru yuvarlanmış değerini verir. π'nin açılımı …3832795… diye sürdüğünden otuzuncu basamaktaki 7 yukarı yuvarlanıp 8 olmuştur; öncekilerin hepsi π'nin basamaklarıyla aynıdır. Bu doğruluk tesadüf değildir: SymPy ondalık açılımı istenen duyarlıkta yeniden hesaplar. `float(...)` ise ifadeyi sıradan bir Python `float`'una çevirir. Sembolik dünyadan sayısal dünyaya geçmenin en basit yolu budur; daha güçlüsünü bölümün sonunda `lambdify` ile göreceğiz.

9.2 Semboller ve İfadeler

Sembolik hesabın ilk adımı, harflerle gösterilen değişkenleri SymPy'ye tanıtmaktır.

Tanım 9.2 (Sembol). SymPy'de bir matematiksel değişkeni temsil eden nesneye **sembol** denir. Semboller `sp.symbols` fonksiyonuyla oluşturulur: `x, y = sp.symbols("x y")` satırı adları `x` ve `y` olan iki sembol üretir ve onları aynı adlı iki Python değişkenine bağlar.

Yani `x` artık bir sayı değil, değeri belirsiz bir harftir. Semboller ve sayılar `+`, `-`, `*`, `/`, `**` işlemleriyle ve `sp.sin`, `sp.exp`, `sp.log` gibi SymPy fonksiyonlarıyla birleştirilerek **ifadeler** (expression) kurulur. Bir ifadedeki sembolün yerine bir sayı ya da başka bir ifade koymak için `subs` metodu kullanılır:

```
import sympy as sp

x, y = sp.symbols("x y")
f = x**2 + 2*x*y + sp.sin(x)
print(f)
print(f.subs(x, 1))
print(f.subs({x: 2, y: 3}))
print(f.subs(x, sp.pi))
print(f.subs(y, x**2))
```

Çıktı:

```
x**2 + 2*x*y + sin(x)
2*y + sin(1) + 1
sin(2) + 16
2*pi*y + pi**2
2*x**3 + x**2 + sin(x)
```

`subs` sonucu kendiliğinden sadeleştirir: $x = \pi$ konunca $\sin\pi = 0$ terimi düşer. $\sin 1$ ve $\sin 2$ ise kesin sayılar olarak kalır, ondalığa çevrilmez. Burada `sp.sin` yerine `math.sin` yazılamaz, çünkü `math.sin` yalnız sayı kabul eder ve sembolle çağrılınca hata verir. NumPy'nin `np.sin` fonksiyonu da sembollerle çalışmaz; sembolik ifadelerde daima `sp.` önekli fonksiyonları kullanırız.

 CC BY-NC-SA 4.0

⚠ Python sayıları SymPy'ye ulaşmadan hesaplanır

`x + 1/3` yazıldığında Python önce kendi kuralıyla `1/3` bölmesini yapar ve bir `float` üretir; SymPy yalnız bu yaklaşık sayıyı görür. Kesin kesir için `sp.Rational(1, 3)` yazılmalıdır. Kuvvet işaretine de dikkat: Python'da üs alma `**` ile yapılır. `^` başka bir işlemdir (bitlerde dışlayan veya, XOR) ve `2 ^ 3` ifadesinin değeri `1`'dir.

```
import sympy as sp

x = sp.symbols("x")
print(x + 1/3)
print(x + sp.Rational(1, 3))
print(2 ^ 3, 2 ** 3)
```

Çıktı:

```
x + 0.333333333333333
x + 1/3
1 8
```

SymPy bir ifadeyi bir metin olarak değil, bir ağaç olarak saklar.

Tanım 9.3 (İfade Ağacı). Bir SymPy ifadesinin **ifade ağacı**, iç düğümleri işlemler ve fonksiyonlar (`Add`, `Mul`, `Pow`, `sin`, ...), yaprakları semboller ve sayılar olan ağaçtır. Kökteki düğümün türü `func` niteliğiyle, alt ağaçları `args` niteliğiyle okunur.

Yani $x^2 + 2xy + \sin x$ ifadesi SymPy için "üç terimin toplamı"dır: kökte bir `Add` düğümü, onun altında bir kuvvet (`Pow`), bir çarpım (`Mul`) ve bir `sin` düğümü vardır. Çıkarma ve bölme için ayrı düğüm yoktur: $a - b$ ifadesi `Add(a, Mul(-1, b))`, a/b ifadesi de `Mul(a, Pow(b, -1))` olarak saklanır. Her alt ağacın ham yapısını `sp.srepr` gösterir:

```
import sympy as sp

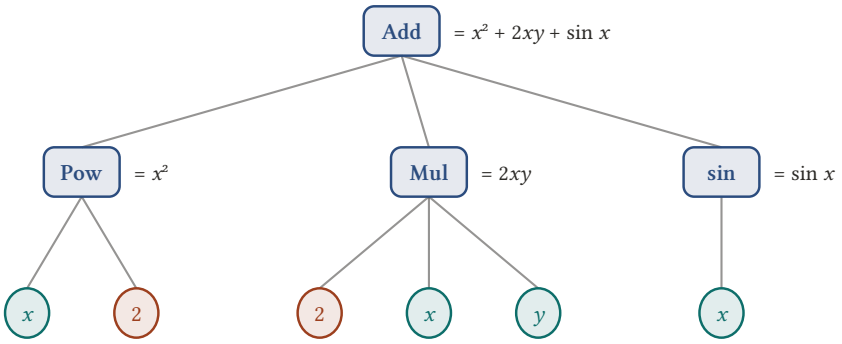
x, y = sp.symbols("x y")
f = x**2 + 2*x*y + sp.sin(x)
print(f.func)
print(f.args)
for term in f.args:
    print(sp.srepr(term))
```

Çıktı:

```
<class 'sympy.core.add.Add'>
(x**2, 2*x*y, sin(x))
Pow(Symbol('x'), Integer(2))
Mul(Integer(2), Symbol('x'), Symbol('y'))
sin(Symbol('x'))
```

Çıktıdan ağacın tamamı okunabilir:

 CC BY-NC-SA 4.0



Şekil 9.1: $x^2 + 2xy + \sin x$ ifadesinin ağacı. Kökteki `Add` düğümünün üç alt ağacı vardır: `Pow`, `Mul` ve `sin`. Yapraklar semboller (x, y) ve tam sayılardır (2). Şekil, yukarıdaki `sp.srepr` çıktılarının çizimidir.

`subs` gibi işlemler bu ağacı dolaşarak çalışır: `f.subs(x, 1)` her `x` yaprağını `1` ile değiştirir, sonra düğümleri aşağıdan yukarıya yeniden hesaplar. SymPy'nin birçok davranışı bu yapıdan gelir. Örneğin iki ifadenin "aynı" olup olmadığı sorusu, ileride göreceğimiz gibi, ağaçların aynı olup olmadığı sorusudur.

9.3 Varsayımlar

Bir sembol hakkında bildiklerimizi SymPy'ye söylemezsek, SymPy en genel durumu düşünür.

Tanım 9.4 (Varsayım). Bir sembol oluşturulurken `real=True`, `positive=True`, `integer=True`, `nonnegative=True` gibi anahtar sözcüklerle verilen bilgilere o sembolün **varsayımları** (assumptions) denir. Varsayımı verilmemiş bir sembol, keyfi bir karmaşık sayıyı temsil eder.

Yani SymPy bir dönüşümü ancak sembolün alabileceği **her** değer için doğruysa yapar. $\sqrt{x^2} = x$ eşitliği $x = -1$ için yanlıştır. $\sqrt{x^2} = |x|$ eşitliği de $x = i$ için yanlıştır: $\sqrt{i^2} = \sqrt{-1} = i$ iken $|i| = 1$'dir. Bu yüzden varsayımsız bir sembolde $\sqrt{x^2}$ olduğu gibi kalır; sembolün reel ya da pozitif olduğunu söyleyince sadeleşir:

```
import sympy as sp

z = sp.symbols("z")
x = sp.symbols("x", real=True)
p = sp.symbols("p", positive=True)
n = sp.symbols("n", integer=True)
print(sp.sqrt(z**2), sp.sqrt(x**2), sp.sqrt(p**2))
print(sp.sin(n * sp.pi), sp.cos(n * sp.pi))
print(p.is_positive, x.is_positive, n.is_integer)
```

Çıktı:

```
sqrt(z**2) Abs(x) p
0 (-1)**n
True None True
```

`n` tam sayı olduğundan SymPy $\sin n\pi = 0$ ve $\cos n\pi = (-1)^n$ dönüşümlerini de yapar. Son satırdaki `is_positive` gibi nitelikler üç değerden birini alır: `True`, `False` ya da `None`. `None` "bilinmiyor" demektir: reel bir x pozitif de olabilir, olmayabilir de. Bir `if` koşulunda `None` yanlış sayılır. Bu yüzden `if x.is_positive:` bloğunun çalışmaması "x pozitif değil" demek değildir; yalnız "pozitif olduğu bilinmiyor" demektir.

Örnek 9.1 (Logaritmanın Çarpım Kuralı ve Varsayımlar). $\ln(ab) = \ln a + \ln b$ dönüşümünü `sp.expand_log` fonksiyonuyla önce varsayımsız, sonra pozitif a, b sembolleriyle yaptırınız. Sonuçların farkını açıklayınız.

Çözüm.

 CC BY-NC-SA 4.0

İki durumu aynı kodda deneyelim. İkinci satırdaki `sp.symbols` çağrısı `a` ve `b` adlarını yeni, pozitif sembollere bağlar:

```
import sympy as sp

a, b = sp.symbols("a b")
print(sp.expand_log(sp.log(a * b)))

a, b = sp.symbols("a b", positive=True)
print(sp.expand_log(sp.log(a * b)))
print(sp.expand_log(sp.log(a**3 / b)))
```

Çıktı:

```
log(a*b)
log(a) + log(b)
3*log(a) - log(b)
```

Varsayımsız sembollerde ifade değişmez, çünkü kural karmaşık sayılarda genel olarak yanlıştır. Örneğin $a = b = -1$ için $\ln(ab) = \ln 1 = 0$'dır; oysa karmaşık logaritmanın esas değeriyle $\ln(-1) = i\pi$ olduğundan $\ln(-1) + \ln(-1) = 2\pi i$ olur. $a, b > 0$ olduğu bilinince kural her değer için doğrudur ve SymPy onu uygular; $\ln(a^3/b) = 3\ln a - \ln b$ dönüşümü de aynı nedenle yapılır. Reel pozitif sayılarla çalışılacaksa sembolleri baştan `positive=True` ile tanımlamak, `force=True` seçeneğiyle SymPy'yi kuralı uygulamaya zorlamaktan daha güvenlidir.
■

9.4 İfadeleri Dönüştürmek

Aynı matematiksel ifadenin birçok yazılışı vardır ve SymPy'de her yazılışa götüren ayrı bir fonksiyon bulunur. En sık kullanılanlar şunlardır:

- `sp.expand`: çarpımları ve kuvvetleri açar.
- `sp.factor`: bir polinomu rasyonel katsayılı indirgenemez çarpanlarına ayırır.
- `sp.collect`: bir sembolün kuvvetlerine göre terimleri toplar.
- `sp.cancel`: bir rasyonel fonksiyonu ortak çarpanları sadeleşmiş pay/payda biçimine getirir.
- `sp.apart`: bir rasyonel fonksiyonu basit kesirlere ayırır.
- `sp.together`: kesirleri ortak paydada toplar.
- `sp.expand_trig`, `sp.trigsimp`: trigonometrik ifadeleri açar ya da sadeleştirir.

Önce polinomlarla başlayalım:

```
import sympy as sp

x, y = sp.symbols("x y")
print(sp.expand((x + 1)**5))
print(sp.factor(x**4 - 1))
print(sp.factor(x**3 - x**2*y - x*y**2 + y**3))
print(sp.factor(x**2 - 2))
print(sp.factor(x**2 - 2, extension=sp.sqrt(2)))
print(sp.collect(sp.expand((x + y + 1)**2), x))
```

Çıktı:

```
x**5 + 5*x**4 + 10*x**3 + 10*x**2 + 5*x + 1
(x - 1)*(x + 1)*(x**2 + 1)
(x - y)**2*(x + y)
x**2 - 2
(x - sqrt(2))*(x + sqrt(2))
x**2 + x*(2*y + 2) + y**2 + 2*y + 1
```

 CC BY-NC-SA 4.0

`factor`, çarpanlara ayırmayı rasyonel sayılar üzerinde yapar. x^2+1 ve x^2-2 rasyonel katsayılı çarpanlara ayrılmaz; bu yüzden ilki bir çarpan olarak kalır, ikincisi de değişmeden geri döner. `extension=sp.sqrt(2)` seçeneği sayı kümesine $\sqrt{2}$'yi ekler; o zaman $x^2-2=(x-\sqrt{2})(x+\sqrt{2})$ ayrışması bulunur.
Rasyonel fonksiyonlarda üç fonksiyon birbirini tamamlar:

```
import sympy as sp

x = sp.symbols("x")
r = (x**3 - 1) / (x**2 - 1)
print(r)
print(sp.cancel(r))
print(sp.apart((x**2 + 1) / (x**3 - x)))
print(sp.together(1/x + 1/(x + 1)))
```

Çıktı:

```
(x**3 - 1)/(x**2 - 1)
(x**2 + x + 1)/(x + 1)
1/(x + 1) + 1/(x - 1) - 1/x
(2*x + 1)/(x*(x + 1))
```

`cancel`, pay ve paydadaki ortak $x-1$ çarpanını sadeleştirdi. Bu, $x=1$ dışında geçerli bir eşitliktir; özgün kesir $x=1$'de tanımsızdır, sadeleşmiş kesir değildir. `apart`, Analiz 2'deki basit kesirlere ayırma algoritmasını uygular (bkz. Analiz 2). `together` ise bunun tersini yapar.
Trigonometrik ifadeler ve genel amaçlı `sp.simplify` için:

```
import sympy as sp

x = sp.symbols("x")
print(sp.expand_trig(sp.sin(2*x)))
print(sp.expand_trig(sp.cos(3*x)))
print(sp.trigsimp(sp.sin(x)**4 - sp.cos(x)**4))
print(sp.simplify(sp.sin(x)**2 + sp.cos(x)**2))
print(sp.simplify((x**2 - 1) / (x - 1)))
```

Çıktı:

```
2*sin(x)*cos(x)
4*cos(x)**3 - 3*cos(x)
-cos(2*x)
1
x + 1
```

`simplify`, yukarıdaki dönüşümlerin birçoğunu sırayla dener ve bulduğu en kısa ifadeyi döndürür. Hangi dönüşümün işe yarayacağını bilmediğimizde kullanışlıdır. Ama yavaştır ve "en sade biçim" kesin tanımlı bir kavram olmadığından sonucun biçimi önceden kestirilemez. Hedef biçim belliyse (çarpanlara ayrılmış, açılmış, basit kesirlere ayrılmış) ona götüren özel fonksiyonu kullanmak daha iyidir.
İki ifadeyi karşılaştırırken bir inceliğe dikkat etmek gerekir.

Tanım 9.5 (Yapısal Eşitlik). SymPy'de `a == b` karşılaştırması, iki ifadenin ağaçlarının aynı olup olmadığını sınar. Buna **yapısal eşitlik** denir. İki ifadenin matematiksel olarak eşit olması, yani değişkenlerin her değeri için aynı sayıyı vermesi ayrı bir sorudur.

Yani $(x+1)^2$ ile x^2+2x+1 matematiksel olarak eşit, yapısal olarak farklıdır: birinin kökünde `Pow`, ötekinin kökünde `Add` vardır. Matematiksel eşitliği sınamak için farkı sadeleştirip sıfırla karşılaştırırız ya da `equals` metodunu kullanırız:

```
import sympy as sp

x = sp.symbols("x")
a = (x + 1)**2
```

 CC BY-NC-SA 4.0

```
b = x**2 + 2*x + 1
print(a == b)
print(sp.expand(a) == b)
print(sp.simplify(a - b) == 0)
print(a.equals(b))
```

Çıktı:

```
False
True
True
True
```

Bir özdeşliği üç adımda doğrulamak

1. Eşitliğin iki tarafını ayrı ifadeler olarak yazın: `lhs` ve `rhs`.
2. Farkı sadeleştirin: `sp.simplify(lhs - rhs)`.
3. Sonuç `0` ise özdeşlik doğrudur. Sıfırdan farklı bir ifade çıkarsa farkı birkaç sayıda `subs` ile hesaplayın: sıfırdan farklı bir değer görülürse iddia yanlıştır. Bu kontrol gereklidir, çünkü `simplify` her sıfırı tanıyamayabilir.

Örnek 9.2 (Bir Trigonometrik Özdeşlik). $(\sin x + \cos x)^2 = 1 + \sin 2x$ özdeşliğini SymPy ile doğrulayınız.

Çözüm.
Tarifin üç adımını uygulayalım:

```
import sympy as sp

x = sp.symbols("x")
lhs = (sp.sin(x) + sp.cos(x))**2
rhs = 1 + sp.sin(2*x)
print(sp.simplify(lhs - rhs))
print(lhs == rhs)
```

Çıktı:

```
0
False
```

Fark sıfıra sadeleştiğinden özdeşlik her x için doğrudur. İkinci satır, iki tarafın yapısal olarak farklı olduğunu hatırlatır (Tanım 9.5). Elle de kolayca görülür:

$$(\sin x + \cos x)^2 = \sin^2 x + \cos^2 x + 2\sin x \cos x = 1 + \sin 2x.$$

■

9.5 Limit

Analiz 1'de ε–δ ile tanımladığımız limit (bkz. Analiz 1) SymPy'de tek bir fonksiyonla hesaplanır: `sp.limit(f, x, a)`. Sonsuz, iki küçük o harfiyle `sp.oo` olarak yazılır.

```
import sympy as sp

x = sp.symbols("x")
n = sp.symbols("n", positive=True, integer=True)
print(sp.limit(sp.sin(x) / x, x, 0))
```

 CC BY-NC-SA 4.0

```
print(sp.limit((1 - sp.cos(x)) / x**2, x, 0))
print(sp.limit((1 + 1/n)**n, n, sp.oo))
print(sp.limit(x * sp.log(x), x, 0, dir="+"))
print(sp.limit(x**2 / sp.exp(x), x, sp.oo))
```

Çıktı:

```
1
1/2
E
0
0
```

Üçüncü satırdaki `E`, SymPy'nin e sayısıdır (bkz. Analiz 1). SymPy limitleri, değişkene küçük sayılar koyarak tahmin etmez; ifadenin seri açılımları üzerinde çalışan bir algoritmayla kesin olarak hesaplar. Bu yüzden $0 \cdot \infty$ ya da ∞/∞ gibi belirsiz biçimler de sorun çıkarmaz.
Tek yönlü limitler `dir` seçeneğiyle istenir: `dir="+"` sağdan, `dir="-"` soldan, `dir="+-"` iki yönlü limit demektir. $\arctan(1/x)$ fonksiyonunun sıfırdaki davranışına bakalım:

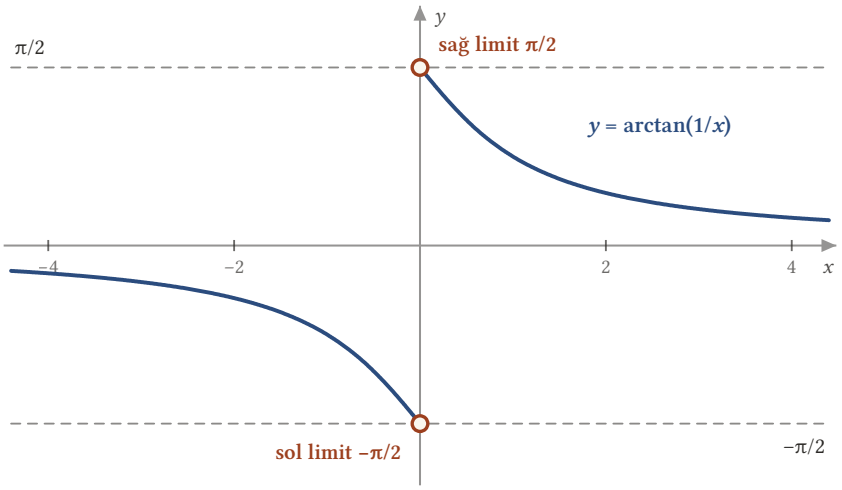
```
import sympy as sp

x = sp.symbols("x")
f = sp.atan(1 / x)
print(sp.limit(f, x, 0, dir="+"))
print(sp.limit(f, x, 0, dir="-"))
try:
    print(sp.limit(f, x, 0, dir="+-"))
except ValueError:
    print("İki yönlü limit yok: sol ve sağ limit farklı.")
```

Çıktı:

```
pi/2
-pi/2
İki yönlü limit yok: sol ve sağ limit farklı.
```

x sağdan sıfıra yaklaşırken $1/x \to +\infty$ olduğundan $\arctan(1/x) \to \pi/2$; soldan yaklaşırken $1/x \to -\infty$ olduğundan $\arctan(1/x) \to -\pi/2$ olur. Sağ ve sol limit farklı olduğundan iki yönlü limit yoktur (bkz. Analiz 1) ve SymPy `ValueError` hatası verir. Hatayı `try`/`except` ile yakaladık.



Şekil 9.2: $y = \arctan(1/x)$ fonksiyonunun grafiği. Sağdan yaklaşırken değerler $\pi/2$'ye, soldan yaklaşırken $-\pi/2$'ye gider; içi boş noktalar bu iki tek yönlü limiti gösterir. Sağ ve sol limit farklı olduğundan sıfırda iki yönlü limit yoktur.

 CC BY-NC-SA 4.0

⚠ Limit fonksiyonu varsayılan olarak sağ limiti alır

`dir` verilmezse `sp.limit` sağ limiti hesaplar. Bu yüzden `sp.limit(1/x, x, 0)` sonucu `oo` olur, oysa $1/x$ fonksiyonunun sıfırda iki yönlü limiti yoktur. İki yönlü limit isteniyorsa `dir="+-"` yazılmalı ya da iki yön ayrı ayrı hesaplanmalıdır:

```
import sympy as sp

x = sp.symbols("x")
print(sp.limit(1 / x, x, 0))
print(sp.limit(1 / x, x, 0, dir="-"))
print(sp.limit(1 / x**2, x, 0, dir="+-"))
```

Çıktı:

```
oo
-oo
oo
```

9.6 Türev

Türev `sp.diff` fonksiyonuyla alınır: `sp.diff(f, x)` birinci türevi, `sp.diff(f, x, n)` n'inci türevi, `sp.diff(f, x, y)` önce x'e, sonra y'ye göre kısmi türevi verir. Türevin tanımı ve kuralları için bkz. Analiz 2.

```
import sympy as sp

x, y = sp.symbols("x y")
print(sp.diff(x**x, x))
print(sp.diff(sp.sin(x)**2, x, 2))
print(sp.diff(sp.exp(x * y**2), x, y))
d = sp.Derivative(sp.atan(x), x)
print(d, "=", d.doit())
```

Çıktı:

```
x**x*(log(x) + 1)
2*(-sin(x)**2 + cos(x)**2)
2*y*(x*y**2 + 1)*exp(x*y**2)
Derivative(atan(x), x) = 1/(x**2 + 1)
```

İlk satır, logaritmik türevle bulunan $(x^x)' = x^x(\ln x + 1)$ formülüdür. İkinci satır $(\sin^2 x)'' = 2(\cos^2 x - \sin^2 x) = 2\cos 2x$ sonucunu açılmış biçimde verir. `sp.Derivative` türevi hesaplamadan, yalnız gösterir; `doit` metodu hesabı yapar. Aynı ayrım integral, limit ve toplam için de vardır: `sp.Integral`, `sp.Limit`, `sp.Sum`.

Türevi bir noktada hesaplamak için önce türev ifadesini bulur, sonra `subs` ile noktayı koyarız. $f(x) = x^3 - 2x$ eğrisinin $x = 1$ noktasındaki teğetini bulalım. Teğet doğrusu $y = f(a) + f'(a)(x - a)$ denklemiyle verilir (bkz. Analiz 2):

```
import sympy as sp

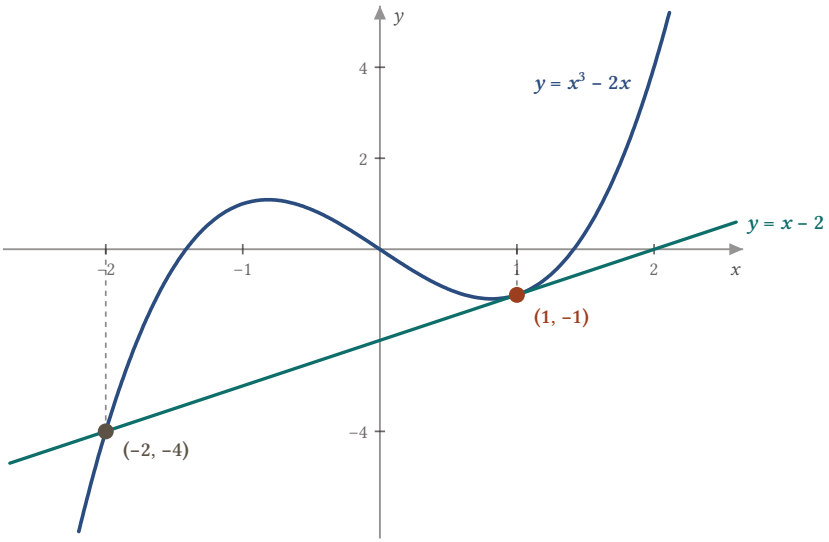
x = sp.symbols("x")
f = x**3 - 2*x
a = 1
slope = sp.diff(f, x).subs(x, a)
tangent = f.subs(x, a) + slope * (x - a)
print("eğim:", slope)
print("teğet: y =", sp.expand(tangent))
print(sp.factor(f - tangent))
```

 CC BY-NC-SA 4.0

Çıktı:

```
eğim: 1
teğet: y = x - 2
(x - 1)**2*(x + 2)
```

Son satır ilginç bir şey söyler: eğri ile teğet arasındaki fark $(x-1)^2(x+2)$'dir. $x=1$'deki **çift** kök teğetliği gösterir; eğri doğruya bu noktada değer ama onu kesmez. Basit kök $x=-2$ ise teğetin eğriyi ikinci kez kestiği $(-2,-4)$ noktasıdır.



Şekil 9.3: $y = x^3 - 2x$ eğrisi ve $x = 1$'deki teğeti $y = x - 2$. Teğet eğriye $(1, -1)$ noktasında değer, farkın çift kökü $x = 1$ budur. Basit kök $x = -2$ ise teğetin eğriyi yeniden kestiği $(-2, -4)$ noktasını verir.

9.7 İntegral

İntegral için tek fonksiyon vardır: `sp.integrate(f, x)` belirsiz integrali, `sp.integrate(f, (x, a, b))` $[a,b]$ üzerindeki belirli integrali hesaplar. Sınırlar `sp.oo` olabilir; birden çok sınır üçlüsü verilirse katlı integral içten dışa doğru hesaplanır.

```
import sympy as sp

x, y = sp.symbols("x y")
print(sp.integrate(x * sp.exp(x), x))
print(sp.integrate(1 / (1 + x**2), x))
print(sp.integrate(sp.sin(x)**2, (x, 0, sp.pi)))
print(sp.integrate(sp.exp(-x**2), (x, -sp.oo, sp.oo)))
print(sp.integrate(1 / x**2, (x, 1, sp.oo)))
print(sp.integrate(x * y, (y, 0, x), (x, 0, 1)))
```

Çıktı:

```
(x - 1)*exp(x)
atan(x)
pi/2
sqrt(pi)
1
1/8
```

İlk satır kısmi integrasyonla bulunan $\int xe^x\,dx = (x-1)e^x$, ikinci satır da temel $\int \frac{dx}{1+x^2} = \arctan x$ integralidir. Ardından $\int_0^{\pi} \sin^2 x\,dx = \pi/2$, Gauss integrali $\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi}$ ve yakınsak bir genelleştirilmiş integral gelir. Son satır, önce y'ye göre 0'dan x'e, sonra x'e göre 0'dan 1'e alınan

 CC BY-NC-SA 4.0

$$\int_0^1 \int_0^x xy\,dy\,dx = \int_0^1 \frac{x^3}{2}\,dx = \frac{1}{8}$$

katlı integralidir.

> ⚠ Belirsiz integralde sabit yoktur
>
> `sp.integrate(f, x)` bir ilkel fonksiyon döndürür, $+C$ sabitini yazmaz. Ayrıca $\int dx/x$ için `log(x)` verir, mutlak değer koymaz. Bu, $x > 0$ aralığında doğru bir ilkeldir; $x < 0$ için ilkel $\ln|x|$'tir. Bulunan ilkeli türev alarak denetlemek iyi bir alışkanlıktır:
>
> ```
> import sympy as sp
>
> x = sp.symbols("x")
> F = sp.integrate(x * sp.exp(x), x)
> print(sp.simplify(sp.diff(F, x) - x * sp.exp(x)))
> print(sp.integrate(1 / x, x))
> ```
>
> Çıktı:
>
> ```
> 0
> log(x)
> ```

Her integralin ilkeli temel fonksiyonlarla yazılamaz. SymPy böyle durumlarda ya integralle tanımlanan özel bir fonksiyon kullanır ya da integrali hesaplanmamış olarak geri verir:

```
import sympy as sp

x = sp.symbols("x")
print(sp.integrate(sp.exp(-x**2), x))
print(sp.integrate(sp.sin(x) / x, x))
I = sp.Integral(x**x, (x, 0, 1))
print(I.doit())
print(I.evalf(20))
```

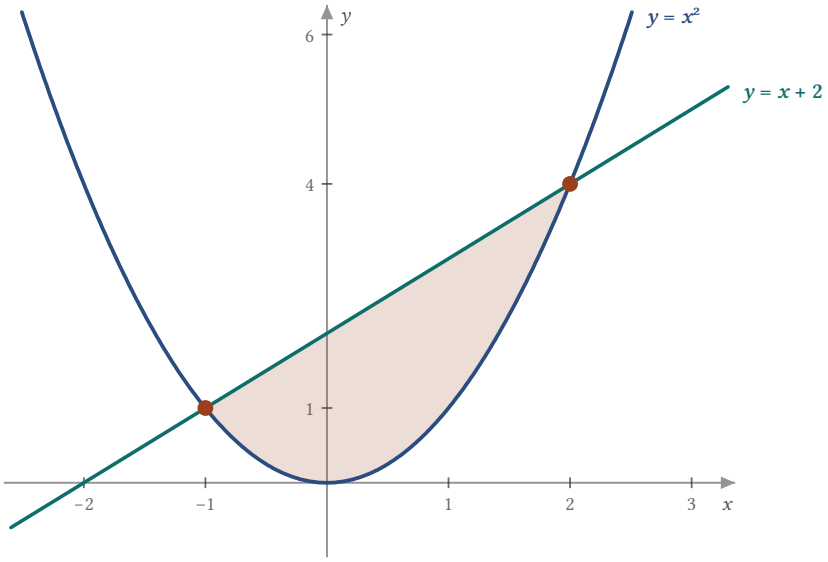
Çıktı:

```
sqrt(pi)*erf(x)/2
Si(x)
Integral(x**x, (x, 0, 1))
0.78343051071213440706
```

`erf` hata fonksiyonu, $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$ ile; `Si` sinüs integrali de $\operatorname{Si}(x) = \int_0^x \frac{\sin t}{t}\,dt$ ile tanımlanır. $\int_0^1 x^x\,dx$ için kapalı bir biçim bulunamadı ve `doit` integrali olduğu gibi döndürdü. `evalf(20)` ise integrali sayısal olarak 20 anlamlı basamakla hesapladı. Sayısal integrali Sayısal Türev ve İntegral bölümünde ayrıntısıyla göreceğiz.

Örnek 9.3 (İki Eğri Arasındaki Alan). $y = x^2$ parabolü ile $y = x + 2$ doğrusu arasında kalan sınırlı bölgenin alanını SymPy ile hesaplayınız.

 CC BY-NC-SA 4.0



Şekil 9.4: $y = x^2$ parabolü ile $y = x + 2$ doğrusu arasında kalan sınırlı bölge (gölgeli). İki eğri bölgenin uç noktalarında kesişir.

Çözüm.
Önce iki eğrinin kesiştiği noktaları buluruz. Bunun için `sp.solve` fonksiyonunu kullanacağız. Fonksiyonu Denklem Çözmek kısmında ayrıntısıyla göreceğiz; burada denklemin çözümlerini bir liste olarak verir. `sp.Eq(top, bottom)` de iki tarafın eşitliğini, yani $x + 2 = x^2$ denklemini yazar. Sonra kesişimler arasında üstteki eğriden alttakini çıkarıp integral alırız (bkz. Analiz 2):

```
import sympy as sp

x = sp.symbols("x")
top = x + 2
bottom = x**2
a, b = sp.solve(sp.Eq(top, bottom), x)
print("kesişimler:", a, b)
print("orta noktada fark:", (top - bottom).subs(x, (a + b) / 2))
area = sp.integrate(top - bottom, (x, a, b))
print("alan:", area, "=", area.evalf())
```

Çıktı:

```
kesişimler: -1 2
orta noktada fark: 9/4
alan: 9/2 = 4.50000000000000
```

Eğriler $x = -1$ ve $x = 2$'de kesişir. Aradaki bir noktada farkın pozitif çıkması, bu aralıkta doğrunun parabolün üstünde kaldığını doğrular. Elle hesap aynı sonucu verir:

$$\int_{-1}^{2} (x + 2 - x^2)\,dx = \left[\frac{x^2}{2} + 2x - \frac{x^3}{3}\right]_{-1}^{2} = \frac{10}{3} + \frac{7}{6} = \frac{9}{2}.$$

■

 CC BY-NC-SA 4.0

9.8 Seriler ve Toplamlar

Bir fonksiyonun bir nokta etrafındaki Taylor açılımı `sp.series(f, x, a, n)` ile bulunur. Sonuç $(x-a)^{n-1}$ terimine kadar olan açılım ile bir $O\left((x-a)^n\right)$ kalan teriminden oluşur. `removeO` metodu kalan terimini atar ve geriye Taylor polinomu kalır (bkz. Analiz 2).

```
import sympy as sp

x = sp.symbols("x")
s = sp.series(sp.exp(x), x, 0, 6)
print(s)
print(s.removeO())
print(sp.series(sp.tan(x), x, 0, 8))
print(sp.series(sp.sqrt(x), x, 4, 3))
```

Çıktı:

```
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
x**5/120 + x**4/24 + x**3/6 + x**2/2 + x + 1
x + x**3/3 + 2*x**5/15 + 17*x**7/315 + O(x**8)
1 - (x - 4)**2/64 + x/4 + O((x - 4)**3, (x, 4))
```

`n` parametresi terim sayısını değil, kalanın kuvvetini belirler: `sp.series(sp.exp(x), x, 0, 6)` en çok x^5 terimini içerir. Son satır $\sqrt{x}$'in 4 etrafındaki açılımıdır. SymPy $2+\frac{x-4}{4}$ toplamını $1+\frac{x}{4}$ olarak yazdığından açılım ilk bakışta tanıdık görünmez; ama $\sqrt{x}\approx 2+\frac{x-4}{4}-\frac{(x-4)^2}{64}$ ile aynıdır.
Sonlu ve sonsuz toplamlar `sp.summation` ile hesaplanır:

```
import sympy as sp

k, n = sp.symbols("k n", integer=True, positive=True)
s = sp.summation(k**2, (k, 1, n))
print(s)
print(sp.factor(s))
print(sp.summation(1 / k**2, (k, 1, sp.oo)))
print(sp.summation(sp.Rational(1, 2)**k, (k, 0, sp.oo)))
```

Çıktı:

```
n**3/3 + n**2/2 + n/6
n*(n + 1)*(2*n + 1)/6
pi**2/6
2
```

İlk iki satır $1^2+2^2+\cdots+n^2=\frac{n(n+1)(2n+1)}{6}$ formülüdür. Ardından Basel problemi olarak bilinen $\sum 1/k^2=\pi^2/6$ eşitliği ve $1/2$ oranlı geometrik serinin toplamı gelir.

Örnek 9.4 (e Sayısına Taylor Polinomlarıyla Yaklaşmak). e^x fonksiyonunun 0 etrafındaki n'inci Taylor polinomu T_n olsun. $|\,e-T_n(1)\,|<10^{-4}$ eşitsizliğini sağlayan en küçük n'yi SymPy ile bulunuz.

Çözüm.
$T_n(1)=\sum_{k=0}^{n}1/k!$ değerini kesir olarak hesaplayıp hatayı `evalf` ile ölçelim:

```
import sympy as sp

x = sp.symbols("x")
for n in range(4, 9):
    T = sp.series(sp.exp(x), x, 0, n + 1).removeO()
    value = T.subs(x, 1)
    error = (sp.E - value).evalf(5)
    print(n, value, error)
```

Çıktı:

```
4 65/24 0.0099485
5 163/60 0.0016152
6 1957/720 0.00022627
7 685/252 2.7860e-5
8 109601/40320 3.0586e-6
```

Hata $n=6$ için $2{,}26\cdot 10^{-4}$, $n=7$ için $2{,}79\cdot 10^{-5}$'tir. İstenen en küçük değer $n=7$'dir ve $e\approx T_7(1)=\frac{685}{252}$ olur. Hata kesin bir kesir olan $T_n(1)$ ile sembolik e arasındaki farktan hesaplandığından, kendisi bir yuvarlama hatası taşımaz. Benzer bir soruyu $5\cdot 10^{-5}$ duyarlıkla, Lagrange kalanıyla ve hesap yapmadan önce çözen bir örnek için bkz. Analiz 2. Oradaki $e-T_n(1)<\frac{3}{(n+1)!}$ sınırı bizim 10^{-4} duyarlığımızda da $n=7$ verir: sınırın 10^{-4}'ten küçük olması için $(n+1)!>30\,000$ gerekir ve $7!=5040<30\,000<40\,320=8!$ olduğundan en küçük seçim $n+1=8$'dir.
■

9.9 Denklem Çözmek

Alan örneğinde kısaca kullandığımız `sp.solve`, SymPy'nin en çok başvurulan fonksiyonlarından biridir. `sp.solve(expr, x)`, expr $=0$ denkleminin çözümlerini bir liste olarak verir. Denklemi iki taraflı yazmak için `sp.Eq(lhs, rhs)` kullanılır. Python'da `=` atama, `==` de yapısal eşitlik olduğundan denklem için ayrı bir nesne gerekir.

```
import sympy as sp

x, a, b, c = sp.symbols("x a b c")
print(sp.solve(x**2 - 5*x + 6, x))
print(sp.solve(sp.Eq(x**3, 8), x))
print(sp.solve(x**2 + 1, x))
print(sp.solve(a*x**2 + b*x + c, x))
```

Çıktı:

```
[2, 3]
[2, -1 - sqrt(3)*I, -1 + sqrt(3)*I]
[-I, I]
[(-b - sqrt(-4*a*c + b**2))/(2*a), (-b + sqrt(-4*a*c + b**2))/(2*a)]
```

`I`, SymPy'nin sanal birimidir: $x^3=8$ denkleminin bir reel, iki karmaşık kökü vardır. Son satır ikinci derece denklemin kök formülüdür. Ancak dikkat: SymPy burada $a\neq 0$ olduğunu sessizce varsaydı. `solve` "genel" durumu çözer; parametrelerin özel değerlerini ayrıca incelemek bize düşer.
Çözümleri belli bir kümede aramak için `sp.solveset` daha uygundur. `domain` seçeneği çözüm kümesini belirler ve sonuç her zaman bir küme olur; eşitsizlikler de çözülebilir:

```
import sympy as sp

x = sp.symbols("x")
print(sp.solveset(x**2 + 1, x, domain=sp.S.Reals))
print(sp.solveset(sp.sin(x), x, domain=sp.Interval(0, 10)))
print(sp.solveset(x**2 - 4 > 0, x, domain=sp.S.Reals))
```

Çıktı:

```
EmptySet
{0, pi, 2*pi, 3*pi}
Union(Interval.open(-oo, -2), Interval.open(2, oo))
```

$x^2+1=0$ denkleminin reel çözümü yoktur (`EmptySet`, boş küme). $[0,10]$ aralığında $\sin x=0$ denkleminin çözümleri $0,\pi,2\pi,3\pi$'dir. $x^2>4$ eşitsizliğinin çözümü de $(-\infty,-2)\cup(2,\infty)$ kümesidir.
Denklem sistemleri için denklemler bir liste, bilinmeyenler ikinci bir liste olarak verilir. Bir çember ile bir doğrunun kesişimini bulalım: $x^2+y^2=5$, $y=x+1$.

 CC BY-NC-SA 4.0

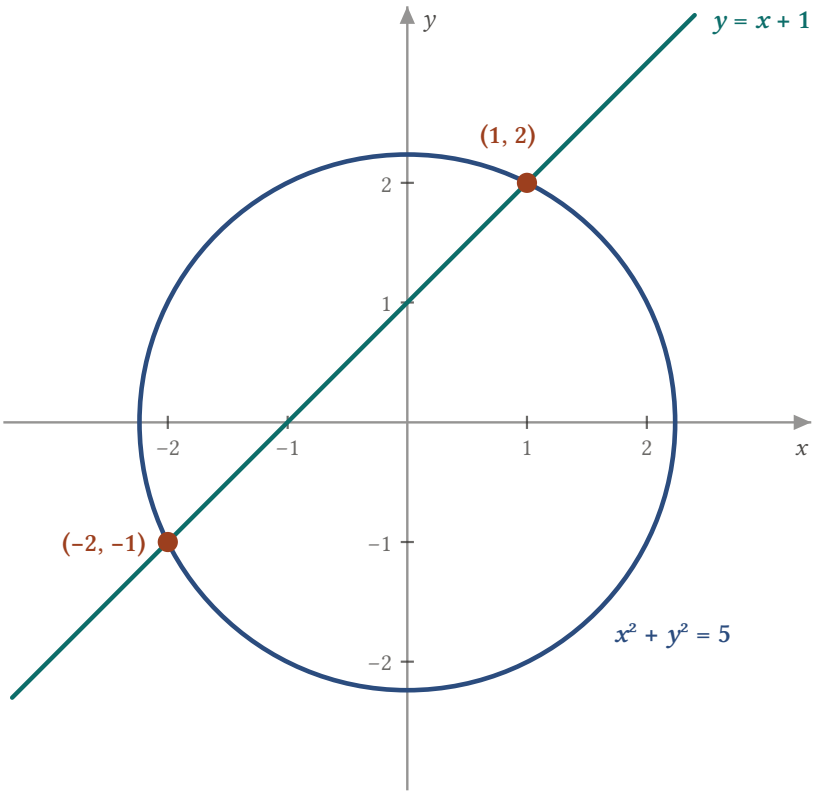
```
import sympy as sp

x, y = sp.symbols("x y")
eqs = [sp.Eq(x**2 + y**2, 5), sp.Eq(y, x + 1)]
sols = sp.solve(eqs, [x, y], dict=True)
print(sols)
for s in sols:
    print(s[x], s[y], [e.subs(s) for e in eqs])
```

Çıktı:

```
[{x: -2, y: -1}, {x: 1, y: 2}]
-2 -1 [True, True]
1 2 [True, True]
```

`dict=True` seçeneği her çözümü `{x: ..., y: ...}` biçiminde bir sözlük olarak verir. Böylece çözüm doğrudan `subs`'a verilebilir. Son iki satırdaki `[True, True]` listeleri, iki çözümün de iki denklemi sağladığını gösterir.



Şekil 9.5: $x^2 + y^2 = 5$ çemberi ile $y = x + 1$ doğrusu $(-2, -1)$ ve $(1, 2)$ noktalarında kesişir. Bu iki nokta, `sp.solve` fonksiyonunun bulduğu iki çözümdür.

Doğrusal sistemlerde `sp.linsolve` çözüm kümesini verir; sonsuz çözüm varsa onu serbest değişkenlere bağlı olarak yazar:

```
import sympy as sp

x, y, z = sp.symbols("x y z")
eqs = [x + y + z - 6, x - y + 2*z - 5, 2*x + y - z - 1]
print(sp.linsolve(eqs, [x, y, z]))
print(sp.linsolve([x + y - 2, 2*x + 2*y - 4], [x, y]))
```

 CC BY-NC-SA 4.0

Çıktı:

```
{(1, 2, 3)}
{(2 - y, y)}
```

İlk sistemin tek çözümü $(1, 2, 3)$'tür. İkinci sistemin iki denklemi aynı doğruyu verdiğinden sonsuz çözüm vardır: her y için $(2 - y, y)$ bir çözümdür.
Her denklemin kapalı biçimde bir çözümü yoktur. $\cos x = x$ denkleminin kökü temel fonksiyonlarla yazılamaz ve `sp.solve` bunu `NotImplementedError` hatasıyla bildirir. Böyle durumlarda `sp.nsolve`, verilen bir başlangıç noktasından sayısal bir yöntemle kökü bulur:

```
import sympy as sp

x = sp.symbols("x")
try:
    sp.solve(sp.cos(x) - x, x)
except NotImplementedError:
    print("solve kapalı biçimde bir çözüm bulamadı.")
print(sp.nsolve(sp.cos(x) - x, x, 0.7))
```

Çıktı:

```
solve kapalı biçimde bir çözüm bulamadı.
0.739085133215161
```

Aynı kökü Newton-Raphson yöntemiyle elle hesaplayan örnek için bkz. Nümerik Analiz. Sayısal kök bulmayı bir sonraki bölümde SciPy ile ayrıntılı olarak ele alacağız.

9.10 Diferansiyel Denklem Çözmek

SymPy bazı diferansiyel denklem türlerini de kapalı biçimde çözer. Bilinmeyen fonksiyon `sp.Function("y")` ile tanımlanır ve `y(x).diff(x)` onun türevini gösterir. Denklem `sp.Eq` ile kurulur, `sp.dsolve` ile çözülür. Başlangıç koşulları `ics` sözlüğüyle verilir.
Birinci mertebeden doğrusal $y' + y = x$ denklemini önce genel olarak, sonra $y(0) = 1$ koşuluyla çözelim:

```
import sympy as sp

x = sp.symbols("x")
y = sp.Function("y")
ode = sp.Eq(y(x).diff(x) + y(x), x)
print(sp.dsolve(ode, y(x)))
sol = sp.dsolve(ode, y(x), ics={y(0): 1})
print(sol)
print(sp.checkodesol(ode, sol))
```

Çıktı:

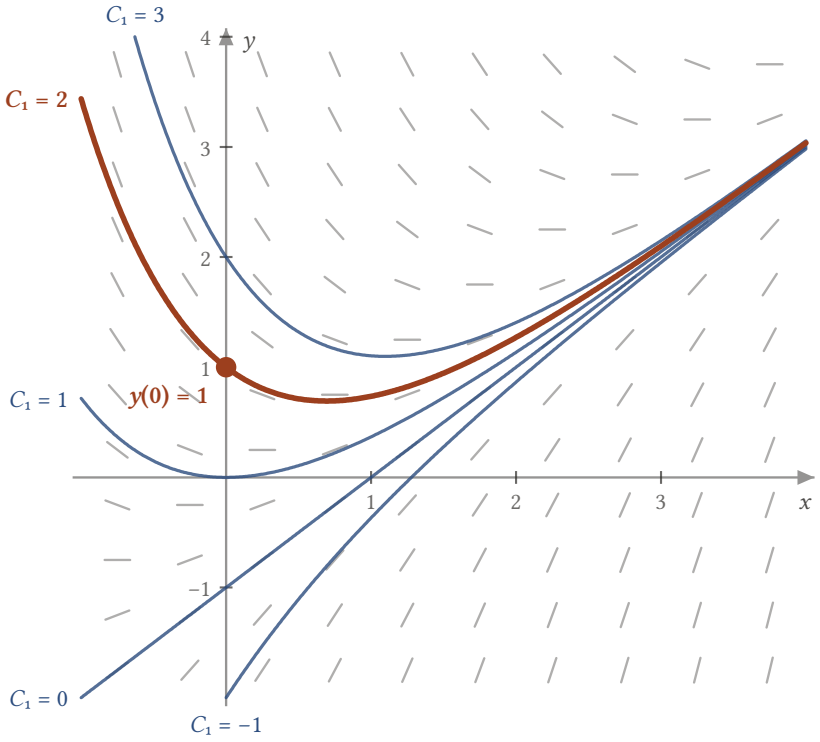
```
Eq(y(x), C1*exp(-x) + x - 1)
Eq(y(x), x - 1 + 2*exp(-x))
(True, 0)
```

Genel çözüm $y = C_1 e^{-x} + x - 1$ ailesidir; `C1` keyfi sabittir. $y(0) = 1$ koşulu $C_1 - 1 = 1$, yani $C_1 = 2$ verir. `sp.checkodesol`, çözümü denkleme koyar ve artığın sıfır olduğunu `(True, 0)` ile bildirir. Aynı sonuca elle de ulaşılır. Denklemi e^{x} integrasyon çarpanıyla çarparsak (bkz. Diferansiyel Denklemler)

$$(e^{x} y)' = x e^{x} \quad \Rightarrow \quad e^{x} y = (x - 1) e^{x} + C_1$$

olur. Aşağıdaki şekil denklemin yön alanını ve aileden birkaç çözümü gösterir:

 CC BY-NC-SA 4.0



Şekil 9.6: $y' = x - y$ denkleminin yön alanı (kısa çizgiler her noktadaki eğimi gösterir) ve $y = C_1 e^{-x} + x - 1$ çözüm ailesinden $C_1 = -1, 0, 1, 2, 3$ eğrileri. Kalın eğri, $(0, 1)$ noktasından geçen $C_1 = 2$ çözümüdür. Bütün çözümler $C_1 = 0$ için elde edilen $y = x - 1$ doğrusuna yaklaşır.

$C_1 = 0$ için çözüm $y = x - 1$ doğrusudur. Diğer bütün çözümler $C_1 e^{-x} \to 0$ olduğundan $x \to \infty$ iken bu doğruya yaklaşır. Kapalı biçimde çözülemeyen denklemleri Diferansiyel Denklemlerin Sayısal Çözümü bölümünde sayısal yöntemlerle çözeceğiz.

9.11 Matrisler

Lineer cebirin hesapları da sembolik olarak, yani kesin kesirlerle yapılabilir. `sp.Matrix` ile kurulan matrislerin girdileri kesin sayılar ya da ifadelerdir. Matris çarpımı `*` ya da `@` ile yapılır; `.T` transpozu, `.det()` determinantı, `.inv()` tersi, `.rank()` rankı verir. NumPy ile Lineer Cebir bölümündeki `np.linalg` fonksiyonlarından farkı, sonuçların yaklaşık ondalık sayılar değil, kesin kesirler olmasıdır.

```
import sympy as sp

A = sp.Matrix([[2, 1, 0], [1, 3, 1], [0, 1, 2]])
print(A.det())
print(A.inv())
print(A * A.inv() == sp.eye(3))
```

Çıktı:

```
8
Matrix([[5/8, -1/4, 1/8], [-1/4, 1/2, -1/4], [1/8, -1/4, 5/8]])
True
```

`sp.eye(3)` 3×3 birim matristir. Ters matrisin girdileri kesin kesirler olduğundan $AA^{-1} = I$ eşitliği yapısal olarak da tam tutar.
Satır indirgenmiş merdiven biçim (bkz. Lineer Cebir) `rref` metoduyla, sıfır uzayının bir tabanı da `nullspace` metoduyla bulunur:

 CC BY-NC-SA 4.0

```
import sympy as sp

B = sp.Matrix([[1, 2, 1, 1],
               [2, 4, 0, 6],
               [1, 2, 3, -1]])
R, pivots = B.rref()
print(R)
print(pivots)
print([v.T for v in B.nullspace()])
```

Çıktı:

```
Matrix([[1, 2, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
(0, 2, 3)
[Matrix([[-2, 1, 0, 0]])]
```

`rref` iki şey döndürür: indirgenmiş matrisi ve pivot sütunlarının indislerini. İndisler Python alışkanlığıyla 0'dan başlar; yani pivotlar 1., 3. ve 4. sütunlardadır. Pivotsuz tek sütun ikincisi olduğundan sıfır uzayı tek boyutludur. Tabanı $(-2, 1, 0, 0)$ vektörüdür: $x_1 + 2x_2 = 0, x_3 = x_4 = 0$. Vektörleri tek satırda yazdırmak için transpozlarını (`v.T`) yazdırdık.
Özdeğer ve özvektörler (bkz. Lineer Cebir) için `charpoly`, `eigenvals`, `eigenvects` ve `diagonalize` metodları vardır. Karakteristik polinomun değişkeni olan λ sembolünü `lam` adlı değişkene bağlıyoruz, çünkü `lambda` Python'da ayrılmış bir sözcüktür (bkz. Tanım 2.12):

```
import sympy as sp

lam = sp.symbols("lambda")
A = sp.Matrix([[4, 1], [2, 3]])
p = A.charpoly(lam).as_expr()
print(p, "=", sp.factor(p))
print(A.eigenvals())
for value, mult, vectors in A.eigenvects():
    print(value, mult, [v.T for v in vectors])
P, D = A.diagonalize()
print(P, D)
print(P * D * P.inv() == A)
```

Çıktı:

```
lambda**2 - 7*lambda + 10 = (lambda - 5)*(lambda - 2)
{5: 1, 2: 1}
2 1 [Matrix([[-1/2, 1]])]
5 1 [Matrix([[1, 1]])]
Matrix([[-1, 1], [2, 1]]) Matrix([[2, 0], [0, 5]])
True
```

Karakteristik polinom $\lambda^2 - 7\lambda + 10 = (\lambda - 2)(\lambda - 5)$'tir. `eigenvals` her özdeğeri cebirsel katıyla birlikte bir sözlük olarak verir. `eigenvects` her özdeğer için katı ve öz uzayın bir tabanını listeler. `diagonalize`, $A = PDP^{-1}$ olacak biçimde P ve köşegen D matrislerini döndürür. P'nin sütunları özvektörlerdir; SymPy $(-1/2, 1)$ özvektörünü 2 ile çarpıp $(-1, 2)$ olarak almıştır (bkz. Lineer Cebir).
Girdileri sembol olan matrislerle de çalışılabilir. θ ve φ açılı iki dönme matrisinin çarpımını hesaplayalım:

```
import sympy as sp

theta, phi = sp.symbols("theta phi", real=True)


def rot(a):
    return sp.Matrix([[sp.cos(a), -sp.sin(a)],
                      [sp.sin(a), sp.cos(a)]])
```

 CC BY-NC-SA 4.0

```
M = sp.simplify(rot(theta) * rot(phi))
for row in M.tolist():
    print(row)
print(M == rot(theta + phi))
print(sp.simplify(rot(theta).det()))
```

Çıktı:

```
[cos(phi + theta), -sin(phi + theta)]
[sin(phi + theta), cos(phi + theta)]
True
1
```

İki dönmenin bileşkesi, açıları toplanmış dönmedir. `simplify` bunu görmek için $\cos(\theta+\varphi) = \cos\theta\cos\varphi - \sin\theta\sin\varphi$ toplam formüllerini kullandı. Dönme matrisinin determinantı da $\cos^2\theta + \sin^2\theta = 1$'dir.

9.12 Sembolikten Sayısala: lambdify

Sembolik bir sonuç çoğu zaman yolun yarısıdır. Bulduğumuz bir türevi binlerce noktada hesaplamak, grafiğini çizmek ya da bir sayısal yöntemde kullanmak isteriz. `subs` ile `evalf` bunu yapabilir, ama her noktada ifade ağacını baştan dolaştığı için yavaştır.

Tanım 9.6 (Lambdify). `sp.lambdify(x, expr, "numpy")` çağrısı, `expr` ifadesini, `x` yerine bir sayı ya da bir NumPy dizisi alan sıradan bir Python fonksiyonuna çevirir. Üretilen fonksiyonda `sp.sin`, `sp.exp` gibi SymPy fonksiyonlarının yerini `np.sin`, `np.exp` gibi NumPy karşılıkları alır.

Yani `lambdify` sembolik dünyadan sayısal dünyaya geçen köprüdür: formülü SymPy ile bir kez türetiriz, sonra NumPy hızıyla bir dizinin bütün elemanları üzerinde hesaplarız (bkz. Vektörizasyon ve Broadcasting).

```
import numpy as np
import sympy as sp

x = sp.symbols("x")
expr = sp.diff(x**2 * sp.exp(-x), x)
print(expr)
f = sp.lambdify(x, expr, "numpy")
xs = np.linspace(0, 4, 5)
print(f(xs))
print(f(1.0), expr.subs(x, 1).evalf())
```

Çıktı:

```
-x**2*exp(-x) + 2*x*exp(-x)
[ 0.          0.36787944  0.         -0.14936121 -0.14652511]
0.36787944117144233 0.367879441171442
```

`f`, x^2e^{-x} fonksiyonunun $(2x - x^2)e^{-x}$ türevini hesaplayan bir NumPy fonksiyonudur. Bir dizi alınca bütün elemanlarda birden hesaplar; tek bir sayıda ise `subs` ve `evalf` ile aynı değeri verir. Hız farkını ölçelim:

```
import time

import numpy as np
import sympy as sp

x = sp.symbols("x")
expr = sp.exp(-x**2) * sp.cos(3 * x)
xs = np.linspace(0, 2, 2000)
```

 CC BY-NC-SA 4.0

```
t0 = time.perf_counter()
slow = np.array([float(expr.subs(x, v)) for v in xs])
t1 = time.perf_counter()
f = sp.lambdify(x, expr, "numpy")
t2 = time.perf_counter()
fast = f(xs)
t3 = time.perf_counter()

print(f"subs ile döngü    : {1000 * (t1 - t0):.0f} ms")
print(f"lambdify (üretim) : {1000 * (t2 - t1):.0f} ms")
print(f"lambdify (hesap)  : {1000 * (t3 - t2):.2f} ms")
print("en büyük fark     :", np.max(np.abs(slow - fast)))
```

Çıktı:

```
subs ile döngü    : 962 ms
lambdify (üretim) : 110 ms
lambdify (hesap)  : 0.05 ms
en büyük fark     : 2.220446049250313e-16
```

Süreler bu notların yazıldığı bilgisayarda ölçülmüştür; başka bir bilgisayarda, hatta aynı bilgisayarda her çalıştırmada biraz farklı çıkar. Oranlar ise kabaca aynı kalır. `subs` ile 2000 noktalık döngü yaklaşık bir saniye sürdü. `lambdify` ile üretilen fonksiyon aynı hesabı milisaniyenin çok altında, binlerce kat hızlı yaptı. Fonksiyonu üretmenin maliyeti bir kerelik bir bedeldir: fonksiyon bir kez üretilir, sonra istenen kadar çağrılır. Sonuçlar ise neredeyse tıpatıp aynıdır; en büyük fark son birkaç ikili basamaktaki yuvarlamadan gelir.

Sembolik sonucu sayısal hesaba üç adımda taşımak

1. Formülü SymPy ile türetin (`diff`, `integrate`, `series`, `solve`, …).
2. `sp.lambdify(x, expr, "numpy")` ile ifadeyi bir Python fonksiyonuna çevirin.
3. Fonksiyonu NumPy dizileriyle çağırıp sonuçları hesaplayın ya da çizin.

Tarifi sinüsün Taylor polinomlarına uygulayalım. T_1, T_3, T_5, T_7 polinomlarını SymPy ile bulup NumPy ile hesaplayacak ve Matplotlib ile Grafik Çizimi bölümündeki gibi $\sin x$ ile aynı eksende çizeceğiz:

```
import matplotlib.pyplot as plt
import numpy as np
import sympy as sp

x = sp.symbols("x")
xs = np.linspace(-2 * np.pi, 2 * np.pi, 400)

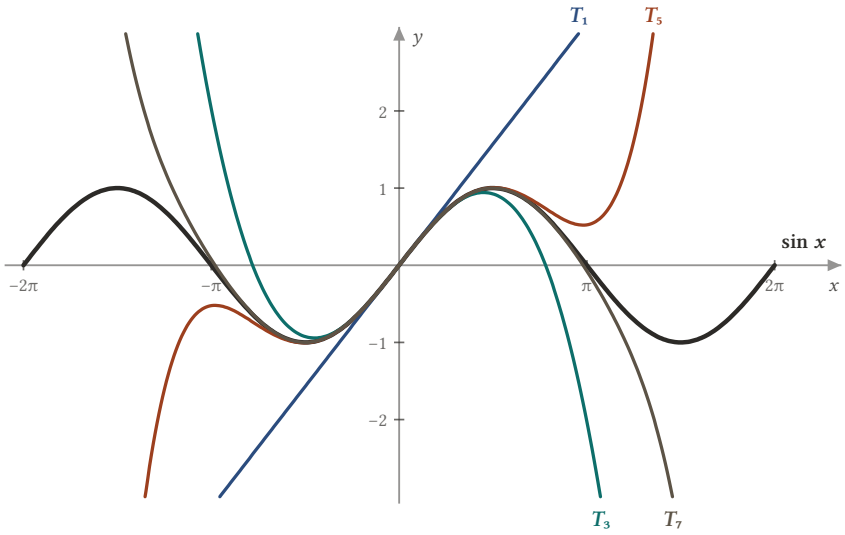
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(xs, np.sin(xs), color="black", lw=2.5, label="sin x")
for n in [1, 3, 5, 7]:
    T = sp.series(sp.sin(x), x, 0, n + 1).removeO()  # 1. adım
    T_num = sp.lambdify(x, T, "numpy")               # 2. adım
    ax.plot(xs, T_num(xs), label=f"T{n}")            # 3. adım
    print(f"T{n}(x) =", T)
ax.set_ylim(-3, 3)
ax.axhline(0, color="gray", lw=0.5)
ax.legend()
fig.savefig("taylor-sin.png", dpi=150)
```

Çıktı:

```
T1(x) = x
T3(x) = -x**3/6 + x
T5(x) = x**5/120 - x**3/6 + x
T7(x) = -x**7/5040 + x**5/120 - x**3/6 + x
```

 CC BY-NC-SA 4.0

Kod, polinomları yazdırır ve grafiği `taylor-sin.png` dosyasına kaydeder. Aşağıdaki şekil aynı grafiğin aynı verilerle çizilmiş hâlidir:



Şekil 9.7: $\sin x$ (kalın) ve Taylor polinomları T_1, T_3, T_5, T_7; Matplotlib kodunun çizdiği grafiğin aynı verilerle (400 nokta, $-3 \le y \le 3$ penceresi) çizilmiş hâli. Derece arttıkça polinom sinüsü daha geniş bir aralıkta izler.

Bütün polinomlar 0 yakınında $\sin x$ ile çakışır; derece arttıkça çakışmanın sürdüğü aralık genişler. Uzakta ise her polinom en yüksek dereceli teriminin işaretine göre $\pm\infty$'a gider. Taylor polinomu yerel bir yaklaşımdır ve hatası Lagrange kalanıyla kestirilir (bkz. Analiz 2).

9.13 Alıştırmalar

Aşağıdaki alıştırmaları önce kendiniz çözmeye çalışın; her çözümde kod ve çıktısı verilmiştir.

Alıştırma 9.1 (Kosinüsün Dördüncü Kuvveti). $\cos^4 x - \sin^4 x = \cos 2x$ özdeşliğini SymPy ile doğrulayınız.

Çözüm.
Adım 1. İki tarafı ayrı ifadeler olarak yazıp farkı sadeleştiririz. Bu, özdeşlik doğrulama tarifidir (bkz. Örnek 9.2).
Adım 2. Sol tarafı çarpanlarına ayırarak özdeşliğin neden doğru olduğunu da görelim:

```
import sympy as sp

x = sp.symbols("x")
lhs = sp.cos(x)**4 - sp.sin(x)**4
rhs = sp.cos(2*x)
print(sp.simplify(lhs - rhs))
print(sp.factor(lhs))
```

Çıktı:

```
0
(-sin(x) + cos(x))*(sin(x) + cos(x))*(sin(x)**2 + cos(x)**2)
```

Adım 3. Fark 0 olduğundan özdeşlik doğrudur. Çarpanlara ayrılmış biçim elle ispatı da verir. `factor`, $a^4 - b^4 = (a-b)(a+b)(a^2+b^2)$ ayrışımını yaptı; ilk iki çarpanın çarpımı $\cos^2 x - \sin^2 x = \cos 2x$, üçüncü çarpan da 1'dir:

 CC BY-NC-SA 4.0

$$\cos^4 x - \sin^4 x = (\cos^2 x - \sin^2 x)(\cos^2 x + \sin^2 x) = \cos 2x \cdot 1.$$

■

Alıştırma 9.2 (Köklü Bir Limit). $\displaystyle\lim_{x\to 0} \frac{\sqrt{x+4}-2}{x}$ limitini SymPy ile hesaplayınız ve sonucu eşlenikle çarparak doğrulayınız.

Çözüm.
Adım 1. Limiti doğrudan hesaplarız.
Adım 2. Payı eşleniği $\sqrt{x+4}+2$ ile çarpınca paydaki köklerin gittiğini görürüz. Böylece kesir $1/(\sqrt{x+4}+2)$ olur ve bunun limiti doğrudan alınır:

```
import sympy as sp

x = sp.symbols("x")
f = (sp.sqrt(x + 4) - 2) / x
print(sp.limit(f, x, 0))
conj = sp.sqrt(x + 4) + 2
print(sp.expand((sp.sqrt(x + 4) - 2) * conj))
print(sp.limit(1 / conj, x, 0))
```

Çıktı:

```
1/4
x
1/4
```

Adım 3. $(\sqrt{x+4}-2)(\sqrt{x+4}+2) = x$ olduğundan $x \neq 0$ için

$$\frac{\sqrt{x+4}-2}{x} = \frac{1}{\sqrt{x+4}+2} \to \frac{1}{2+2} = \frac{1}{4}$$

bulunur. İki yol aynı sonucu verir.
■

Alıştırma 9.3 (Bir Kübik Polinomun Yerel Ekstremumları). $f(x) = x^3 - 3x + 1$ fonksiyonunun yerel ekstremum noktalarını ve değerlerini SymPy ile bulunuz.

Çözüm.
Adım 1. Kritik noktalar $f'(x) = 0$ denkleminin çözümleridir.
Adım 2. Her kritik noktada ikinci türevin işaretine bakarız: $f''(c) < 0$ ise yerel maksimum, $f''(c) > 0$ ise yerel minimum vardır (bkz. Analiz 2).

```
import sympy as sp

x = sp.symbols("x")
f = x**3 - 3*x + 1
f1 = sp.diff(f, x)
f2 = sp.diff(f, x, 2)
critical = sp.solve(f1, x)
print("f'(x) =", f1, "  kritik noktalar:", critical)
for c in critical:
    print(c, "f'' =", f2.subs(x, c), "f =", f.subs(x, c))
```

Çıktı:

 CC BY-NC-SA 4.0

```
f'(x) = 3*x**2 - 3   kritik noktalar: [-1, 1]
-1 f'' = -6 f = 3
1 f'' = 6 f = -1
```

Adım 3. $f'(x) = 3x^2 - 3 = 0$ denkleminin kökleri ± 1'dir. $f''(-1) = -6 < 0$ olduğundan $x = -1$'de yerel maksimum vardır ve değeri $f(-1) = 3$'tür. $f''(1) = 6 > 0$ olduğundan $x = 1$'de yerel minimum vardır ve değeri $f(1) = -1$'dir.
■

Alıştırma 9.4 (Kısmi İntegrasyonlu Bir Belirli İntegral). $\displaystyle\int_0^1 x^2 e^x \, dx$ integralini kesin olarak hesaplayınız ve ondalık değerini veriniz.

Çözüm.
Adım 1. Önce ilkeli, sonra belirli integrali hesaplarız.
Adım 2. Kesin sonucu `evalf` ile ondalığa çevirirız:

```
import sympy as sp

x = sp.symbols("x")
F = sp.integrate(x**2 * sp.exp(x), x)
print(F)
I = sp.integrate(x**2 * sp.exp(x), (x, 0, 1))
print(I, "=", I.evalf())
```

Çıktı:

```
(x**2 - 2*x + 2)*exp(x)
-2 + E = 0.718281828459045
```

Adım 3. İlkel $(x^2 - 2x + 2)e^x$'tir; iki kez kısmi integrasyonla elle de bulunur. Sınırları koyarsak

$$\int_0^1 x^2 e^x \, dx = (1 - 2 + 2)e - 2 = e - 2 \approx 0{,}718281828$$

olur.
■

Alıştırma 9.5 (Basit Kesirlerle İntegral). $\displaystyle\int \frac{3x + 5}{(x - 1)(x + 2)} \, dx$ integralini önce `apart` ile basit kesirlere ayırarak, sonra `integrate` ile hesaplayınız.

Çözüm.
Adım 1. `apart` ile kesri basit kesirlere ayırırız.
Adım 2. İntegrali alır ve bulunan ilkeli türev alarak denetleriz:

```
import sympy as sp

x = sp.symbols("x")
r = (3*x + 5) / ((x - 1) * (x + 2))
print(sp.apart(r))
F = sp.integrate(r, x)
print(F)
print(sp.simplify(sp.diff(F, x) - r))
```

 CC BY-NC-SA 4.0

Çıktı:

```
1/(3*(x + 2)) + 8/(3*(x - 1))
8*log(x - 1)/3 + log(x + 2)/3
0
```

Adım 3. Ayrışım

$$\frac{3x+5}{(x-1)(x+2)} = \frac{8}{3(x-1)} + \frac{1}{3(x+2)}$$

biçimindedir. Gerçekten $x=1$ koyunca $A = 8/3$, $x=-2$ koyunca $B = (-1)/(-3) = 1/3$ bulunur. Buradan

$$\int \frac{3x+5}{(x-1)(x+2)}\,dx = \frac{8}{3}\ln|\,x-1\,| + \frac{1}{3}\ln|\,x+2\,| + C$$

olur. SymPy mutlak değerleri ve sabiti yazmaz; bu farkı İntegral kısmındaki uyarıda görmüştük (bkz. Analiz 2).
■

Alıştırma 9.6 (Logaritmanın Taylor Polinomu). $\ln(1+x)$ fonksiyonunun 0 etrafındaki beşinci Taylor polinomu T_5'i bulunuz. $T_5(0{,}1)$ değerinin $\ln 1{,}1$ sayısından farkını hesaplayıp $0{,}1^6/6$ sınırıyla karşılaştırınız.

Çözüm.
Adım 1. `series` ile x^5 terimine kadar açılımı alırız.
Adım 2. $0{,}1$'i `sp.Rational(1, 10)` olarak koyarız. Böylece $T_5(0{,}1)$ kesin bir kesir olur; hesaplanan fark yuvarlamadan değil, yalnız açılımın kesilmesinden gelir:

```
import sympy as sp

x = sp.symbols("x")
T5 = sp.series(sp.log(1 + x), x, 0, 6).removeO()
print(T5)
h = sp.Rational(1, 10)
approx = T5.subs(x, h)
exact = sp.log(1 + h)
print(approx, "=", approx.evalf(12))
print("ln(1.1) =", exact.evalf(12))
print("hata =", (approx - exact).evalf(5))
print("sınır =", (h**6 / 6).evalf(5))
```

Çıktı:

```
x**5/5 - x**4/4 + x**3/3 - x**2/2 + x
285931/3000000 = 0.0953103333333
ln(1.1) = 0.0953101798043
hata = 1.5353e-7
sınır = 1.6667e-7
```

Adım 3. $T_5(x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5}$'tir. Hata yaklaşık $1{,}535 \cdot 10^{-7}$'dir ve $0{,}1^6/6 \approx 1{,}667 \cdot 10^{-7}$ sınırının altında kalır. $x = 0{,}1$ için açılım alterne bir seridir ve terimlerin mutlak değeri azalır. Bu yüzden hata, atılan ilk terimin mutlak değerinden küçüktür; atılan ilk terim de $-x^6/6$'dır (bkz. Analiz 2).
■

 CC BY-NC-SA 4.0

Alıştırma 9.7 (Teleskopik Bir Toplam). $\displaystyle\sum_{k=1}^{n} \frac{1}{k(k+1)}$ toplamının kapalı biçimini ve $n \to \infty$ iken limitini SymPy ile bulunuz.

Çözüm.
Adım 1. Genel terimi `apart` ile basit kesirlere ayırırız; ikinci argüman `k`, ayrıştırmanın hangi değişkene göre yapılacağını söyler.
Adım 2. Kısmi toplamı `summation` ile hesaplar, sonra limitini alır ve sonsuz toplamla karşılaştırırız:

```
import sympy as sp

k, n = sp.symbols("k n", integer=True, positive=True)
a = 1 / (k * (k + 1))
print(sp.apart(a, k))
S = sp.simplify(sp.summation(a, (k, 1, n)))
print(S)
print(sp.limit(S, n, sp.oo), sp.summation(a, (k, 1, sp.oo)))
```

Çıktı:

```
-1/(k + 1) + 1/k
n/(n + 1)
1 1
```

Adım 3. $\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$ olduğundan toplamda ardışık terimler birbirini götürür:

$$\sum_{k=1}^{n}\left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n+1} = \frac{n}{n+1}.$$

Limiti 1'dir; serinin toplamı da 1'dir (bkz. Analiz 2).
■

Alıştırma 9.8 (Doğrusal Olmayan Bir Sistem). $x^2 + y^2 = 25$, $xy = 12$ sisteminin bütün çözümlerini SymPy ile bulunuz ve doğrulayınız.

Çözüm.
Adım 1. Denklemleri `sp.Eq` ile yazıp `solve`'a veririz. Sözlük seçeneği olmadan `solve` çözümleri (x, y) demetleri olarak döndürür.
Adım 2. Her çözümü iki denkleme koyup hepsinin sağlandığını `all` ile denetleriz:

```
import sympy as sp

x, y = sp.symbols("x y")
eqs = [sp.Eq(x**2 + y**2, 25), sp.Eq(x * y, 12)]
sols = sp.solve(eqs, [x, y])
print(sols)
print(all(e.subs({x: a, y: b}) for a, b in sols for e in eqs))
```

Çıktı:

```
[(-4, -3), (-3, -4), (3, 4), (4, 3)]
True
```

Adım 3. Dört çözüm vardır: $(3,4)$, $(4,3)$, $(-3,-4)$, $(-4,-3)$. Elle de bulunur: $(x+y)^2 = 25 + 24 = 49$ ve $(x-y)^2 = 25 - 24 = 1$ olduğundan $x + y = \pm 7$, $x - y = \pm 1$'dir. Dört işaret seçimi dört çözümü verir.

 CC BY-NC-SA 4.0

■

Alıştırma 9.9 (Köşegenleştirmeyle Matris Kuvveti). $A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$ matrisini köşegenleştirerek A^n için genel bir formül bulunuz ve $n = 10$ için doğrulayınız.

Çözüm.
Adım 1. `diagonalize` ile $A = PDP^{-1}$ ayrışımını buluruz.
Adım 2. $A^n = PD^nP^{-1}$'dir ve köşegen matrisin kuvveti köşegen girdilerin kuvvetidir. n'yi negatif olmayan bir tam sayı sembolü olarak tanımlarsak SymPy formülü sembolik olarak hesaplar:

```
import sympy as sp

n = sp.symbols("n", integer=True, nonnegative=True)
A = sp.Matrix([[2, 1], [1, 2]])
P, D = A.diagonalize()
print(P, D)
An = P * D**n * P.inv()
print(An)
print(An.subs(n, 10) == A**10, A**10)
```

Çıktı:

```
Matrix([[-1, 1], [1, 1]]) Matrix([[1, 0], [0, 3]])
Matrix([[3**n/2 + 1/2, 3**n/2 - 1/2], [3**n/2 - 1/2, 3**n/2 + 1/2]])
True Matrix([[29525, 29524], [29524, 29525]])
```

Adım 3. Özdeğerler 1 ve 3'tür, özvektörler $(-1, 1)$ ve $(1, 1)$'dir. Buradan

$$A^n = \frac{1}{2}\begin{pmatrix} 3^n + 1 & 3^n - 1 \\ 3^n - 1 & 3^n + 1 \end{pmatrix}$$

bulunur. $n = 10$ için $3^{10} = 59049$ olduğundan köşegen girdiler 29525, diğerleri 29524'tür ve doğrudan hesaplanan A^{10} ile aynıdır.
■

Alıştırma 9.10 (Parametreli Bir Matrisin Tersinirliği). $K = \begin{pmatrix} 1 & k & 0 \\ k & 1 & k \\ 0 & k & 1 \end{pmatrix}$ matrisinin hangi k değerleri için tersinir olmadığını SymPy ile bulunuz.

Çözüm.
Adım 1. Bir kare matris ancak ve ancak determinantı sıfırdan farklıysa tersinirdir (bkz. Lineer Cebir). Determinantı k'ye bağlı bir ifade olarak hesaplarız.
Adım 2. Determinantı sıfır yapan k değerlerini `solve` ile bulur, bu değerlerde rankı da denetleriz:

```
import sympy as sp

k = sp.symbols("k")
K = sp.Matrix([[1, k, 0], [k, 1, k], [0, k, 1]])
d = K.det()
print(d)
roots = sp.solve(d, k)
print(roots)
for r in roots:
    print(r, K.subs(k, r).rank())
```

 CC BY-NC-SA 4.0

Çıktı:

```
1 - 2*k**2
[-sqrt(2)/2, sqrt(2)/2]
-sqrt(2)/2 2
sqrt(2)/2 2
```

Adım 3. $\det K = 1 - 2k^2$'dir ve $k = \pm\frac{\sqrt{2}}{2}$ için sıfır olur. Bu iki değerde rank 2'ye düşer, yani K tersinir değildir. Diğer bütün k değerlerinde K tersinirdir.
■

Alıştırma 9.11 (Sembolik Türevle Newton Yöntemi). $x^3 - 2x - 5 = 0$ denkleminin kökünü Newton yöntemiyle $x_0 = 2$'den başlayarak beş adımda hesaplayınız. Türevi `sp.diff` ile bulunuz, fonksiyonları `lambdify` ile sayısallaştırınız ve sonucu `sp.nsolve` ile karşılaştırınız.

Çözüm.
Adım 1. Newton yinelemesi $x_{k+1} = x_k - f(x_k)/f'(x_k)$'dir (bkz. Nümerik Analiz). Türevi elle yazmak yerine SymPy'ye bulduruyoruz.
Adım 2. f ve f''yi `lambdify` ile Python fonksiyonlarına çevirip yinelemeyi sıradan bir döngüyle yaparız:

```
import sympy as sp

x = sp.symbols("x")
expr = x**3 - 2*x - 5
f = sp.lambdify(x, expr, "numpy")
fp = sp.lambdify(x, sp.diff(expr, x), "numpy")
xk = 2.0
for k in range(1, 6):
    xk = xk - f(xk) / fp(xk)
    print(k, xk)
print(sp.nsolve(expr, x, 2))
```

Çıktı:

```
1 2.1
2 2.094568121104185
3 2.094551481698199
4 2.0945514815423265
5 2.0945514815423265
2.09455148154233
```

Adım 3. $f'(x) = 3x^2 - 2$'dir. İlk adım $x_1 = 2 - \frac{-1}{10} = 2{,}1$'dir. Yineleme dördüncü adımda $2{,}0945514815423265$ değerine oturur ve `nsolve` aynı kökü verir. Doğru basamak sayısının her adımda kabaca ikiye katlanması, Newton yönteminin karesel yakınsamasıdır.
■

Bu bölümde SymPy ile kesin hesap yapmayı öğrendik: semboller ve varsayımlar, ifade dönüşümleri, limit, türev, integral, seri, denklem, diferansiyel denklem ve matris hesapları. Sembolik sonuçları da `lambdify` ile NumPy'ye taşıdık. Ancak birçok problemin kapalı biçimde çözümü yoktur. $\cos x = x$ denkleminde gördüğümüz gibi o zaman sayısal yöntemlere başvururuz. Bir sonraki bölüm Kök Bulma ve Optimizasyon, denklemlerin köklerini ve fonksiyonların minimumlarını SciPy ile sayısal olarak bulmayı anlatıyor.

 CC BY-NC-SA 4.0

10. Kök Bulma ve Optimizasyon

Bir gezegenin yörüngedeki konumunu veren Kepler denklemi $E - e\sin E = M$, ya da $\cos x = x$ ve $\tan x = x$ gibi denklemlerin çözümü kapalı bir formülle yazılamaz. Beşinci ve daha yüksek dereceden polinomlar için de genel bir kök formülü yoktur. Bu tür denklemleri sayısal olarak çözeriz: bir başlangıç tahmininden yola çıkar, onu adım adım iyileştirir ve istediğimiz kadar basamağa ulaşınca dururuz.
Optimizasyon, yani bir fonksiyonun en küçük ya da en büyük değerini aramak, kök bulmanın yakın akrabasıdır. Türevlenebilir bir fonksiyonun bir iç noktadaki yerel ekstremumunda türev sıfırdır; dolayısıyla minimum aramak çoğu zaman türevin kökünü aramak demektir. Bir kutunun hacmini en büyük yapan ölçüler, bir noktanın bir eğriye en yakın noktası, bir modelin verilere en iyi uyan parametreleri hep bu türden sorulardır.
Bu bölümde önce ikiye bölme ve Newton yöntemlerini kendimiz yazıyoruz. Yöntemlerin teorisi Nümerik Analiz notlarındadır; burada onların Python'daki karşılığını kuruyoruz. Ardından aynı işleri `scipy.optimize` modülüyle yapıyoruz: tek değişkenli denklemler, denklem sistemleri, tek ve çok değişkenli minimizasyon ve kısıtlı problemler.

10.1 Kökleri Ayırmak: İşaret Değişimi

Bir kökü sayısal olarak aramaya başlamadan önce onun yaklaşık nerede olduğunu bilmemiz gerekir.

Tanım 10.1 (Kök Parantezi). f, $[a, b]$ aralığında sürekli bir fonksiyon olsun. $f(a)$ ile $f(b)$ zıt işaretliyse, yani $f(a)\,f(b) < 0$ ise $[a, b]$ aralığına f için bir **kök parantezi** (bracket) denir.

Yani parantez, içinde en az bir kök bulunduğunu kesin olarak bildiğimiz bir aralıktır. Bunu Bolzano Teoremi garanti eder (bkz. Analiz 1): sürekli bir fonksiyon negatif bir değerden pozitif bir değere geçerken sıfırdan geçmek zorundadır. Parantezin içinde birden çok kök de olabilir; teorem yalnız en az bir kök bulunduğunu söyler.
Parantez bulmanın en kolay yolu, f'yi sık bir ızgarada hesaplayıp ardışık iki değerin işaretine bakmaktır. NumPy ile bu iş tek satırdır: `y[:-1] * y[1:]` dizisi komşu değerlerin çarpımlarını verir ve bir çarpım negatifse o hücrede işaret değişir (Vektörizasyon ve Broadcasting bölümündeki dilimleme ve vektörel işlemler). Örnek olarak $f(x) = x^3 - 3x + 1$ polinomunu $[-2{,}5;\, 2{,}5]$ aralığında, adımı $0{,}5$ olan bir ızgarada tarayalım.

```
import numpy as np


def f(x):
    return x**3 - 3*x + 1


x = np.linspace(-2.5, 2.5, 11)       # adım 0.5 olan ızgara
y = f(x)
# komşu iki noktada f'nin işareti değişiyorsa arada bir kök vardır
i = np.nonzero(y[:-1] * y[1:] < 0)[0]
brackets = list(zip(x[i], x[i + 1]))
for a, b in brackets:
    print(f"[{a}, {b}] aralığında işaret değişiyor")
```

Çıktı:

```
[-2.0, -1.5] aralığında işaret değişiyor
[0.0, 0.5] aralığında işaret değişiyor
[1.5, 2.0] aralığında işaret değişiyor
```

 CC BY-NC-SA 4.0

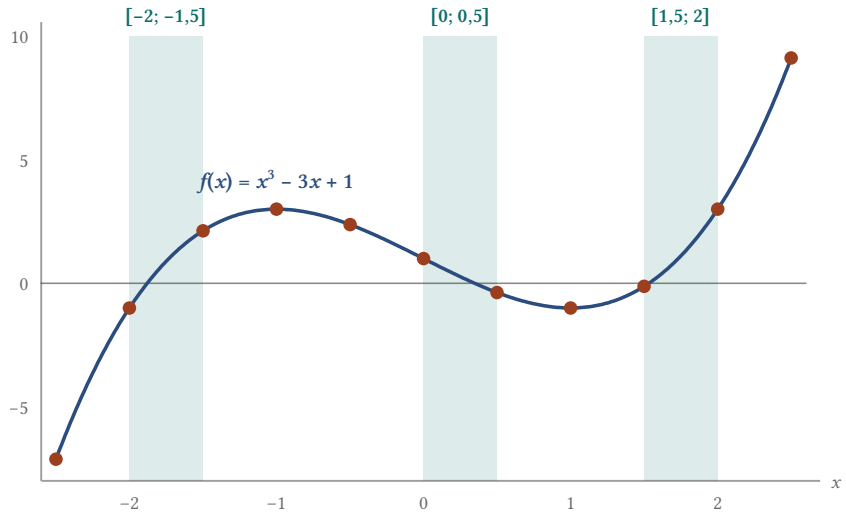
`np.nonzero(...)[0]` koşulun doğru olduğu indisleri verir; `i` indisi, işaretin `x[i]` ile `x[i + 1]` arasında değiştiğini söyler. Taramayı bir grafikle de görmek iyi bir alışkanlıktır. Aşağıdaki kod önceki kodun devamıdır ve Matplotlib ile Grafik Çizimi bölümündeki nesne yönelimli arayüzü kullanır; `axvspan` işaretin değiştiği aralıkları gölgeler.

```
import matplotlib.pyplot as plt

xx = np.linspace(-2.5, 2.5, 401)
fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(xx, f(xx), label="f(x) = x³ − 3x + 1")
ax.plot(x, y, "o", label="ızgara noktaları")
ax.axhline(0, color="gray", linewidth=0.8)
for a, b in brackets:
    ax.axvspan(a, b, alpha=0.2)      # işaret değişen aralıklar
ax.set_xlabel("x")
ax.legend()
fig.savefig("isaret_tarama.png", dpi=150)
print("grafik isaret_tarama.png dosyasına kaydedildi")
```

Çıktı:

```
grafik isaret_tarama.png dosyasına kaydedildi
```



Şekil 10.1: $f(x) = x^3 - 3x + 1$ fonksiyonunun grafiği ve adımı 0,5 olan ızgaranın 11 noktası; Matplotlib kodunun çizdiği grafiğin aynısı. Komşu iki ızgara noktasında işaretin değiştiği üç aralık gölgelidir: her birinde bir kök vardır.

Üç parantez bulduk. Kübik bir polinomun en çok üç kökü olduğundan bütün kökler bu aralıklardadır. Bu polinomun köklerini kapalı biçimde de biliyoruz: $x = 2\cos\theta$ yazılırsa $x^3 - 3x = 2(4\cos^3\theta - 3\cos\theta)$ olur ve bu ifade $2\cos 3\theta$'ya eşittir. Denklem $\cos 3\theta = -\tfrac{1}{2}$ hâline gelir ve kökler

$$2\cos\frac{2\pi}{9} \approx 1{,}532, \qquad 2\cos\frac{4\pi}{9} \approx 0{,}347, \qquad 2\cos\frac{8\pi}{9} \approx -1{,}879$$

bulunur. Bu kesin değerler, bölüm boyunca yöntemlerin gerçek hatasını ölçmemizi sağlayacak.

 CC BY-NC-SA 4.0

⚠ Izgara kökleri kaçırabilir

İşaret taraması iki durumda kök kaçırır. Birincisi, iki kök aynı hücreye düşerse: $(x-0{,}1)(x-0{,}3)$ fonksiyonunun iki kökü de $[0;0{,}5]$ hücresindedir, ama $f(0)=0{,}03$ ve $f(0{,}5)=0{,}08$ aynı işaretli olduğundan tarama hiçbir şey bulmaz. İkincisi, çift katlı köklerde: $(x-1)^2$ fonksiyonu $x=1$'de sıfır olur ama işaret değiştirmez. Izgarayı sıklaştırmak ilk sorunu hafifletir, ikincisini çözmez. Bir de kök tam bir ızgara noktasına denk gelirse çarpım 0 olur ve `< 0` testi onu atlar. Grafiğe bakmak her durumda iyi bir kontroldür.

10.2 İkiye Bölme Yöntemini Yazmak

Elimizde bir parantez varsa onu sürekli yarıya indirerek köke istediğimiz kadar yaklaşabiliriz. Yöntemin ayrıntılı anlatımı Nümerik Analiz notlarındadır; burada onu bir Python fonksiyonuna çeviriyoruz.

💡 İkiye bölme dört adımda

1. $f(a)\,f(b)<0$ olduğunu denetle; değilse dur, çünkü parantez yoktur.
2. Orta nokta $p=(a+b)/2$'yi ve $f(p)$'yi hesapla.
3. $f(a)$ ile $f(p)$ zıt işaretliyse kök $[a,p]$ aralığındadır, $b=p$ al; değilse kök $[p,b]$ aralığındadır, $a=p$ al.
4. Yarı uzunluk $(b-a)/2$ toleransın altına inene kadar 2. ve 3. adımları tekrarla; son orta noktayı döndür.

Bu tarifi $f(x)=x^3-3x+1$ polinomunun $[1{,}5;\,2]$ parantezindeki köküne, 10^{-10} toleransla uygulayalım.

```
import math


def bisection(f, a, b, tol=1e-10, max_iter=100):
    """f'nin [a, b] parantezindeki bir kökünü ikiye bölmeyle bulur.

    (p, n) döndürür: p yaklaşık kök, n hesaplanan orta nokta sayısı.
    """
    fa = f(a)
    if fa * f(b) > 0:
        raise ValueError("f(a) ile f(b) aynı işaretli: parantez yok")
    for n in range(1, max_iter + 1):
        p = (a + b) / 2
        fp = f(p)
        if fp == 0 or (b - a) / 2 < tol:
            return p, n
        if fa * fp < 0:          # kök [a, p] içinde
            b = p
        else:                    # kök [p, b] içinde
            a, fa = p, fp
    raise RuntimeError("istenen hassasiyete ulaşılamadı")


def f(x):
    return x**3 - 3*x + 1


p, n = bisection(f, 1.5, 2.0)
exact = 2 * math.cos(2 * math.pi / 9)
print(f"kök ≈ {p:.12f}, {n} orta nokta")
print(f"gerçek hata: {abs(p - exact):.1e}")
print("öngörülen adım sayısı:", math.ceil(math.log2(0.5 / 1e-10)))
```

Çıktı:

 CC BY-NC-SA 4.0

```
kök ≈ 1.532088886190, 33 orta nokta
gerçek hata: 4.8e-11
öngörülen adım sayısı: 33
```

Fonksiyon parantez koşulunu en başta denetler ve sağlanmıyorsa bir `ValueError` fırlatır (Sınıflar, Hata Yakalama ve Dosyalar). `fa` değerini saklayarak her adımda f'yi yalnız bir kez çağırırız; f'nin hesabı pahalıysa asıl maliyet çağrı sayısıdır. `max_iter` sınırı, bir hata yüzünden sonsuz döngüye girmeyi önler. Kodun son satırı, kaç orta nokta gerektiğini önceden hesaplıyor. Bunu şu sınır sağlar.

Önerme 10.1 (İkiye Bölme Yönteminin Hata Sınırı). *$f \in C[a,b]$ ve $f(a)\,f(b) < 0$ olsun. İkiye bölme yönteminin n-inci orta noktası p_n ise f'nin $[a,b]$ aralığındaki bir p kökü için*

$$\mid p_n - p \mid \leq \frac{b-a}{2^n}$$

olur.

İspatı için bkz. Nümerik Analiz. Hatanın bir ε toleransından küçük olması için $\frac{b-a}{2^n} < \varepsilon$, yani $n > \log_2 \frac{b-a}{\varepsilon}$ yeterlidir. Örneğimizde $b-a = 0{,}5$ ve $\varepsilon = 10^{-10}$ olduğundan $n > \log_2(5 \cdot 10^9) \approx 32{,}2$, yani $n = 33$ çıkar. Fonksiyonun saydığı orta nokta sayısı da tam budur ve gerçek hata $4{,}8 \cdot 10^{-11}$ ile sınırın altında kalır.

10.3 Newton Yöntemini Yazmak

İkiye bölme güvenilirdir ama yavaştır: hata sınırı her adımda yalnız yarıya iner. Fonksiyonun türevini de kullanırsak çok daha hızlı ilerleyebiliriz. Kontrol Yapıları ve Fonksiyonlar bölümünde karekökü hesaplarken kullandığımız iterasyon, aslında Newton yönteminin $x^2 - a = 0$ denklemine uygulanmasıydı. Genel biçimde yöntem

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

kuralıyla çalışır: f yerine onun x_n noktasındaki teğetini koyar ve teğetin kökünü bir sonraki yaklaşım olarak alır (bkz. Nümerik Analiz). Yöntemlerin hızını karşılaştırabilmek için önce hız kavramını kesinleştirelim.

Tanım 10.2 (Yakınsama Mertebesi). (x_n) dizisi p'ye yakınsasın ve $e_n = \mid x_n - p \mid$ olsun. Bir $\alpha \geq 1$ ve bir $\lambda > 0$ sabiti için

$$\lim_{n \to \infty} \frac{e_{n+1}}{e_n^{\alpha}} = \lambda$$

ise dizi p'ye **α mertebesinden** yakınsar denir. $\alpha = 1$ ise (bu durumda $\lambda < 1$ istenir) yakınsama **doğrusal**, $\alpha = 2$ ise **karesel**dir.

Yani doğrusal yakınsamada hata her adımda kabaca sabit bir oranla küçülür ve doğru basamak sayısı her adımda sabit bir miktar artar. Karesel yakınsamada ise yeni hata eski hatanın karesiyle orantılıdır; hata 10^{-4} iken bir sonraki adımda 10^{-8} civarına iner, yani doğru basamak sayısı her adımda yaklaşık iki katına çıkar.

Teorem 10.1 (Newton Yönteminin Karesel Yakınsaması). *f, p'yi içeren bir aralıkta iki kez sürekli türevlenebilir, $f(p) = 0$ ve $f'(p) \neq 0$ olsun. x_0, p'ye yeterince yakın seçilirse Newton dizisi p'ye yakınsar ve hiçbir adımda $x_n = p$ olmuyorsa*

$$\lim_{n \to \infty} \frac{\mid x_{n+1} - p \mid}{\mid x_n - p \mid^2} = \left| \frac{f''(p)}{2f'(p)} \right|$$

olur. Dolayısıyla $f''(p) \neq 0$ ise yakınsama karesel, $f''(p) = 0$ ise karesel yakınsamadan da hızlıdır.

> ***İspat.***

 CC BY-NC-SA 4.0

Yakınsama Nümerik Analiz notlarında ispatlanmıştır; burada yalnız hızı gösteriyoruz. Taylor Teoremi'ne göre p ile x_n arasında bir ξ_n için

$$0 = f(p) = f(x_n) + f'(x_n)(p - x_n) + \frac{f''(\xi_n)}{2}(p - x_n)^2$$

olur. Bu eşitliği $f'(x_n)$'e bölüp $\frac{f(x_n)}{f'(x_n)} = x_n - x_{n+1}$ eşitliğini kullanırsak

$$x_{n+1} - p = \frac{f''(\xi_n)}{2f'(x_n)}\,(x_n - p)^2$$

bulunur. $x_n \to p$ olduğundan $\xi_n \to p$'dir. f' ile f'' sürekli olduğundan $f'(x_n) \to f'(p)$ ve $f''(\xi_n) \to f''(p)$ olur. Her iki tarafın mutlak değerini $|\,x_n - p\,|^2$'ye bölüp limite geçmek iddiayı verir. ■

Teoremi kodla gözleyelim. Aşağıdaki `newton_raphson` fonksiyonu bütün iterasyonları bir listede saklar; böylece her adımın hatasını gerçek kök $p = 2\cos\frac{2\pi}{9}$ ile karşılaştırabiliriz.

```
import math


def newton_raphson(f, df, x0, tol=1e-12, max_iter=50):
    """Newton yöntemi: (yaklaşık kök, bütün iterasyonlar) döndürür."""
    xs = [x0]
    for _ in range(max_iter):
        x = xs[-1]
        d = df(x)
        if d == 0:
            raise ZeroDivisionError(f"x = {x} noktasında türev sıfır")
        xs.append(x - f(x) / d)
        if abs(xs[-1] - x) < tol:
            return xs[-1], xs
    raise RuntimeError("Newton yöntemi yakınsamadı")


def f(x):
    return x**3 - 3*x + 1


def df(x):
    return 3*x**2 - 3


p = 2 * math.cos(2 * math.pi / 9)          # gerçek kök
root, xs = newton_raphson(f, df, 2.0)
e = [abs(x - p) for x in xs]
for n, (x, err) in enumerate(zip(xs, e)):
    print(f"n={n}  x_n={x:.15f}  hata={err:.1e}")
for n in range(1, 4):
    print(f"e_{n+1} / e_{n}^2 = {e[n+1] / e[n]**2:.4f}")
print("öngörülen oran:", round(p / (p**2 - 1), 4))
```

Çıktı:

```
n=0  x_n=2.000000000000000  hata=4.7e-01
n=1  x_n=1.666666666666667  hata=1.3e-01
n=2  x_n=1.548611111111111  hata=1.7e-02
n=3  x_n=1.532390161865380  hata=3.0e-04
n=4  x_n=1.532088989397224  hata=1.0e-07
n=5  x_n=1.532088886237968  hata=1.2e-14
n=6  x_n=1.532088886237956  hata=0.0e+00
```

 CC BY-NC-SA 4.0

```
e_2 / e_1^2 = 0.9123
e_3 / e_2^2 = 1.1036
e_4 / e_3^2 = 1.1365
öngörülen oran: 1.1372
```

Hata sütunu 10^{-1}, 10^{-2}, 10^{-4}, 10^{-7}, 10^{-14} mertebelerinden geçiyor: üs her adımda yaklaşık iki katına çıkıyor. Altıncı adımda hata 0'dır: x_6, gerçek kökün `float` olarak hesaplanan değerine eşittir ve makine duyarlığına ulaşılmıştır. e_{n+1}/e_n^2 oranları da teoremin söylediği sabite yaklaşıyor: $f''(x) = 6x$ ve $f'(x) = 3x^2 - 3$ olduğundan

$$\left|\frac{f''(p)}{2f'(p)}\right| = \frac{6p}{2(3p^2-3)} = \frac{p}{p^2-1} \approx 1{,}1372$$

bulunur.

 CC BY-NC-SA 4.0

⚠ Newton her zaman yakınsamaz

Teorem 10.1, x_0'ın köke *yeterince yakın* olmasını ister. Uzak bir başlangıçta yöntem döngüye girebilir ya da ıraksayabilir. Aşağıdaki kodun ilk yarısı $f(x) = x^3 - 2x + 2$ için $x_0 = 0$'dan, ikinci yarısı $f(x) = \arctan x$ için $x_0 = 1{,}5$'ten başlar.

```
import math


def f(x):
    return x**3 - 2*x + 2


def df(x):
    return 3*x**2 - 2

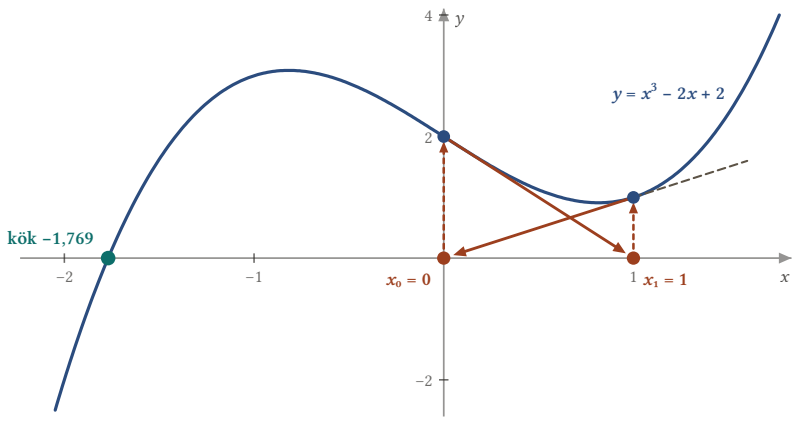

x = 0.0
for n in range(1, 5):
    x = x - f(x) / df(x)
    print(f"x_{n} = {x}")

x = 1.5                                   # arctan için Newton
for n in range(1, 6):
    x = x - math.atan(x) * (1 + x**2)
    print(f"x_{n} = {x:.6g}")
```

Çıktı:

```
x_1 = 1.0
x_2 = 0.0
x_3 = 1.0
x_4 = 0.0
x_1 = -1.69408
x_2 = 2.32113
x_3 = -5.11409
x_4 = 32.2957
x_5 = -1575.32
```

İlk denklemde $x_1 = 0 - \frac{2}{-2} = 1$ ve $x_2 = 1 - \frac{1}{1} = 0$ olduğundan iterasyon 0 ile 1 arasında sonsuza kadar döner. İkincisinde arctan'ın türevi $\frac{1}{1+x^2}$ uzaklarda çok küçük olduğundan teğetler neredeyse yataydır ve her adım bir öncekinden daha uzağa savrulur. ($|\, x_0 \,|$, yaklaşık 1,3917452 olan bir eşiğin üstündeyse iterasyon ıraksar, altındaysa 0'a yakınsar.)



Şekil 10.2: $f(x) = x^3 - 2x + 2$ için Newton yöntemi $x_0 = 0$'dan başlayınca: 0'daki teğet ekseni 1'de, 1'deki teğet ekseni yeniden 0'da keser. İterasyon 0 ile 1 arasında döner ve −1,769 civarındaki köke hiç yaklaşmaz.

 CC BY-NC-SA 4.0

Çare, Newton'u bir parantezle desteklemektir; aşağıda tanıtacağımız `brentq` tam bunu yapar.

10.4 SciPy ile Tek Değişkenli Denklemler

Kendi fonksiyonlarımız yöntemleri anlamak için değerlidir; gerçek işte ise `scipy.optimize` modülünün test edilmiş ve kenar durumları düşünülmüş fonksiyonlarını kullanırız. Tek değişkenli denklemler için üç araç tanıyacak, sonra yöntemlerin hızlarını karşılaştıracağız.

10.4.1 `brentq`: parantezli ve hızlı

`brentq(f, a, b)` çağrısı Brent yöntemini uygular. Yöntem kökü hiçbir zaman parantezin dışına bırakmaz, ama mümkün olduğunda yarıya bölmek yerine sekant ve ters karesel interpolasyon adımları atar. Böylece ikiye bölmenin güvenilirliğini sekant yönteminin hızıyla birleştirir. Durma toleransları `xtol` (varsayılan $2 \cdot 10^{-12}$) ve `rtol` (varsayılan değeri makine epsilonunun dört katı, yaklaşık $8{,}9 \cdot 10^{-16}$) parametreleridir. `full_output=True` verilirse kökün yanında, iterasyon ve fonksiyon çağrısı sayılarını taşıyan bir `RootResults` nesnesi de döner.

```
import numpy as np
from scipy.optimize import brentq


def f(x):
    return x**3 - 3*x + 1


brackets = [(-2.0, -1.5), (0.0, 0.5), (1.5, 2.0)]
exact = 2 * np.cos(2 * np.array([4, 2, 1]) * np.pi / 9)
for (a, b), p in zip(brackets, exact):
    r, info = brentq(f, a, b, full_output=True)
    print(f"[{a}, {b}]: kök = {r:.15f}, hata = {abs(r - p):.1e}, "
          f"{info.function_calls} f çağrısı")
```

Çıktı:

```
[-2.0, -1.5]: kök = -1.879385241571423, hata = 3.9e-13, 8 f çağrısı
[0.0, 0.5]: kök = 0.347296355333861, hata = 1.1e-16, 8 f çağrısı
[1.5, 2.0]: kök = 1.532088886237956, hata = 0.0e+00, 8 f çağrısı
```

Üç kökün her biri yalnız 8 fonksiyon çağrısıyla bulundu. Kendi ikiye bölme fonksiyonumuz 10^{-10} hassasiyet için iki uç nokta ve 33 orta nokta, toplam 35 çağrı harcamıştı. İlk kökteki $3{,}9 \cdot 10^{-13}$ hata `xtol` toleransının altındadır; daha fazla basamak gerekirse `xtol` küçültülebilir.

10.4.2 `root_scalar`: bütün yöntemlere tek kapı

`root_scalar` fonksiyonu farklı yöntemlere tek bir arayüzden ulaştırır. Parantezli yöntemler (`bisect`, `brentq`) `bracket` ister, açık yöntemler ise bir başlangıç noktası `x0` ister. Sekant yöntemi isteğe bağlı ikinci bir nokta `x1` de alır; verilmezse SciPy onu `x0`'ın hemen yanında kendisi seçer. Newton yönteminde türev `fprime` ile verilir; verilmezse SciPy türevi sonlu farklarla yaklaşık hesaplar. Halley yöntemi ise hem `fprime` hem de ikinci türev `fprime2` ister. Aşağıda beş yöntemi aynı köke uyguluyoruz. `root_scalar(f, **kw)` yazımı sözlükteki anahtarları isimli argümanlar olarak geçirir; örneğin ilk satır `root_scalar(f, method="bisect", bracket=[1.5, 2.0])` çağrısıyla aynıdır.

```
from scipy.optimize import root_scalar


def f(x):
    return x**3 - 3*x + 1


def df(x):
    return 3*x**2 - 3
```

 CC BY-NC-SA 4.0

```
def d2f(x):
    return 6*x


# her yöntemin istediği bilgiler bir sözlükte
calls = [
    {"method": "bisect", "bracket": [1.5, 2.0]},
    {"method": "brentq", "bracket": [1.5, 2.0]},
    {"method": "secant", "x0": 2.0, "x1": 1.9},
    {"method": "newton", "x0": 2.0, "fprime": df},
    {"method": "halley", "x0": 2.0, "fprime": df, "fprime2": d2f},
]
for kw in calls:
    sol = root_scalar(f, **kw)
    print(f"{kw['method']:7s} kök={sol.root:.15f}  "
          f"iterasyon={sol.iterations:2d}  çağrı={sol.function_calls}")
```

Çıktı:

```
bisect  kök=1.532088886238853  iterasyon=38  çağrı=40
brentq  kök=1.532088886237956  iterasyon= 7  çağrı=8
secant  kök=1.532088886237956  iterasyon= 7  çağrı=8
newton  kök=1.532088886237956  iterasyon= 6  çağrı=12
halley  kök=1.532088886237956  iterasyon= 4  çağrı=12
```

SciPy'ın `bisect` yöntemi varsayılan `xtol` $= 2 \cdot 10^{-12}$ toleransına inmek için 38 adım atıyor. Newton 6 iterasyonda bitiyor, ama her iterasyonda hem f'yi hem f''yü çağırdığından 12 çağrı yapıyor. Halley yöntemi ikinci türevi de kullanır ve üçüncü mertebeden yakınsar; 4 iterasyonda, her birinde üç çağrıyla biter. Hangi yöntemin "ucuz" olduğu, türevleri hesaplamanın f'yi hesaplamaya göre ne kadar pahalı olduğuna bağlıdır.

10.4.3 `newton`: birçok başlangıç noktası birden

`scipy.optimize.newton(f, x0, fprime=None, fprime2=None)` fonksiyonu türev verilmezse sekant, `fprime` verilirse Newton, ikisi de verilirse Halley yöntemini kullanır. Asıl gücü şudur: `x0` bir NumPy dizisi olursa her bileşen için ayrı bir Newton iterasyonu, hepsi birlikte ve vektörel olarak yürütülür.

```
import numpy as np
from scipy.optimize import newton


def f(x):
    return x**3 - 3*x + 1


def df(x):
    return 3*x**2 - 3


print(newton(f, 2.0, fprime=df))          # tek başlangıç noktası
x0 = np.linspace(-3, 3, 12)               # 12 başlangıç noktası birden
roots = newton(f, x0, fprime=df)
for a, r in zip(x0, roots):
    print(f"x0 = {a:6.3f}  ->  {r:9.6f}")
```

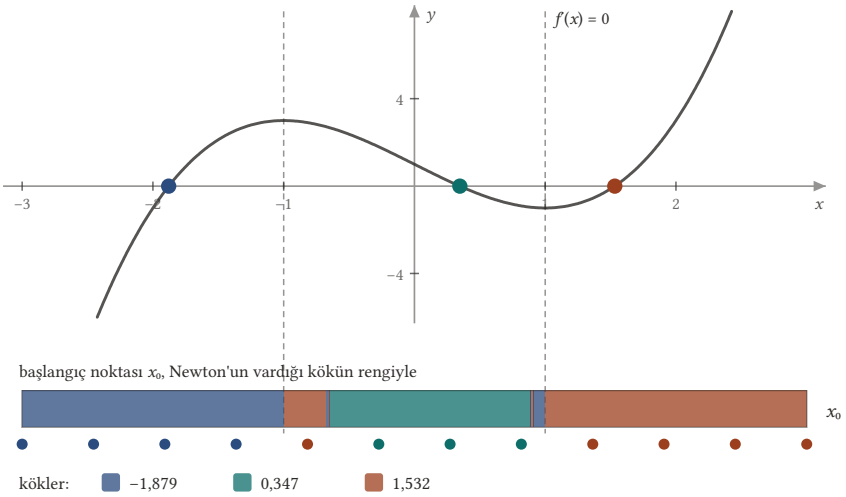
Çıktı:

```
1.532088886237956
x0 = -3.000  ->  -1.879385
x0 = -2.455  ->  -1.879385
x0 = -1.909  ->  -1.879385
x0 = -1.364  ->  -1.879385
```

 CC BY-NC-SA 4.0

```
x0 = -0.818  ->   1.532089
x0 = -0.273  ->   0.347296
x0 =  0.273  ->   0.347296
x0 =  0.818  ->   0.347296
x0 =  1.364  ->   1.532089
x0 =  1.909  ->   1.532089
x0 =  2.455  ->   1.532089
x0 =  3.000  ->   1.532089
```

Başlangıç noktalarının çoğu kendisine en yakın köke gidiyor, ama $x_0 = -0{,}818$ en yakın kök olan $-1{,}879$'a değil, en uzaktaki $1{,}532$'ye gidiyor. Sebep, bu noktanın $f'(x) = 0$ olan $x = -1$'e yakın olmasıdır: orada teğet neredeyse yataydır ve x eksenini çok uzakta keser. Aynı kodu $[-3; 3]$ aralığındaki 1201 noktalık bir ızgarada (türevin sıfır olduğu ve Newton'un hiç başlayamadığı $x = \pm 1$ noktaları çıkarılarak) çalıştırıp her başlangıç noktasını vardığı kökün rengine boyarsak aşağıdaki şekli elde ederiz.



Şekil 10.3: Üstte $f(x) = x^3 - 3x + 1$ ve üç kökü. Alttaki şeritte $[-3; 3]$ aralığındaki her x_0 başlangıç noktası, `newton` fonksiyonunun oradan vardığı kökün rengine boyanmıştır (1201 noktalık ızgara); şeridin altındaki 12 nokta, metindeki kodun başlangıç noktalarıdır. Türevin sıfır olduğu $x = \pm 1$ yakınında renkler birbirine karışır.

Newton yönteminde başlangıç noktası yalnız yakınsayıp yakınsamamayı değil, hangi köke yakınsandığını da belirler. Belirli bir kökü istiyorsak parantezli bir yöntem kullanmak en güvenlisidir.

10.4.4 Yakınsama hızlarının karşılaştırılması

Tanım 10.2 ile tanımladığımız hızı üç yöntem için yan yana görelim. Aşağıdaki kod ikiye bölmenin, sekantın ($x_0 = 2, x_1 = 1{,}9$) ve Newton'un ($x_0 = 2$) ilk altı adımının hatalarını, yine $p = 2\cos\frac{2\pi}{9}$ köküne göre yazdırır. Sekant yöntemi Newton'un türevi yerine son iki noktadan geçen kirişin eğimini kullanır (bkz. Nümerik Analiz). İki başlangıç noktasıyla başladığından sekantın n-inci adımı x_{n+1}'i üretir; bu yüzden kod `sec[n + 1]` değerini yazdırır ve tablonun her satırında üç yöntemin de n adım sonraki hatası yer alır.

```
import math


def f(x):
    return x**3 - 3*x + 1


def df(x):
```

 CC BY-NC-SA 4.0

```
    return 3*x**2 - 3


p = 2 * math.cos(2 * math.pi / 9)          # gerçek kök

bis, a, b = [], 1.5, 2.0                   # ikiye bölme: orta noktalar
for _ in range(6):
    m = (a + b) / 2
    bis.append(m)
    if f(a) * f(m) < 0:
        b = m
    else:
        a = m

sec = [2.0, 1.9]                           # sekant: x0 = 2, x1 = 1.9
for _ in range(6):
    u, v = sec[-2], sec[-1]
    sec.append(v - f(v) * (v - u) / (f(v) - f(u)))

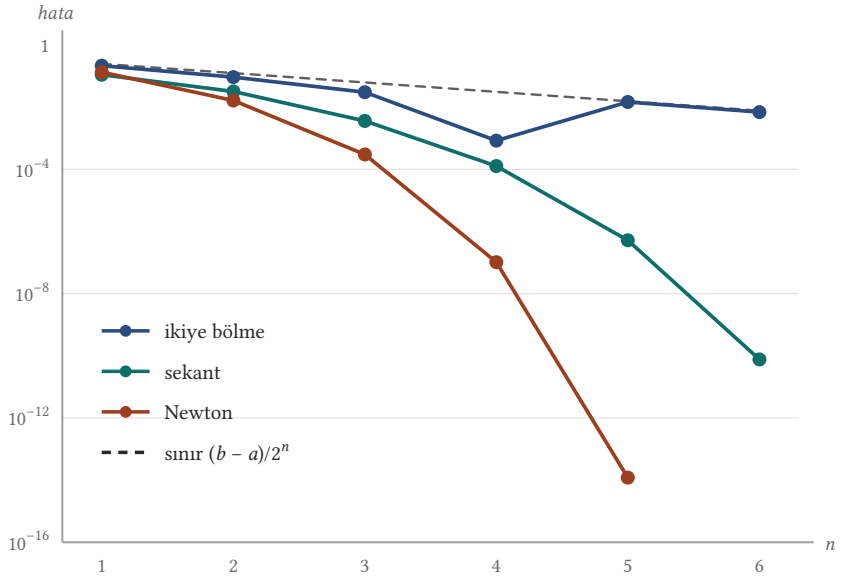
nwt = [2.0]                                # Newton: x0 = 2
for _ in range(6):
    x = nwt[-1]
    nwt.append(x - f(x) / df(x))

print(" n  ikiye bölme    sekant    Newton")
for n in range(1, 7):
    print(f"{n:2d}  {abs(bis[n - 1] - p):9.1e}  {abs(sec[n + 1] - p):9.1e}"
          f"  {abs(nwt[n] - p):9.1e}")
```

Çıktı:

```
 n  ikiye bölme    sekant    Newton
 1    2.2e-01    1.1e-01    1.3e-01
 2    9.3e-02    3.2e-02    1.7e-02
 3    3.0e-02    3.6e-03    3.0e-04
 4    8.4e-04    1.3e-04    1.0e-07
 5    1.5e-02    5.2e-07    1.2e-14
 6    7.0e-03    7.6e-11    0.0e+00
```

 CC BY-NC-SA 4.0



Şekil 10.4: Aynı kök ($x^3 - 3x + 1 = 0$'ın 1,532 civarındaki kökü) için üç yöntemin n adım sonraki hatası, logaritmik ölçekte; tablodaki sayıların aynısı. İkiye bölmenin hatası sınırın altında kalır ama inişi yavaştır; sekantın ve Newton'un eğrisi giderek dikleşir. Newton'un hatası 0 olan 6. adımı çizilmedi.

Logaritmik eksende doğrusal yakınsama bir doğru, daha yüksek mertebeden yakınsama ise giderek dikleşen bir eğri verir. İkiye bölmenin hatası tek tek adımlarda büyüyebilir (4. adımda orta nokta tesadüfen köke çok yakın düşmüştür), ama hiçbir zaman Önerme 10.1 sınırını aşmaz ve sınır her adımda yarıya iner. Sekant yönteminin mertebesi altın oran $\frac{1+\sqrt{5}}{2} \approx 1{,}618$'dir; Newton'un mertebesi ise 2'dir.

Örnek 10.1 (Kepler Denklemi). Bir gezegenin yörüngedeki konumu, yörüngenin dış merkezliği e ve ortalama anomali M verildiğinde Kepler denkleminin

$$E - e\sin E = M$$

çözümü olan E açısından (dış merkez anomaliden) hesaplanır. $e = 0{,}5$ için $M = 0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}, \pi$ değerlerine karşılık gelen E değerlerini tek bir `newton` çağrısıyla bulunuz.

Çözüm.
Denklemi kök problemine çevirmek. $g(E) = E - e\sin E - M$ diyelim. $g'(E) = 1 - e\cos E$ olduğundan $g'(E) \geq 1 - e = 0{,}5 > 0$'dır ve g kesin artandır; her M için tek bir kök vardır ve Newton yönteminde türev hiçbir zaman sıfır olmaz.
Başlangıç noktası. $e\sin E$ terimi en çok $0{,}5$ olduğundan E, M'den fazla uzaklaşmaz; doğal başlangıç $E_0 = M$'dir.
Kod. `kepler` fonksiyonu `M` dizisini kullandığından dizi hâlindeki bir `E` için bütün hesaplar bileşen bileşen yapılır; `newton` her bileşene kendi iterasyonunu uygular.

```
import numpy as np
from scipy.optimize import newton

e = 0.5                                  # dış merkezlik
M = np.linspace(0, np.pi, 5)             # ortalama anomaliler


def kepler(E):
    return E - e * np.sin(E) - M


def dkepler(E):
```

```
    return 1 - e * np.cos(E)


E = newton(kepler, M, fprime=dkepler)     # başlangıç: E0 = M
for m, val in zip(M, E):
    print(f"M = {m:.6f}  ->  E = {val:.12f}")
print(f"en büyük artık: {np.max(np.abs(kepler(E))):.1e}")
```

Çıktı:

```
M = 0.000000  ->  E = 0.000000000000
M = 0.785398  ->  E = 1.261703055253
M = 1.570796  ->  E = 2.020979938090
M = 2.356194  ->  E = 2.609753954400
M = 3.141593  ->  E = 3.141592653590
en büyük artık: 2.2e-16
```

Kontrol. $M = 0$ ve $M = \pi$ için $\sin E = 0$ olduğundan $E = M$ kesin çözümdür ve kod da bunları verdi. $M = \frac{\pi}{2}$ için

$$2{,}020980 - 0{,}5\sin(2{,}020980) \approx 1{,}570796 \approx \frac{\pi}{2}$$

bulunur. Bütün denklemlerin artığı $2{,}2 \cdot 10^{-16}$ civarındadır. ■

10.5 Denklem Sistemleri

Birden çok bilinmeyenli denklem sistemlerinde de aynı fikir işler; türevin yerini Jacobi matrisi alır. $F : \mathbb{R}^n \to \mathbb{R}^n$ için $F(\mathbf{x}) = \mathbf{0}$ sistemini çözmek istiyoruz. F'nin $\mathbf{x}_k$ noktasındaki doğrusal yaklaşımı $F(\mathbf{x}_k) + J(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k)$'dir; burada J, F'nin Jacobi matrisidir (bkz. Analiz 4). Bu yaklaşımı sıfır yapan nokta bir sonraki iterasyondur:

$$J(\mathbf{x}_k)\,\mathbf{s}_k = -F(\mathbf{x}_k), \qquad \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k.$$

Her adımda ters matris hesaplamak yerine bir lineer sistem çözeriz; NumPy ile Lineer Cebir bölümünde `solve`'un `inv`'den neden daha iyi olduğunu görmüştük. Örnek olarak

$$x^2 + y^2 = 4, \qquad e^x + y = 1$$

sistemini ele alalım. $F(x, y) = (x^2 + y^2 - 4,\, e^x + y - 1)$ için Jacobi matrisi

$$J(x, y) = \begin{pmatrix} 2x & 2y \\ e^x & 1 \end{pmatrix}$$

olur. $(1, -1)$ noktasından başlayalım.

```
import numpy as np


def F(v):
    x, y = v
    return np.array([x**2 + y**2 - 4, np.exp(x) + y - 1])


def J(v):
    x, y = v
    return np.array([[2*x, 2*y],
                     [np.exp(x), 1.0]])
```

```
v = np.array([1.0, -1.0])                 # başlangıç noktası
for k in range(1, 7):
    s = np.linalg.solve(J(v), -F(v))      # J s = -F sistemini çöz
    v = v + s
    print(f"{k}: x = {v[0]:.12f}, y = {v[1]:.12f}, "
          f"|s| = {np.linalg.norm(s):.1e}")
```

Çıktı:

```
1: x = 1.075765685480, y = -1.924234314520, |s| = 9.3e-01
2: x = 1.009470792611, y = -1.737844860783, |s| = 2.0e-01
3: x = 1.004188579849, y = -1.729653231335, |s| = 9.7e-03
4: x = 1.004168738694, y = -1.729637287086, |s| = 2.5e-05
5: x = 1.004168738475, y = -1.729637287026, |s| = 2.3e-10
6: x = 1.004168738475, y = -1.729637287026, |s| = 0.0e+00
```

Adım uzunlukları $|\,\mathbf{s}_k\,|$ tek değişkenli durumdaki gibi karesel hızla küçülüyor: 10^{-2}, 10^{-5}, 10^{-10}. SciPy'da aynı işi `fsolve` ve `root` yapar. `fsolve(F, x0, fprime=J)`, Newton yöntemini uzak başlangıçlarda da sağlam kalacak biçimde değiştiren "hybrid" (Powell) yöntemini kullanır; Jacobi matrisi verilmezse onu sonlu farklarla kendisi hesaplar. `root(F, x0, jac=J)` aynı yöntemi daha düzenli bir arayüzle sunar ve sonucu, başarının ve fonksiyon çağrısı sayısının da içinde olduğu bir `OptimizeResult` nesnesi olarak döndürür.

```
import numpy as np
from scipy.optimize import fsolve, root


def F(v):
    x, y = v
    return np.array([x**2 + y**2 - 4, np.exp(x) + y - 1])


def J(v):
    x, y = v
    return np.array([[2*x, 2*y],
                     [np.exp(x), 1.0]])

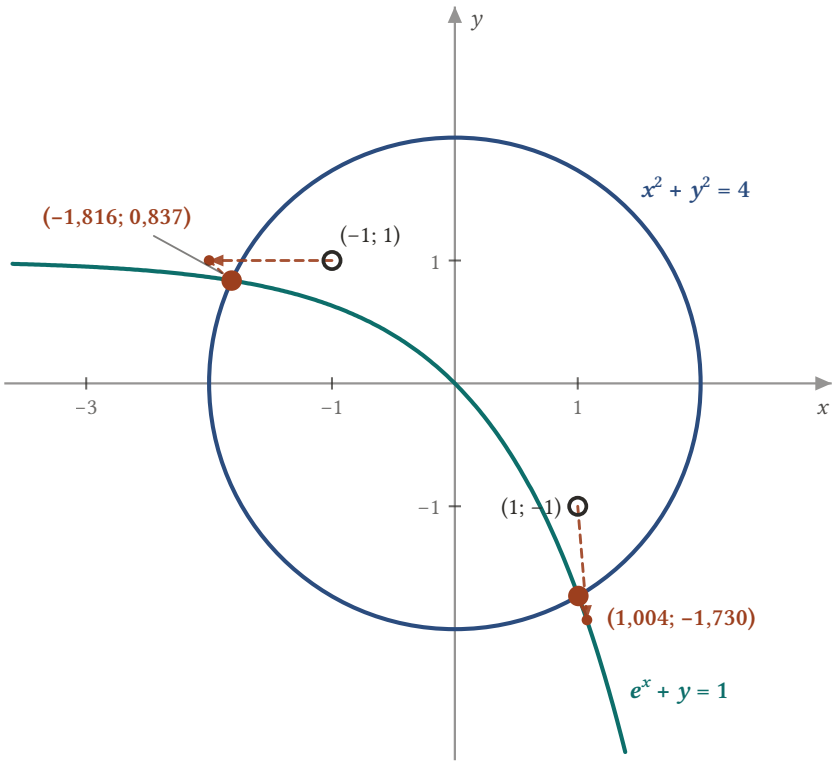

for start in ([1.0, -1.0], [-1.0, 1.0]):
    v = fsolve(F, start, fprime=J)
    print(f"fsolve, başlangıç {start}: x = {v[0]:.12f}, "
          f"y = {v[1]:.12f}")

sol = root(F, [-1.0, 1.0], jac=J)         # varsayılan yöntem: "hybr"
print(sol.success, sol.message)
print("x =", sol.x, " nfev =", sol.nfev)
print("F(x) =", F(sol.x))
```

Çıktı:

```
fsolve, başlangıç [1.0, -1.0]: x = 1.004168738475, y = -1.729637287026
fsolve, başlangıç [-1.0, 1.0]: x = -1.816264068824, y = 0.837367799891
True The solution converged.
x = [-1.81626407  0.8373678 ]  nfev = 10
F(x) = [-2.91411340e-12 -2.17825757e-13]
```

 CC BY-NC-SA 4.0



Şekil 10.5: $x^2 + y^2 = 4$ çemberi ile $e^x + y = 1$ eğrisinin iki kesişim noktası. Kesikli oklar Newton yönteminin $(1; -1)$ ve $(-1; 1)$ başlangıç noktalarından attığı ilk adımlardır; her başlangıç kendisine yakın olan çözüme gider.

İki farklı başlangıç noktası, sistemin iki farklı çözümünü verdi; şekil, çemberle eğrinin tam iki noktada kesiştiğini gösteriyor. `fsolve` yakınsamadığında yalnız bir uyarı yazar ve yine bir dizi döndürür. Bu yüzden sonucu F'ye koyup artığı denetlemek ya da `root` kullanıp `sol.success` değerine bakmak iyi bir alışkanlıktır.

10.6 Tek Değişkenli Optimizasyon

Şimdi kök bulmadan minimum bulmaya geçiyoruz. Yerel ve mutlak minimum kavramları Analiz 2 notlarındadır. Fermat Teoremi'ne göre türevlenebilir bir fonksiyonun bir iç noktadaki yerel ekstremumunda türev sıfırdır (bkz. Analiz 2).

`minimize_scalar(f)` tek değişkenli bir fonksiyonun yerel minimumunu türev kullanmadan arar. Varsayılan Brent yöntemi, altın oran aramasını parabolik interpolasyonla birleştirir. Aramanın nereden başlayacağı `bracket` ile seçilebilir; `method="bounded"` ve `bounds=(a, b)` ise aramayı bir aralığa hapseder. Sonuç bir `OptimizeResult` nesnesidir: `x` minimum noktası, `fun` oradaki değer, `nfev` fonksiyon çağrısı sayısıdır. SciPy'da yalnız minimize eden fonksiyonlar vardır; f'nin maksimumu, $-f$'nin minimumu olarak bulunur.

Örnek olarak iki yerel minimumu olan $f(x) = x^4 - 4x^2 + x$ fonksiyonunu üç farklı çağrıyla minimize edelim.

```
from scipy.optimize import minimize_scalar

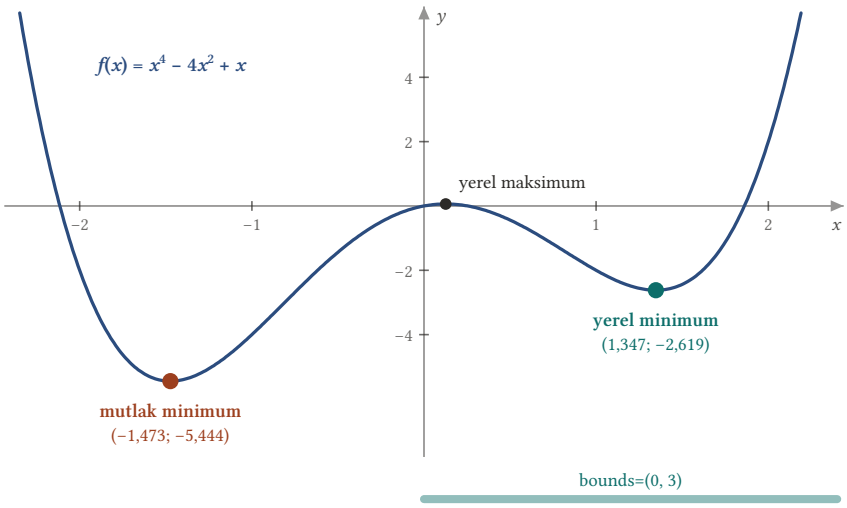

def f(x):
    return x**4 - 4*x**2 + x


r1 = minimize_scalar(f)                              # varsayılan: Brent
```

```
r2 = minimize_scalar(f, bracket=(-3, -2))
r3 = minimize_scalar(f, bounds=(0, 3), method="bounded")
for name, r in [("varsayılan", r1), ("bracket=(-3, -2)", r2),
                ("bounds=(0, 3)", r3)]:
    print(f"{name:16s}  x = {r.x:.6f}  f(x) = {r.fun:.6f}  "
          f"nfev = {r.nfev}")
```

Çıktı:

```
varsayılan        x = 1.346997  f(x) = -2.618556  nfev = 15
bracket=(-3, -2)  x = -1.472998  f(x) = -5.444192  nfev = 14
bounds=(0, 3)     x = 1.346998  f(x) = -2.618556  nfev = 12
```



Şekil 10.6: $f(x) = x^4 - 4x^2 + x$ fonksiyonunun iki yerel minimumu. `bracket=(-3, -2)` ile yapılan arama soldaki mutlak minimumu, varsayılan arama ve `bounds=(0, 3)` ile yapılan arama sağdaki yerel minimumu bulur.

Varsayılan arama $(0, 1)$ çiftinden başlayıp yokuş aşağı ilerler ve sağdaki vadiye düşer: $f(1{,}347) \approx -2{,}619$ yalnız bir yerel minimumdur. `bracket=(-3, -2)` aramayı soldan başlatır ve asıl, mutlak minimumu $f(-1{,}473) \approx -5{,}444$'ü bulur. `bounds=(0, 3)` ise aramayı $[0, 3]$ aralığıyla sınırladığı için yine sağdaki minimumu verir. Yöntem, başladığı vadinin dibini bulur; vadiyi seçmek bize kalır.
Kök bulma ile optimizasyon arasındaki bağı kodda da görelim: $f'(x) = 4x^3 - 8x + 1$ türevinin köklerini işaret taraması ve `brentq` ile bulup ikinci türev $f''(x) = 12x^2 - 8$ ile sınıflandıralım.

```
import numpy as np
from scipy.optimize import brentq


def f(x):
    return x**4 - 4*x**2 + x


def df(x):
    return 4*x**3 - 8*x + 1


def d2f(x):
    return 12*x**2 - 8
```

```
x = np.linspace(-3, 3, 61)
d = df(x)
for i in np.nonzero(d[:-1] * d[1:] < 0)[0]:
    c = brentq(df, x[i], x[i + 1])
    kind = "minimum" if d2f(c) > 0 else "maksimum"
    print(f"x = {c:9.6f}   f(x) = {f(c):9.6f}   yerel {kind}")
```

Çıktı:

```
x = -1.472998   f(x) = -5.444192   yerel minimum
x =  0.126000   f(x) =  0.062748   yerel maksimum
x =  1.346997   f(x) = -2.618556   yerel minimum
```

Türevin üç kökü vardır: iki yerel minimum ve aralarında bir yerel maksimum. `minimize_scalar` sonuçları bu tablodaki minimumlarla beş ondalık basamağa kadar örtüşüyor.

10.7 Çok Değişkenli Optimizasyon

Çok değişkenli bir $f:\mathbb{R}^n\to\mathbb{R}$ fonksiyonunu minimize etmek için `minimize` fonksiyonu kullanılır. Bir yerel minimumda gradyan sıfırdır (bkz. Analiz 4); Hessian matrisi pozitif tanımlı olan bir kritik nokta ise kesin yerel minimumdur (bkz. Analiz 4).
Yöntemleri sınamak için klasik bir örnek Rosenbrock fonksiyonudur:

$$f(x,y)=(1-x)^2+100\,(y-x^2)^2.$$

İki karenin toplamı olduğundan $f\ge 0$'dır ve minimum değer $f(1,1)=0$'dır. Sayısal olarak zor olmasının sebebi, minimumun $y=x^2$ parabolü boyunca uzanan dar ve kıvrık bir vadinin dibinde olmasıdır: yöntemler vadiye çabuk iner, ama vadinin içinde minimuma doğru yavaş ilerler. Alışılmış başlangıç noktası $(-1{,}2;\,1)$'dir.
İlk yöntem **Nelder–Mead**'dir. Türev kullanmaz: düzlemde bir üçgeni (genel olarak $n+1$ köşeli bir simpleksi) yansıtma, genişletme ve büzme hamleleriyle fonksiyonun azaldığı yöne kaydırır.

```
import numpy as np
from scipy.optimize import minimize


def f(v):
    x, y = v
    return (1 - x)**2 + 100 * (y - x**2)**2


x0 = np.array([-1.2, 1.0])                # alışılmış başlangıç noktası
res = minimize(f, x0, method="Nelder-Mead")
print(res.message)
print("x =", res.x, " f(x) =", res.fun)
print("iterasyon:", res.nit, " f çağrısı:", res.nfev)
```

Çıktı:

```
Optimization terminated successfully.
x = [1.00002202 1.00004222]  f(x) = 8.177661197416674e-10
iterasyon: 85  f çağrısı: 159
```

Nelder–Mead minimumu buldu, ama 159 fonksiyon çağrısıyla ve ancak 10^{-4} mertebesinde bir doğrulukla; varsayılan durma toleransları `xatol` ve `fatol` 10^{-4}'tür. İkinci yöntem **BFGS**'tir: gradyanı kullanır ve Hessian matrisinin bir yaklaşımını adım adım güncelleyen bir yarı-Newton yöntemidir. Gradyan

$$\nabla f(x,y)=\big(-2(1-x)-400x(y-x^2),\;200(y-x^2)\big)$$

olarak `jac` parametresiyle verilebilir; verilmezse SciPy onu sonlu farklarla yaklaşık hesaplar. Aşağıdaki kod önceki kodun devamıdır.

 CC BY-NC-SA 4.0

```
def grad(v):
    x, y = v
    return np.array([-2 * (1 - x) - 400 * x * (y - x**2),
                     200 * (y - x**2)])


r1 = minimize(f, x0, method="BFGS")              # gradyan verilmedi
r2 = minimize(f, x0, method="BFGS", jac=grad)    # gradyan verildi
for name, r in [("gradyansız", r1), ("gradyanlı", r2)]:
    print(f"{name:10s} x = {r.x}  nit = {r.nit}  "
          f"nfev = {r.nfev}  njev = {r.njev}")
```

Çıktı:

```
gradyansız x = [0.99999328 0.99998655]  nit = 31  nfev = 114  njev = 38
gradyanlı  x = [0.99999997 0.99999995]  nit = 32  nfev = 39  njev = 39
```

Sonuçtaki `nit` iterasyon sayısı, `njev` ise gradyanın kaç kez hesaplandığıdır. Gradyan verilmeyince SciPy her gradyanı sonlu farklarla yaklaşık hesaplar ve bunun için iki ek fonksiyon çağrısı yapar. Burada f 38 noktada hesaplanmış, her noktada gradyan için 2 çağrı daha yapılmıştır: $38 + 2 \cdot 38 = 114$. Gradyan verilince fonksiyon çağrısı 39'a düşer ve sonuç daha doğrudur. Gradyanı biliyorsak vermek neredeyse her zaman kazançtır; elle türev almak zahmetliyse SymPy ile Sembolik Hesap bölümündeki `diff` ve `lambdify` bu işi yapar.

Yöntemlerin izlediği yolu görmek için `callback` parametresi kullanılır: SciPy her iterasyondan sonra bu fonksiyonu o anki noktayla çağırır. Aşağıdaki kod yine öncekilerin devamıdır.

```
paths = {}
for method, jac in [("Nelder-Mead", None), ("BFGS", grad)]:
    path = [x0]
    minimize(f, x0, method=method, jac=jac,
             callback=lambda xk: path.append(xk.copy()))
    paths[method] = np.array(path)
    print(f"{method:11s}: {len(path)} nokta, son nokta {path[-1]}")
```

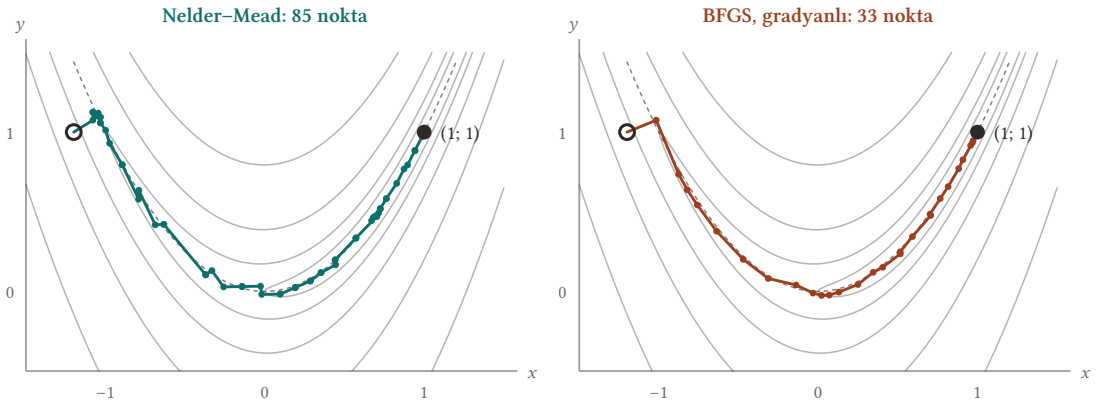
Çıktı:

```
Nelder-Mead: 85 nokta, son nokta [1.00002202 1.00004222]
BFGS       : 33 nokta, son nokta [0.99999997 0.99999995]
```

Nokta sayılarında küçük bir ayrıntı var. BFGS yolunda 32 iterasyonun noktalarına `x0` eklenince 33 nokta olur. Nelder–Mead ise başlangıç simpleksini de bir iterasyon sayar; bu yüzden 85 iterasyonda callback 84 kez çağrılır ve `x0` ile birlikte 85 nokta olur.

`xk.copy()` önemlidir: optimizasyon fonksiyonu aynı diziyi yerinde güncelleyebilir, kopyalamadan eklenen bütün liste elemanları sonunda aynı diziyi gösterebilir.

 CC BY-NC-SA 4.0



Şekil 10.7: Rosenbrock fonksiyonunun eş yükselti eğrileri ($f = 1, 4, 16, 64, 256$) ve kesikli $y = x^2$ vadisi. Solda Nelder–Mead yönteminin, sağda gradyanı verilen BFGS yönteminin callback ile kaydedilen noktaları; ikisi de içi boş daireyle gösterilen (−1,2; 1) noktasından başlayıp muz biçimli vadiyi izleyerek (1; 1) minimumuna varır.

Son olarak bulunan noktanın gerçekten minimum olduğunu denetleyelim: gradyan sıfıra yakın olmalı, Hessian pozitif tanımlı olmalıdır. Bu kod da öncekilerin devamıdır.

```
def hessian(v):
    x, y = v
    return np.array([[2 - 400 * (y - x**2) + 800 * x**2, -400 * x],
                     [-400 * x, 200.0]])


print("gradyan:", grad(r2.x))
H = hessian(np.array([1.0, 1.0]))
print(H)
print("özdeğerler:", np.linalg.eigvalsh(H))
print("koşul sayısı:", np.linalg.cond(H))
```

Çıktı:

```
gradyan: [-1.70451579e-06  8.23259305e-07]
[[ 802. -400.]
 [-400.  200.]]
özdeğerler: [3.99360767e-01 1.00160064e+03]
koşul sayısı: 2508.009601277225
```

Gradyan 10^{-6} mertebesindedir ve Hessian'ın iki özdeğeri de pozitiftir; $(1, 1)$ kesin yerel minimumdur. Koşul sayısının yaklaşık 2508 olması, eş yükselti eğrilerinin bir yönde diğerine göre çok uzamış olduğunu söyler. Dar vadinin sayısal karşılığı budur (NumPy ile Lineer Cebir bölümündeki koşul sayısı).

⚠ Optimizasyon yerel minimum bulur

Bu bölümdeki bütün yöntemler başlangıç noktasının çevresindeki bir yerel minimumu arar. Fonksiyonun birden çok minimumu varsa hangisinin bulunacağı `x0`'a bağlıdır (alıştırmalardaki Himmelblau fonksiyonu buna bir örnektir). Mutlak minimum için birkaç farklı başlangıçtan çalıştırıp en iyi sonucu seçmek basit ve etkili bir yoldur; SciPy'da ayrıca `differential_evolution`, `basinhopping` ve `shgo` gibi küresel arama yöntemleri de vardır. Konveks fonksiyonlarda ise her yerel minimum mutlak minimumdur.

 CC BY-NC-SA 4.0

10.8 Kısıtlı Optimizasyon

Gerçek problemlerde değişkenler çoğu zaman serbest değildir: uzunluklar pozitif olmalı, bütçe aşılmamalı, hacim sabit kalmalıdır. Bu tür problemleri önce genel bir biçimde yazalım.

Tanım 10.3 (Kısıtlı Optimizasyon Problemi). $f, g_1, \dots, g_m, h_1, \dots, h_k : \mathbb{R}^n \to \mathbb{R}$ fonksiyonları verilsin.

$$\begin{aligned} \text{en küçük yap:} \quad & f(\mathbf{x}) \\ \text{kısıtlar:} \quad & g_i(\mathbf{x}) \geq 0, \quad i = 1, \dots, m, \\ & h_j(\mathbf{x}) = 0, \quad j = 1, \dots, k \end{aligned}$$

biçimindeki probleme **kısıtlı optimizasyon problemi** denir. f **amaç fonksiyonu**, $g_i(\mathbf{x}) \geq 0$ koşulları **eşitsizlik kısıtları**, $h_j(\mathbf{x}) = 0$ koşulları **eşitlik kısıtları**dır. Bütün kısıtları sağlayan noktaların kümesine **uygun bölge** denir.

Yani amaç fonksiyonunu yalnız uygun bölgedeki noktalar arasında en küçük yapmaya çalışırız. SciPy'ın `minimize` fonksiyonu kısıtları tam bu biçimde bekler: `{"type": "ineq", "fun": g}` sözlüğü $g(\mathbf{x}) \geq 0$, `{"type": "eq", "fun": h}` sözlüğü $h(\mathbf{x}) = 0$ anlamına gelir. Değişkenlerin alt ve üst sınırları ayrıca `bounds` ile, her değişken için bir `(alt, üst)` çifti olarak verilir; `None` o yönde sınır olmadığını söyler. Kısıtlı problemler için en çok kullanılan yöntem, her adımda problemi karesel bir alt probleme yaklaştıran `SLSQP`'dir; `trust-constr` yöntemi de aynı işi `LinearConstraint` ve `NonlinearConstraint` nesneleriyle yapar.
Örnek olarak $f(x, y) = (x-2)^2 + (y-1)^2$ fonksiyonunu $x + y \leq 2$, $x \geq 0$, $y \geq 0$ üçgeninde minimize edelim. Geometrik olarak bu, üçgenin $(2, 1)$ noktasına en yakın noktasını aramaktır. $x + y \leq 2$ kısıtını önce SciPy'ın istediği biçime, $2 - x - y \geq 0$ biçimine çeviriyoruz.

```
import numpy as np
from scipy.optimize import minimize


def f(v):
    x, y = v
    return (x - 2)**2 + (y - 1)**2

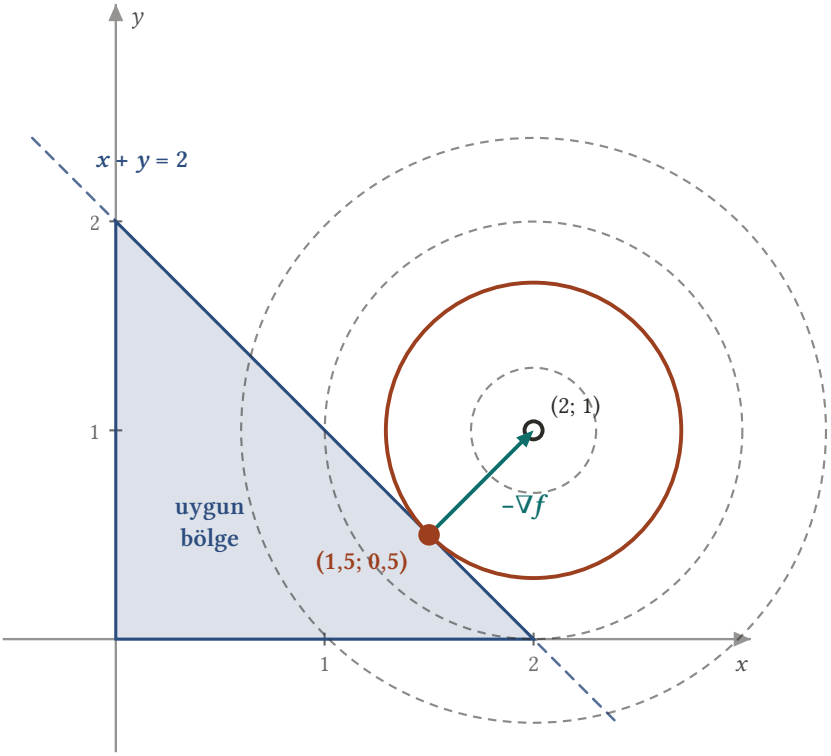

def g(v):                          # x + y <= 2  yerine  2 - x - y >= 0
    return 2 - v[0] - v[1]


cons = [{"type": "ineq", "fun": g}]
bounds = [(0, None), (0, None)]          # x >= 0, y >= 0
res = minimize(f, x0=[0.0, 0.0], method="SLSQP",
               bounds=bounds, constraints=cons)
print(res.message)
print(f"x = {res.x[0]:.6f}, y = {res.x[1]:.6f}, f = {res.fun:.6f}")
print(f"g(x, y) = {g(res.x):.1e}")
print("f'nin gradyanı:", 2 * (res.x - [2, 1]))
```

Çıktı:

```
Optimization terminated successfully
x = 1.500000, y = 0.500000, f = 0.500000
g(x, y) = -3.3e-16
f'nin gradyanı: [-1. -1.]
```

 CC BY-NC-SA 4.0



Şekil 10.8: Uygun bölge $x + y \le 2$, $x, y \ge 0$ üçgenidir. Kesikli çemberler (2; 1) merkezli eş yükselti eğrileridir; minimum, üçgene değen en küçük çemberin değme noktası (1,5; 0,5)'tir. Orada $-\nabla f = (1; 1)$ vektörü $x + y = 2$ doğrusuna diktir.

$(2, 1)$ noktası uygun bölgede değildir ($2 + 1 > 2$). Bu yüzden minimum üçgenin $x + y = 2$ kenarı üzerindedir: $(2, 1)$'in bu doğru üzerindeki dik izdüşümü $(1{,}5;\ 0{,}5)$ noktasıdır ve $f = 0{,}5$ olur. Kısıtın değeri 0'dır (yazdırılan $-3{,}3 \cdot 10^{-16}$ yuvarlama hatasıdır), yani kısıt sınırdadır. Optimum noktada $\nabla f = (-1, -1)$, kısıt doğrusunun normali $(1, 1)$ ile paraleldir: $\nabla f + \lambda \nabla (x + y - 2) = \mathbf{0}$ eşitliği $\lambda = 1$ ile sağlanır. Bu, Lagrange çarpanları koşuludur (bkz. Analiz 4).

> ⚠ Eşitsizliğin yönü
>
> SciPy'da `"ineq"` her zaman $g(\mathbf{x}) \ge 0$ demektir. $x + y \le 2$ kısıtını çevirmeden `"fun": lambda v: v[0] + v[1] - 2` diye yazarsak $x + y \ge 2$ problemini çözmüş oluruz. Bu yeni problemde $(2, 1)$ uygundur, çözücü kısıtsız minimumu $(2, 1)$'i bulur ve "başarılı" der. Hata mesajı çıkmaz; yanlış problemin doğru çözümünü alırız.

Örnek 10.2 (En Az Malzemeli Silindir Kutu). Hacmi $1000\ \text{cm}^3$ olan kapaklı bir silindir kutunun yüzey alanını en küçük yapan r yarıçapını ve h yüksekliğini `minimize` ile bulunuz.

Çözüm.
Problemi kurmak. Yüzey alanı iki daire ile yan yüzeyin toplamıdır:

$$A(r, h) = 2\pi r^2 + 2\pi r h.$$

$V = 1000$ olmak üzere hacim koşulu $\pi r^2 h - V = 0$ eşitlik kısıtıdır. Uzunluklar pozitif olmalıdır; `bounds` ile $r, h \ge 0{,}1$ istiyoruz.
Gradyan. $\frac{\partial A}{\partial r} = 4\pi r + 2\pi h$ ve $\frac{\partial A}{\partial h} = 2\pi r$'dir. Gradyanı `jac` ile veriyoruz.

```python
import numpy as np
from scipy.optimize import minimize

V = 1000.0                                  # hacim, cm³


def area(v):
    r, h = v
    return 2 * np.pi * r**2 + 2 * np.pi * r * h


def area_grad(v):
    r, h = v
    return np.array([4 * np.pi * r + 2 * np.pi * h, 2 * np.pi * r])


def volume_gap(v):                          # π r² h - V = 0 olmalı
    r, h = v
    return np.pi * r**2 * h - V


res = minimize(area, x0=[5.0, 10.0], jac=area_grad, method="SLSQP",
               bounds=[(0.1, None), (0.1, None)],
               constraints=[{"type": "eq", "fun": volume_gap}])
r, h = res.x
print(f"r = {r:.4f} cm, h = {h:.4f} cm, alan = {res.fun:.2f} cm²")
print(f"h / r = {h / r:.4f}")
r_exact = (V / (2 * np.pi)) ** (1 / 3)
print(f"kapalı biçim: r = {r_exact:.4f}, h = {2 * r_exact:.4f}")
```

Çıktı:

```
r = 5.4193 cm, h = 10.8385 cm, alan = 553.58 cm²
h / r = 2.0000
kapalı biçim: r = 5.4193, h = 10.8385
```

Kapalı biçimle karşılaştırma. Kısıttan $h = \frac{V}{\pi r^2}$ çekilirse $A(r) = 2\pi r^2 + \frac{2V}{r}$ olur. $A'(r) = 4\pi r - \frac{2V}{r^2} = 0$ denkleminden $r^3 = \frac{V}{2\pi}$, yani $r = \sqrt[3]{V/(2\pi)} \approx 5{,}4193$ bulunur. Bu durumda $\pi r^3 = \frac{V}{2}$ olduğundan $h = \frac{V}{\pi r^2} = 2r$'dir: en ekonomik kutunun yüksekliği çapına eşittir. Sayısal sonuç bununla yazdırılan bütün basamaklarda örtüşüyor.
Bir uyarı. Aynı çağrı `jac=area_grad` olmadan, yani gradyan sonlu farklarla hesaplanarak yapılırsa SLSQP bu başlangıç noktasından $r \approx 5{,}5242$, $h \approx 10{,}4306$ noktasında durur ve yine "başarılı" der. Sayısal bir optimizasyonun sonucunu her zaman bağımsız bir yolla, burada kapalı biçimle, denetlemek gerekir. ■

10.9 Alıştırmalar

Aşağıdaki alıştırmalarda bölümün araçlarını yeni denklemlere ve optimizasyon problemlerine uyguluyoruz; her çözümde kod ve çalıştırılmış çıktısı verilmiştir.

Alıştırma 10.1 (İkiye Bölmede Adım Sayısı). $x^3 - x - 2 = 0$ denkleminin $[1, 2]$ aralığındaki kökünü 10^{-8} hassasiyetle bulmak için ikiye bölme yönteminin kaç orta nokta hesaplaması gerektiğini Önerme 10.1 ile önceden belirleyiniz ve bölümdeki `bisection` fonksiyonuyla doğrulayınız.

Çözüm.
Parantez. $f(x) = x^3 - x - 2$ için $f(1) = -2 < 0$ ve $f(2) = 4 > 0$ olduğundan $[1, 2]$ bir kök parantezidir.

Öngörü. $\frac{2-1}{2^n} < 10^{-8}$, yani $2^n > 10^8$ olmalıdır. $\log_2 10^8 \approx 26{,}58$ olduğundan $n = 27$ orta nokta yeterlidir.
Doğrulama.

```
import math


def bisection(f, a, b, tol=1e-10, max_iter=100):
    """f'nin [a, b] parantezindeki bir kökünü ikiye bölmeyle bulur."""
    fa = f(a)
    if fa * f(b) > 0:
        raise ValueError("f(a) ile f(b) aynı işaretli: parantez yok")
    for n in range(1, max_iter + 1):
        p = (a + b) / 2
        fp = f(p)
        if fp == 0 or (b - a) / 2 < tol:
            return p, n
        if fa * fp < 0:
            b = p
        else:
            a, fa = p, fp
    raise RuntimeError("istenen hassasiyete ulaşılamadı")


def f(x):
    return x**3 - x - 2


print("öngörülen n:", math.ceil(math.log2((2 - 1) / 1e-8)))
p, n = bisection(f, 1.0, 2.0, tol=1e-8)
print(f"p = {p:.10f}, n = {n}, f(p) = {f(p):.1e}")
```

Çıktı:

```
öngörülen n: 27
p = 1.5213797018, n = 27, f(p) = -3.0e-08
```

Fonksiyon da 27 orta noktada durdu. Denklemin Cardano formülünden gelen gerçek kökü

$$\sqrt[3]{1+\sqrt{26/27}} + \sqrt[3]{1-\sqrt{26/27}} \approx 1{,}5213797068$$

sayısıdır; bulunan değerin hatası yaklaşık $5 \cdot 10^{-9}$ olup 10^{-8}'in altındadır. ■

Alıştırma 10.2 (Tanjant Denkleminin İlk Pozitif Kökü). $\tan x = x$ denkleminin en küçük pozitif kökünü `brentq` ile bulunuz.

Çözüm.
Kökün yeri. $g(x) = \tan x - x$ diyelim. $\left(0, \frac{\pi}{2}\right)$ aralığında $\tan x > x$ olduğundan kök yoktur. $\left(\frac{\pi}{2}, \pi\right)$ aralığında $\tan x < 0 < x$ olduğundan yine kök yoktur. $\left(\pi, \frac{3\pi}{2}\right)$ aralığında $g(\pi) = -\pi < 0$'dır ve $x \to \frac{3\pi}{2}^-$ iken $g(x) \to +\infty$ olur; g bu aralıkta sürekli ve artan olduğundan tam bir kök vardır.
Tuzak. $[1, 2]$ aralığında da işaret değişir: $g(1) \approx 0{,}557$ ve $g(2) \approx -4{,}185$. Ama bu aralıkta $\tan x$'in $\frac{\pi}{2}$ kutbu vardır; g sürekli olmadığından Tanım 10.1 koşulu sağlanmaz.
Kod. Önce yanlış parantezin ne verdiğine, sonra doğru paranteze bakalım. Sağ uç, kutbun 10^{-6} solunda seçildi.

```python
import numpy as np
from scipy.optimize import brentq


def g(x):
    return np.tan(x) - x


# yanlış parantez: [1, 2] içinde tan x'in kutbu π/2 var
bad = brentq(g, 1.0, 2.0)
print(f"[1, 2]: x = {bad:.10f}, g(x) = {g(bad):.2e}")

# doğru parantez: (π, 3π/2), sağ uç kutbun hemen solunda
a, b = np.pi, 1.5 * np.pi - 1e-6
print(f"g(a) = {g(a):.4f}, g(b) = {g(b):.4e}")
r = brentq(g, a, b)
print(f"x = {r:.12f}, g(x) = {g(r):.1e}")
```

Çıktı:

```
[1, 2]: x = 1.5707963268, g(x) = 1.44e+12
g(a) = -3.1416, g(b) = 1.0000e+06
x = 4.493409457909, g(x) = 8.9e-16
```

`brentq` yanlış parantezde $\frac{\pi}{2} \approx 1{,}5707963268$ noktasını döndürdü; orada g'nin değeri 10^{12} mertebesindedir, yani bu bir kök değil kutuptur. İşaret değişimi ancak sürekli bir fonksiyon için kök garantisi verir; sonucu g'ye koyup denetlemek bu tür hataları yakalar. Doğru parantezde en küçük pozitif kök $x \approx 4{,}493409457909$ bulunur. ■

Alıştırma 10.3 (Kosinüs Denklemi için Üç Yöntem). $\cos x = x$ denkleminin kökünü `root_scalar` ile `brentq` ($[0, 1]$ parantezi), sekant ($x_0 = 0, x_1 = 1$) ve Newton ($x_0 = 1$) yöntemleriyle bulunuz ve iterasyon ile fonksiyon çağrısı sayılarını karşılaştırınız.

Çözüm.
Hazırlık. $f(x) = \cos x - x$ için $f(0) = 1 > 0$ ve $f(1) = \cos 1 - 1 \approx -0{,}460 < 0$ olduğundan $[0, 1]$ bir parantezdir. Türev $f'(x) = -\sin x - 1$'dir ve $[0, 1]$ aralığında $-1{,}85$ ile -1 arasında kalır, hiç sıfır olmaz.
Kod.

```python
import numpy as np
from scipy.optimize import root_scalar


def f(x):
    return np.cos(x) - x


def df(x):
    return -np.sin(x) - 1


for kw in [{"method": "brentq", "bracket": [0.0, 1.0]},
           {"method": "secant", "x0": 0.0, "x1": 1.0},
           {"method": "newton", "x0": 1.0, "fprime": df}]:
    sol = root_scalar(f, **kw)
    print(f"{kw['method']:7s} x = {sol.root:.15f}  "
          f"iterasyon = {sol.iterations}  çağrı = {sol.function_calls}")
```

 CC BY-NC-SA 4.0

Çıktı:

```
brentq  x = 0.739085133215156  iterasyon = 7  çağrı = 8
secant  x = 0.739085133215161  iterasyon = 6  çağrı = 7
newton  x = 0.739085133215161  iterasyon = 4  çağrı = 8
```

Karşılaştırma. Üç yöntem de kök $x \approx 0{,}739085133215$ için anlaşıyor; `brentq` sonucunun son basamağındaki fark `xtol` toleransının içindedir. Newton en az iterasyonu (4) yapıyor, ama her iterasyonda f ve f' çağrıldığından toplam 8 çağrı harcıyor. Sekant 6 iterasyonda 7 çağrıyla bitiyor; türevi hesaplamak pahalıysa sekant en ekonomik seçenektir. ■

Alıştırma 10.4 (Sinüs ile Doğrunun Bütün Kesişimleri). $\sin x = \dfrac{x}{10}$ denkleminin bütün reel köklerini bulunuz.

Çözüm.
Kökler nerede olabilir? $|\sin x| \le 1$ olduğundan bir kökte $\left|\frac{x}{10}\right| \le 1$, yani $x \in [-10, 10]$ olmalıdır. Bu sonlu aralığı taramak yeterlidir.
Izgara seçimi. $x = 0$ bir köktür. 201 noktalı bir ızgara $(-10, -9{,}9, \ldots, 10)$ 0'ı tam içerir; o noktada $f = 0$ olur, çarpım testi `< 0` sağlanmaz ve kök kaçar. Bu yüzden 0'ı içermeyen 200 noktalık bir ızgara alıyoruz. Adım yaklaşık 0,1 olduğundan iki kökün aynı hücreye düşmesi de beklenmez.

```
import numpy as np
from scipy.optimize import brentq


def f(x):
    return np.sin(x) - x / 10


x = np.linspace(-10, 10, 200)              # adım 20/199 ≈ 0.1
y = f(x)
i = np.nonzero(y[:-1] * y[1:] < 0)[0]
roots = [brentq(f, x[k], x[k + 1]) for k in i]
print(len(roots), "kök:")
for r in roots:
    print(f"  {r: .10f}   f = {f(r): .1e}")
```

Çıktı:

```
7 kök:
  -8.4232039324   f = -3.2e-15
  -7.0681743581   f = -2.2e-16
  -2.8523418945   f =  9.2e-14
   0.0000000000   f =  1.2e-17
   2.8523418945   f = -9.2e-14
   7.0681743581   f =  2.2e-16
   8.4232039324   f =  3.2e-15
```

Sonuç. Denklemin 7 kökü vardır: 0, $\pm 2{,}8523418945$, $\pm 7{,}0681743581$ ve $\pm 8{,}4232039324$. f tek fonksiyon olduğundan kökler 0'a göre simetriktir. ■

Alıştırma 10.5 (Döngüye Giren Newton İterasyonunu Kurtarmak). $x^3 - 2x + 2 = 0$ denkleminin reel kökünü, SciPy'ın `newton` fonksiyonunun $x_0 = 0$ başlangıcında neden başarısız olduğunu da göstererek güvenilir bir yöntemle bulunuz.

 CC BY-NC-SA 4.0

Çözüm.
Kaç reel kök var? $f(x) = x^3 - 2x + 2$ için $f'(x) = 3x^2 - 2$ türevi $x = \pm\sqrt{2/3} \approx \pm 0{,}816$ noktalarında sıfırdır. Yerel minimum değeri $f\left(\sqrt{2/3}\right) \approx 0{,}911 > 0$ olduğundan f, $x > -0{,}816$ için hiç sıfır olmaz; tek reel kök sol taraftadır.
Newton neden başarısız? "Newton her zaman yakınsamaz" uyarısında gördüğümüz gibi, $x_0 = 0$'dan başlayan iterasyon $0, 1, 0, 1, \ldots$ döngüsüne girer. SciPy 50 iterasyon sonra bir `RuntimeError` fırlatır; bunu `try`/`except` ile yakalıyoruz.
Parantezle çözüm. $f(-2) = -2 < 0$ ve $f(-1) = 3 > 0$ olduğundan $[-2, -1]$ bir parantezdir.

```
from scipy.optimize import brentq, newton


def f(x):
    return x**3 - 2*x + 2


def df(x):
    return 3*x**2 - 2


try:
    newton(f, 0.0, fprime=df)
except RuntimeError as err:
    print("Newton, x0 = 0:", err)

print("f(-2) =", f(-2.0), " f(-1) =", f(-1.0))
r = brentq(f, -2.0, -1.0)
print(f"brentq: {r:.15f}")
print(f"Newton, x0 = -2: {newton(f, -2.0, fprime=df):.15f}")
```

Çıktı:

```
Newton, x0 = 0: Failed to converge after 50 iterations, value is 0.0.
f(-2) = -2.0  f(-1) = 3.0
brentq: -1.769292354238632
Newton, x0 = -2: -1.769292354238631
```

Kök $x \approx -1{,}769292354239$'dur. Parantezin bir ucundan, $x_0 = -2$'den başlatılan Newton da aynı köke yakınsıyor: başlangıç doğru seçilince Newton yine hızlı ve doğrudur. ■

Alıştırma 10.6 (Elips ile Parabolün Kesişimi). $\dfrac{x^2}{4} + y^2 = 1$ elipsi ile $y = x^2 - 1$ parabolünün bütün kesişim noktalarını `fsolve` ile bulunuz ve sonucu kesin değerlerle karşılaştırınız.

Çözüm.
Kesin çözüm. $y = x^2 - 1$ elips denklemine konursa $\frac{x^2}{4} + (x^2 - 1)^2 = 1$ olur. $u = x^2$ dersek $u^2 - \frac{7}{4}u = 0$, yani $u = 0$ ya da $u = \frac{7}{4}$ bulunur. Kesişim noktaları $(0, -1)$ ve $\left(\pm\frac{\sqrt{7}}{2}, \frac{3}{4}\right)$'tür.
Sayısal çözüm. Başlangıç noktalarını kaba bir çizimden okuyoruz: kesişimlerden biri elipsin alt tepesi yakınında, ikisi sağda ve solda üst yarı düzlemdedir. Bu yüzden $(0{,}5; -1{,}5)$, $(2, 1)$ ve $(-2, 1)$ noktalarından başlıyoruz.

```
import numpy as np
from scipy.optimize import fsolve


def F(v):
    x, y = v
```

```
    return np.array([x**2 / 4 + y**2 - 1, y - x**2 + 1])


found = []
for start in ([0.5, -1.5], [2.0, 1.0], [-2.0, 1.0]):
    v = fsolve(F, start)
    found.append(v)
    print(f"başlangıç {start}: x = {v[0]: .12f}, y = {v[1]: .12f}")

print("√7/2 =", np.sqrt(7) / 2)
```

Çıktı:

```
başlangıç [0.5, -1.5]: x =  0.000000014406, y = -1.000000000000
başlangıç [2.0, 1.0]: x =  1.322875655532, y =  0.750000000000
başlangıç [-2.0, 1.0]: x = -1.322875655532, y =  0.750000000000
√7/2 = 1.3228756555322954
```

Karşılaştırma. Yan noktalar $\frac{\sqrt{7}}{2} \approx 1{,}322875655532$ ve $y = 0{,}75$ ile bütün basamaklarda örtüşüyor. $(0, -1)$ noktasında ise x yalnız $1{,}4 \cdot 10^{-8}$ doğrulukla bulundu. Sebep, iki eğrinin bu noktada birbirine teğet olmasıdır: elipsin alt tepesi ile parabolün tepesi aynı yatay teğete sahiptir. Jacobi matrisi

$$J(x, y) = \begin{pmatrix} x/2 & 2y \\ -2x & 1 \end{pmatrix}$$

$(0, -1)$ noktasında tekildir ve Newton türü yöntemler orada yalnız doğrusal hızla yakınsar. Bu, tek değişkenli durumdaki çift katlı kökün ($u = x^2 = 0$) sistemlerdeki karşılığıdır. ■

Alıştırma 10.7 (En Büyük Hacimli Açık Kutu). 30×20 cm'lik bir kartonun dört köşesinden kenarı x olan kareler kesilip kenarlar yukarı katlanarak üstü açık bir kutu yapılıyor. Kutunun hacmini en büyük yapan x değerini `minimize_scalar` ile bulunuz.

Çözüm.
Model. Kutunun tabanı $(30 - 2x) \times (20 - 2x)$, yüksekliği x'tir:

$$V(x) = x(30 - 2x)(20 - 2x), \qquad 0 < x < 10.$$

Maksimum için minimum. V'nin maksimumu $-V$'nin minimumudur. Aralık bilindiğinden `method="bounded"` ve `bounds=(0, 10)` kullanıyoruz.

```
import numpy as np
from scipy.optimize import minimize_scalar


def volume(x):
    return x * (30 - 2*x) * (20 - 2*x)


res = minimize_scalar(lambda x: -volume(x), bounds=(0, 10),
                      method="bounded")
print(f"x = {res.x:.6f} cm, hacim = {volume(res.x):.4f} cm³")
x_exact = (25 - 5 * np.sqrt(7)) / 3
print(f"kapalı biçim: x = {x_exact:.6f}, hacim = {volume(x_exact):.4f}")
```

Çıktı:

```
x = 3.923748 cm, hacim = 1056.3059 cm³
kapalı biçim: x = 3.923748, hacim = 1056.3059
```

Kapalı biçim. $V(x) = 4x^3 - 100x^2 + 600x$ ve $V'(x) = 12x^2 - 200x + 600$'dür. $V'(x) = 0$ denklemi $3x^2 - 50x + 150 = 0$ olur; kökleri $x = \frac{25 \pm 5\sqrt{7}}{3}$'tür. $\frac{25+5\sqrt{7}}{3} \approx 12{,}74$ aralığın dışındadır. $V(0) = V(10) = 0$ ve aralığın içinde $V > 0$ olduğundan kalan kritik nokta $x = \frac{25-5\sqrt{7}}{3} \approx 3{,}923748$ cm maksimumdur; hacim yaklaşık $1056{,}31\ \text{cm}^3$'tür. Sayısal sonuç bununla örtüşüyor. ■

Alıştırma 10.8 (Parabole En Yakın Nokta). $y = x^2$ parabolünün $(3, 0)$ noktasına en yakın noktasını ve bu en kısa uzaklığı bulunuz.

Çözüm.
Amaç fonksiyonu. Parabolün (x, x^2) noktasının $(3, 0)$'a uzaklığının karesi $d(x) = (x-3)^2 + x^4$'tür. Karekök artan bir fonksiyon olduğundan uzaklığı ve karesini aynı x en küçük yapar; karekökten kurtulmak hesabı kolaylaştırır.
Kod. `minimize_scalar` ile minimumu buluyor, sonra aynı noktayı $d'(x) = 4x^3 + 2(x-3)$ türevinin kökü olarak `brentq` ile doğruluyoruz.

```
import numpy as np
from scipy.optimize import brentq, minimize_scalar


def dist2(x):                       # (x, x²) ile (3, 0) arası uzaklığın karesi
    return (x - 3)**2 + x**4


res = minimize_scalar(dist2)
print(f"x = {res.x:.8f}, nokta = ({res.x:.6f}, {res.x**2:.6f})")
print(f"uzaklık = {np.sqrt(res.fun):.8f}, √5 = {np.sqrt(5):.8f}")

c = brentq(lambda x: 4*x**3 + 2*(x - 3), 0, 2)     # dist2'nin türevi
print("türevin kökü:", c)
```

Çıktı:

```
x = 1.00000000, nokta = (1.000000, 1.000000)
uzaklık = 2.23606798, √5 = 2.23606798
türevin kökü: 1.0
```

Kesin sonuç. $d'(x) = 0$ denklemi $2x^3 + x - 3 = 0$, yani $(x-1)(2x^2 + 2x + 3) = 0$ olur. İkinci çarpanın diskriminantı $4 - 24 < 0$ olduğundan tek kritik nokta $x = 1$'dir. En yakın nokta $(1, 1)$, en kısa uzaklık $\sqrt{(1-3)^2 + 1^2} = \sqrt{5} \approx 2{,}2361$'dir. ■

Alıştırma 10.9 (Karesel Bir Fonksiyonun Minimumu). $f(x, y) = x^2 + xy + y^2 - 3x$ fonksiyonunun minimumunu gradyanı vererek `minimize` ile bulunuz ve sonucu $\nabla f = \mathbf{0}$ lineer sistemini çözerek doğrulayınız.

Çözüm.
Gradyan ve Hessian. $\nabla f(x, y) = (2x + y - 3,\ x + 2y)$'dir. $\nabla f = \mathbf{0}$ koşulu

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$$

lineer sistemidir. Katsayı matrisi aynı zamanda f'nin Hessian matrisidir.
Kod.

 CC BY-NC-SA 4.0

```python
import numpy as np
from scipy.optimize import minimize


def f(v):
    x, y = v
    return x**2 + x*y + y**2 - 3*x


def grad(v):
    x, y = v
    return np.array([2*x + y - 3, x + 2*y])


res = minimize(f, x0=[0.0, 0.0], jac=grad, method="BFGS")
print("minimize:", res.x, " f =", res.fun, " nit =", res.nit)

A = np.array([[2.0, 1.0], [1.0, 2.0]])     # ∇f = 0  <=>  A v = b
b = np.array([3.0, 0.0])
print("solve   :", np.linalg.solve(A, b))
print("Hessian özdeğerleri:", np.linalg.eigvalsh(A))
```

Çıktı:

```
minimize: [ 2.00000101 -1.00000124]  f = -2.9999999999986944  nit = 5
solve   : [ 2. -1.]
Hessian özdeğerleri: [1. 3.]
```

Sonuç. Lineer sistemin çözümü $(2,-1)$'dir ve $f(2,-1) = 4 - 2 + 1 - 6 = -3$ olur. Hessian'ın özdeğerleri 1 ve 3 pozitif olduğundan $(2,-1)$ kesin yerel minimumdur; f konveks bir karesel fonksiyon olduğundan bu aynı zamanda mutlak minimumdur. BFGS aynı noktayı 5 iterasyonda, varsayılan gradyan toleransının izin verdiği 10^{-6} doğrulukla buldu. ■

Alıştırma 10.10 (Himmelblau Fonksiyonunun Minimumları). Himmelblau fonksiyonu adıyla bilinen

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

fonksiyonunu `minimize` ile $(1,1)$, $(-1,1)$, $(-1,-1)$ ve $(1,-1)$ başlangıç noktalarından minimize ediniz ve bulunan minimumları karşılaştırınız.

Çözüm.
Gradyan. $a = x^2 + y - 11$ ve $b = x + y^2 - 7$ dersek $f = a^2 + b^2$ ve zincir kuralıyla

$$\frac{\partial f}{\partial x} = 4xa + 2b, \qquad \frac{\partial f}{\partial y} = 2a + 4yb$$

bulunur.
Kod.

```python
import numpy as np
from scipy.optimize import minimize


def f(v):
    x, y = v
    return (x**2 + y - 11)**2 + (x + y**2 - 7)**2
```

```python
def grad(v):
    x, y = v
    a, b = x**2 + y - 11, x + y**2 - 7
    return np.array([4*x*a + 2*b, 2*a + 4*y*b])


for start in ([1, 1], [-1, 1], [-1, -1], [1, -1]):
    res = minimize(f, start, jac=grad, method="BFGS")
    print(f"başlangıç {str(start):8s} -> x = {res.x[0]: .6f}, "
          f"y = {res.x[1]: .6f}, f = {res.fun:.1e}")
```

Çıktı:

```
başlangıç [1, 1]   -> x =  3.000000, y =  2.000000, f = 2.1e-17
başlangıç [-1, 1]  -> x = -2.805118, y =  3.131312, f = 7.8e-14
başlangıç [-1, -1] -> x = -3.779310, y = -3.283186, f = 4.4e-13
başlangıç [1, -1]  -> x =  3.584428, y = -1.848127, f = 7.2e-16
```

Karşılaştırma. Dört başlangıç noktası dört farklı minimum verdi. Her birinde $f \approx 0$'dır. f iki karenin toplamı olduğundan $f \geq 0$'dır; dolayısıyla dört noktanın dördü de mutlak minimumdur. Örneğin $(3, 2)$ için $a = 9 + 2 - 11 = 0$ ve $b = 3 + 4 - 7 = 0$'dır. `minimize` hangi minimumun "havzasında" başlarsa onu bulur; başlangıç noktasını değiştirmek farklı minimumları keşfetmenin basit bir yoludur. ■

Alıştırma 10.11 (Düzleme En Yakın Nokta). $x + 2y + 3z = 14$ düzleminin orijine en yakın noktasını kısıtlı optimizasyonla bulunuz.

Çözüm.
Problem. Uzaklığın karesini, $f(\mathbf{v}) = x^2 + y^2 + z^2 = \mathbf{v} \cdot \mathbf{v}$'yi, $\mathbf{n} \cdot \mathbf{v} - 14 = 0$ eşitlik kısıtı altında en küçük yapıyoruz; burada $\mathbf{n} = (1, 2, 3)$ düzlemin normalidir. Gradyan $\nabla f = 2\mathbf{v}$'dir.

```python
import numpy as np
from scipy.optimize import minimize


def f(v):
    return v @ v                          # x² + y² + z²


n = np.array([1.0, 2.0, 3.0])             # düzlemin normal vektörü
cons = [{"type": "eq", "fun": lambda v: n @ v - 14}]
res = minimize(f, x0=np.zeros(3), jac=lambda v: 2 * v,
               method="SLSQP", constraints=cons)
print("nokta:", res.x)
print(f"uzaklık = {np.sqrt(res.fun):.10f}, √14 = {np.sqrt(14):.10f}")
```

Çıktı:

```
nokta: [1.00000013 2.00000006 2.99999989]
uzaklık = 3.7416573674, √14 = 3.7416573868
```

Kesin sonuç. Lagrange koşulu $2\mathbf{v} = \lambda \mathbf{n}$, en yakın noktanın normal doğrultusunda olduğunu söyler: $\mathbf{v} = t\,\mathbf{n}$. Kısıttan $t\,(\mathbf{n} \cdot \mathbf{n}) = 14t = 14$, yani $t = 1$ bulunur. En yakın nokta $(1, 2, 3)$, uzaklık $\sqrt{14} \approx 3{,}7416573868$'dir. SLSQP bunu varsayılan toleransının izin verdiği yaklaşık 10^{-7} doğrulukla verdi. ■

 CC BY-NC-SA 4.0

Bu bölümde gradyanları ya elle yazdık ya da SciPy'ın sonlu farklarla yaklaşık hesaplamasına bıraktık. Sonlu farkların ne kadar doğru olduğu ve integrallerin sayısal olarak nasıl hesaplandığı bir sonraki bölümün, Sayısal Türev ve İntegral bölümünün konusudur.

 CC BY-NC-SA 4.0

11. Sayısal Türev ve İntegral

Analiz derslerinde türev ve integral, kurallar ve formüllerle hesaplanır. Uygulamada ise fonksiyon çoğu zaman bir formül olarak değil, bir ölçüm tablosu ya da başka bir programın çıktısı olarak elimizdedir. Formül olsa bile ilkel her zaman bulunamaz: e^{-x^2} ve $\sin x/x$ fonksiyonlarının ilkelleri elementer fonksiyonlarla yazılamaz. SymPy ile Sembolik Hesap bölümünde SymPy'nin bu integraller için `erf` ve `Si` adlı özel fonksiyonlara başvurduğunu görmüştük. Böyle durumlarda türev ve integral, fonksiyonun sonlu sayıda noktadaki değerlerinden **sayısal** olarak hesaplanır.

Bu bölümde önce fark bölümleriyle türev alacağız ve hatanın adım boyuna bağlı davranışını inceleyeceğiz: adımı küçültmek bir noktaya kadar işe yarar, sonra sonucu bozar. Ardından ızgara verisinin türevini veren `np.gradient` fonksiyonunu, yamuk ve Simpson kurallarını (önce kendi kodumuzla, sonra `np.trapezoid` ve `scipy.integrate.simpson` ile), SciPy'nin hata tahmini de veren uyarlamalı `quad` fonksiyonunu ve katlı integraller için `dblquad` ile `tplquad` fonksiyonlarını kullanacağız. Formüllerin türetilmesi ve hata terimlerinin ispatı Nümerik Analiz notlarındadır (bkz. Nümerik Analiz ve Nümerik Analiz). Burada onları Python'da uygulayıp sınıyoruz.

Vektörizasyon ve Broadcasting bölümünde fark bölümlerini dizilerle hesaplamış ve merkezi farkın hatasının h^2 ile küçüldüğünü görmüştük. Şimdi aynı formüllerin ne kadar doğru olabileceğini ve doğruluğu neyin sınırladığını soruyoruz.

11.1 Fark Bölümleriyle Türev

Türevin tanımındaki limiti küçük ama sıfır olmayan bir adımda durdurmak, akla gelen ilk yaklaşımdır.

Tanım 11.1 (İleri ve Geri Fark Bölümü). f, x'i içeren bir aralıkta tanımlı ve $h > 0$ olsun.

$$D_h^+ f(x) = \frac{f(x+h) - f(x)}{h}, \qquad D_h^- f(x) = \frac{f(x) - f(x-h)}{h}$$

sayılarına f'nin x'teki **ileri fark bölümü** ve **geri fark bölümü** (forward and backward difference quotient), h sayısına da **adım** (step size) denir.

Yani ileri fark x'in sağındaki, geri fark ise solundaki bir noktayla kurulan kirişin eğimidir. $h \to 0$ iken ikisi de $f'(x)$'e yaklaşır; türevin tanımı tam olarak budur.

Tanım 11.2 (Merkezi Fark Bölümü). f, x'i içeren bir aralıkta tanımlı ve $h > 0$ olsun.

$$D_h f(x) = \frac{f(x+h) - f(x-h)}{2h}$$

sayısına f'nin x'teki **merkezi fark bölümü** (central difference quotient) denir.

Yani merkezi fark, x'in iki yanında eşit uzaklıktaki noktalardan geçen kirişin eğimidir. İleri ve geri fark bölümlerinin aritmetik ortalamasına eşittir:

$$D_h f(x) = \frac{D_h^+ f(x) + D_h^- f(x)}{2}.$$

İki formülü Python fonksiyonu olarak yazıp $f(x) = e^x$ için $x = 1$'de deneyelim. Kesin türev $f'(1) = e$'dir.

```
import math


def forward_diff(f, x, h):
    """İleri fark bölümü: (f(x + h) - f(x)) / h."""
```

 CC BY-NC-SA 4.0

```
    return (f(x + h) - f(x)) / h


def central_diff(f, x, h):
    """Merkezi fark bölümü: (f(x + h) - f(x - h)) / (2h)."""
    return (f(x + h) - f(x - h)) / (2 * h)


x = 1.0
exact = math.exp(x)                 # (e^x)' = e^x
print("     h    ileri fark hatası   merkezi fark hatası")
for h in [0.1, 0.01, 0.001]:
    e_fwd = abs(forward_diff(math.exp, x, h) - exact)
    e_cen = abs(central_diff(math.exp, x, h) - exact)
    print(f"{h:6.0e}   {e_fwd:17.3e}   {e_cen:19.3e}")
```

Çıktı:

```
     h    ileri fark hatası   merkezi fark hatası
 1e-01           1.406e-01             4.533e-03
 1e-02           1.364e-02             4.530e-05
 1e-03           1.360e-03             4.530e-07
```

Hata ileri farkta h ile, merkezi farkta h^2 ile orantılı küçülüyor: h on kat küçülünce birinci sütun on kat, ikinci sütun yüz kat küçülüyor. Bunun nedeni Taylor açılımıdır.

Önerme 11.1 (Fark Bölümlerinin Kesme Hatası). *$h > 0$ olsun.*

1. *$f, [x, x+h]$ aralığında C^2 sınıfındaysa $x < \xi < x+h$ olan bir ξ için*

$$D_h^+ f(x) - f'(x) = \frac{h}{2} f''(\xi)$$

   olur.
2. *$f, [x-h, x+h]$ aralığında C^3 sınıfındaysa $x-h < \eta < x+h$ olan bir η için*

$$D_h f(x) - f'(x) = \frac{h^2}{6} f'''(\eta)$$

   olur.

Birinci eşitliğin ispatı için bkz. Nümerik Analiz. İkinci eşitlik oradaki üç-nokta orta nokta formülüdür (bkz. Nümerik Analiz).

Yani hata ifadesindeki h'nin kuvveti yaklaşımın **mertebesini** (order) verir: ileri fark bölümünün hatası $O(h)$, merkezi farkınki $O(h^2)$'dir (bkz. Nümerik Analiz). İleri farka birinci mertebeden, merkezi farka ikinci mertebeden bir yaklaşım denir. $f(x) = e^x$ için $f'' = f''' = e^x$ olduğundan $x = 1$ civarında hatalar yaklaşık $\frac{e}{2}h$ ve $\frac{e}{6}h^2$'dir. $h = 0{,}001$ için bunlar yaklaşık $1{,}36 \cdot 10^{-3}$ ve $4{,}53 \cdot 10^{-7}$ eder; tablodaki sayılarla uyuşur.

11.2 Adımı Seçmek: Kesme ve Yuvarlama Hatası

Hata formülleri, h'yi olabildiğince küçük seçmeyi önerir gibidir. Deneyelim: aynı türevi $h = 10^{-1}, 10^{-2}, \ldots, 10^{-15}$ adımlarıyla hesaplayalım.

```
import math

x = 1.0
exact = math.exp(x)
print("     h    ileri fark hatası   merkezi fark hatası")
for k in range(1, 16):
```

 CC BY-NC-SA 4.0

```
    h = 10.0**-k
    fwd = (math.exp(x + h) - math.exp(x)) / h
    cen = (math.exp(x + h) - math.exp(x - h)) / (2 * h)
    print(f"{h:6.0e}   {abs(fwd - exact):17.3e}"
          f"   {abs(cen - exact):19.3e}")
```

Çıktı:

```
     h    ileri fark hatası   merkezi fark hatası
1e-01           1.406e-01             4.533e-03
1e-02           1.364e-02             4.530e-05
1e-03           1.360e-03             4.530e-07
1e-04           1.359e-04             4.531e-09
1e-05           1.359e-05             5.859e-11
1e-06           1.359e-06             1.635e-10
1e-07           1.399e-07             5.859e-11
1e-08           6.603e-09             6.603e-09
1e-09           2.154e-07             6.603e-09
1e-10           1.548e-06             6.727e-07
1e-11           3.263e-05             1.043e-05
1e-12           4.323e-04             2.103e-04
1e-13           4.559e-04             4.559e-04
1e-14           9.338e-03             9.338e-03
1e-15           3.903e-01             1.683e-01
```

Tablo beklenmedik bir şey söylüyor. İleri farkın hatası $h = 10^{-8}$'e kadar küçülüyor, sonra büyümeye başlıyor; $h = 10^{-15}$'te türevin ilk ondalık basamağı bile yanlış. Merkezi fark en iyi sonucunu $h = 10^{-5}$ civarında veriyor; $h = 10^{-7}$'de aynı küçüklükte bir hatanın çıkması ise bir rastlantıdır ve nedenini birazdan göreceğiz. Bozulmanın kaynağı paydaki çıkarmadır:

```
import math

h = 1e-12
a = math.exp(1.0 + h)
b = math.exp(1.0)
print(f"e^(1 + h)   = {a:.17f}")
print(f"e^1         = {b:.17f}")
print(f"fark        = {a - b:.6e}")
print(f"e * h       = {math.e * h:.6e}")
print(f"(1 + h) - 1 = {(1.0 + h) - 1.0:.6e}")
```

Çıktı:

```
e^(1 + h)   = 2.71828182846176380
e^1         = 2.71828182845904509
fark        = 2.718714e-12
e * h       = 2.718282e-12
(1 + h) - 1 = 1.000089e-12
```

e^{1+h} ile e^1'in ilk on ondalık basamağı aynıdır. İki sayı da makinede yaklaşık 16 anlamlı basamakla saklandığı için farklarının yalnız ilk dört basamağı doğru çıkar: $2{,}718282 \cdot 10^{-12}$ olması gereken fark $2{,}718714 \cdot 10^{-12}$ bulundu. Bu, Python ile İlk Adımlar bölümünde gördüğümüz sadeleşme hatasıdır: birbirine çok yakın iki sayının farkında anlamlı basamaklar kaybolur. Adımın kendisi bile tam değildir: $1 + h$ makinede tam gösterilemediği için $(1 + h) - 1$ farkı 10^{-12} yerine $1{,}000089 \cdot 10^{-12}$ çıkar. Bu hataların etkisi bölme sırasında $1/h$ ile büyür.

Önerme 11.2 (Yuvarlama Hatasıyla Birlikte Toplam Hata). *f'nin değerleri makinede $\tilde{f}$ olarak hesaplansın ve x'in yakınındaki her t için $|\,\tilde{f}(t) - f(t)\,| \le \delta$ olsun. Çıkarma ve bölmenin kendi yuvarlamasını ihmal edelim. $\tilde{D}_h^+ f(x)$ ve $\tilde{D}_h f(x)$, fark bölümlerinin $\tilde{f}$ değerleriyle hesaplanan karşılıkları olsun.*

1. x'in yakınındaki her t için $|\,f''(t)\,| \le M_2$ ise

$$\left| \tilde{D}_h^+ f(x) - f'(x) \right| \le \frac{M_2\, h}{2} + \frac{2\delta}{h}$$

olur. Sağ taraf en küçük değerini $h_+ = 2\sqrt{\delta/M_2}$ *adımında alır.*

2. x*'in yakınındaki her* t *için* $|\, f'''(t)\,| \le M_3$ *ise*

$$\left| \tilde{D}_h f(x) - f'(x) \right| \le \frac{M_3\, h^2}{6} + \frac{\delta}{h}$$

olur. Sağ taraf en küçük değerini $h_0 = \sqrt[3]{3\delta/M_3}$ *adımında alır.*

İspat.
İleri fark. $\tilde{f}(t) = f(t) + r(t)$ *yazalım;* $|\, r(t)\,| \le \delta$*'dır. Bu durumda*

$$\tilde{D}_h^+ f(x) = D_h^+ f(x) + \frac{r(x+h) - r(x)}{h}$$

olur. Önerme 11.1 gereği birinci terimin $f'(x)$*'ten farkı* $\frac{h}{2}\,|\, f''(\xi)\,| \le \frac{M_2 h}{2}$*'dir. İkinci terimin mutlak değeri en çok* $\frac{|\, r(x+h)\,| + |\, r(x)\,|}{h} \le \frac{2\delta}{h}$*'dir. Üçgen eşitsizliği birinci sınırı verir.*
Merkezi fark. *Aynı ayrıştırmayla*

$$\tilde{D}_h f(x) - f'(x) = \left(D_h f(x) - f'(x) \right) + \frac{r(x+h) - r(x-h)}{2h}$$

olur. Parantezin mutlak değeri en çok $\frac{M_3 h^2}{6}$*, ikinci terimin mutlak değeri en çok* $\frac{2\delta}{2h} = \frac{\delta}{h}$*'dir.*
En küçük değer. $A, B > 0$ *için* $g(h) = Ah + B/h$ *fonksiyonunun türevi* $g'(h) = A - B/h^2$*'dir. Bu türev yalnız* $h = \sqrt{B/A}$*'da sıfır olur ve* $g''(h) = 2B/h^3 > 0$ *olduğundan burası* $(0, \infty)$*'daki en küçük değerdir.* $A = M_2/2$ *ve* $B = 2\delta$ *için* $h_+ = \sqrt{4\delta/M_2} = 2\sqrt{\delta/M_2}$ *bulunur. Benzer biçimde* $g(h) = Ah^2 + B/h$ *için* $g'(h) = 2Ah - B/h^2 = 0$ *denklemi* $h^3 = B/(2A)$ *verir ve* $g''(h) = 2A + 2B/h^3 > 0$*'dır.* $A = M_3/6$ *ve* $B = \delta$ *için* $h_0^3 = 3\delta/M_3$ *olur.*
■

Yani toplam hata iki kaynaktan gelir: h küçüldükçe azalan **kesme hatası** (truncation error) ve h küçüldükçe büyüyen **yuvarlama hatası** (round-off error) (bkz. Nümerik Analiz ve Nümerik Analiz). Makinede δ kabaca $\varepsilon\,|\, f(x)\,|$ kadardır; burada $\varepsilon = 2^{-52} \approx 2{,}2 \cdot 10^{-16}$ makine epsilonudur. f ve türevleri 1 mertebesindeyse en iyi adımlar ileri fark için $h_+ \approx \sqrt{\varepsilon} \approx 10^{-8}$, merkezi fark için $h_0 \approx \sqrt[3]{\varepsilon} \approx 10^{-5}$ olur. Bu adımlarda hata sınırları da yaklaşık $\sqrt{\varepsilon} \approx 10^{-8}$ ve $\varepsilon^{2/3} \approx 10^{-11}$'dir. Demek ki merkezi fark yalnız daha hızlı yakınsamakla kalmaz, ulaşabildiği doğruluk da daha yüksektir.

Örnek 11.1 (Üstel Fonksiyonda En İyi Adım). $f(x) = e^x$ için $x = 1$'de $\delta = \varepsilon e$ ve $M_2 = M_3 = e$ alın. Önerme 11.2 ile verilen en iyi adımları ve bu adımlardaki hata sınırlarını hesaplayıp adım taramasının tablosuyla karşılaştırın.

Çözüm.
Formüller. $h_+ = 2\sqrt{\delta/M_2}$ ve $h_0 = \sqrt[3]{3\delta/M_3}$ adımlarını ve sınırların bu adımlardaki değerlerini doğrudan koda dökeriz:

```
import math

eps = 2.0**-52                  # makine epsilonu
x = 1.0
delta = eps * math.exp(x)       # bir f değerindeki yuvarlama hatası
M = math.exp(x)                 # x yakınında |f''| ve |f'''| yaklaşık e

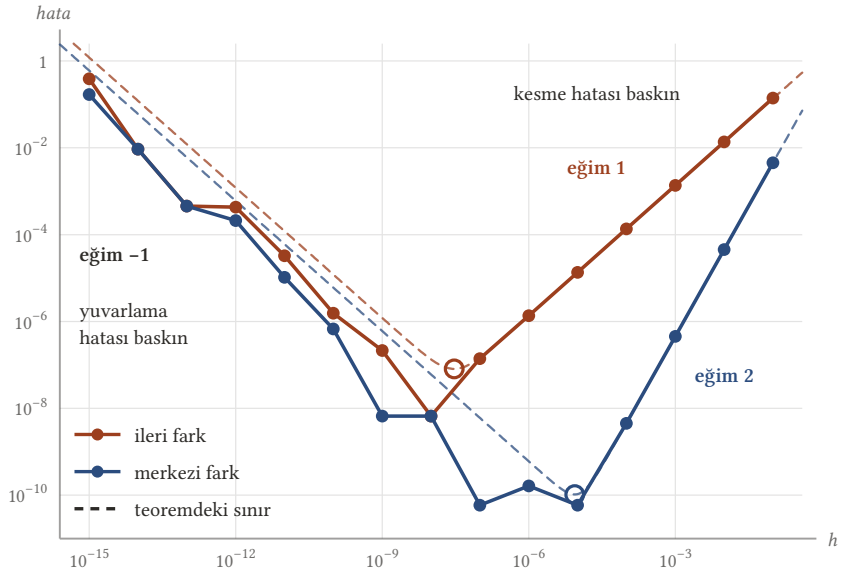
h_fwd = 2 * math.sqrt(delta / M)
bound_fwd = M * h_fwd / 2 + 2 * delta / h_fwd
h_cen = (3 * delta / M) ** (1 / 3)
bound_cen = M * h_cen**2 / 6 + delta / h_cen
print(f"ileri fark  : h = {h_fwd:.2e}   sınır = {bound_fwd:.2e}")
print(f"merkezi fark: h = {h_cen:.2e}   sınır = {bound_cen:.2e}")
```

Çıktı:

```
ileri fark  : h = 2.98e-08   sınır = 8.10e-08
merkezi fark: h = 8.73e-06   sınır = 1.04e-10
```

Karşılaştırma. İleri fark için önerilen adım $h_+ \approx 3 \cdot 10^{-8}$'dir; tabloda en küçük hata ($6{,}6 \cdot 10^{-9}$) $h = 10^{-8}$'de çıkmıştı. Merkezi fark için önerilen adım $h_0 \approx 8{,}7 \cdot 10^{-6}$'dır; tabloda en küçük hata ($5{,}9 \cdot 10^{-11}$) $h = 10^{-5}$'te çıkmıştı. Aynı hata $h = 10^{-7}$'de de çıkmıştı, ama orada yuvarlama hataları tesadüfen birbirini büyük ölçüde götürmüştür. Sınır $h = 10^{-7}$'de $6 \cdot 10^{-9}$'dur, yani bu adımda hata yüz kat büyük de çıkabilirdi; nitekim aradaki $h = 10^{-6}$ adımında hata yeniden $1{,}6 \cdot 10^{-10}$'dur. Güvenilir olan yalnız $h = 10^{-5}$'teki sonuçtur. Gerçek hatalar sınırların altında kalır, çünkü sınır en kötü durumu varsayar: iki yuvarlama hatasının hem en büyük hem de ters işaretli olduğunu. Yine de sınırlar en iyi adımın büyüklüğünü doğru tahmin eder.
■

Aşağıdaki şekil, tablonun iki sütununu ve teoremdeki iki sınırı aynı logaritmik eksenlerde gösteriyor. Hata eğrileri V biçimindedir: sağ kolda kesme hatası, sol kolda yuvarlama hatası baskındır.



Şekil 11.1: e^x fonksiyonunun $x = 1$'deki türevi için ileri ve merkezi farkın hatası, iki eksen de logaritmik; adım taraması kodunun tablosundaki sayıların aynısı. Sağda kesme hatası baskındır ve doğrular 1 ile 2 eğimiyle iner. Solda yuvarlama hatası baskındır ve hata $1/h$ gibi büyür. Kesikli eğriler teoremdeki sınırlardır; içi boş halkalar sınırın en küçük olduğu adımları, yaklaşık $3{\cdot}10^{-8}$ ve $9{\cdot}10^{-6}$ değerlerini gösterir.

⚠ Adımı küçültmek doğruluğu her zaman artırmaz

Fark bölümlerinde h küçüldükçe yuvarlama hatası $1/h$ gibi büyür. 10^{-12} gibi çok küçük bir adım, 10^{-5} gibi orta bir adımdan çok daha kötü sonuç verir. Ayrıca adım, x'in ve f'nin ölçeğine göre seçilmelidir: $x = 10^6$ civarında 10^{-5} gibi bir adım x'in son basamaklarına düşer; $x + h$ toplamında adımın yalnız birkaç anlamlı basamağı korunur.

 CC BY-NC-SA 4.0

Fark bölümüyle türev üç adımda

1. Merkezi farkı seç. İleri ya da geri farkı yalnız fonksiyon x'in bir yanında tanımlı değilse kullan.
2. Adımı ölçekle: $\delta \approx \varepsilon \mid f(x) \mid$ ve $M_3 \approx \mid f'''(x) \mid$ alarak $h = \sqrt[3]{3\delta/M_3}$ hesapla. Bu bilgiler yoksa $h = \sqrt[3]{\varepsilon}\max(1, \mid x \mid)$ kaba ama makul bir başlangıçtır.
3. Türevi h ve $2h$ adımlarıyla hesaplayıp karşılaştır. İki sonucun farkı hatanın büyüklüğü hakkında fikir verir; fark beklenenden büyükse adımı yeniden düşün.

Örnek 11.2 (Büyük Bir Noktada Logaritmanın Türevi). $f(x) = \ln x$ fonksiyonunun $x = 10^6$ noktasındaki türevini merkezi farkla, adımı yukarıdaki üç adıma göre seçerek hesaplayın. Sonucu ezbere seçilmiş $h = 10^{-5}$ adımının sonucuyla karşılaştırın. Kesin değer $f'(10^6) = 10^{-6}$'dır.

Çözüm.
1. adım. $\ln x$, 10^6'nın iki yanında da tanımlıdır; merkezi farkı kullanırız.
2. adım. $\delta = \varepsilon \ln(10^6) \approx 3{,}1 \cdot 10^{-15}$'tir. $f'''(x) = 2/x^3$ olduğundan $M_3 = 2 \cdot 10^{-18}$ alırız. Önerilen adım $h = \sqrt[3]{3\delta/M_3} \approx 16{,}6$ çıkar. Bu adım ilk bakışta büyük görünür, ama 10^6'nın yanında küçüktür: $\ln x$ bu ölçekte çok yavaş değişir.
3. adım. h ile $2h$ adımlarının sonuçlarını karşılaştırırız. Kıyas için $h = 10^{-5}$ adımını da deneriz:

```
import math


def central_diff(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)


eps = 2.0**-52
x = 1.0e6
exact = 1 / x                       # (ln x)' = 1/x

# 2. adım: delta = eps |ln x| ve M3 = |f'''(x)| = 2 / x^3
delta = eps * abs(math.log(x))
M3 = 2 / x**3
h = (3 * delta / M3) ** (1 / 3)
print(f"önerilen h = {h:.2f}")

# 3. adım: h ile 2h karşılaştırılır; kıyas için ezbere h = 1e-5
d1 = central_diff(math.log, x, h)
d2 = central_diff(math.log, x, 2 * h)
print(f"D(h) ile D(2h) arasındaki bağıl fark: {abs(d1 - d2) / d1:.1e}")
for step in [h, 2 * h, 1e-5]:
    d = central_diff(math.log, x, step)
    rel = abs(d - exact) / exact
    print(f"h = {step:8.3g}   bağıl hata = {rel:.1e}")
```

Çıktı:

```
önerilen h = 16.63
D(h) ile D(2h) arasındaki bağıl fark: 2.7e-10
h =     16.6   bağıl hata = 1.0e-10
h =     33.3   bağıl hata = 3.7e-10
h =    1e-05   bağıl hata = 8.3e-08
```

Sonuç. $D(h)$ ile $D(2h)$ arasındaki bağıl fark $2{,}7 \cdot 10^{-10}$'dur; gerçek bağıl hata da bu mertebededir ($1{,}0 \cdot 10^{-10}$). Teoremdeki sınır, $h = 10^{-5}$ adımında δ/h teriminden gelen $3 \cdot 10^{-4}$ kadar bir bağıl hataya izin verir. Bu çalıştırmada yuvarlama hataları kısmen birbirini götürdüğü için gerçek hata daha küçük çıktı, ama yine de önerilen adımdakinin yaklaşık 800 katıdır.
■

 CC BY-NC-SA 4.0

11.3 Izgarada Türev: np.gradient

Şimdiye kadar f'yi istediğimiz noktada hesaplayabiliyorduk. Fonksiyon bir tablo olarak verildiğinde ise adımı biz seçemeyiz; elimizdeki noktalarla yetinmek zorundayız.

Tanım 11.3 (np.gradient ile Izgarada Türev). $x_0 < x_1 < \ldots < x_n$ noktalarında $y_k = f(x_k)$ değerleri verilsin. `np.gradient(y, x)` çağrısı, k indisli elemanı $f'(x_k)$'ye bir yaklaşım olan $n+1$ elemanlı bir dizi döndürür.

- İç noktalarda $(0 < k < n)$ x_{k-1}, x_k, x_{k+1} noktalarından geçen ikinci dereceden polinomun x_k'deki türevini alır. Aralıklar eşitse bu, merkezi fark bölümüdür: $(y_{k+1} - y_{k-1})/(2h)$.
- Uç noktalarda varsayılan olarak tek yönlü fark bölümünü kullanır: $(y_1 - y_0)/(x_1 - x_0)$ ve $(y_n - y_{n-1})/(x_n - x_{n-1})$. `edge_order=2` verilirse ilk ya da son üç noktadan geçen ikinci dereceden polinomun uçtaki türevini alır.

Noktalar eşit aralıklıysa `x` dizisi yerine adım da verilebilir: `np.gradient(y, h)`.

Yani `np.gradient` iç noktalarda ikinci mertebeden, uçlarda ise varsayılan hâlde yalnız birinci mertebeden bir yaklaşım verir; `edge_order=2` uçları da ikinci mertebeye çıkarır. Eşit aralıkta bunlar Nümerik Analiz notlarındaki üç-nokta formülleridir (bkz. Nümerik Analiz). Dokuz noktalı bir ızgarada $\cos x$'in türevini hesaplayıp kesin türev $-\sin x$ ile karşılaştıralım:

```
import numpy as np

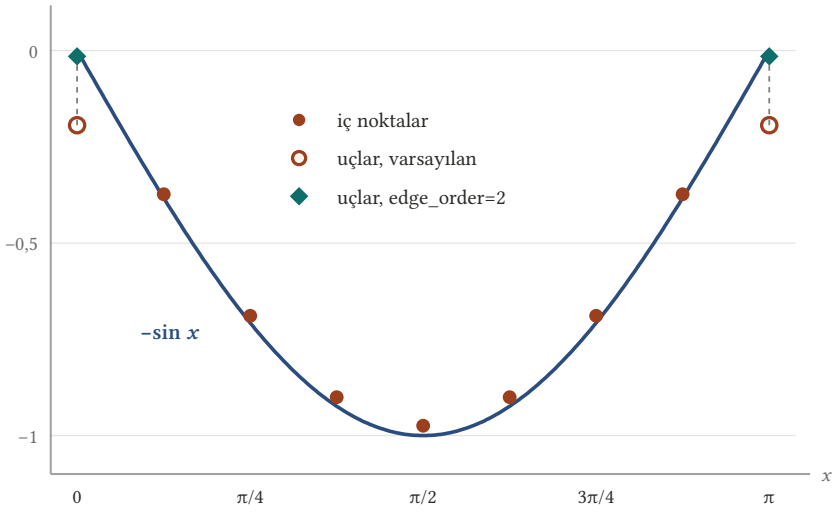
x = np.linspace(0, np.pi, 9)              # adım pi/8
y = np.cos(x)
exact = -np.sin(x)
g1 = np.gradient(y, x)                    # uçlarda birinci mertebe
g2 = np.gradient(y, x, edge_order=2)      # uçlarda ikinci mertebe
print(np.round(g1, 4))
print(np.round(g2, 4))
print(np.round(exact, 4) + 0.0)           # + 0.0: -0. yerine 0. yazar
err1 = np.abs(g1 - exact)
err2 = np.abs(g2 - exact)
print("iç noktalarda en büyük hata:", round(err1[1:-1].max(), 4))
print("uçlarda hata, varsayılan   :", np.round(err1[[0, -1]], 4))
print("uçlarda hata, edge_order=2 :", np.round(err2[[0, -1]], 4))
```

Çıktı:

```
[-0.1938 -0.3729 -0.6891 -0.9003 -0.9745 -0.9003 -0.6891 -0.3729 -0.1938]
[-0.0148 -0.3729 -0.6891 -0.9003 -0.9745 -0.9003 -0.6891 -0.3729 -0.0148]
[ 0.     -0.3827 -0.7071 -0.9239 -1.     -0.9239 -0.7071 -0.3827  0.    ]
iç noktalarda en büyük hata: 0.0255
uçlarda hata, varsayılan   : [0.1938 0.1938]
uçlarda hata, edge_order=2 : [0.0148 0.0148]
```

İç noktalardaki en büyük hata 0,0255'tir. Bu, Önerme 11.1 ile uyumludur: $h = \pi/8$ için $\frac{h^2}{6} \approx 0{,}0257$ eder. Uçlarda ise varsayılan tek yönlü fark 0,19 kadar sapar; `edge_order=2` bu sapmayı 0,015'e indirir.

 CC BY-NC-SA 4.0



Şekil 11.2: `np.gradient` kodunun dokuz noktada bulduğu türevler ve kesin türev $-\sin x$. İç noktalardaki merkezi farklar eğriye çok yakındır. Uçlardaki tek yönlü fark ise varsayılan hâlde 0,19 kadar sapar (içi boş halkalar); `edge_order=2` ile bu sapma 0,015'e iner (yeşil karolar).

Örnek 11.3 (Düzensiz Aralıklı Ölçümlerden Hız). Bir cismin konumu $t = 0; 0{,}5; 1{,}2; 2; 2{,}5; 3$ saniyelerinde ölçülmüş olsun ve konum $s(t) = 2t^2$ metre kuralına uysun. Bu altı ölçümden cismin hızını `np.gradient` ile hesaplayın ve kesin hız $v(t) = 4t$ ile karşılaştırın.

Çözüm.
Çağrı. Ölçüm anları eşit aralıklı olmadığı için zaman dizisini ikinci argüman olarak veririz. Karşılaştırma için son satırda `t` dizisini bilerek vermiyoruz:

```
import numpy as np

t = np.array([0.0, 0.5, 1.2, 2.0, 2.5, 3.0])      # saniye
s = 2 * t**2                                       # metre
print(np.gradient(s, t))
print(np.round(np.gradient(s, t, edge_order=2), 6))
print(4 * t)                                       # kesin hız
print(np.gradient(s))                              # YANLIŞ: adım 1 sanılır
```

Çıktı:

```
[ 1.   2.   4.8  8.  10.  11. ]
[ 0.   2.   4.8  8.  10.  12. ]
[ 0.   2.   4.8  8.  10.  12. ]
[0.5  1.44 3.75 4.81 5.   5.5 ]
```

Sonuç. İç noktalarda sonuç tam olarak kesin hızdır, çünkü üç noktadan geçen ikinci dereceden polinom s'nin kendisidir. Uçlarda varsayılan tek yönlü fark 1 ve 11 verir (kesin değerler 0 ve 12); `edge_order=2` uçları da düzeltir. Son satır sık yapılan bir hatadır: `x` verilmezse `np.gradient` noktalar arasındaki uzaklığı 1 sayar ve sonuç anlamsız olur.
■

`np.gradient` çok boyutlu dizilerde her eksen boyunca ayrı ayrı çalışır ve kısmi türevleri bir demet olarak döndürür. $f(x, y) = \sin x \cos y$ fonksiyonunun kısmi türevlerini $[0, 2] \times [0, 1]$ üzerindeki bir ızgarada hesaplayalım:

 CC BY-NC-SA 4.0

```
import numpy as np

x = np.linspace(0, 2, 201)                    # adım 0.01
y = np.linspace(0, 1, 101)                    # adım 0.01
X, Y = np.meshgrid(x, y, indexing="ij")       # X[i, j] = x[i], Y[i, j] = y[j]
F = np.sin(X) * np.cos(Y)
Fx, Fy = np.gradient(F, x, y, edge_order=2)   # eksen 0 -> x, eksen 1 -> y
i, j = 100, 50
print(X[i, j], Y[i, j])
print(Fx[i, j], np.cos(1) * np.cos(0.5))
print(Fy[i, j], -np.sin(1) * np.sin(0.5))
print(np.abs(Fx - np.cos(X) * np.cos(Y)).max())
print(np.abs(Fy + np.sin(X) * np.sin(Y)).max())
```

Çıktı:

```
1.0 0.5
0.4741519791538522 0.4741598817790379
-0.40341595643361927 -0.4034226801113349
3.333216667877892e-05
2.791296891380135e-05
```

`indexing="ij"` seçeneği, dizinin 0 numaralı ekseninin x'e, 1 numaralı ekseninin y'ye karşılık gelmesini sağlar; böylece `np.gradient(F, x, y)` önce $\partial f/\partial x$'i, sonra $\partial f/\partial y$'yi verir. $(1;\,0{,}5)$ noktasında iki kısmi türevin de ilk dört ondalık basamağı doğrudur. Bütün ızgaradaki en büyük hata $3 \cdot 10^{-5}$ civarındadır; adım $0{,}01$ olduğundan bu, ikinci mertebeden bir yöntemden beklenen büyüklüktür.

11.4 Yamuk ve Simpson Kuralları

İntegrale geçelim. Belirli integrali hesaplamak için $[a, b]$ aralığını n eşit parçaya böler, her parçada f'yi integrali kolay alınan bir polinomla değiştiririz.

Tanım 11.4 (Bileşik Yamuk Kuralı). $n \geq 1$, $h = (b-a)/n$ ve $k = 0, 1, \ldots, n$ için $x_k = a + kh$ olsun.

$$T_n = h\left[\frac{f(x_0)}{2} + f(x_1) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}\right]$$

sayısına $\int_a^b f(x)\,dx$ için **bileşik yamuk kuralı** (composite trapezoidal rule) yaklaşımı denir.

Yani her $[x_k, x_{k+1}]$ parçasında f'yi uçlardaki değerleri birleştiren doğruyla değiştirip n yamuğun alanını toplarız (tek parça için bkz. Nümerik Analiz). İç noktalar iki yamuğa birden katıldığı için ağırlıkları h, uç noktaların ağırlığı $h/2$'dir.

Tanım 11.5 (Bileşik Simpson Kuralı). n çift, $h = (b-a)/n$ ve $k = 0, 1, \ldots, n$ için $x_k = a + kh$ olsun.

$$S_n = \frac{h}{3}\Big[\, f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots \\ + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\,\Big]$$

sayısına $\int_a^b f(x)\,dx$ için **bileşik Simpson kuralı** (composite Simpson's rule) yaklaşımı denir.

Yani $[x_0, x_2], [x_2, x_4], \ldots$ ikili parçalarının her birinde f'yi üç noktadan geçen bir parabolle değiştirip parabollerin altındaki alanları toplarız (tek parça için bkz. Nümerik Analiz). Ağırlıklar tek indisli noktalarda $4h/3$, iç çift indisli noktalarda $2h/3$, uçlarda $h/3$'tür.
İki kuralı NumPy dilimleriyle birkaç satırda yazabiliriz: `y[1:-1]` iç noktaları, `y[1:-1:2]` tek indisli noktaları, `y[2:-1:2]` ise iç çift indisli noktaları seçer. Örnek olarak ilkeli elementer olmayan

$$\int_0^2 e^{-x^2}\,dx = \frac{\sqrt{\pi}}{2}\,\mathrm{erf}(2)$$

 CC BY-NC-SA 4.0

integralini alalım; kesin değeri `math.erf` ile hesaplarız.

```
import math

import numpy as np


def trapezoid_rule(f, a, b, n):
    """Bileşik yamuk kuralı, n eşit alt aralık."""
    x = np.linspace(a, b, n + 1)
    y = f(x)
    h = (b - a) / n
    return h * (y[0] / 2 + y[1:-1].sum() + y[-1] / 2)


def simpson_rule(f, a, b, n):
    """Bileşik Simpson kuralı; n çift olmalı."""
    if n % 2:
        raise ValueError("Simpson kuralı için n çift olmalı")
    x = np.linspace(a, b, n + 1)
    y = f(x)
    h = (b - a) / n
    return h / 3 * (y[0] + 4 * y[1:-1:2].sum()
                    + 2 * y[2:-1:2].sum() + y[-1])


def f(x):
    return np.exp(-x**2)

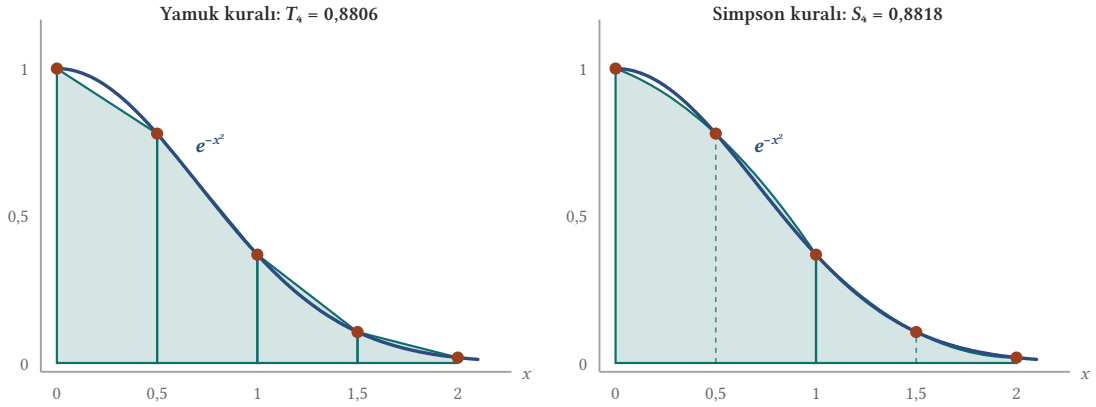

exact = math.sqrt(math.pi) / 2 * math.erf(2)
print(f"kesin değer: {exact:.10f}")
print(f"T_4 = {trapezoid_rule(f, 0, 2, 4):.10f}")
print(f"S_4 = {simpson_rule(f, 0, 2, 4):.10f}")
print("   n   yamuk hatası   oran   Simpson hatası   oran")
old_t = old_s = None
for n in [4, 8, 16, 32, 64, 128, 256]:
    et = abs(trapezoid_rule(f, 0, 2, n) - exact)
    es = abs(simpson_rule(f, 0, 2, n) - exact)
    rt = f"{old_t / et:6.2f}" if old_t else "     -"
    rs = f"{old_s / es:6.2f}" if old_s else "     -"
    print(f"{n:4d}   {et:12.3e}  {rt}   {es:14.3e}  {rs}")
    old_t, old_s = et, es
```

Çıktı:

```
kesin değer: 0.8820813908
T_4 = 0.8806186341
S_4 = 0.8818124253
   n   yamuk hatası   oran   Simpson hatası   oran
   4      1.463e-03       -        2.690e-04       -
   8      3.776e-04    3.87        1.588e-05   16.94
  16      9.515e-05    3.97        9.942e-07   15.97
  32      2.383e-05    3.99        6.212e-08   16.01
  64      5.961e-06    4.00        3.882e-09   16.00
 128      1.490e-06    4.00        2.426e-10   16.00
 256      3.726e-07    4.00        1.516e-11   16.00
```

$n = 4$ için iki yaklaşımı aşağıdaki şekilde görüyoruz. Simpson kuralı aynı beş fonksiyon değeriyle yamuk kuralından beş kat daha küçük hata veriyor.

 CC BY-NC-SA 4.0



Şekil 11.3: e^{-x^2} fonksiyonunun $[0, 2]$ aralığındaki integrali, $n = 4$ alt aralıkla; kuralları kendimiz yazdığımız kodun ilk iki sonucu. Solda her alt aralıktaki yamuk, sağda ikişer alt aralığı kaplayan iki parabol. Kesin değer 0,8821'dir: parabollerin altındaki alan yamukların alanından daha yakındır.

Tablonun "oran" sütunları, n ikiye katlanınca hatanın yamuk kuralında 4'e, Simpson kuralında 16'ya bölündüğünü gösteriyor. Bunun nedeni hata terimleridir.

Önerme 11.3 (Bileşik Kuralların Hatası). *$h = (b-a)/n$ olsun.*

1. *$f \in C^2[a,b]$ ise bir $\mu \in (a,b)$ için*

$$\int_a^b f(x)\,dx - T_n = -\frac{(b-a)\,h^2}{12}\,f''(\mu)$$

olur.

2. *n çift ve $f \in C^4[a,b]$ ise bir $\mu \in (a,b)$ için*

$$\int_a^b f(x)\,dx - S_n = -\frac{(b-a)\,h^4}{180}\,f^{(4)}(\mu)$$

olur.

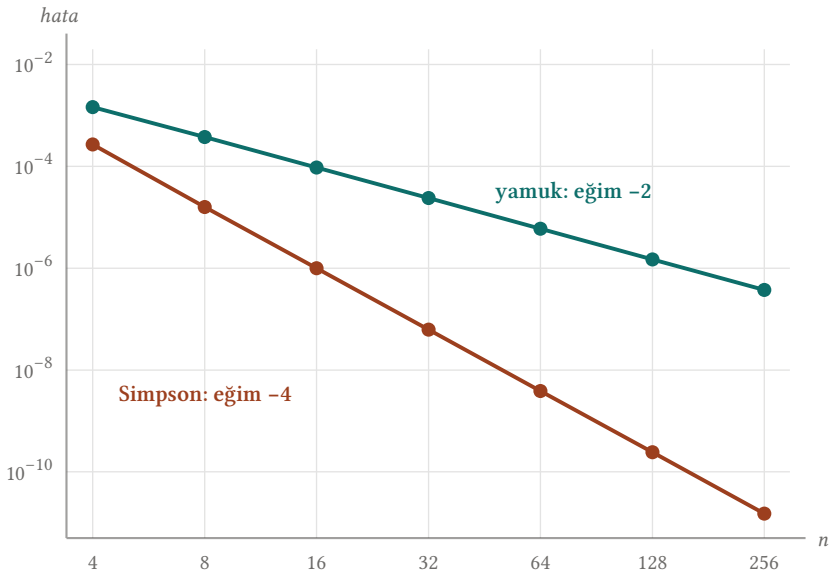
İspat.
Yamuk. *Tek bir $[x_k, x_{k+1}]$ parçasında yamuk kuralının hatası, bir $\xi_k \in (x_k, x_{k+1})$ için $-\frac{h^3}{12}f''(\xi_k)$'dir (bkz. Nümerik Analiz). n parçayı toplarız:*

$$\int_a^b f(x)\,dx - T_n = -\frac{h^3}{12}\sum_{k=0}^{n-1} f''(\xi_k).$$

$\frac{1}{n}\sum_k f''(\xi_k)$ ortalaması, $f''(\xi_k)$ sayılarının en küçüğü ile en büyüğü arasındadır. f'' sürekli olduğundan, ara değer teoremini bu iki değerin alındığı noktalar arasında uygularsak ortalamaya eşit olan bir $f''(\mu)$ değeri buluruz; μ bu iki nokta arasında, dolayısıyla (a,b) içindedir. Böylece toplam $nf''(\mu)$ olur ve $nh^3 = (b-a)h^2$ eşitliği birinci formülü verir.
Simpson. *Her $[x_{2j}, x_{2j+2}]$ parçasında Simpson kuralının hatası, bir ξ_j için $-\frac{h^5}{90}f^{(4)}(\xi_j)$'dir (bkz. Nümerik Analiz). $n/2$ parçayı toplayıp aynı ortalama argümanını kullanırız: toplam hata $-\frac{h^5}{90}\cdot\frac{n}{2}\,f^{(4)}(\mu)$ olur ve $\frac{n}{2}h^5 = \frac{(b-a)h^4}{2}$ eşitliği ikinci formülü verir.*
■

Yani yamuk kuralı ikinci, Simpson kuralı dördüncü mertebedendir. n ikiye katlanınca h yarıya iner; yamuk kuralının hatası yaklaşık $2^2 = 4$'e, Simpson kuralınınki $2^4 = 16$'ya bölünür. Simpson kuralı derecesi en çok 3 olan polinomları tam integre eder, çünkü onların dördüncü türevi sıfırdır. Aynı tablo logaritmik eksenlerde iki doğru verir ve doğruların eğimleri mertebeleri gösterir:

 CC BY-NC-SA 4.0



Şekil 11.4: Kuralları kendimiz yazdığımız kodun tablosu, iki eksen de logaritmik. n her ikiye katlandığında yamuk kuralının hatası 4'e, Simpson kuralınınki 16'ya bölünür: doğruların eğimleri -2 ve -4'tür, yani hatalar h^2 ve h^4 ile orantılıdır.

11.5 Örneklenmiş Veriyle İntegral

Kendi yazdığımız fonksiyonlar f'yi bir Python fonksiyonu olarak alıyordu. Bir ölçüm tablosunda ise elimizde yalnız sayılar vardır; NumPy ve SciPy'nin hazır fonksiyonları doğrudan bu sayılarla çalışır.

Tanım 11.6 (np.trapezoid ve scipy.integrate.simpson). `y` dizisi, `x` dizisindeki noktalarda fonksiyonun değerlerini tutsun.

- `np.trapezoid(y, x)` her $[x_k, x_{k+1}]$ parçasında bir yamuğun alanını hesaplayıp toplar: $\sum_k (x_{k+1} - x_k)\,\frac{y_k + y_{k+1}}{2}$. Noktaların eşit aralıklı olması gerekmez. Eşit aralıkta `x` yerine `dx=h` de verilebilir.
- `scipy.integrate.simpson(y, x=x)` bileşik Simpson kuralını uygular. Alt aralık sayısı tek olduğunda da çalışır; son aralığı ayrı bir formülle hesaba katar.

Yani bu iki fonksiyon, yukarıda kendimiz yazdığımız kuralların veri dizisi alan sürümleridir. NumPy'nin eski sürümlerindeki `np.trapz` adı kaldırılmıştır; yeni kodda `np.trapezoid` kullanılır.

```
import numpy as np
from scipy.integrate import simpson

x = np.linspace(0, 2, 5)                # n = 4 alt aralık, h = 0.5
y = np.exp(-x**2)
print(np.trapezoid(y, x))               # x noktaları verilerek
print(np.trapezoid(y, dx=0.5))          # eşit aralıkta yalnız adım
print(simpson(y, x=x))

# Düzensiz aralıklar: yamuk kuralı her parçanın kendi genişliğini kullanır
xu = np.array([0.0, 0.2, 0.5, 1.0, 1.4, 2.0])
print(np.trapezoid(np.exp(-xu**2), xu))
```

Çıktı:

```
0.8806186341245394
0.8806186341245394
```

 CC BY-NC-SA 4.0

```
0.8818124252941161
0.8931873236709345
```

İlk üç sayı, kendi kodumuzun $n = 4$ için bulduğu T_4 ve S_4 değerleriyle aynıdır. Son satırda noktalar düzensiz aralıklıdır; `np.trapezoid` her parçanın kendi genişliğini kullanır.

> **Bileşik kuralla integral üç adımda**
>
> 1. Noktaları ve değerleri kur: `x = np.linspace(a, b, n + 1)` ve `y = f(x)`. Simpson kuralı için n'yi çift seç.
> 2. `np.trapezoid(y, x)` ya da `simpson(y, x=x)` ile yaklaşımı hesapla.
> 3. n'yi ikiye katlayıp yeniden hesapla. Yeni yaklaşımın hatası yamuk kuralında yaklaşık $(T_{2n} - T_n)/3$, Simpson kuralında yaklaşık $(S_{2n} - S_n)/15$'tir.

Örnek 11.4 (İkiye Katlayarak Hata Tahmini). $\int_1^2 \frac{dx}{x} = \ln 2$ integraline yamuk ve Simpson kurallarıyla $n = 8, 16, 32$ alt aralık için yaklaşın. Her ikiye katlamada yeni yaklaşımın hatasını tahmin edin ve tahmini gerçek hatayla karşılaştırın.

Çözüm.
Tahminin gerekçesi. I kesin değer olsun. Yamuk kuralında $I - T_n \approx 4\,(I - T_{2n})$ olduğundan

$$T_{2n} - T_n = (I - T_n) - (I - T_{2n}) \approx 3\,(I - T_{2n})$$

olur, yani $I - T_{2n} \approx (T_{2n} - T_n)/3$'tür. Simpson kuralında aynı hesap 16 çarpanıyla $I - S_{2n} \approx (S_{2n} - S_n)/15$ verir.
Kod. Üç adımı bir döngüde uygularız:

```python
import math

import numpy as np
from scipy.integrate import simpson

exact = math.log(2)
old_t = old_s = None
for n in [8, 16, 32]:
    # 1. adım: noktalar ve değerler
    x = np.linspace(1, 2, n + 1)
    y = 1 / x
    # 2. adım: kurallar
    t = np.trapezoid(y, x)
    s = simpson(y, x=x)
    # 3. adım: n'yi ikiye katlayınca hata tahmini
    if old_t is not None:
        est_t = (t - old_t) / 3
        est_s = (s - old_s) / 15
        print(f"n = {n:2d}  yamuk: tahmin {est_t:.2e}"
              f"  gerçek {exact - t:.2e}")
        print(f"        Simpson: tahmin {est_s:.2e}"
              f"  gerçek {exact - s:.2e}")
    old_t, old_s = t, s
```

Çıktı:

```
n = 16  yamuk: tahmin -2.44e-04  gerçek -2.44e-04
        Simpson: tahmin -4.59e-07  gerçek -4.72e-07
n = 32  yamuk: tahmin -6.10e-05  gerçek -6.10e-05
        Simpson: tahmin -2.95e-08  gerçek -2.97e-08
```

 CC BY-NC-SA 4.0

Sonuç. Yamuk kuralı için tahmin ile gerçek hata, yazılan üç basamakta aynıdır. Simpson kuralında da tahmin gerçek hatanın yüzde üçü içindedir. Hatalar negatiftir, yani iki kural da integrali fazla tahmin eder. Önerme 11.3 bunu açıklar: $f(x) = 1/x$ için $f''(x) = 2/x^3$ ve $f^{(4)}(x) = 24/x^5$ pozitiftir. Yamuk kuralında bunun geometrik anlamı da vardır: $1/x$ dışbükey olduğu için kirişler eğrinin üstünde kalır. Kesin değeri bilmediğimiz durumlarda bu tahmin, kaç alt aralığın yeteceğine karar vermek için kullanılır.
■

Örnek 11.5 (Hız Tablosundan Yol). Bir aracın hızı ilk bir dakika boyunca beş saniyede bir okunmuş ve sırasıyla 0; 8,5; 14,6; 19,0; 22,1; 24,3; 25,9; 27,1; 27,9; 28,5; 28,9; 29,2; 29,5 m/s bulunmuş olsun. Aracın bu sürede aldığı yolu yamuk ve Simpson kurallarıyla hesaplayın. Tablo $v(t) = 30\,(1 - e^{-t/15})$ modelinden üretilip bir ondalığa yuvarlanmıştır; sonuçları modelin verdiği $\int_0^{60} v(t)\,dt = 1350 + 450\,e^{-4}$ değeriyle karşılaştırın.

Çözüm.
Veri. 13 ölçüm 12 eşit alt aralık verir; n çift olduğundan Simpson kuralı doğrudan uygulanır.

```
import math

import numpy as np
from scipy.integrate import simpson

t = np.arange(0, 61, 5.0)                          # 0, 5, ..., 60 saniye
v = np.array([0.0, 8.5, 14.6, 19.0, 22.1, 24.3, 25.9,
              27.1, 27.9, 28.5, 28.9, 29.2, 29.5])   # m/s
print("yamuk  :", np.trapezoid(v, t))
print("Simpson:", simpson(v, x=t))
print("model  :", 1350 + 450 * math.exp(-4))
```

Çıktı:

```
yamuk  : 1353.75
Simpson: 1357.8333333333335
model  : 1358.2420374999303
```

Sonuç. Simpson kuralı modelin değerinden 0,4 metre, yamuk kuralı 4,5 metre uzakta kaldı. Verideki yuvarlama tek başına sonucu $60 \cdot 0{,}05 = 3$ metreye kadar değiştirebileceğinden Simpson sonucu, verinin izin verdiği doğruluktadır. Yamuk kuralının hatası ise çoğunlukla kuralın kendisinden gelir: v içbükeydir ve kirişler eğrinin altında kalır.
■

Bazen integralin yalnız son değeri değil, üst sınırın fonksiyonu olarak bütün gidişatı gerekir: $F(x) = \int_a^x f(t)\,dt$. `scipy.integrate.cumulative_trapezoid(y, x, initial=0)` her x_k için $\int_{x_0}^{x_k} f(t)\,dt$ integralinin yamuk kuralıyla bulunan yaklaşımını verir; `initial=0` ilk elemanı 0 yapar. Örnek olarak SymPy ile Sembolik Hesap bölümünde karşılaştığımız sinüs integrali fonksiyonunu,

$$\operatorname{Si}(x) = \int_0^x \frac{\sin t}{t}\,dt,$$

$[0, 20]$ aralığında hesaplayalım. $t = 0$'da $\sin t/t$ ifadesi $0/0$ olur. `np.sinc(u)` ise $\sin(\pi u)/(\pi u)$ değerini hesaplar ve $u = 0$'da 1 verir; bu yüzden `np.sinc(t / np.pi)` sorunu ortadan kaldırır.

```
import numpy as np
from scipy.integrate import cumulative_trapezoid
from scipy.special import sici
```

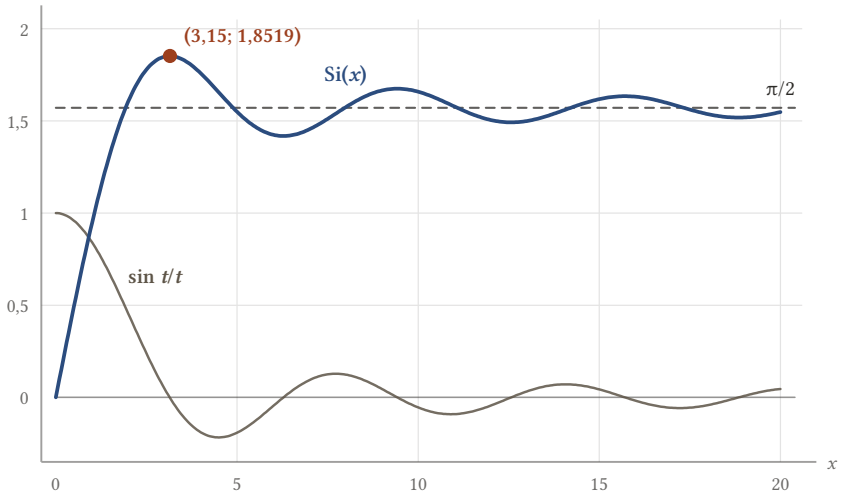
 CC BY-NC-SA 4.0

```
t = np.linspace(0, 20, 401)                # adım 0.05
y = np.sinc(t / np.pi)                     # sin(t)/t; t = 0'da 1
Si = cumulative_trapezoid(y, t, initial=0)  # Si[k] = t[0]..t[k] integrali
exact = sici(t)[0]                         # sici, (Si, Ci) ikilisini verir
print(len(Si), Si[-1])
print("en büyük hata:", np.abs(Si - exact).max())
k = Si.argmax()
print("en büyük değer:", t[k], Si[k])
```

Çıktı:

```
401 1.5482454765212605
en büyük hata: 9.086494840016002e-05
en büyük değer: 3.1500000000000004 1.8518598626730711
```

Sonuçlar, SciPy'nin özel fonksiyonu `scipy.special.sici` ile 10^{-4} duyarlıkla uyuşuyor. `sici(t)` iki dizi döndürür, sinüs integrali Si ve kosinüs integrali Ci; ilkini `[0]` indisiyle aldık. En büyük değer $x = \pi$ civarındadır, çünkü Si'nin türevi olan $\sin x / x$ ilk kez orada sıfır olur ve işaret değiştirir.



Şekil 11.5: `cumulative_trapezoid` kodunun 401 noktada hesapladığı Si(x) değerleri (mavi) ve integrali alınan $\sin t/t$ fonksiyonu (gri). Si(x), $\sin t/t$ pozitifken artar, negatifken azalır; en büyük değerine $x = \pi$ civarında ulaşır ve $\pi/2$ doğrusu etrafında sönen salınımlarla ona yaklaşır.

Şekil, $x \to \infty$ iken Si(x)'in $\pi/2$'ye yaklaştığını düşündürüyor. Bu genelleştirilmiş integrali birazdan `quad` ile hesaplayacağız.

11.6 Uyarlamalı İntegral: quad

İntegrand bir Python fonksiyonu olarak verildiğinde en pratik araç `scipy.integrate.quad` fonksiyonudur. Bu fonksiyon noktaları kendisi seçer ve sonucun yanında bir hata tahmini de verir.

Tanım 11.7 (scipy.integrate.quad). `quad(f, a, b)` çağrısı $\int_a^b f(x)\,dx$ için bir `(değer, hata)` ikilisi döndürür; buradaki `hata`, değerin mutlak hatası için bir tahmindir. `quad` her alt aralıkta integrali ve hatasını yüksek dereceli bir kuralla tahmin eder ve toplam hata tahmini istenen sınırın altına inene kadar hata tahmini en büyük olan alt aralığı ikiye böler. Sınırlar `np.inf` ya da `-np.inf` olabilir. Varsayılan hedef, mutlak ya da bağıl hatanın yaklaşık $1{,}5 \cdot 10^{-8}$'in altına inmesidir (`epsabs` ve `epsrel` parametreleri).

Yani `quad` bizden n istemez: zor bölgelerde alt aralıkları sıklaştırır, kolay bölgelerde geniş tutar. Bu tür yöntemlere **uyarlamalı** (adaptive) integral yöntemleri denir. `quad`, uzun yıllardır kullanılan QUADPACK kütüphanesinin algoritmalarına dayanır.

 CC BY-NC-SA 4.0

```python
import math

import numpy as np
from scipy.integrate import quad


def f(x):
    return np.exp(-x**2)


value, err = quad(f, 0, 2)
exact = math.sqrt(math.pi) / 2 * math.erf(2)
print(value, err)
print("gerçek hata:", abs(value - exact))
info = quad(f, 0, 2, full_output=1)[2]
print("fonksiyon çağrısı:", info["neval"])
```

Çıktı:

```
0.8820813907624215 9.793070696178202e-15
gerçek hata: 0.0
fonksiyon çağrısı: 21
```

`full_output=1` verilirse `quad` üçüncü bir eleman olarak ayrıntılı bir bilgi sözlüğü de döndürür; kodda bu demetin `[2]` indisli elemanını aldık. Sözlüğün `neval` anahtarı, fonksiyonun kaç kez hesaplandığını verir. `quad` sonucu yalnız 21 fonksiyon değeriyle makine duyarlığında buldu; Simpson kuralı 257 noktayla ancak 10^{-11}'e inmişti. Hata tahmini 10^{-14} mertebesindedir ve gerçek hata ondan da küçüktür.
İntegrand başka parametrelere de bağlıysa bu parametrelerin değerleri `args` ile verilir.

Örnek 11.6 (Bessel Fonksiyonlarını İntegralle Hesaplamak). n bir tam sayı olmak üzere birinci tür Bessel fonksiyonu

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(n\tau - x\sin\tau)\,d\tau$$

integraliyle verilebilir. $J_0(2{,}5)$, $J_1(2{,}5)$ ve $J_2(2{,}5)$ değerlerini `quad` ile hesaplayıp `scipy.special.jv` fonksiyonunun değerleriyle karşılaştırın.

Çözüm.
Parametreler. İntegrand τ'nun yanında n ve x'e de bağlıdır. `quad` integrali her zaman fonksiyonun **ilk** argümanına göre alır; geri kalan argümanların değerleri `args` demetiyle verilir:

```python
import numpy as np
from scipy.integrate import quad
from scipy.special import jv


def integrand(tau, n, x):
    return np.cos(n * tau - x * np.sin(tau))


x = 2.5
for n in [0, 1, 2]:
    value, err = quad(integrand, 0, np.pi, args=(n, x))
    print(n, value / np.pi, jv(n, x))
```

Çıktı:

 CC BY-NC-SA 4.0

```
0 -0.048383776468198046 -0.048383776468197914
1 0.49709410246427405 0.4970941024642741
2 0.44605905843961724 0.44605905843961724
```

Sonuç. Üç değer de SciPy'nin özel fonksiyonuyla son bir iki basamak dışında aynıdır. `args` yerine `lambda tau: integrand(tau, n, x)` de yazılabilirdi; `args` daha okunaklıdır.
■

11.6.1 Gauss Kuralları

`quad`'ın 21 değerle bu kadar iyi sonuç vermesinin nedeni, düğümleri akıllıca seçmesidir. Yamuk ve Simpson kuralları düğümleri eşit aralıklı alır. Düğümleri de serbest bırakırsak, n düğümlü bir kural derecesi en çok $2n-1$ olan bütün polinomları tam integre edecek biçimde kurulabilir. Bu kurallara **Gauss–Legendre kuralları** denir; ispatlarını burada vermeyeceğiz. NumPy'de `np.polynomial.legendre.leggauss(n)` fonksiyonu $[-1,1]$ aralığı için düğümleri ve ağırlıkları verir:

```python
import numpy as np

t, w = np.polynomial.legendre.leggauss(3)     # 3 düğüm ve ağırlık
print(t)
print(w)
for k in range(7):
    approx = (w * t**k).sum()
    exact = (1 - (-1) ** (k + 1)) / (k + 1)   # [-1, 1] üzerinde x^k
    print(f"k = {k}   Gauss = {approx:.12f}   kesin = {exact:.12f}")
```

Çıktı:

```
[-0.77459667  0.          0.77459667]
[0.55555556 0.88888889 0.55555556]
k = 0   Gauss = 2.000000000000   kesin = 2.000000000000
k = 1   Gauss = 0.000000000000   kesin = 0.000000000000
k = 2   Gauss = 0.666666666667   kesin = 0.666666666667
k = 3   Gauss = 0.000000000000   kesin = 0.000000000000
k = 4   Gauss = 0.400000000000   kesin = 0.400000000000
k = 5   Gauss = 0.000000000000   kesin = 0.000000000000
k = 6   Gauss = 0.240000000000   kesin = 0.285714285714
```

Üç düğümlü kural $k=0,1,\dots,5$ için kesin sonucu verir ve $k=6$'da yanılır: $2\cdot 3-1=5$. Düğümler simetriktir ve aralığın uç noktaları düğüm değildir. $\int_0^2 e^{-x^2}\,dx$ integralini $x=1+t$ değişkeniyle $[-1,1]$'e taşıyıp düğüm sayısını artıralım:

```python
import math

import numpy as np

exact = math.sqrt(math.pi) / 2 * math.erf(2)
for n in range(2, 9):
    t, w = np.polynomial.legendre.leggauss(n)
    # [0, 2] -> [-1, 1]: x = 1 + t, dx = dt
    approx = (w * np.exp(-(1 + t) ** 2)).sum()
    print(f"n = {n}   hata = {abs(approx - exact):.1e}")
```

Çıktı:

```
n = 2   hata = 3.7e-02
n = 3   hata = 3.2e-03
n = 4   hata = 1.5e-04
n = 5   hata = 3.4e-06
n = 6   hata = 3.5e-08
```

 CC BY-NC-SA 4.0

```
n = 7   hata = 6.8e-09
n = 8   hata = 3.4e-10
```

Sekiz fonksiyon değeriyle hata $3 \cdot 10^{-10}$'a iniyor; Simpson kuralı bu doğruluk için yaklaşık 130 nokta istiyordu. `quad` her alt aralıkta 21 noktalı bir Gauss–Kronrod kuralı kullanır: bu kural 10 noktalı Gauss kuralının düğümlerine 11 düğüm ekler. Değer olarak 21 noktalı kuralın sonucu alınır; iki kuralın sonuçları arasındaki fark ise hata tahmininin temelidir. `quad` bu farkı deneyimle belirlenmiş bir formülle ölçekler ve tahminin yuvarlama düzeyinin altına inmesine izin vermez. İlk `quad` örneğindeki $9{,}8 \cdot 10^{-15}$ tahmini bu yuvarlama tabanıdır: $50\,\varepsilon \cdot 0{,}8821 \approx 9{,}8 \cdot 10^{-15}$, yani makine epsilonunun 50 katı ile integralin değerinin çarpımı.

11.6.2 Zor İntegrandlar

Uyarlamalı bölmenin gücü, integrand bir yerde hızla değiştiğinde ortaya çıkar. $x = 0{,}3$'te dar ve yüksek bir tepesi olan

$$f(x) = \frac{1}{(x - 0{,}3)^2 + 10^{-4}}$$

fonksiyonunu $[0, 1]$'de integre edelim. Kesin değer $100\,(\arctan 70 + \arctan 30)$'dur. `quad`'ın hangi noktalara baktığını görmek için fonksiyonu, her çağrıda x'i bir listeye ekleyecek biçimde yazıyoruz:

```
import math

import numpy as np
from scipy.integrate import quad

calls = []


def f(x):
    calls.append(x)                    # quad'ın baktığı noktaları kaydet
    return 1 / ((x - 0.3)**2 + 1e-4)

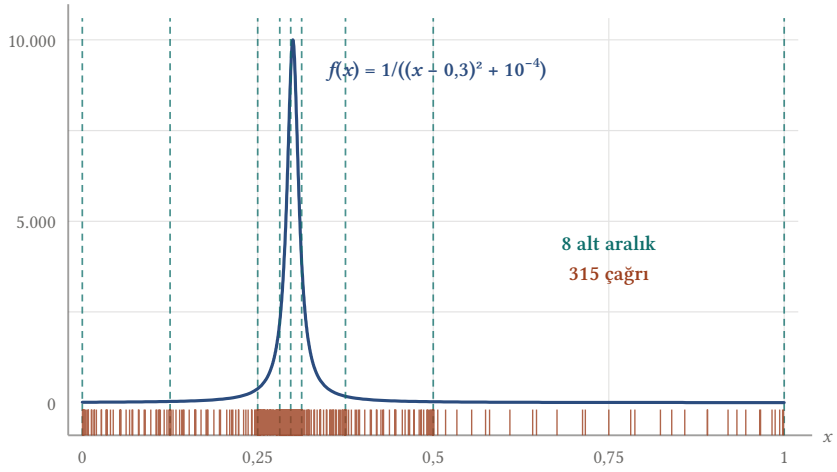

exact = 100 * (math.atan(70) + math.atan(30))
value, err, info = quad(f, 0, 1, full_output=1)
print(value, err, abs(value - exact))
print("çağrı sayısı:", len(calls), "  alt aralık:", info["last"])
left = info["alist"][:info["last"]]            # alt aralıkların sol uçları
ends = np.sort(np.concatenate([left, [1.0]]))  # sağ uç 1'i de ekle
print(ends)
x = np.linspace(0, 1, 1001)
print("yamuk, 1001 nokta, hata:", abs(np.trapezoid(f(x), x) - exact))
```

Çıktı:

```
309.3986915124147 2.3722903395168027e-08 2.2737367544323206e-13
çağrı sayısı: 315   alt aralık: 8
[0.       0.125    0.25     0.28125  0.296875 0.3125   0.375    0.5
 1.      ]
yamuk, 1001 nokta, hata: 6.6448414486330876e-06
```

Bilgi sözlüğünde `info["last"]` alt aralık sayısı, `info["alist"]` alt aralıkların sol uçlarıdır; bu sol uçlara `np.concatenate` ile son sağ uç 1'i ekleyip sıralayınca bütün uçları elde ederiz. `quad` 315 fonksiyon değeriyle 10^{-13} hataya ulaştı; 1001 eşit aralıklı noktayla yamuk kuralı ise $7 \cdot 10^{-6}$ hatada kaldı. Şekil, alt aralıkların tepeye doğru nasıl daraldığını gösteriyor.

 CC BY-NC-SA 4.0



Şekil 11.6: `quad`'ın $x = 0{,}3$'teki dar tepeyi nasıl çözdüğü: tepe fonksiyonu kodunun kaydettiği veriler. Kesikli çizgiler `quad`'ın böldüğü 8 alt aralığın uçları, eksenin altındaki kısa çizgiler fonksiyonun hesaplandığı 315 nokta. Alt aralıklar tepenin yanında ikiye bölüne bölüne daralır; fonksiyonun düz olduğu yerde tek geniş aralık yeter.

`quad`, sonsuz aralıkları ve uç noktada sonsuza giden integrandları da çoğu zaman sorunsuz işler (genelleştirilmiş integraller için bkz. Analiz 3 ve Analiz 3):

```
import math

import numpy as np
from scipy.integrate import quad

print(quad(lambda x: np.exp(-x**2), 0, np.inf), math.sqrt(math.pi) / 2)
print(quad(lambda x: np.exp(-x**2), -np.inf, np.inf), math.sqrt(math.pi))
print(quad(lambda x: 1 / np.sqrt(x), 0, 1))
print(quad(np.log, 0, 1))
```

Çıktı:

```
(0.8862269254527579, 7.101318390472462e-09) 0.8862269254527579
(1.7724538509055159, 1.4202636780944923e-08) 1.7724538509055159
(1.9999999999999984, 5.773159728050814e-15)
(-0.9999999999999999, 1.1102230246251563e-15)
```

$\int_0^\infty e^{-x^2}\,dx = \sqrt{\pi}/2$, $\int_{-\infty}^\infty e^{-x^2}\,dx = \sqrt{\pi}$, $\int_0^1 x^{-1/2}\,dx = 2$ ve $\int_0^1 \ln x\,dx = -1$ değerleri son bir iki basamağa kadar doğru. Sonsuz aralıkta `quad` önce bir değişken değiştirmeyle aralığı sonlu yapar. Uçtaki tekillikleri ise integrandı uç noktada hiç hesaplamadan, uca doğru sıklaşan alt aralıklar ve bir ekstrapolasyonla aşar. Tekillik aralığın içindeyse `quad`'a yerini söylemek gerekir:

```
import numpy as np
from scipy.integrate import quad


def f(x):
    return 1 / np.sqrt(abs(x))


with np.errstate(divide="ignore"):       # 1/0 uyarısını sustur
    print(quad(f, -1, 1))
print(quad(f, -1, 1, points=[0]))
```

Çıktı:

 CC BY-NC-SA 4.0

```
(inf, inf)
(3.9999999999999813, 5.684341886080802e-14)
```

$1/\sqrt{|\,x\,|}$ fonksiyonu $x = 0$'da sonsuza gider. 21 noktalı kuralın orta düğümü $[-1, 1]$'in tam ortası olduğundan `quad` fonksiyonu 0'da hesaplamaya çalıştı ve `inf` döndürdü. `points=[0]` parametresi aralığı tekil noktadan böler. Tekillik böylece iki alt aralığın ucuna gelir ve sonuç doğru çıkar: $\int_{-1}^{1} |\,x\,|^{-1/2}\,dx = 4$. Iraksak bir integralde ise `quad` yine bir sayı döndürür ve yalnız bir uyarı verir. $\int_0^1 dx/x$ integrali ıraksaktır:

```
import warnings

from scipy.integrate import quad

with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter("always")       # uyarıları yakala
    value, err = quad(lambda x: 1 / x, 0, 1)
print(value, err)
print(caught[0].category.__name__)
print(str(caught[0].message).splitlines()[0])
```

Çıktı:

```
41.67684067538809 9.35056037314051
IntegrationWarning
The maximum number of subdivisions (50) has been achieved.
```

`quad` 50 alt aralık sınırına takılıp 41,68 değerini ve 9,35 gibi büyük bir hata tahmini döndürdü; sonuç anlamsızdır. `warnings.catch_warnings(record=True)` bloğu, blok içinde verilen uyarıları bir listede toplar; böylece uyarıyı program içinde denetleyebiliriz.

> ⚠ Uyarıları ve hata tahminini okuyun
>
> `quad` başarısız olduğunda hata vermez; `IntegrationWarning` türünde bir uyarı basar ve bir sayı döndürmeye devam eder. Hata tahmini değere göre büyükse ya da bir uyarı çıktıysa sonuç güvenilir değildir. Tersine, küçük bir hata tahmini de kanıt değildir: integrand `quad`'ın baktığı noktaların arasında beklenmedik biçimde davranıyorsa tahmin yanılabilir. Önemli bir sonucu bağımsız bir yolla, örneğin değişken değiştirerek ya da integrali parçalara bölerek yeniden hesaplayın.

Yavaş sönen salınımlar da `quad`'ı zorlar. Şekildeki $\mathrm{Si}(x)$'in limiti olan $\int_0^\infty \frac{\sin x}{x}\,dx = \frac{\pi}{2}$ integralini (bkz. Analiz 3) doğrudan hesaplamaya çalışmak başarısız olur. Çare, salınımı `quad`'a ağırlık olarak bildirmektir: `weight="sin", wvar=1` seçenekleri $\int g(x)\sin x\,dx$ biçimindeki integraller için özel bir yöntem kullanır.

```
import math
import warnings

import numpy as np
from scipy.integrate import quad


def sinc(x):
    return np.sinc(x / np.pi)            # sin(x)/x


with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter("always")
    print(quad(sinc, 0, np.inf))
print(str(caught[0].message).splitlines()[0])

# [0, 1] olağan; [1, oo) üzerinde 1/x, sin(x) ağırlığıyla
part1, _ = quad(sinc, 0, 1)
part2, _ = quad(lambda x: 1 / x, 1, np.inf, weight="sin", wvar=1)
print(part1 + part2, math.pi / 2)
```

 CC BY-NC-SA 4.0

Çıktı:

```
(2.2478679634674115, 3.2903230521899287)
The integral is probably divergent, or slowly convergent.
1.5707963268467806 1.5707963267948966
```

Doğrudan çağrı 2,25 gibi yanlış bir değer ve "integral muhtemelen ıraksak ya da yavaş yakınsak" uyarısı verdi. İntegrali $[0,1]$ ve $[1,\infty)$ olarak ikiye bölüp ikinci parçayı $g(x)=1/x$ ve $\sin x$ ağırlığıyla hesaplayınca sonuç $\pi/2$'ye $5\cdot 10^{-11}$ yakınlıkta çıktı.

> quad ile güvenilir integral üç adımda
>
> 1. İntegrandı incele: aralık sonsuz mu, integrand bir yerde sonsuza gidiyor mu, yavaş sönen bir salınımı var mı?
> 2. Zorlukları `quad`'a bildir: tekil noktaları aralığın ucuna getir (aralığı böl ya da `points` ver), sonsuz aralığı ayrı bir parça yap, salınımlar için `weight` kullan.
> 3. Uyarıları ve hata tahminini oku; sonucu bilinen bir değerle ya da başka bir yolla denetle.

Örnek 11.7 (Gamma Fonksiyonunun Yarımdaki Değeri). $\Gamma(1/2)=\int_0^\infty t^{-1/2}e^{-t}\,dt$ değerini `quad` ile hesaplayın ve $\sqrt{\pi}$ ile karşılaştırın (bkz. Analiz 3).

Çözüm.
1. adım. İntegrand $t=0$'da sonsuza gider ve aralık sonsuzdur; iki zorluk birden vardır.
2. adım. İntegrali $t=1$'den bölüp iki parçayı ayrı hesaplarız: $[0,1]$'de tekillik aralığın ucundadır, $[1,\infty)$'da integrand üstel olarak söner.

```
import math

import numpy as np
from scipy.integrate import quad


def g(t):
    return np.exp(-t) / np.sqrt(t)


left, err1 = quad(g, 0, 1)            # t = 0'da tekil
right, err2 = quad(g, 1, np.inf)      # sonsuz aralık
print(left, err1)
print(right, err2)
print(left + right)
print(math.sqrt(math.pi), math.gamma(0.5))
```

Çıktı:

```
1.4936482656248504 4.0730441241976223e-10
0.2788055852806195 4.764513549588497e-09
1.77245385090547
1.7724538509055159 1.7724538509055159
```

3. adım. Uyarı çıkmadı ve iki parçanın hata tahminleri 10^{-9} mertebesinde kaldı. Toplam, $\sqrt{\pi}$ ile ve `math.gamma(0.5)` ile $5\cdot 10^{-14}$ yakınlıkta uyuşuyor.
■

11.7 Katlı İntegraller: dblquad ve tplquad

`quad`'ı iç içe kullanarak katlı integraller de hesaplanabilir; SciPy bunu hazır fonksiyonlarla yapar.

Tanım 11.8 (dblquad ile İki Katlı İntegral). `dblquad(func, a, b, gfun, hfun)` çağrısı

 CC BY-NC-SA 4.0

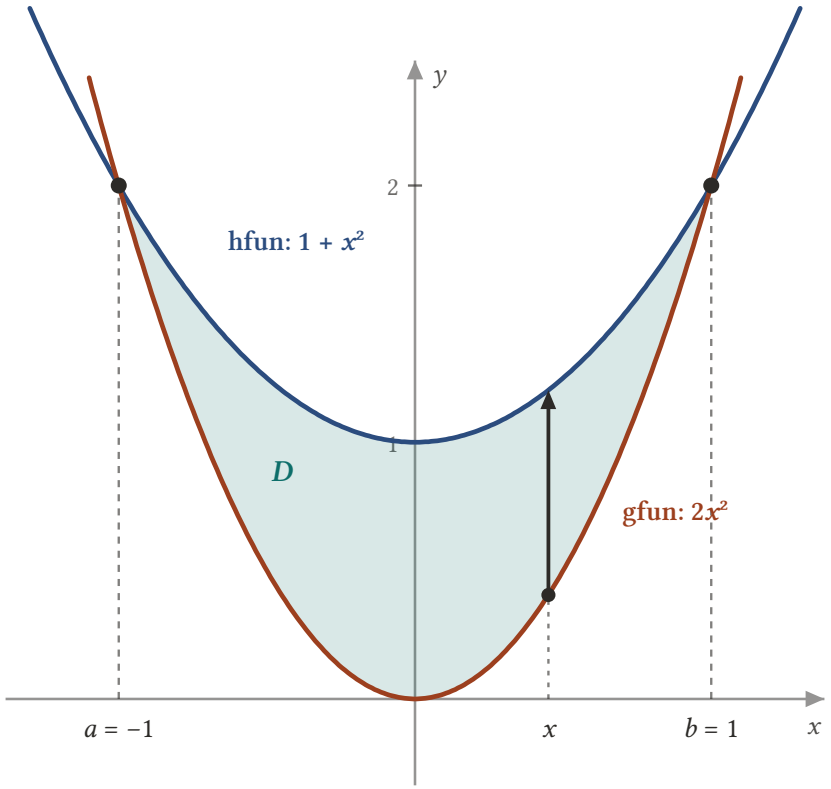
$$\int_a^b \int_{g(x)}^{h(x)} \text{func}(y, x)\, dy\, dx$$

ardışık integralini hesaplar ve bir `(değer, hata)` ikilisi döndürür. `gfun` ve `hfun`, x'e bağlı alt ve üst sınırları veren fonksiyonlardır; sınırlar sabitse sayı olarak da verilebilirler. `func` fonksiyonunun **ilk** argümanı iç değişken y, ikincisi dış değişken x'tir.

Yani `dblquad`, birinci tip bir bölgede (bkz. İntegral Calculus) yazılan ardışık integrali (bkz. İntegral Calculus) iç içe iki `quad` çağrısıyla hesaplar: dış integral x üzerinden, iç integral y üzerinden alınır.

> ⚠ dblquad'da argüman sırası
>
> `func(y, x)` sırası, matematikteki $f(x, y)$ alışkanlığının tersidir. `lambda x, y: ...` yazmak hata vermez ama sessizce başka bir integral hesaplar; integrand x ile y'de simetrik değilse sonuç yanlış çıkar. `tplquad`'da sıra `func(z, y, x)`'tir: en içteki değişken her zaman önce gelir.

Örnek 11.8 (İki Parabol Arasındaki Bölgede İntegral). D, $y = 2x^2$ ve $y = 1 + x^2$ parabollerinin sınırladığı bölge olmak üzere $\iint_D (x + 2y)\, dA$ integralini `dblquad` ile hesaplayın. Bu integral İntegral Calculus notlarında elle hesaplanmış ve $\frac{32}{15}$ bulunmuştur (bkz. İntegral Calculus).



Şekil 11.7: `dblquad` örneğindeki D bölgesi. Dış integral x'i $a = -1$'den $b = 1$'e götürür; her x için iç integral y'yi alttaki `gfun(x)` $= 2x^2$ parabolünden üstteki `hfun(x)` $= 1 + x^2$ parabolüne kadar tarar (ok).

Çözüm.
Sınırlar. Paraboller $2x^2 = 1 + x^2$, yani $x = \pm 1$ iken kesişir. $-1 \le x \le 1$ için $x^2 \le 1$ olduğundan alttaki parabol $y = 2x^2$, üstteki $y = 1 + x^2$'dir. Dolayısıyla $a = -1$, $b = 1$, `gfun` fonksiyonu $2x^2$ ve `hfun` fonksiyonu $1 + x^2$'dir.
Kod.

 CC BY-NC-SA 4.0

```python
from fractions import Fraction

from scipy.integrate import dblquad

value, err = dblquad(lambda y, x: x + 2 * y,       # önce y, sonra x
                     -1, 1,                        # x sınırları
                     lambda x: 2 * x**2,           # alttaki parabol
                     lambda x: 1 + x**2)           # üstteki parabol
print(value, err)
print(float(Fraction(32, 15)))
```

Çıktı:

```
2.1333333333333333 2.3684757858670007e-14
2.1333333333333333
```

Sonuç. `dblquad`, $\frac{32}{15} = 2{,}1333\ldots$ değerini 10^{-14} mertebesinde bir hata tahminiyle verdi; elle yapılan hesapla aynı.
■

Tanım 11.9 (tplquad ile Üç Katlı İntegral). `tplquad(func, a, b, gfun, hfun, qfun, rfun)` çağrısı

$$\int_{a}^{b} \int_{g(x)}^{h(x)} \int_{q(x,y)}^{r(x,y)} \mathrm{func}(z, y, x)\, dz\, dy\, dx$$

ardışık integralini hesaplar. `gfun` ve `hfun` x'e, `qfun` ve `rfun` ise x ile y'ye bağlı sınırlardır; sabit sınırlar sayı olarak verilebilir.

Yani `tplquad`, tip 1 bir cisim için (bkz. İntegral Calculus) yazılan ardışık integrali hesaplar: en içte z, ortada y, dışta x üzerinden integral alınır.

Örnek 11.9 (Bir Yüzeyin Altındaki Cisimde Üç Katlı İntegral). E, birinci oktantta $z = 12xy$ yüzeyi ile $y = x$, $x = 1$ ve $z = 0$ düzlemleri arasında kalan cisim olsun. $\iiint_E z\, dV$ integralini `tplquad` ile hesaplayın ve İntegral Calculus notlarında bulunan 4 sonucuyla karşılaştırın (bkz. İntegral Calculus).

Çözüm.
Sınırlar. Cisim $0 \le x \le 1$, $0 \le y \le x$ ve $0 \le z \le 12xy$ eşitsizlikleriyle betimlenir. Sabit alt sınırları doğrudan `0` olarak yazarız:

```python
from scipy.integrate import tplquad

value, err = tplquad(lambda z, y, x: z,
                     0, 1,                          # x sınırları
                     0, lambda x: x,                # y sınırları
                     0, lambda x, y: 12 * x * y)    # z sınırları
print(value, err)
```

Çıktı:

```
4.0 7.890024717707689e-13
```

Sonuç. Değer 4, hata tahmini 10^{-12} mertebesinde; elle bulunan sonuçla aynı.
■

 CC BY-NC-SA 4.0

Dört ve daha çok katlı integraller için `scipy.integrate.nquad` vardır. Ancak boyut arttıkça iç içe kuralların maliyeti üstel büyür: her boyutta 21 nokta kullanılırsa d boyutta en az 21^d fonksiyon değeri gerekir. Yüksek boyutlarda bu yüzden Monte Carlo yöntemleri tercih edilir (bkz. Rastgele Sayılar ve Monte Carlo Yöntemleri).

11.8 Alıştırmalar

Alıştırma 11.1 (Küp Fonksiyonunda İleri Farkın Hatası). $f(x) = x^3$ için $x = 2$'deki ileri fark bölümünün hatasının tam olarak $6h + h^2$ olduğunu gösterin ve bunu $h = 0{,}1;\ 0{,}01;\ 0{,}001$ için Python ile doğrulayın.

Çözüm.
Hesap. $(2+h)^3 = 8 + 12h + 6h^2 + h^3$ olduğundan

$$D_h^+ f(2) = \frac{(2+h)^3 - 8}{h} = 12 + 6h + h^2$$

olur. $f'(2) = 3 \cdot 2^2 = 12$ olduğundan hata $6h + h^2$'dir. Bu, Önerme 11.1 ile uyumludur: $f''(\xi) = 6\xi$ olduğundan $\frac{h}{2} f''(\xi) = 3h\xi$'dir ve $\xi = 2 + h/3$ için $3h\xi = 6h + h^2$ olur.
Kod.

```
def f(x):
    return x**3


x = 2.0
for h in [0.1, 0.01, 0.001]:
    err = (f(x + h) - f(x)) / h - 12.0     # f'(2) = 12
    print(f"h = {h:5.3f}   hata = {err:.10f}"
          f"   6h + h^2 = {6 * h + h**2:.10f}   hata/h = {err / h:.4f}")
```

Çıktı:

```
h = 0.100   hata = 0.6100000000   6h + h^2 = 0.6100000000   hata/h = 6.1000
h = 0.010   hata = 0.0601000000   6h + h^2 = 0.0601000000   hata/h = 6.0100
h = 0.001   hata = 0.0060010000   6h + h^2 = 0.0060010000   hata/h = 6.0010
```

Sonuç. Kayan noktalı hesap on ondalık basamağa kadar formülle aynı. Hata h ile orantılıdır ve hata bölü h oranı $f''(2)/2 = 6$'ya yaklaşır: ileri fark birinci mertebedendir.
■

Alıştırma 11.2 (Beş Noktalı Merkezi Fark). Beş noktalı merkezi fark formülü

$$f'(x) \approx \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h}$$

ile verilir ve kesme hatası $\frac{h^4}{30} \mid f^{(5)}(\xi) \mid$ kadardır. $f(x) = \sin x$ için $x = 1$'de bu formülü $h = 10^{-1}, \ldots, 10^{-6}$ adımlarıyla deneyin. Hatanın h^4 ile küçüldüğü adımları ve en iyi adımı bulun.

Çözüm.
Toplam hata. Formüldeki katsayıların mutlak değerlerinin toplamı $1 + 8 + 8 + 1 = 18$'dir. Her fonksiyon değeri en çok δ kadar yanlışsa yuvarlamadan gelen hata en çok $\frac{18\delta}{12h} = \frac{3\delta}{2h}$ olur. Toplam hata sınırı $E(h) = \frac{M_5 h^4}{30} + \frac{3\delta}{2h}$'dir. $E'(h) = \frac{4M_5 h^3}{30} - \frac{3\delta}{2h^2} = 0$ denklemi $h^5 = \frac{45\delta}{4M_5}$ verir. $\delta = \varepsilon \sin 1$ ve $M_5 = \cos 1$ alırız.
Kod.

```python
import math


def five_point(f, x, h):
    """Beş noktalı merkezi fark formülü."""
    return (f(x - 2 * h) - 8 * f(x - h)
            + 8 * f(x + h) - f(x + 2 * h)) / (12 * h)


x = 1.0
exact = math.cos(x)
old = None
for k in range(1, 7):
    h = 10.0**-k
    err = abs(five_point(math.sin, x, h) - exact)
    ratio = f"{old / err:8.0f}" if old else "       -"
    print(f"h = {h:6.0e}   hata = {err:.3e}   oran = {ratio}")
    old = err
eps = 2.0**-52
h_best = (45 * eps * math.sin(x) / (4 * math.cos(x))) ** (1 / 5)
print(f"teorik en iyi h = {h_best:.1e}")
```

Çıktı:

```
h =  1e-01   hata = 1.799e-06   oran =       -
h =  1e-02   hata = 1.801e-10   oran =    9988
h =  1e-03   hata = 1.497e-13   oran =    1203
h =  1e-04   hata = 5.385e-14   oran =       3
h =  1e-05   hata = 4.665e-12   oran =       0
h =  1e-06   hata = 1.847e-11   oran =       0
teorik en iyi h = 1.3e-03
```

Sonuç. $h = 10^{-1}$'den 10^{-2}'ye geçerken hata yaklaşık 10^4 kat küçülüyor: formül dördüncü mertebedendir. $h = 10^{-1}$'deki hata da $\frac{h^4}{30}\cos 1 \approx 1{,}80 \cdot 10^{-6}$ tahminiyle uyuşuyor. $h = 10^{-3}$'te yuvarlama hatası öne geçmeye başlıyor ve oran düşüyor: bu adımda kesme hatası yalnız $1{,}8 \cdot 10^{-14}$ kadardır, gerçek hata ise $1{,}5 \cdot 10^{-13}$'tür. Teorik en iyi adım $1{,}3 \cdot 10^{-3}$'tür. Gerçekten de en küçük hatalar 10^{-3} ve 10^{-4} adımlarında, 10^{-13} mertebesinde çıktı. Merkezi farkla ulaşılabilen en iyi hata $\varepsilon^{2/3} \approx 10^{-11}$ mertebesindeydi; dört fonksiyon değeri kullanan bu formül yaklaşık yüz kat daha iyi sonuç veriyor. ■

Alıştırma 11.3 (Karmaşık Adımla Türev). f, gerçel eksende gerçel değerler alan analitik bir fonksiyon olsun. Taylor açılımından $f'(x) \approx \operatorname{Im} f(x+ih)/h$ yaklaşımının hatasının h^2 mertebesinde olduğunu gösterin. Bu **karmaşık adım** (complex step) yaklaşımını $f(x) = e^x \sin x$ için $x = 1$'de $h = 10^{-2}$, 10^{-5}, 10^{-10}, 10^{-20} ve 10^{-100} adımlarıyla deneyin ve h çok küçükken bile sadeleşme hatası oluşmamasının nedenini açıklayın.

Çözüm.
Taylor açılımı. $i^2 = -1$ ve $i^3 = -i$ olduğundan

$$f(x+ih) = f(x) + ihf'(x) - \frac{h^2}{2}f''(x) - i\,\frac{h^3}{6}f'''(x) + \cdots$$

olur. x gerçel ve f'nin türevleri gerçel olduğundan sanal kısım $hf'(x) - \frac{h^3}{6}f'''(x) + \cdots$'dır. h'ye bölünce

$$\frac{\operatorname{Im} f(x+ih)}{h} = f'(x) - \frac{h^2}{6}f'''(x) + \cdots$$

 CC BY-NC-SA 4.0

bulunur; hata h^2 mertebesindedir.
Kod. `np.exp` ve `np.sin` karmaşık sayılarla da çalışır; `math.exp` ve `math.sin` çalışmaz. Kesin türev $f'(x) = e^x(\sin x + \cos x)$'tir.

```
import math

import numpy as np


def f(z):
    return np.exp(z) * np.sin(z)       # karmaşık sayılarla da çalışır


x = 1.0
exact = math.e * (math.sin(1) + math.cos(1))
for h in [1e-2, 1e-5, 1e-10, 1e-20, 1e-100]:
    d = f(x + 1j * h).imag / h
    print(f"h = {h:7.0e}   türev = {d:.16f}   hata = {abs(d - exact):.1e}")
```

Çıktı:

```
h =   1e-02   türev = 3.7560765145542852   hata = 2.7e-05
h =   1e-05   türev = 3.7560492271220163   hata = 2.7e-11
h =   1e-10   türev = 3.7560492270947270   hata = 4.4e-16
h =   1e-20   türev = 3.7560492270947274   hata = 0.0e+00
h =  1e-100   türev = 3.7560492270947279   hata = 4.4e-16
```

Sonuç. Formülde birbirine yakın iki sayının farkı yoktur: $hf'(x)$ terimi sanal kısım olarak doğrudan hesaplanır. Bu yüzden yuvarlama hatası h küçüldükçe büyümez. $h \le 10^{-10}$ için kesme hatası (h^2 mertebesinde) makine duyarlığının altına düşer ve sonuç son basamağına kadar doğru çıkar; $h = 10^{-100}$ bile sorun yaratmaz. Yöntemin bedeli, f'nin karmaşık sayılarla hesaplanabilmesinin gerekmesidir.
■

Alıştırma 11.4 (Yuvarlanmış Ölçümlerden İvme). Yukarı atılan bir topun yüksekliği $s(t) = 20t - 4{,}9t^2$ metre olsun. Yükseklik $[0, 4]$ saniye aralığında Δt aralıklarla ölçülüp santimetreye yuvarlanmış olsun. `np.gradient`'i iki kez uygulayarak hızı ve ivmeyi $\Delta t = 0{,}1;\ 0{,}05;\ 0{,}01$ için hesaplayın ve en büyük hataları kesin değerler $v(t) = 20 - 9{,}8t$ ve $a = -9{,}8$ ile karşılaştırın.

Çözüm.
Kesme hatası yok. s ikinci dereceden bir polinomdur; `np.gradient` (`edge_order=2` ile) böyle bir fonksiyonun türevini tam verir. Hızın kendisi de birinci dereceden olduğundan ikinci uygulama da tamdır. Demek ki bütün hata yükseklik değerlerinin yuvarlanmasından gelir.
Kod.

```
import numpy as np

for dt in [0.1, 0.05, 0.01]:
    t = np.arange(0, 4 + dt / 2, dt)
    s = np.round(20 * t - 4.9 * t**2, 2)      # santimetreye yuvarlanmış
    v = np.gradient(s, t, edge_order=2)
    a = np.gradient(v, t, edge_order=2)
    err_v = np.abs(v - (20 - 9.8 * t)).max()
    err_a = np.abs(a + 9.8).max()
    print(f"dt = {dt:4.2f}   hız hatası = {err_v:.3f}"
          f"   ivme hatası = {err_a:.2f}")
```

Çıktı:

```
dt = 0.10   hız hatası = 0.040   ivme hatası = 0.45
dt = 0.05   hız hatası = 0.100   ivme hatası = 3.20
dt = 0.01   hız hatası = 0.464   ivme hatası = 40.20
```

Sonuç. Ölçüm aralığı küçüldükçe hatalar büyüyor. Yuvarlama hatası en çok $\delta = 0{,}005$ metredir. Hızda bu hata $\delta/\Delta t$ ile, ivmede iki kez bölündüğü için $\delta/\Delta t^2$ ile büyür: Δt on kat küçülünce ivme hatası yaklaşık yüz kat büyüdü ($0{,}45$'ten $40{,}2$'ye). Türev almak verideki hatayı büyütür. Gürültülü bir veriden türev alırken çözüm aralığı küçültmek değil, veriye önce düzgün bir eğri uydurmaktır (bkz. Polinomlar, İnterpolasyon ve Eğri Uydurma).
■

Alıştırma 11.5 (Yamuk Kuralı İçin Yeterli Alt Aralık Sayısı). $\int_0^1 e^x\,dx$ integralini bileşik yamuk kuralıyla 10^{-6}'dan küçük hatayla hesaplamak için Önerme 11.3 sınırına göre kaç alt aralık yeter? Bu sayı için gerçek hatayı hesaplayın ve gerçekte yeten en küçük alt aralık sayısını Python ile bulun.

Çözüm.
Sınır. $[0,1]$'de $|\,f''(x)\,| = e^x \le e$ ve $h = 1/n$ olduğundan hata en çok $\frac{e}{12n^2}$'dir. $\frac{e}{12n^2} \le 10^{-6}$ eşitsizliği $n \ge \sqrt{e/(12\cdot 10^{-6})} \approx 475{,}9$ verir; sınır $n = 476$'yı önerir.
Kod. Kesin değer $e - 1$'dir. Gerçekte yeten en küçük n'yi $n = 1$'den başlayıp sayarak buluruz:

```
import math

import numpy as np


def trap_error(n):
    x = np.linspace(0, 1, n + 1)
    return abs(np.trapezoid(np.exp(x), x) - (math.e - 1))


tol = 1e-6
n_bound = math.ceil(math.sqrt(math.e / (12 * tol)))
print("sınırın verdiği n:", n_bound, "  hata:", trap_error(n_bound))
n = 1
while trap_error(n) > tol:
    n += 1
print("yeterli en küçük n:", n, "  hata:", trap_error(n))
print("bir eksiği:", n - 1, "  hata:", trap_error(n - 1))
```

Çıktı:

```
sınırın verdiği n: 476   hata: 6.319740037952215e-07
yeterli en küçük n: 379   hata: 9.96861172941621e-07
bir eksiği: 378   hata: 1.0021425469464162e-06
```

Sonuç. Sınırın önerdiği $n = 476$'da hata $6{,}3\cdot 10^{-7}$'dir; gerçekte $n = 379$ yeter. Sınır kötümserdir, çünkü f'' yerine onun $[0,1]$'deki en büyük değeri e'yi kullanır. Oysa $f''(\mu)$ değeri f'''nin bir ortalamasıdır ve e^x'in $[0,1]$'deki ortalaması $e - 1 \approx 1{,}72$'dir. Sınırdaki e yerine bu değer yazılırsa $n \ge 378{,}4$ bulunur; bu da gerçekte yeten $n = 379$ ile uyuşur.
■

Alıştırma 11.6 (Simpson Kuralında Beklenmedik Hız). $\int_0^1 \frac{4}{1+x^2}\,dx = \pi$ integralini `simpson` ile $n = 2, 4, 8, 16, 32, 64$ alt aralık için hesaplayın. Ardışık hataların oranına bakın ve beklenen 16 yerine neden başka bir oran çıktığını açıklayın.

 CC BY-NC-SA 4.0

Çözüm.
Kod. Hataların yanında f''''nün uç noktalardaki değerlerini SymPy ile hesaplıyoruz; nedenini birazdan göreceğiz:

```
import math

import numpy as np
import sympy as sp
from scipy.integrate import simpson

old = None
for n in [2, 4, 8, 16, 32, 64]:
    x = np.linspace(0, 1, n + 1)
    err = abs(simpson(4 / (1 + x**2), x=x) - math.pi)
    ratio = f"{old / err:6.1f}" if old else "     -"
    print(f"n = {n:2d}   hata = {err:.3e}   oran = {ratio}")
    old = err

t = sp.symbols("t")
d3 = sp.diff(4 / (1 + t**2), t, 3)
print(d3.subs(t, 0), d3.subs(t, 1))
```

Çıktı:

```
n =  2   hata = 8.259e-03   oran =      -
n =  4   hata = 2.403e-05   oran =  343.8
n =  8   hata = 1.511e-07   oran =  159.0
n = 16   hata = 2.365e-09   oran =   63.9
n = 32   hata = 3.696e-11   oran =   64.0
n = 64   hata = 5.773e-13   oran =   64.0
0 0
```

Oran. Küçük n'lerden sonra oran 16 değil, $64 = 2^6$ çıkıyor: hata h^6 ile küçülüyor.
Açıklama. Önerme 11.3 hatayı $-\frac{(b-a)h^4}{180} f^{(4)}(\mu)$ olarak verir. İspattaki toplam bir Riemann toplamı gibidir: $(b-a) f^{(4)}(\mu)$, $\int_a^b f^{(4)}(x)\,dx = f'''(b) - f'''(a)$ değerine yaklaşır. Daha ayrıntılı bir analiz (Euler–Maclaurin formülü) hatanın

$$\int_a^b f(x)\,dx - S_n = -\frac{h^4}{180}\left[\, f'''(b) - f'''(a) \,\right] + O(h^6)$$

biçiminde açıldığını gösterir. Burada $f'''(0) = f'''(1) = 0$ olduğu için h^4 terimi yok olur ve geriye h^6 mertebesindeki terim kalır.
■

Alıştırma 11.7 (Periyodik Fonksiyonda Yamuk Kuralı). $\int_0^{2\pi} e^{\cos x}\,dx = 2\pi I_0(1)$'dir; burada I_0, `scipy.special.i0` ile hesaplanan değiştirilmiş Bessel fonksiyonudur. Bu integrali yamuk kuralıyla $n = 2, 4, 8, 16, 32$ alt aralık için hesaplayın ve hatanın neden h^2'den çok daha hızlı küçüldüğünü açıklayın.

Çözüm.
Kod.

```
import numpy as np
from scipy.special import i0

exact = 2 * np.pi * i0(1)
print(exact)
```

 CC BY-NC-SA 4.0

```
for n in [2, 4, 8, 16, 32]:
    x = np.linspace(0, 2 * np.pi, n + 1)
    err = abs(np.trapezoid(np.exp(np.cos(x)), x) - exact)
    print(f"n = {n:2d}   hata = {err:.1e}")
```

Çıktı:

```
7.954926521012844
n =  2   hata = 1.7e+00
n =  4   hata = 3.4e-02
n =  8   hata = 1.3e-06
n = 16   hata = 0.0e+00
n = 32   hata = 0.0e+00
```

Açıklama. Hata $n = 4$'ten $n = 8$'e geçerken yirmi binden fazla kat küçüldü; $n = 16$'da makine duyarlığına ulaştı. Simpson kuralındaki akıl yürütmenin yamuk kuralı için karşılığı, hatanın

$$\int_a^b f(x)\,dx - T_n = -\frac{h^2}{12}\,\big[\,f'(b) - f'(a)\,\big] + O(h^4)$$

biçiminde açılmasıdır; sonraki terimler de f''', $f^{(5)}$, … türevlerinin uç noktalardaki değerlerinin farklarını içerir. $e^{\cos x}$ fonksiyonu 2π periyotlu ve sonsuz kez türevlenebilir olduğundan bütün türevleri 0 ile 2π'de aynı değeri alır. Bu yüzden açılımın bütün terimleri yok olur ve hata h'nin her kuvvetinden hızlı küçülür. Periyodik fonksiyonların tam periyot üzerindeki integralinde en iyi kural, en basit kural olan yamuk kuralıdır.
■

Alıştırma 11.8 (Sinüs Eğrisinin Uzunluğu). $y = \sin x$ eğrisinin $0 \le x \le \pi$ arasındaki yayının uzunluğunu

$$L = \int_0^{\pi} \sqrt{1 + \cos^2 x}\,dx$$

formülüyle `quad` kullanarak hesaplayın. Sonucu, eğri üzerindeki $n + 1$ eşit aralıklı noktayı birleştiren kırık çizgilerin $n = 4, 16, 64, 256$ için uzunluklarıyla karşılaştırın.

Çözüm.
Kırık çizgi. Ardışık noktalar arasındaki kirişin boyu $\sqrt{(\Delta x)^2 + (\Delta y)^2}$'dir; `np.diff` farkları, `np.hypot` bu karekökü verir.

```
import numpy as np
from scipy.integrate import quad

L, err = quad(lambda x: np.sqrt(1 + np.cos(x)**2), 0, np.pi)
print(L, err)
for n in [4, 16, 64, 256]:
    x = np.linspace(0, np.pi, n + 1)
    chords = np.hypot(np.diff(x), np.diff(np.sin(x)))   # kiriş boyları
    print(f"n = {n:3d}   kırık çizgi = {chords.sum():.10f}"
          f"   fark = {L - chords.sum():.1e}")
```

Çıktı:

```
3.8201977890277115 1.3015768476030976e-13
n =   4   kırık çizgi = 3.7900913085   fark = 3.0e-02
n =  16   kırık çizgi = 3.8182749650   fark = 1.9e-03
```

```
n =  64   kırık çizgi = 3.8200775044   fark = 1.2e-04
n = 256   kırık çizgi = 3.8201902708   fark = 7.5e-06
```

Sonuç. İntegral elementer fonksiyonlarla hesaplanamaz (bir eliptik integraldir); `quad` onu 10^{-13} hata tahminiyle $L \approx 3{,}8201977890$ olarak verdi. Kırık çizgilerin uzunlukları bu değere alttan yaklaşır, çünkü her kiriş, gerdiği yaydan kısadır. n dört katına çıkınca fark yaklaşık 16 kat küçülüyor; kırık çizginin hatası h^2 ile orantılıdır.
■

Alıştırma 11.9 (Normal Dağılımda Bir, İki ve Üç Standart Sapma). Standart normal dağılımın yoğunluğu $\varphi(x) = e^{-x^2/2}/\sqrt{2\pi}$'dir (bkz. Olasılık Teorisi). Z standart normal dağılımlı olmak üzere $k = 1, 2, 3$ için $P(|\,Z\,| \le k) = \int_{-k}^{k} \varphi(x)\,dx$ olasılıklarını `quad` ile hesaplayın ve $\operatorname{erf}(k/\sqrt{2})$ değerleriyle karşılaştırın.

Çözüm.
Denetim. Önce φ'nin bütün doğru üzerindeki integralinin 1 olduğunu, yani φ'nin gerçekten bir yoğunluk olduğunu doğrularız. Sonra her k için integrali alırız. $x = \sqrt{2}\,u$ değişken değiştirmesi $\int_{-k}^{k} \varphi(x)\,dx = \operatorname{erf}(k/\sqrt{2})$ verir.

```
import math

import numpy as np
from scipy.integrate import quad


def phi(x):
    return np.exp(-x**2 / 2) / np.sqrt(2 * np.pi)


print(quad(phi, -np.inf, np.inf))
for k in [1, 2, 3]:
    p, err = quad(phi, -k, k)
    print(f"k = {k}   olasılık = {p:.10f}"
          f"   erf = {math.erf(k / math.sqrt(2)):.10f}")
```

Çıktı:

```
(0.9999999999999998, 1.0178191320905743e-08)
k = 1   olasılık = 0.6826894921   erf = 0.6826894921
k = 2   olasılık = 0.9544997361   erf = 0.9544997361
k = 3   olasılık = 0.9973002039   erf = 0.9973002039
```

Sonuç. Olasılıklar yaklaşık 0,6827, 0,9545 ve 0,9973'tür ve on ondalık basamağa kadar `math.erf` ile aynıdır. İstatistikteki "68–95–99,7 kuralı" bu üç sayıdır.
■

Alıştırma 11.10 (Paraboloidin Altındaki Hacim). $z = 1 - x^2 - y^2$ paraboloidi ile $z = 0$ düzlemi arasındaki cismin hacmini `dblquad` ile önce dik koordinatlarda, sonra kutupsal koordinatlarda hesaplayın ve sonuçları $\pi/2$ ile karşılaştırın (bkz. İntegral Calculus).

Çözüm.
Sınırlar. Taban birim disktir. Dik koordinatlarda $-1 \le x \le 1$ ve $-\sqrt{1-x^2} \le y \le \sqrt{1-x^2}$'dir. Kutupsal koordinatlarda $0 \le \theta \le 2\pi$, $0 \le r \le 1$'dir ve $dA = r\,dr\,d\theta$ olduğundan integrand $(1 - r^2)\,r$ olur. `dblquad`'da iç değişken önce gelir: dik koordinatlarda `func(y, x)`, kutupsal koordinatlarda `func(r, theta)`.

 CC BY-NC-SA 4.0

```
import numpy as np
from scipy.integrate import dblquad

# Dik koordinatlar: -1 <= x <= 1, -sqrt(1 - x^2) <= y <= sqrt(1 - x^2)
v1, e1 = dblquad(lambda y, x: 1 - x**2 - y**2, -1, 1,
                 lambda x: -np.sqrt(1 - x**2),
                 lambda x: np.sqrt(1 - x**2))
# Kutupsal koordinatlar: 0 <= theta <= 2 pi, 0 <= r <= 1, dA = r dr dtheta
v2, e2 = dblquad(lambda r, theta: (1 - r**2) * r, 0, 2 * np.pi, 0, 1)
print(v1, e1)
print(v2, e2)
print(np.pi / 2)
```

Çıktı:

```
1.5707963267740037 2.2114190221070963e-08
1.5707963267948966 1.743934249004316e-14
1.5707963267948966
```

Sonuç. Kutupsal koordinatlardaki sonuç son basamağına kadar $\pi/2$'dir; dik koordinatlardaki sonuç ise $2 \cdot 10^{-11}$ kadar sapıyor. Fark integrandın düzgünlüğünden gelir. Kutupsal koordinatlarda iki integral de polinom integralidir. Dik koordinatlarda iç integral $\frac{4}{3}(1-x^2)^{3/2}$ fonksiyonunu verir; bu fonksiyonun $x = \pm 1$'de ikinci türevi sınırsızdır ve dış integrali zorlaştırır. İki sonuç da varsayılan $1{,}5 \cdot 10^{-8}$ hedefinin çok altındadır.
■

Alıştırma 11.11 (Bir Dörtyüzlünün Hacmi). $x + 2y + z = 2$, $x = 2y$, $x = 0$ ve $z = 0$ düzlemleriyle sınırlı dörtyüzlünün hacmini `tplquad` ile hesaplayın (bkz. İntegral Calculus).

Çözüm.
Sınırlar. Dörtyüzlü alttan $z = 0$, üstten $z = 2 - x - 2y$ düzlemiyle sınırlıdır. Üst düzlem $z = 0$ düzlemini $y = 1 - x/2$ doğrusu boyunca keser. Bu doğru $y = x/2$ doğrusuyla $x = 1$'de buluşur. Dolayısıyla cisim

$$0 \le x \le 1, \qquad \frac{x}{2} \le y \le 1 - \frac{x}{2}, \qquad 0 \le z \le 2 - x - 2y$$

eşitsizlikleriyle betimlenir. Hacim, 1 fonksiyonunun bu cisim üzerindeki integralidir.

```
from scipy.integrate import tplquad

value, err = tplquad(lambda z, y, x: 1.0,
                     0, 1,                                  # x
                     lambda x: x / 2, lambda x: 1 - x / 2,  # y
                     0, lambda x, y: 2 - x - 2 * y)         # z
print(value, err, 1 / 3)
```

Çıktı:

```
0.3333333333333333 2.210813483580885e-14 0.3333333333333333
```

Sonuç. Hacim $\frac{1}{3}$'tür; İntegral Calculus notlarındaki elle hesapla aynı.
■

Bu bölümde türev ve integrali fonksiyonun sonlu sayıda noktadaki değerlerinden hesapladık. Fark bölümlerinde kesme hatası ile yuvarlama hatasının dengesini, yamuk ve Simpson kurallarında mertebenin

anlamını, `quad` ile de uyarlamalı yöntemlerin gücünü ve sınırlarını gördük. Bu formüllerin çoğu aslında bir interpolasyon polinomunun türevi ya da integralidir. Sonraki bölümde, Polinomlar, İnterpolasyon ve Eğri Uydurma, verilen noktalardan geçen ya da onlara en iyi uyan fonksiyonları kuracağız.

 CC BY-NC-SA 4.0

12. Polinomlar, İnterpolasyon ve Eğri Uydurma

Polinomlar bilgisayarın en rahat hesapladığı fonksiyonlardır: değerleri yalnız toplama ve çarpmayla bulunur, türevleri ve integralleri yine polinomdur. Bu yüzden sayısal hesabın büyük bir kısmı, karmaşık bir fonksiyonu ya da yalnız birkaç noktada bilinen bir büyüklüğü bir polinomla veya polinom parçalarıyla değiştirme fikrine dayanır. Sayısal Türev ve İntegral bölümündeki yamuk ve Simpson kuralları da aslında bir interpolasyon polinomunun integralidir.

Elimizdeki veriyle iki farklı soru sorabiliriz. Değerler kesinse (bir tablodan okunmuş ya da pahalı bir hesaptan gelmişse) verilen noktaların hepsinden geçen bir fonksiyon isteriz; buna **interpolasyon** denir. Değerler ölçümden geliyor ve gürültü içeriyorsa bütün noktalardan geçmeye çalışmak gürültüyü de kopyalamak olur. O zaman noktaların gidişine en iyi uyan basit bir eğri ararız; buna da **eğri uydurma** denir (bkz. Nümerik Analiz).

Bu bölümde önce NumPy'ın `Polynomial` sınıfıyla polinomlar üzerinde hesap yapıyor ve köklerin katsayılara ne kadar duyarlı olabildiğini görüyoruz. Ardından interpolasyon polinomunu üç yoldan kuruyor, eşit aralıklı noktalarda derece artınca ortaya çıkan Runge olayını ve onu gideren Çebişev düğümlerini inceliyoruz. Sonra SciPy'ın kübik spline'larına geçiyoruz. En sonda `polyfit` ve `curve_fit` ile ölçüm verisine model uyduruyor ve artıklara bakarak modelin yeterli olup olmadığına karar vermeyi öğreniyoruz. Yöntemlerin teorisi Nümerik Analiz notlarındadır; burada onların Python'daki karşılığını kuruyoruz.

12.1 NumPy'da Polinomlar

Polinomlarla çalışmak için NumPy'ın `numpy.polynomial` paketini kullanacağız.

Tanım 12.1 (Polynomial Nesnesi). `numpy.polynomial.Polynomial` sınıfının bir örneğine **`Polynomial` nesnesi** denir. `Polynomial([a0, a1, ..., an])` çağrısı

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

polinomunu temsil eden bir nesne kurar. Katsayılar **artan derece** sırasıyla verilir ve nesnenin `coef` niteliğinde saklanır.

Yani bir `Polynomial` nesnesi, katsayı listesini taşıyan ve bir polinom gibi davranan bir Python nesnesidir: fonksiyon gibi çağrılır, başka polinomlarla toplanıp çarpılır, türevi ve integrali alınır. Sınıflar, Hata Yakalama ve Dosyalar bölümünün alıştırmalarında yazdığımız `Poly` sınıfı bu fikrin küçük bir örneğiydi. $p(x) = (x-1)(x-2)(x-3)$ polinomuyla başlayalım. Açılmış hâli $-6 + 11x - 6x^2 + x^3$ olduğundan katsayı listesi `[-6, 11, -6, 1]`'dir:

```
import numpy as np
from numpy.polynomial import Polynomial

p = Polynomial([-6, 11, -6, 1])      # -6 + 11x - 6x^2 + x^3
print(p)
print(p.coef, p.degree())
print(p(4))
print(p(np.array([0.0, 1.0, 2.5])))
```

Çıktı:

```
-6.0 + 11.0 x - 6.0 x**2 + 1.0 x**3
[-6. 11. -6.  1.] 3
6.0
[-6.     0.    -0.375]
```

`degree` metodu polinomun derecesini verir. Nesne bir sayıyla çağrılınca bir sayı, bir NumPy dizisiyle çağrılınca dizinin her elemanı için ayrı bir değer döndürür; örneğin $p(2{,}5) = 1{,}5 \cdot 0{,}5 \cdot (-0{,}5) = -0{,}375$'tir. Polinomun yazdırılış biçimi işletim sistemine bağlıdır. Bu notlardaki çıktılar Windows'ta alındı ve kuvvetleri `x**2` biçiminde gösteriyor. Linux, macOS ve Colab'da aynı polinom `-6.0 + 11.0·x - 6.0·x² + 1.0·x³` biçiminde yazılır. İsterseniz programın başında `np.polynomial.set_default_printstyle("ascii")` çağırarak biçimi her yerde aynı yapabilirsiniz.
Polinomlar arasındaki işlemler alışılmış operatörlerle yazılır. `divmod(p, q)` bölümü ve kalanı birlikte verir; `deriv` türevi, `integ` ise sabit terimi sıfır olan ilkel fonksiyonu döndürür:

```
from numpy.polynomial import Polynomial

p = Polynomial([-6, 11, -6, 1])
q = Polynomial([1, 1])              # 1 + x
print(p + q)
print(p * q)
print(q**3)
quot, rem = divmod(p, q)
print(quot, "|", rem)
print(p.deriv())
P = p.integ()                       # P(0) = 0 olan ilkel fonksiyon
print(P)
print(P(1) - P(0))
```

Çıktı:

```
-5.0 + 12.0 x - 6.0 x**2 + 1.0 x**3
-6.0 + 5.0 x + 5.0 x**2 - 5.0 x**3 + 1.0 x**4
1.0 + 3.0 x + 3.0 x**2 + 1.0 x**3
18.0 - 7.0 x + 1.0 x**2 | -24.0
11.0 - 12.0 x + 3.0 x**2
0.0 - 6.0 x + 5.5 x**2 - 2.0 x**3 + 0.25 x**4
-2.25
```

Bölme satırı $p(x) = (x+1)(x^2 - 7x + 18) - 24$ eşitliğini söylüyor. Kalan teoremine göre kalan $p(-1) = (-2)(-3)(-4) = -24$ olmalıdır ve öyle. Son satır, ilkel fonksiyonun uç noktalardaki farkı olarak

$$\int_0^1 (x-1)(x-2)(x-3)\,dx = -\frac{9}{4}$$

integralini hesapladı.
Bir polinomun değerini hesaplamanın en ekonomik yolu, onu iç içe çarpımlar biçiminde yazmaktır.

Tanım 12.2 (Horner Yöntemi). $p(x) = a_0 + a_1 x + \ldots + a_n x^n$ polinomunun bir x noktasındaki değerini

$$b_n = a_n, \qquad b_k = a_k + x\,b_{k+1} \quad (k = n-1, \ldots, 1, 0)$$

yinelemesiyle hesaplayıp $p(x) = b_0$ almaya **Horner yöntemi** denir.

Yani polinom

$$p(x) = a_0 + x\left(a_1 + x\left(a_2 + \ldots + x\,(a_{n-1} + x\,a_n)\cdots\right)\right)$$

biçiminde içten dışa doğru hesaplanır. Her adımda bir çarpma ve bir toplama yapıldığından toplam maliyet n çarpma ve n toplamadır; her $a_k x^k$ terimini ayrı ayrı hesaplamak ise yaklaşık $n^2/2$ çarpma ister. Yöntem birkaç satırda yazılır. NumPy işlemleri dizilere eleman eleman uygulandığından aynı kod bütün bir diziyle de çalışır:

```
import numpy as np
from numpy.polynomial import Polynomial
```

 CC BY-NC-SA 4.0

```
def horner(coef, x):
    """Katsayıları artan sırada verilen polinomun x'teki değeri."""
    result = 0.0
    for a in reversed(coef):
        result = result * x + a
    return result


p = Polynomial([-6, 11, -6, 1])
x = np.array([0.0, 1.0, 2.5, 4.0])
print(horner(p.coef, x))
print(p(x))
```

Çıktı:

```
[-6.     0.    -0.375  6.   ]
[-6.     0.    -0.375  6.   ]
```

`Polynomial` nesnesinin değer hesabı da aynı sonucu veriyor.

⚠ İki ayrı katsayı sırası

NumPy'da polinomlar için iki arayüz vardır. Eski arayüzün fonksiyonları (`np.polyval`, `np.polyfit`, `np.roots`, `np.poly1d`) katsayıları **en yüksek dereceden başlayarak** ister; `Polynomial` sınıfı ise **artan sırada**. Aynı listeyi iki arayüze vermek hiçbir hata mesajı üretmez, yalnız yanlış sonuç verir:

```
import numpy as np
from numpy.polynomial import Polynomial

c = [1, -6, 11, -6]          # x^3 - 6x^2 + 11x - 6, büyük dereceden başlar
print(np.polyval(c, 4))
print(np.roots(c))
print(Polynomial(c)(4))      # yanlış: Polynomial c'yi artan sırada okur
print(Polynomial(c[::-1])(4))
```

Çıktı:

```
6
[3. 2. 1.]
-231.0
6.0
```

`Polynomial(c)` listeyi $1 - 6x + 11x^2 - 6x^3$ olarak okudu ve $x = 4$'te -231 verdi. Yeni kodda `Polynomial` sınıfını kullanın; eski fonksiyonlarla çalışan bir kodla karşılaşınca listeyi `c[::-1]` ile ters çevirin.

12.2 Kökler

Bir `Polynomial` nesnesinin köklerini `roots` metodu verir; `Polynomial.fromroots` ise tersini yaparak verilen köklerden baş katsayısı 1 olan polinomu kurar.

```
from numpy.polynomial import Polynomial

p = Polynomial([-6, 11, -6, 1])
print(p.roots())
q = Polynomial([5, 0, 1])                # 5 + x^2
print(q.roots())
r = Polynomial.fromroots([1, 1, -2])     # (x - 1)^2 (x + 2)
print(r)
print(r.roots())
```

Çıktı:

 CC BY-NC-SA 4.0

```
[1. 2. 3.]
[0.-2.23606798j 0.+2.23606798j]
2.0 - 3.0 x + 0.0 x**2 + 1.0 x**3
[-2.          0.99999998  1.00000002]
```

x^2+5 polinomunun kökleri $\pm i\sqrt{5} \approx \pm 2{,}23607\,i$'dir; kökler karmaşık olunca sonuç da karmaşık sayılardan oluşan bir dizidir ve Python sanal birimi `j` ile yazar. Son satıra dikkat: $(x-1)^2(x+2)$ polinomunun çift katlı kökü 1, 0,99999998 ve 1,00000002 olarak bulundu. Hata 10^{-8} mertebesindedir, yani makine hassasiyeti $2{,}2 \cdot 10^{-16}$'nın kareköküne yakındır. Katlı köklerin bu duyarlılığını alıştırmalarda yeniden göreceğiz (Alıştırma 12.3).
Beşinci ve daha yüksek dereceden polinomlar için genel bir kök formülü olmadığından kökleri bulan yöntem sayısal olmak zorundadır. NumPy'ın yolu, problemi bir özdeğer problemine çevirmektir.

Tanım 12.3 (Eşlik Matrisi). Baş katsayısı 1 olan

$$p(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1} + x^n$$

polinomu verilsin. Alt köşegeni birlerden, son sütunu $-a_0, -a_1, \ldots, -a_{n-1}$ sayılarından oluşan ve öteki elemanları sıfır olan

$$C = \begin{pmatrix} 0 & 0 & \cdots & 0 & -a_0 \\ 1 & 0 & \cdots & 0 & -a_1 \\ 0 & 1 & \cdots & 0 & -a_2 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & -a_{n-1} \end{pmatrix}$$

$n \times n$ matrisine p'nin **eşlik matrisi** (companion matrix) denir.

Yani eşlik matrisi, polinomun katsayılarını bir matrise yerleştirmenin standart bir yoludur. Önemi, özdeğerlerinin (bkz. Lineer Cebir) polinomun kökleri olmasından gelir.

Önerme 12.1 (Eşlik Matrisinin Özdeğerleri). *r, baş katsayısı 1 olan p polinomunun bir kökü ise r, p'nin eşlik matrisi C'nin bir özdeğeridir. Özel olarak p'nin n farklı kökü varsa C'nin özdeğerleri tam olarak bu köklerdir.*

> ***İspat.***
> *$v = (1, r, r^2, \ldots, r^{n-1})$ satır vektörünü alıp vC çarpımını sütun sütun hesaplayalım. $j < n$ için C'nin j'inci sütununda yalnız $(j+1)$'inci satırda bir 1 vardır; bu yüzden vC'nin j'inci bileşeni v'nin $(j+1)$'inci bileşenidir, yani $r^j = r \cdot r^{j-1}$'dir. Son sütunda $-a_0, \ldots, -a_{n-1}$ sayıları bulunur ve $p(r) = 0$ olduğundan*
>
> $$-(a_0 + a_1 r + \ldots + a_{n-1}r^{n-1}) = r^n = r \cdot r^{n-1}$$
>
> *olur. Demek ki $vC = r\,v$'dir, yani $C^T v^T = r\,v^T$. İlk bileşeni 1 olduğundan $v \neq 0$'dır; dolayısıyla r, C^T'nin bir özdeğeridir. $\det(C^T - \lambda I) = \det(C - \lambda I)$ olduğundan C ile C^T'nin özdeğerleri aynıdır ve r, C'nin de bir özdeğeridir. $n \times n$ bir matrisin en çok n özdeğeri olduğundan p'nin n farklı kökü varsa C'nin özdeğerleri tam olarak bunlardır.* ■

Aslında daha fazlası doğrudur: C'nin karakteristik polinomu (bkz. Lineer Cebir) tam olarak p'dir; böylece katlı kökler de katlılıklarıyla birlikte özdeğer olur. Bunu $(x-1)(x-2)(x-3)$ için hem sayısal hem sembolik olarak görelim. `polycompanion` fonksiyonu, NumPy'ın kullandığı eşlik matrisini verir:

```
import numpy as np
import sympy as sp
from numpy.polynomial import polynomial as P

c = np.array([-6.0, 11.0, -6.0, 1.0])    # (x - 1)(x - 2)(x - 3)
C = P.polycompanion(c)
print(C)
```

```
print(np.linalg.eigvals(C))
x = sp.symbols("x")
print(sp.Matrix(C.astype(int)).charpoly(x).as_expr())
```

Çıktı:

```
[[  0.   0.   6.]
 [  1.   0. -11.]
 [  0.   1.   6.]]
[1.+0.j 2.+0.j 3.+0.j]
x**3 - 6*x**2 + 11*x - 6
```

Matris tanımdaki biçimdedir: son sütunda 6, -11, 6 sayıları $-a_0$, $-a_1$, $-a_2$'dir. `np.linalg.eigvals` özdeğerleri karmaşık sayı olarak döndürdü, ama sanal kısımları sıfırdır. SymPy'ın `charpoly` metodu (SymPy ile Sembolik Hesap) karakteristik polinomun p'nin kendisi olduğunu gösteriyor. `roots` metodu da bu matrisin özdeğerlerini `np.linalg.eigvals` ile hesaplar.

12.3 Köklerin Duyarlılığı

Özdeğer hesabı çok güvenilir bir algoritmadır, ama bu, katsayılardan hesaplanan köklerin her zaman doğru olduğu anlamına gelmez. Kökleri $1, 2, \ldots, 20$ olan

$$w(x) = (x-1)(x-2)\cdots(x-20)$$

Wilkinson polinomuna bakalım. Kökleri tam sayıdır ve birbirinden iyi ayrılmıştır; hesaplaması en kolay görünen polinomlardan biri olmalı.

```
import numpy as np
from numpy.polynomial import Polynomial

w = Polynomial.fromroots(np.arange(1, 21))   # (x - 1)(x - 2)...(x - 20)
print(w.coef[19], w.coef[0])
r = w.roots()
err = np.abs(r - np.arange(1, 21))
print(np.round(r[10:16], 4))
print(f"en büyük hata {err.max():.3f}, kök {np.argmax(err) + 1}")
```

Çıktı:

```
-210.0 2.43290200817664e+18
[11.0485 11.9087 13.1831 13.774  15.2444 15.792 ]
en büyük hata 0.244, kök 15
```

x^{19}'un katsayısı -210, sabit terim $20! \approx 2{,}43 \cdot 10^{18}$'dir. Yazdırılan kökler 11'den 16'ya kadar olanlardır: 15 yerine 15,2444 bulundu ve en büyük hata 0,244. Kodda bir yanlışlık yok. Bulunan kökler, katsayıları w'nin katsayılarından yuvarlama hatası düzeyinde farklı olan bir polinomun neredeyse tam kökleridir; sorun w'nin köklerinin katsayılara aşırı duyarlı olmasıdır. Bunu James Wilkinson'ın ünlü deneyiyle görelim: yalnız x^{19}'un katsayısını $2^{-23} \approx 1{,}2 \cdot 10^{-7}$ kadar azaltalım.

```
import numpy as np
from numpy.polynomial import Polynomial

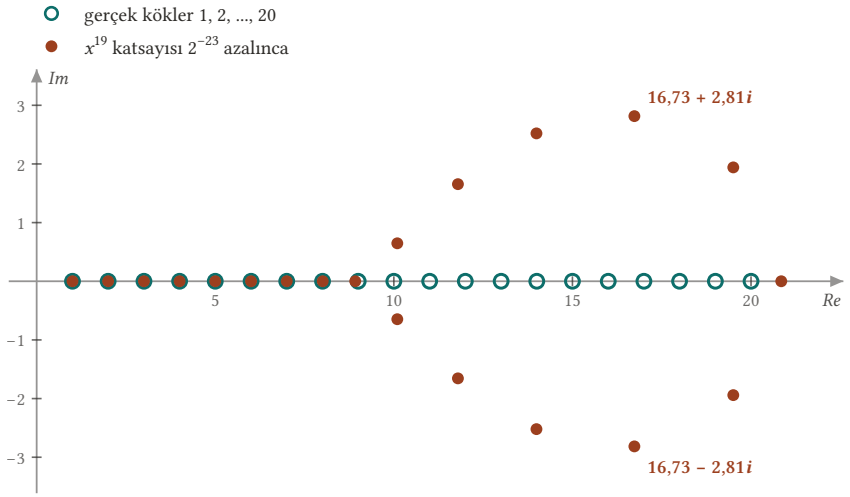
w = Polynomial.fromroots(np.arange(1, 21))
c = w.coef.copy()
c[19] -= 2.0**-23                    # x^19 katsayısı: -210 - 2^(-23)
r = Polynomial(c).roots()
r = r[np.argsort(r.real)]
print("karmaşık kök sayısı:", np.sum(r.imag != 0))
for z in r[r.imag >= 0][6:]:         # eşlenik çiftlerden yalnız biri
    print(f"{z.real:8.4f} {z.imag:+8.4f}i")
```

Çıktı:

 CC BY-NC-SA 4.0

```
karmaşık kök sayısı: 10
  6.9997  +0.0000i
  8.0075  +0.0000i
  8.9153  +0.0000i
 10.0943  +0.6478i
 11.7945  +1.6542i
 13.9930  +2.5192i
 16.7310  +2.8127i
 19.5025  +1.9403i
 20.8469  +0.0000i
```

Yirmi kökün onu karmaşık oldu; eşlenik çiftlerden yalnız üst yarı düzlemdekileri yazdırdık (ilk altı kök $1, \ldots, 6$'ya çok yakın olduğu için atlandı). Bulunan köklerin hepsi karmaşık düzlemde şöyle görünüyor:



Şekil 12.1: Wilkinson polinomu $w(x) = (x-1)(x-2)\cdots(x-20)$ için x^{19} katsayısı -210'dan $-210 - 2^{-23}$'e değiştirilince bulunan kökler (dolu noktalar) ve w'nin gerçek kökleri (içi boş). Yaklaşık 10^{-7}'lik bir değişiklik on kökü karmaşık düzleme taşır.

Katsayıdaki bağıl değişiklik $2^{-23}/210 \approx 5{,}7 \cdot 10^{-10}$ iken kökler $2{,}8$ birime kadar yer değiştirdi. Köklerin katsayılardan hesaplanması bu yüzden kötü koşullu bir problem olabilir: girdideki çok küçük bir değişiklik çıktıyı çok değiştirir. Aynı kavramı lineer sistemler için NumPy ile Lineer Cebir bölümünde koşul sayısıyla ölçmüştük. Bir polinom katsayılarıyla değil de kökleriyle ya da değerleriyle biliniyorsa, onu katsayılara çevirmeden çalışmak çoğu zaman daha güvenlidir. Bölümün geri kalanında interpolasyonda da aynı ilkeyi izleyeceğiz.

12.4 İnterpolasyon Polinomu

Şimdi bölümün asıl sorusuna geçelim: verilen noktalardan geçen polinomu bulmak.

Teorem 12.1 (İnterpolasyon Polinomunun Varlığı ve Tekliği). *$x_0, x_1, \ldots, x_n$ birbirinden farklı sayılar ve $y_0, y_1, \ldots, y_n$ keyfi sayılar olsun. Derecesi en fazla n olan ve $k = 0, 1, \ldots, n$ için $p(x_k) = y_k$ koşulunu sağlayan bir ve yalnız bir p polinomu vardır.*

İspatı için bkz. Nümerik Analiz. Bu polinoma verinin interpolasyon polinomu, x_k noktalarına da **düğümler** denir.
Polinomu bulmanın en doğrudan yolu, bilinmeyen katsayıları bir lineer sistemin çözümü olarak yazmaktır. $p(x) = a_0 + a_1 x + \ldots + a_n x^n$ için $p(x_k) = y_k$ koşulları

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

sistemini verir. Katsayı matrisi, NumPy Dizileri bölümünde `np.vander` ile kurduğumuz Vandermonde matrisidir ve teorem, düğümler farklıyken bu sistemin tek bir çözümü olduğunu söyler. $(0,1)$, $(1,3)$, $(2,2)$ ve $(3,5)$ noktalarından geçen kübik polinomu bulalım:

```
import numpy as np
from numpy.polynomial import Polynomial

x = np.array([0.0, 1.0, 2.0, 3.0])
y = np.array([1.0, 3.0, 2.0, 5.0])
V = np.vander(x, increasing=True)    # sütunlar: 1, x, x^2, x^3
a = np.linalg.solve(V, y)
p = Polynomial(a)
print(p)
print(p(x))
print(p(1.5))
```

Çıktı:

```
1.0 + 5.83333333 x - 5.0 x**2 + 1.16666667 x**3
[1. 3. 2. 5.]
2.437500000000001
```

Polinom dört düğümde de verilen değerleri alıyor. Kesin katsayılar $a_1 = \frac{35}{6}$ ve $a_3 = \frac{7}{6}$'dır, yani

$$p(x) = 1 + \frac{35}{6}\,x - 5x^2 + \frac{7}{6}\,x^3.$$

$p(1{,}5) = \frac{39}{16} = 2{,}4375$ değeri son basamakta küçük bir yuvarlama hatasıyla çıktı.
Aynı polinomu `Polynomial.fit(x, y, deg)` de verir. Bu yöntem asıl olarak en küçük kareler uydurması için yazılmıştır ve onu bölümün sonunda kullanacağız. Nokta sayısı `deg + 1` olduğunda interpolasyon polinomu artıkları sıfır yaptığından en küçük kareler çözümü tam olarak interpolasyon polinomudur.

```
import numpy as np
from numpy.polynomial import Polynomial

x = np.array([0.0, 1.0, 2.0, 3.0])
y = np.array([1.0, 3.0, 2.0, 5.0])
p = Polynomial.fit(x, y, deg=3)
print(p)
print(p.domain, p.window)
print(p.convert())
print(p(1.5))
```

Çıktı:

```
2.4375 - 1.9375 (-1.0 + 0.66666667x) + 0.5625 (-1.0 + 0.66666667x)**2 +
3.9375 (-1.0 + 0.66666667x)**3
[0. 3.] [-1.  1.]
1.0 + 5.83333333 x - 5.0 x**2 + 1.16666667 x**3
2.4374999999999996
```

 CC BY-NC-SA 4.0

⚠ Polynomial.fit ölçeklenmiş bir değişken kullanır

Çıktının ilk iki satırı şaşırtıcı görünebilir. `fit`, verinin bulunduğu $[0, 3]$ aralığını (`domain`) $[-1, 1]$ aralığına (`window`) taşıyan $u = -1 + \frac{2}{3}x$ dönüşümünü yapar ve katsayıları u'nun kuvvetlerine göre saklar. Yazdırılan polinom

$$2{,}4375 - 1{,}9375u + 0{,}5625u^2 + 3{,}9375u^3$$

polinomudur. $u = 0$ noktası $x = 1{,}5$'e karşılık geldiğinden sabit terim $p(1{,}5)$'tir. Değer hesapları doğru çıkar, çünkü nesne dönüşümü kendisi yapar; ama `p.coef` alışılmış katsayılar değildir. Alışılmış katsayılar için `convert()` çağrılmalıdır. Ölçeklemenin nedeni sayısaldır: $[-1, 1]$ aralığında u'nun hiçbir kuvveti 1'i aşmaz ve Vandermonde matrisi çok daha iyi koşullu olur (bkz. NumPy ile Lineer Cebir bölümündeki Vandermonde alıştırması).

Lineer sistem çözmeden yazılan bir biçim de vardır. Lagrange katsayı polinomları

$$L_k(x) = \prod_{j \neq k} \frac{x - x_j}{x_k - x_j}, \qquad k = 0, 1, \ldots, n$$

ile interpolasyon polinomu $p(x) = \sum_{k=0}^{n} y_k L_k(x)$ biçiminde yazılır. L_k polinomu x_k'de 1, öteki düğümlerde 0 değerini alır; bu yüzden toplam her düğümde doğru değeri verir. $L_0, \ldots, L_n$ polinomları, derecesi en fazla n olan polinomların uzayı için bir taban oluşturduğundan bunlara Lagrange taban polinomları da denir. Formülü broadcasting ile vektörel olarak yazabiliriz (Vektörizasyon ve Broadcasting): `np.delete(xn, k)`, `xn`'den `k` indisli elemanı çıkarılmış yeni bir dizi döndürür; `x[:, None] - others` farkı, her değerlendirme noktası için bir satır, her öteki düğüm için bir sütun içeren bir matristir ve `np.prod(..., axis=1)` her satırın çarpımını alır.

```
import numpy as np


def lagrange_basis(xn, k, x):
    """k'inci Lagrange taban polinomunun x noktalarındaki değerleri."""
    others = np.delete(xn, k)                # x_j, j != k
    return np.prod((x[:, None] - others) / (xn[k] - others), axis=1)


def lagrange_eval(xn, yn, x):
    """Lagrange biçimindeki interpolasyon polinomunun değerleri."""
    return sum(yn[k] * lagrange_basis(xn, k, x) for k in range(len(xn)))

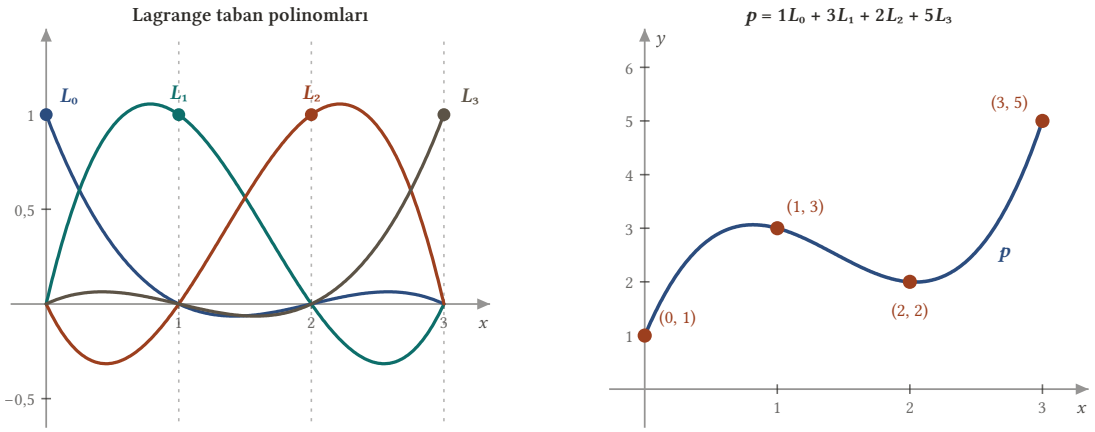

xn = np.array([0.0, 1.0, 2.0, 3.0])
yn = np.array([1.0, 3.0, 2.0, 5.0])
print(lagrange_basis(xn, 1, xn))             # düğümlerde 0, 1, 0, 0
print(lagrange_eval(xn, yn, np.array([1.5, 2.5])))
```

Çıktı:

```
[ 0.  1. -0.  0.]
[2.4375 2.5625]
```

İlk satır L_1'in dört düğümdeki değerleridir. Oradaki `-0.` eksi işaretli sıfırdır: kayan noktalı sayılar sıfırın işaretini de saklar, ama değeri yine 0'dır. İkinci satırdaki $p(1{,}5) = 2{,}4375$ Vandermonde yoluyla bulduğumuz değerdir; $p(2{,}5) = \frac{41}{16} = 2{,}5625$'tir. Taban polinomlarını ve onların ağırlıklı toplamını yan yana çizelim:

 CC BY-NC-SA 4.0



Şekil 12.2: Sol: 0, 1, 2, 3 düğümlerinin Lagrange taban polinomları. Her L_k kendi düğümünde 1, öteki düğümlerde 0 değerini alır (noktalı çizgiler düğümleri gösterir). Sağ: bu polinomların veriyle ağırlıklı toplamı olan interpolasyon polinomu $p(x) = 1 + 35x/6 - 5x^2 + 7x^3/6$ dört noktanın hepsinden geçer.

Lagrange biçimi kavramsal olarak açıktır, ama her yeni noktada bütün L_k'leri baştan hesaplar ve nokta başına maliyeti n^2 mertebesindedir. Aynı formülün küçük bir cebirle elde edilen bir yazılışı hem daha hızlı hem de sayısal olarak daha güvenilirdir.

Önerme 12.2 (Barisentrik İnterpolasyon Formülü). *$x_0, \dots, x_n$ birbirinden farklı düğümler ve*

$$w_k = \frac{1}{\prod_{j \neq k} (x_k - x_j)}, \qquad k = 0, 1, \dots, n$$

olsun. Düğümlerden farklı her x için interpolasyon polinomu

$$p(x) = \frac{\sum_{k=0}^{n} \frac{w_k\, y_k}{x - x_k}}{\sum_{k=0}^{n} \frac{w_k}{x - x_k}}$$

eşitliğini sağlar.

İspat.
$\ell(x) = \prod_{j=0}^{n} (x - x_j)$ yazalım. x bir düğüm değilse $L_k(x)$'in payı $\ell(x)/(x - x_k)$'ye, paydası da $1/w_k$'ye eşittir; yani

$$L_k(x) = \ell(x)\, \frac{w_k}{x - x_k}$$

olur ve Lagrange biçimi

$$p(x) = \ell(x) \sum_{k=0}^{n} \frac{w_k\, y_k}{x - x_k}$$

hâlini alır. Aynı eşitliği bütün y_k değerleri 1 olan veriye uygulayalım. Bu verinin interpolasyon polinomu sabit 1 polinomudur, çünkü bu polinom bütün düğümlerden geçer ve teklik (Teorem 12.1) başka bir polinoma yer bırakmaz. Dolayısıyla

$$1 = \ell(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k}$$

olur. Önceki eşitliği bu eşitliğe bölmek $\ell(x)$ çarpanını yok eder ve iddia edilen formülü verir. ■

Yani w_k ağırlıkları düğümlerden bir kez, n^2 mertebesinde işlemle hesaplanır; sonra her yeni noktada polinomun değeri n mertebesinde işlemle bulunur. Formülün bir güzelliği de bütün w_k'ler aynı sabitle çarpıldığında sonucun değişmemesidir, çünkü pay ve payda aynı sabitle çarpılır. SciPy'ın `BarycentricInterpolator` sınıfı bu formülü kullanır ve ağırlıkları, çok büyük ya da çok küçük sayılar oluşmasın diye ortak bir sabitle ölçekler:

```
import numpy as np
from scipy.interpolate import BarycentricInterpolator

xn = np.array([0.0, 1.0, 2.0, 3.0])
yn = np.array([1.0, 3.0, 2.0, 5.0])
diff = xn[:, None] - xn[None, :]     # diff[k, j] = x_k - x_j
np.fill_diagonal(diff, 1.0)          # j = k çarpanını dışarıda bırak
w = 1 / diff.prod(axis=1)
print(w)
b = BarycentricInterpolator(xn, yn)
print(b(np.array([1.5, 2.5])))
print(b.wi / w)                      # SciPy'ın ağırlıkları w'nin katı
```

Çıktı:

```
[-0.16666667  0.5        -0.5         0.16666667]
[2.4375 2.5625]
[0.421875 0.421875 0.421875 0.421875]
```

Ağırlıklarımız $-\frac{1}{6}$, $\frac{1}{2}$, $-\frac{1}{2}$, $\frac{1}{6}$'dır; SciPy'ın ağırlıkları bunların $0{,}421875 = \frac{27}{64}$ katıdır ve değerler yine $2{,}4375$ ile $2{,}5625$ çıktı. SciPy'daki eski `scipy.interpolate.lagrange` fonksiyonu sonucu tek terimlilerin katsayılarıyla verir, çok sayıda düğümde güvenilmezdir ve kullanımdan kaldırılmıştır; onun yerine `BarycentricInterpolator` kullanılmalıdır.

İnterpolasyon polinomu bir f fonksiyonunun değerlerinden kurulduğunda, polinomun f'den ne kadar saptığını şu teorem söyler.

Teorem 12.2 (İnterpolasyon Hatası). *$x_0, \dots, x_n$ noktaları $[a,b]$ aralığında birbirinden farklı, $f \in C^{n+1}[a,b]$ ve p, f'nin bu düğümlerdeki interpolasyon polinomu olsun. Her $x \in [a,b]$ için*

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\,\omega(x), \qquad \omega(x) = \prod_{k=0}^{n}(x - x_k)$$

eşitliğini sağlayan ve x'e bağlı olan bir $\xi \in (a,b)$ sayısı vardır.

İspatı için bkz. Nümerik Analiz. Hata iki çarpandan oluşur: $f^{(n+1)}(\xi)/(n+1)!$ çarpanı fonksiyona, $\omega(x)$ çarpanı ise yalnız düğümlerin yerine bağlıdır. Fonksiyonu değiştiremeyiz, ama düğümleri seçebiliyorsak ikinci çarpanı küçük tutmaya çalışmalıyız. Bunun neden önemli olduğunu Runge olayı gösterir.

12.5 Runge Olayı

Düğüm sayısını artırınca interpolasyon polinomunun fonksiyona yaklaşmasını bekleriz. Weierstrass Yaklaşım Teoremi de kapalı bir aralıkta sürekli her fonksiyona polinomlarla istenildiği kadar yaklaşılabildiğini söyler. Ama bu, eşit aralıklı noktalardaki interpolasyonun fonksiyona yaklaştığı anlamına gelmez. Carl Runge'un 1901'de verdiği örneği inceleyelim:

$$f(x) = \frac{1}{1 + 25x^2}, \qquad x \in [-1, 1].$$

f'yi $n+1$ eşit aralıklı düğümde interpolasyonla yaklaşıyor ve en büyük hatayı 2001 noktalı sık bir ızgarada ölçüyoruz:

```
import numpy as np
from scipy.interpolate import BarycentricInterpolator
```

 CC BY-NC-SA 4.0

```
def f(x):
    return 1 / (1 + 25 * x**2)


xx = np.linspace(-1, 1, 2001)        # hatayı ölçtüğümüz sık ızgara
for n in range(2, 21, 2):
    xn = np.linspace(-1, 1, n + 1)   # n + 1 eşit aralıklı düğüm
    p = BarycentricInterpolator(xn, f(xn))
    err = np.max(np.abs(p(xx) - f(xx)))
    print(f"n = {n:2d}   en büyük hata = {err:9.3e}")
```

Çıktı:

```
n =  2   en büyük hata = 6.462e-01
n =  4   en büyük hata = 4.384e-01
n =  6   en büyük hata = 6.169e-01
n =  8   en büyük hata = 1.045e+00
n = 10   en büyük hata = 1.916e+00
n = 12   en büyük hata = 3.663e+00
n = 14   en büyük hata = 7.195e+00
n = 16   en büyük hata = 1.439e+01
n = 18   en büyük hata = 2.919e+01
n = 20   en büyük hata = 5.982e+01
```

Hata $n = 4$'ten sonra azalmıyor, büyüyor ve $n = 20$'de 60'a yaklaşıyor. Oysa f son derece düzgün bir fonksiyondur, her mertebeden türevi vardır.

Tanım 12.4 (Runge Olayı). Eşit aralıklı düğümlerde kurulan interpolasyon polinomlarının, derece arttıkça aralığın uçlarına yakın giderek büyüyen salınımlar yapmasına ve en büyük hatanın sıfıra gitmek yerine büyümesine **Runge olayı** (Runge's phenomenon) denir.

Yani daha çok veri her zaman daha iyi bir polinom demek değildir. Hata teoremindeki iki çarpana bakalım. Runge fonksiyonunun yüksek mertebeden türevleri n ile çok hızlı büyür. $\omega(x)$ ise eşit aralıklı düğümlerde aralığın ortasında küçük, uçlarına yakın büyüktür; salınımların uçlarda başlaması bundandır. Çare, düğümleri ω'yı aralık boyunca küçük tutacak biçimde seçmektir.

12.6 Çebişev Düğümleri

Düğümleri aralığın uçlarına doğru sıklaştırırsak ω'yı aralığın her yerinde aynı ölçüde küçük tutabiliriz.

Tanım 12.5 (Çebişev Düğümleri). $n \geq 0$ için $[-1, 1]$ aralığındaki

$$x_k = \cos \frac{(2k+1)\pi}{2n+2}, \qquad k = 0, 1, \ldots, n$$

sayılarına $n+1$ **Çebişev düğümü** (Chebyshev nodes) denir. Bir $[a, b]$ aralığındaki Çebişev düğümleri, bu sayıların $t \mapsto \frac{a+b}{2} + \frac{b-a}{2}\, t$ dönüşümüyle taşınmasıyla elde edilir.

Yani birim çemberin üst yarısını $n+1$ eşit yaya bölüp her yayın orta noktasını x eksenine dik olarak izdüşürüyoruz. Çember üzerinde eşit aralıklı olan noktalar, eksen üzerinde uçlara doğru sıklaşır. Bu seçimin değeri şu önermededir.

Önerme 12.3 (Çebişev Düğümlerinde Düğüm Polinomu). *$x_0, \ldots, x_n$, $[-1, 1]$ aralığındaki $n+1$ Çebişev düğümü ve $\omega(x) = (x - x_0)\cdots(x - x_n)$ olsun. Her $x \in [-1, 1]$ için $|\,\omega(x)\,| \leq 2^{-n}$'dir ve bu sınıra $x = \pm 1$'de ulaşılır.*

> ***İspat.***
> *$T_0(x) = 1$, $T_1(x) = x$ ve $k \geq 1$ için $T_{k+1}(x) = 2x\,T_k(x) - T_{k-1}(x)$ yinelemesiyle tanımlanan polinomlara bakalım. Tümevarımla, $k \geq 1$ için T_k'nin derecesi k ve baş katsayısı 2^{k-1}'dir: T_1 için doğrudur ve yinelemede en yüksek dereceli terim yalnız $2x\,T_k$'den gelir.*

 CC BY-NC-SA 4.0

Her θ için $T_k(\cos\theta) = \cos k\theta$ olduğunu yine tümevarımla gösterelim. $k = 0$ ve $k = 1$ için açıktır. Kosinüsün toplam formülünden gelen

$$\cos(k+1)\theta + \cos(k-1)\theta = 2\cos\theta\cos k\theta$$

özdeşliğinden ve tümevarım hipotezinden

$$T_{k+1}(\cos\theta) = 2\cos\theta\cos k\theta - \cos(k-1)\theta = \cos(k+1)\theta$$

bulunur.
Şimdi T_{n+1}'e bakalım. $\theta_k = \frac{(2k+1)\pi}{2n+2}$ için $(n+1)\theta_k = \frac{(2k+1)\pi}{2}$ olduğundan $T_{n+1}(x_k) = \cos(n+1)\theta_k = 0$'dır. $\theta_0 < \theta_1 < \ldots < \theta_n$ sayıları $(0,\pi)$ aralığında olduğundan $x_k = \cos\theta_k$ düğümleri birbirinden farklıdır. Böylece derecesi $n+1$ olan T_{n+1}'in $n+1$ kökü tam olarak düğümlerdir. Baş katsayısı 2^n olduğundan $T_{n+1}(x) = 2^n\,\omega(x)$ olur. $x \in [-1,1]$ ise bir $\theta \in [0,\pi]$ için $x = \cos\theta$ yazılabilir ve

$$|\ \omega(x)\ | = 2^{-n}\,|\cos(n+1)\theta\ | \le 2^{-n}$$

bulunur. $x = 1$ ($\theta = 0$) ve $x = -1$ ($\theta = \pi$) noktalarında eşitlik vardır. ■

İspatta kullanılan T_k polinomlarına **Çebişev polinomları** denir; NumPy'da `numpy.polynomial.Chebyshev` sınıfıyla temsil edilirler. Önermeyi eşit aralıklı düğümlerle karşılaştıralım. ω'nın en büyük mutlak değerini yine sık bir ızgarada ölçüyoruz:

```
import numpy as np


def omega(xn, x):
    """omega(x) = (x - x_0)(x - x_1)...(x - x_n)"""
    return np.prod(x[:, None] - xn, axis=1)


def cheb_nodes(n):
    """[-1, 1] aralığında n + 1 Çebişev düğümü."""
    k = np.arange(n + 1)
    return np.cos((2 * k + 1) * np.pi / (2 * n + 2))

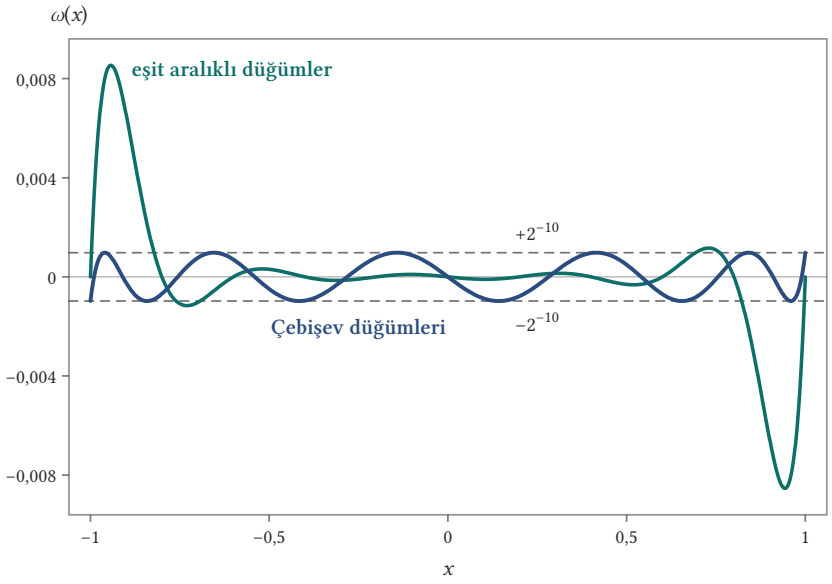

xx = np.linspace(-1, 1, 2001)
for n in (5, 10, 15, 20):
    eq = np.max(np.abs(omega(np.linspace(-1, 1, n + 1), xx)))
    ch = np.max(np.abs(omega(cheb_nodes(n), xx)))
    print(f"n = {n:2d}   eşit: {eq:9.3e}   Çebişev: {ch:9.3e}"
          f"   2^(-n): {2.0**-n:9.3e}")
```

Çıktı:

```
n =  5   eşit: 6.923e-02   Çebişev: 3.125e-02   2^(-n): 3.125e-02
n = 10   eşit: 8.532e-03   Çebişev: 9.766e-04   2^(-n): 9.766e-04
n = 15   eşit: 1.345e-03   Çebişev: 3.052e-05   2^(-n): 3.052e-05
n = 20   eşit: 2.336e-04   Çebişev: 9.537e-07   2^(-n): 9.537e-07
```

Çebişev düğümlerindeki en büyük değer önermedeki 2^{-n} sınırına eşit; eşit aralıklı düğümlerdeki değer ise $n = 20$'de bunun yaklaşık 245 katı. $n = 10$ için iki ω grafiği farkı açıkça gösteriyor:

 CC BY-NC-SA 4.0



Şekil 12.3: 11 düğüm ($n = 10$) için $\omega(x) = (x - x_0)\cdots(x - x_{10})$ çarpımı. Eşit aralıklı düğümlerde çarpım uçlara yakın $8{,}5\cdot10^{-3}$'e çıkar; Çebişev düğümlerinde aralığın her yerinde aynı yükseklikte salınır ve hiçbir zaman $2^{-10} \approx 9{,}8\cdot10^{-4}$ sınırını (kesikli çizgiler) aşmaz.

Şimdi Runge fonksiyonunu iki tür düğümle, $n = 10$ için çizelim. Kod iki alt grafikli bir şekil kurar (Matplotlib ile Grafik Çizimi); `sharey=True` iki grafiğe aynı y eksenini verir.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import BarycentricInterpolator


def f(x):
    return 1 / (1 + 25 * x**2)

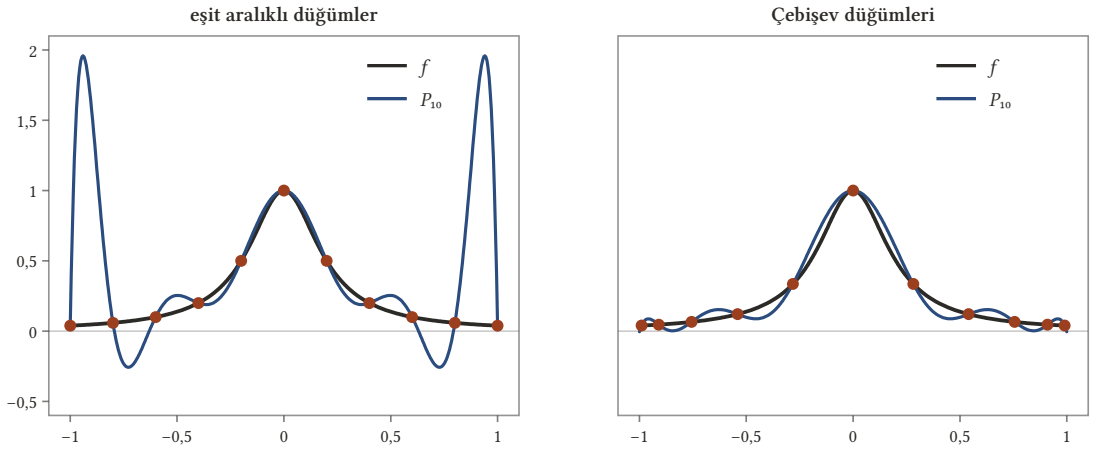

n = 10
k = np.arange(n + 1)
nodes = {"eşit aralıklı düğümler": np.linspace(-1, 1, n + 1),
         "Çebişev düğümleri": np.cos((2 * k + 1) * np.pi / (2 * n + 2))}
xx = np.linspace(-1, 1, 401)
fig, axes = plt.subplots(1, 2, figsize=(9, 3.5), sharey=True)
for ax, (title, xn) in zip(axes, nodes.items()):
    p = BarycentricInterpolator(xn, f(xn))
    ax.plot(xx, f(xx), color="black", label="f")
    ax.plot(xx, p(xx), label="P10")
    ax.plot(xn, f(xn), "o")
    ax.set_title(title)
    ax.set_ylim(-0.6, 2.1)
    ax.legend()
    err = np.max(np.abs(p(xx) - f(xx)))
    print(f"{title}: en büyük hata {err:.4f}")
fig.savefig("runge.png", dpi=150)
```

Çıktı:

```
eşit aralıklı düğümler: en büyük hata 1.9156
Çebişev düğümleri: en büyük hata 0.1092
```

 CC BY-NC-SA 4.0

Kod en büyük hataları yazdırır ve grafiği `runge.png` dosyasına kaydeder. Aşağıdaki şekil aynı grafiğin aynı verilerle çizilmiş hâlidir:



Şekil 12.4: $f(x) = 1/(1 + 25x^2)$ ve 11 düğümden geçen onuncu dereceden interpolasyon polinomu P_{10}; yukarıdaki Matplotlib kodunun çizdiği grafiğin aynı verilerle çizilmiş hâli. Eşit aralıklı düğümlerde polinom uçlara yakın büyük salınımlar yapar (en büyük hata 1,92); uçlarda sıklaşan Çebişev düğümlerinde salınım kaybolur (en büyük hata 0,11).

Derece arttıkça iki düğüm türünün nasıl davrandığını bir tabloda görelim. NumPy'ın `Chebyshev.interpolate` sınıf metodu, bir fonksiyonu Çebişev düğümlerinde interpolasyonla yaklaşan polinomu tek satırda kurar:

```
import numpy as np
from numpy.polynomial import Chebyshev
from scipy.interpolate import BarycentricInterpolator


def f(x):
    return 1 / (1 + 25 * x**2)


xx = np.linspace(-1, 1, 2001)
for n in range(4, 41, 4):
    k = np.arange(n + 1)
    xe = np.linspace(-1, 1, n + 1)
    xc = np.cos((2 * k + 1) * np.pi / (2 * n + 2))
    ee = np.max(np.abs(BarycentricInterpolator(xe, f(xe))(xx) - f(xx)))
    ec = np.max(np.abs(BarycentricInterpolator(xc, f(xc))(xx) - f(xx)))
    print(f"n = {n:2d}   eşit: {ee:9.2e}   Çebişev: {ec:9.2e}")
c = Chebyshev.interpolate(f, 10)     # Çebişev düğümlerinde interpolasyon
print(f"Chebyshev.interpolate, n = 10: {np.max(np.abs(c(xx) - f(xx))):.2e}")
```

Çıktı:

```
n =  4   eşit:  4.38e-01   Çebişev:  4.02e-01
n =  8   eşit:  1.05e+00   Çebişev:  1.71e-01
n = 12   eşit:  3.66e+00   Çebişev:  6.92e-02
n = 16   eşit:  1.44e+01   Çebişev:  3.26e-02
n = 20   eşit:  5.98e+01   Çebişev:  1.53e-02
n = 24   eşit:  2.57e+02   Çebişev:  6.95e-03
n = 28   eşit:  1.13e+03   Çebişev:  3.08e-03
n = 32   eşit:  5.06e+03   Çebişev:  1.40e-03
n = 36   eşit:  2.29e+04   Çebişev:  6.41e-04
```

 CC BY-NC-SA 4.0

```
n = 40   eşit:  1.05e+05   Çebişev:  2.89e-04
Chebyshev.interpolate, n = 10: 1.09e-01
```

Eşit aralıklı düğümlerde hata $n = 40$'ta 10^5'i geçti. Çebişev düğümlerinde ise düzenli olarak azalıyor: n her 4 arttığında kabaca 2,2 kat küçülüyor. `Chebyshev.interpolate` aynı 0,109 hatasını verdi, çünkü o da aynı düğümleri kullanır. Türevi sınırlı her f için Çebişev düğümlerindeki interpolasyon polinomlarının f'ye düzgün yakınsadığı bilinir; eşit aralıklı düğümlerde böyle bir güvence yoktur. Düğümleri seçebildiğimiz her durumda Çebişev düğümlerini tercih etmeliyiz.

12.7 Parçalı Polinomlar ve Spline

Düğümleri seçemiyorsak, örneğin veri eşit aralıklı bir tablodan geliyorsa, başka bir yol izleriz: dereceyi artırmak yerine aralığı küçük parçalara bölüp her parçada düşük dereceli bir polinom kullanırız. En basit seçim, ardışık düğümleri doğru parçalarıyla birleştirmektir ve NumPy'da bunu `np.interp` yapar:

```
import numpy as np


def f(x):
    return 1 / (1 + 25 * x**2)


xn = np.linspace(-1, 1, 11)                  # 10 eşit alt aralık
xx = np.linspace(-1, 1, 2001)
L = np.interp(xx, xn, f(xn))                 # parçalı doğrusal
print(f"en büyük hata: {np.max(np.abs(L - f(xx))):.4f}")
print(np.interp([0.05, 0.5], xn, f(xn)))
```

Çıktı:

```
en büyük hata: 0.0674
[0.875 0.15 ]
```

`np.interp(x, xn, yn)`, artan sırada verilmiş `xn` düğümleri arasında doğrusal interpolasyon yapar: 0,05 noktası 0 ile 0,2 düğümleri arasında kaldığından değer 1 ile 0,5'in ağırlıklı ortalaması olan 0,875'tir. Aynı 11 düğümle en büyük hata 0,067; aynı düğümlerdeki P_{10} polinomunun hatası ise 1,92'ydi. Bu yöntemin hatasına hata teoremi doğrudan bir sınır verir.

Önerme 12.4 (Parçalı Doğrusal İnterpolasyonun Hatası). *$f \in C^2[a,b]$, her $t \in [a,b]$ için $|\, f''(t) \,| \le M$ ve $a = x_0 < x_1 < \dots < x_n = b$ ardışık düğümleri arasındaki en büyük uzaklık h olsun. Düğümlerde f ile çakışan parçalı doğrusal L fonksiyonu için her $x \in [a,b]$ noktasında*

$$|\, f(x) - L(x) \,| \le \frac{M h^2}{8}$$

olur.

> ***İspat.***
> *x, bir $[x_i, x_{i+1}]$ aralığında olsun. Bu aralıkta L, f'nin x_i ve x_{i+1} düğümlerindeki birinci dereceden interpolasyon polinomudur. Hata teoremi (Teorem 12.2) $n = 1$ ile bir ξ için*
>
> $$|\, f(x) - L(x) \,| = \frac{|\, f''(\xi) \,|}{2}\,(x - x_i)(x_{i+1} - x)$$
>
> *verir. Toplamı sabit olan iki pozitif sayının çarpımı, sayılar eşitken en büyüktür; bu yüzden $(x - x_i)(x_{i+1} - x)$ çarpımı aralığın orta noktasında en büyük değeri olan $\frac{(x_{i+1} - x_i)^2}{4}$'ü alır. $|\, f''(\xi) \,| \le M$ ve $x_{i+1} - x_i \le h$ olduğundan*
>
> $$|\, f(x) - L(x) \,| \le \frac{M}{2} \cdot \frac{h^2}{4} = \frac{M h^2}{8}$$
>
> *bulunur.* ■

 CC BY-NC-SA 4.0

Yani düğüm sayısını iki katına çıkarmak hatayı yaklaşık dörtte birine indirir. Daha iyisini yapmak için parçaları daha yüksek dereceden seçer ve birleşme noktalarında türevlerin de uyuşmasını isteriz.

Tanım 12.6 (Kübik Spline). $a = x_0 < x_1 < \ldots < x_n = b$ düğümleri ve $y_0, y_1, \ldots, y_n$ değerleri verilsin. Aşağıdaki üç koşulu sağlayan $S : [a, b] \to \mathbb{R}$ fonksiyonuna bu verinin bir **kübik spline**'ı (cubic spline) denir:

1. Her $[x_i, x_{i+1}]$ alt aralığında S, derecesi en fazla 3 olan bir polinomdur.
2. Her $i = 0, 1, \ldots, n$ için $S(x_i) = y_i$'dir.
3. S, S' ve S'' fonksiyonları $[a, b]$ aralığında süreklidir.

Yani spline, düğümlerde birbirine türevleriyle birlikte pürüzsüzce eklenmiş kübik parçalardan oluşur. Adını, teknik çizimde eğri çizmek için kullanılan esnek cetvelden alır. Bilinmeyenleri sayalım. n parçanın her biri 4 katsayı taşır; toplam $4n$ bilinmeyen vardır. Her parçanın iki ucunda veriyle çakışması $2n$ koşul, iç düğümlerde S' ile S'''nün sürekliliği $2(n-1)$ koşul verir. Toplam $4n-2$ koşul eder; spline'ı tek kılmak için iki koşul daha gerekir. Bu iki koşul genellikle uç noktalara konur ve her seçim ayrı bir spline türü verir.

Tanım 12.7 (Doğal Spline). $S''(a) = S''(b) = 0$ koşullarını sağlayan kübik spline'a **doğal spline** (natural spline) denir.

Yani doğal spline'ın uçlarda eğriliği sıfırdır: uç düğümlerin ötesinde bir doğru gibi devam etmeye hazırdır. Fonksiyon hakkında ek bilgi istemez, ama gerçek f'' uçlarda sıfır değilse orada hatası büyür (Alıştırma 12.8).

Tanım 12.8 (Sabitlenmiş Spline). Değerleri spline'a verilen fonksiyon f olsun. $S'(a) = f'(a)$ ve $S'(b) = f'(b)$ koşullarını sağlayan kübik spline'a **sabitlenmiş spline** (clamped spline) denir.

Yani uçlardaki eğimi biz veririz. Ek bilgi kullandığı için genellikle en doğru seçenektir; ama uçlardaki türevler çoğu zaman bilinmez.

Tanım 12.9 (Düğüm Olmama Koşulu). İlk iki parçanın ve son iki parçanın aynı kübik polinom olmasını, yani S''''nün x_1 ve x_{n-1} düğümlerinde de sürekli olmasını isteyen koşula **düğüm olmama koşulu** (not-a-knot condition) denir.

Yani x_1 ve x_{n-1} noktalarında parça değişmez; bu iki nokta yalnız spline'ın geçtiği noktalar olarak kalır. Değeri, türevleri ve üçüncü türevi bir noktada çakışan iki kübik polinom özdeş olduğundan iki tanım aynı şeyi söyler. Bu koşul ek bilgi istemez ve uçlarda doğal spline'dan daha doğrudur. SciPy'ın varsayılan seçimi budur.

Spline ile interpolasyon üç adımda

1. Düğümleri artan sırada bir `x` dizisine, değerleri bir `y` dizisine koyun.
2. `S = CubicSpline(x, y, bc_type=...)` ile spline'ı kurun. Seçenekler `"not-a-knot"` (varsayılan), `"natural"` ve uç türevlerini veren `((1, da), (1, db))` biçimidir.
3. `S(t)` ile değer, `S(t, 1)` ve `S(t, 2)` ile türev, `S.integrate(a, b)` ile integral hesaplayın.

Örnek 12.1 (Runge Fonksiyonunun Spline'ı). $f(x) = \frac{1}{1+25x^2}$ fonksiyonunun $[-1, 1]$ aralığındaki 11 eşit aralıklı düğümdeki değerlerinden kübik spline kurunuz. En büyük hatayı bulunuz; $x = 0{,}3$'teki değeri ve türevi ile $[-1, 1]$ üzerindeki integrali gerçek değerlerle karşılaştırınız.

Çözüm.
Tarifin üç adımını uygulayalım. Gerçek türev $f'(x) = -\frac{50x}{(1+25x^2)^2}$, $x = 0{,}3$'te $-\frac{15}{(1+2{,}25)^2}$'dir. Gerçek integral, $\int \frac{dx}{1+25x^2} = \frac{1}{5}\arctan 5x$ olduğundan $\frac{2}{5}\arctan 5$'tir.

```
import numpy as np
from scipy.interpolate import CubicSpline
```

```
def f(x):
    return 1 / (1 + 25 * x**2)


xn = np.linspace(-1, 1, 11)                  # 10 eşit alt aralık
xx = np.linspace(-1, 1, 2001)
S = CubicSpline(xn, f(xn))                   # varsayılan: not-a-knot
print(f"en büyük hata: {np.max(np.abs(S(xx) - f(xx))):.4f}")
print(S.c.shape)                             # 4 katsayı, 10 parça
print(S(0.3), f(0.3))
print(S(0.3, 1), -15 / (1 + 25 * 0.3**2)**2)        # birinci türev
print(S.integrate(-1, 1), 0.4 * np.arctan(5))       # integral
```

Çıktı:

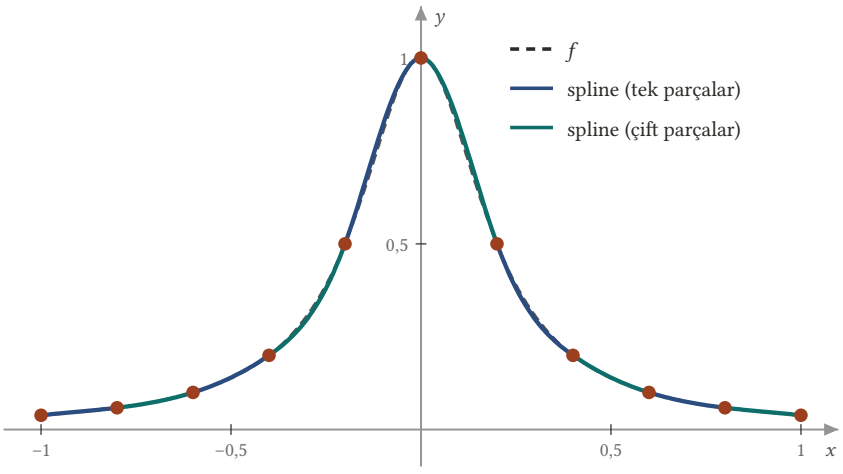
```
en büyük hata: 0.0220
(4, 10)
0.29733288239958944 0.3076923076923077
-1.3660027056024633 -1.4201183431952662
0.5519677815614747 0.5493603067780064
```

Spline'ın en büyük hatası 0,022'dir; bu, aynı düğümlerdeki P_{10} polinomunun hatasının (1,92) yaklaşık 87'de biridir. `S.c` dizisinin şekli $(4, 10)$'dur: i'inci sütun, $[x_i, x_{i+1}]$ parçasında

$$S(x) = c_0(x - x_i)^3 + c_1(x - x_i)^2 + c_2(x - x_i) + c_3$$

yazılışının dört katsayısıdır. $x = 0{,}3$'te değer 0,2973 (gerçeği 0,3077), türev $-1{,}366$ (gerçeği $-1{,}420$), integral 0,55197 (gerçeği 0,54936) çıktı. ■

Spline'ın parçalarını farklı renklerle çizersek birleşme yerleri görünür:



Şekil 12.5: $f(x) = 1/(1 + 25x^2)$ (kesikli) ve onun 11 eşit aralıklı düğümdeki değerlerinden `CubicSpline` ile kurulan kübik spline. Renkler on kübik parçayı birbirinden ayırır; parçalar düğümlerde birinci ve ikinci türevleriyle birlikte birleşir. En büyük hata 0,022'dir; aynı düğümlerdeki P_{10} polinomunun hatası 1,92'ydi.

Düğüm sayısı artınca spline'ın hatası, parçalı doğrusal interpolasyonunkinden çok daha hızlı azalır. Bu hızı ölçelim:

 CC BY-NC-SA 4.0

```python
import numpy as np
from scipy.interpolate import CubicSpline

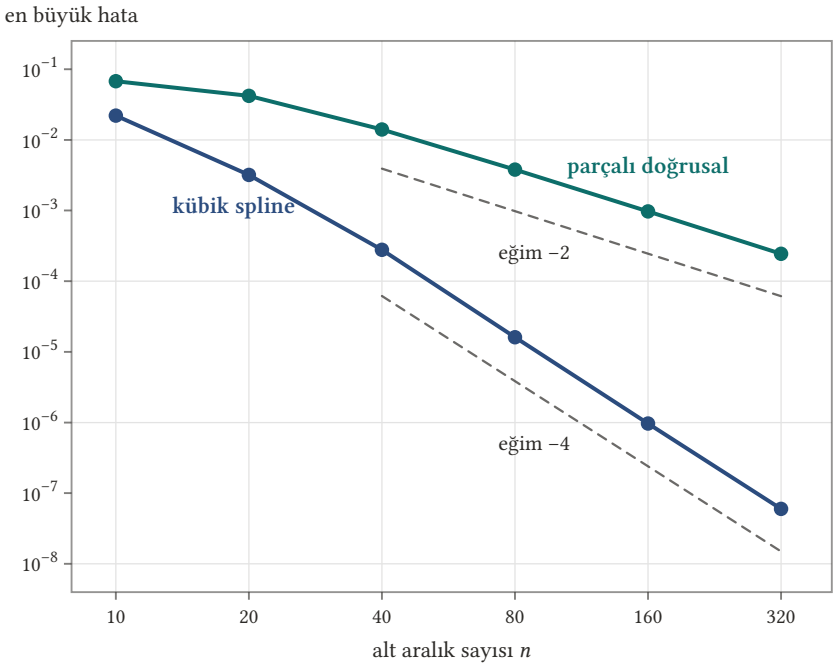

def f(x):
    return 1 / (1 + 25 * x**2)


xx = np.linspace(-1, 1, 20001)
for n in (10, 20, 40, 80, 160, 320):          # alt aralık sayısı
    xn = np.linspace(-1, 1, n + 1)
    e_lin = np.max(np.abs(np.interp(xx, xn, f(xn)) - f(xx)))
    e_spl = np.max(np.abs(CubicSpline(xn, f(xn))(xx) - f(xx)))
    print(f"n = {n:3d}   h = {2 / n:.5f}   doğrusal: {e_lin:8.2e}"
          f"   spline: {e_spl:8.2e}")
```

Çıktı:

```
n =  10   h = 0.20000   doğrusal: 6.74e-02   spline: 2.20e-02
n =  20   h = 0.10000   doğrusal: 4.18e-02   spline: 3.18e-03
n =  40   h = 0.05000   doğrusal: 1.40e-02   spline: 2.78e-04
n =  80   h = 0.02500   doğrusal: 3.80e-03   spline: 1.61e-05
n = 160   h = 0.01250   doğrusal: 9.70e-04   spline: 9.68e-07
n = 320   h = 0.00625   doğrusal: 2.44e-04   spline: 5.98e-08
```

$n = 40$'tan sonra alt aralık sayısı iki katına çıkınca parçalı doğrusal interpolasyonun hatası yaklaşık 4 kat, spline'ınki ise yaklaşık 16 kat küçülüyor; örneğin $9{,}70 \cdot 10^{-4} / 2{,}44 \cdot 10^{-4} \approx 4{,}0$ ve $9{,}68 \cdot 10^{-7} / 5{,}98 \cdot 10^{-8} \approx 16{,}2$. Daha küçük n'de oranlar daha küçüktür (doğrusal için 1,6 ve 3,0, spline için 6,9 ve 11,4), çünkü h, fonksiyonun 0 çevresindeki dar tepesine göre henüz büyüktür. İki eksen de logaritmik olunca, h^p ile orantılı bir hata eğimi $-p$ olan bir doğru çizer:



Şekil 12.6: $f(x) = 1/(1 + 25x^2)$ için parçalı doğrusal interpolasyonun (`np.interp`) ve kübik spline'ın (`CubicSpline`) en büyük hatası, iki ekseni de logaritmik ölçekte; tablodaki sayıların aynısı. $n = 40$'tan sonra alt aralık sayısı iki katına çıkınca hatalar yaklaşık 4 ve 16 kat küçülür; daha küçük n'de iki eğri de daha yavaş iner. Kesikli çizgiler bu hızlara karşılık gelen -2 ve -4 eğimlerini gösterir.

 CC BY-NC-SA 4.0

Spline'daki h^4 hızı genel bir teoremdir.

Teorem 12.3 (Sabitlenmiş Spline'ın Hatası). *$f \in C^4[a,b]$, her $t \in [a,b]$ için $|\, f^{(4)}(t)\,| \le M$ ve S, f'nin $a = x_0 < \dots < x_n = b$ düğümlerindeki sabitlenmiş spline'ı olsun. Ardışık düğümler arasındaki en büyük uzaklık h ise her $x \in [a,b]$ için*

$$|\, f(x) - S(x)\,| \le \frac{5}{384}\, M h^4$$

olur.

İspatı bu bölümün kapsamını aşıyor; sonucu sayısal olarak doğrulayacağız. Düğüm olmama koşulu için de hata h^4 ile orantılı olarak azalır. Doğal spline'da ise f'' uçlarda sıfır değilse hata uçlara yakın yalnız h^2 hızıyla azalır.

Örnek 12.2 (Sinüsün Sabitlenmiş Spline'ı). $[0, \pi]$ aralığında $\sin x$ fonksiyonunu $n = 4, 8, 16, 32, 64$ eşit alt aralıkla sabitlenmiş spline ile yaklaşınız. En büyük hatayı teoremdeki sınırla (Teorem 12.3) karşılaştırınız.

Çözüm.
Uçlardaki türevler $\cos 0 = 1$ ve $\cos \pi = -1$'dir. `bc_type=((1, 1.0), (1, -1.0))` seçeneği her uçta birinci türevi (1) verilen değere eşitler. $|\sin^{(4)} x\,| = |\sin x\,| \le 1$ olduğundan $M = 1$ alırız; $h = \pi/n$'dir.

```
import numpy as np
from scipy.interpolate import CubicSpline

xx = np.linspace(0, np.pi, 20001)
prev = None
for n in (4, 8, 16, 32, 64):
    xn = np.linspace(0, np.pi, n + 1)
    h = np.pi / n
    # sabitlenmiş spline: S'(0) = cos 0 = 1, S'(pi) = cos pi = -1
    S = CubicSpline(xn, np.sin(xn), bc_type=((1, 1.0), (1, -1.0)))
    err = np.max(np.abs(S(xx) - np.sin(xx)))
    bound = 5 / 384 * h**4                  # max |sin''''| = 1
    ratio = "" if prev is None else f"   oran: {prev / err:5.2f}"
    print(f"n = {n:2d}   hata: {err:8.2e}   sınır: {bound:8.2e}{ratio}")
    prev = err
```

Çıktı:

```
n =  4   hata: 1.12e-03   sınır: 4.95e-03
n =  8   hata: 6.32e-05   sınır: 3.10e-04   oran: 17.77
n = 16   hata: 3.89e-06   sınır: 1.94e-05   oran: 16.26
n = 32   hata: 2.42e-07   sınır: 1.21e-06   oran: 16.06
n = 64   hata: 1.51e-08   sınır: 7.56e-08   oran: 16.01
```

Hata her satırda sınırın altında kalıyor ve ardışık iki hatanın oranı 16'ya yaklaşıyor: h yarıya inince hata $2^4 = 16$ kat küçülüyor. Sınır gerçek hatanın yaklaşık 5 katıdır; teorem en kötü durumu kapsar. ■

12.8 En Küçük Kareler ile Polinom Uydurma

Ölçüm verisinde her değer bir miktar gürültü taşır ve verinin bütün noktalarından geçmeye çalışmak gürültüyü de modele katmak olur. Bunun yerine düşük dereceli bir p polinomu seçip m ölçüm (x_i, y_i) için

$$\sum_{i=1}^{m} \big(\, y_i - p(x_i)\big)^2$$

 CC BY-NC-SA 4.0

toplamını en küçük yapan katsayıları ararız. Bu, sütunları $1, x, \ldots, x^d$ olan bir Vandermonde matrisiyle kurulan ve denklem sayısı bilinmeyen sayısından fazla olan bir sistemin en küçük kareler çözümüdür (bkz. NumPy ile Lineer Cebir ve Nümerik Analiz). NumPy'da bu işi iki fonksiyon yapar: `Polynomial.fit(x, y, deg)` ve eski arayüzden `np.polyfit(x, y, deg)`. Uydurmadan sonra $r_i = y_i - p(x_i)$ artıklarına bakmak, modelin yeterli olup olmadığını anlamanın en iyi yoludur.

Tanım 12.10 (Artık Grafiği). (x_i, y_i) verisine bir $\hat{y}$ modeli uydurulmuş olsun. $r_i = y_i - \hat{y}(x_i)$ artıklarının x_i'ye karşı çizildiği grafiğe **artık grafiği** (residual plot) denir.

Yani artık grafiği, modelin açıklayamadığı kısmı gösterir. Model verinin yapısını yakalamışsa geriye yalnız gürültü kalır ve artıklar sıfırın iki yanına düzensizce dağılır. Artıklarda bir desen görülüyorsa (bir kemer, bir dalga ya da uçlara doğru açılan bir yelpaze) model bir şeyi kaçırıyordur.

> **Ölçüm verisine üç adımda polinom uydurmak**
>
> 1. Bir derece seçin; mümkünse dereceyi problemin kendisinden ya da verinin grafiğinden belirleyin.
> 2. `p = Polynomial.fit(x, y, deg)` ile uydurun; alışılmış katsayılar için `p.convert().coef` kullanın.
> 3. `r = y - p(x)` artıklarını hesaplayıp çizin. Artıklarda bir desen varsa dereceyi artırın ya da modeli değiştirin.

Örnek 12.3 (Yukarı Atılan Bir Cismin Yüksekliği). 25 m yükseklikten 3 m/s hızla yukarı atılan bir cismin yüksekliği $t \in [0, 2]$ s aralığında 0,1 s arayla 21 kez ölçülüyor. Ölçümleri üretmek için $h(t) = 25 + 3t - 4{,}905t^2$ değerlerine standart sapması 0,05 m olan normal dağılımlı gürültü ekleniyor. Veriye birinci ve ikinci dereceden polinom uydurup artık kareleri toplamlarını karşılaştırınız.

Çözüm.
`rng.normal(0, 0.05, t.size)` ortalaması 0, standart sapması 0,05 olan normal dağılımdan `t.size` tane sayı üretir (Rastgele Sayılar ve Monte Carlo Yöntemleri); tohum sabit olduğundan her çalıştırmada aynı veri çıkar. Tarifin ilk iki adımını iki derece için uygulayalım:

```
import numpy as np
from numpy.polynomial import Polynomial

rng = np.random.default_rng(16)
t = np.linspace(0, 2, 21)                          # s
h = 25 + 3 * t - 4.905 * t**2 + rng.normal(0, 0.05, t.size)   # m

for deg in (1, 2):
    p = Polynomial.fit(t, h, deg).convert()
    r = h - p(t)                                   # artıklar
    print(f"derece {deg}: katsayılar {np.round(p.coef, 4)}")
    print(f"   artık kareleri toplamı {np.sum(r**2):.4f}")
```

Çıktı:

```
derece 1: katsayılar [28.1307 -6.8326]
   artık kareleri toplamı 54.0890
derece 2: katsayılar [25.022   2.9842 -4.9084]
   artık kareleri toplamı 0.0435
```

Doğrunun artık kareleri toplamı 54,1, parabolünkü 0,0435'tir; parabol yaklaşık 1200 kat daha iyi uyuyor. Parabolün katsayıları 25,022; 2,984; $-4{,}908$ ve verinin üretildiği 25; 3; $-4{,}905$ değerlerine çok yakın. ■

Tarifin üçüncü adımı artıklara bakmaktır. Aşağıdaki kod üç alt grafik kurar: ilkinde veri ve iki polinom, ötekilerde iki modelin artıkları vardır.

 CC BY-NC-SA 4.0

```
import matplotlib.pyplot as plt
import numpy as np
from numpy.polynomial import Polynomial

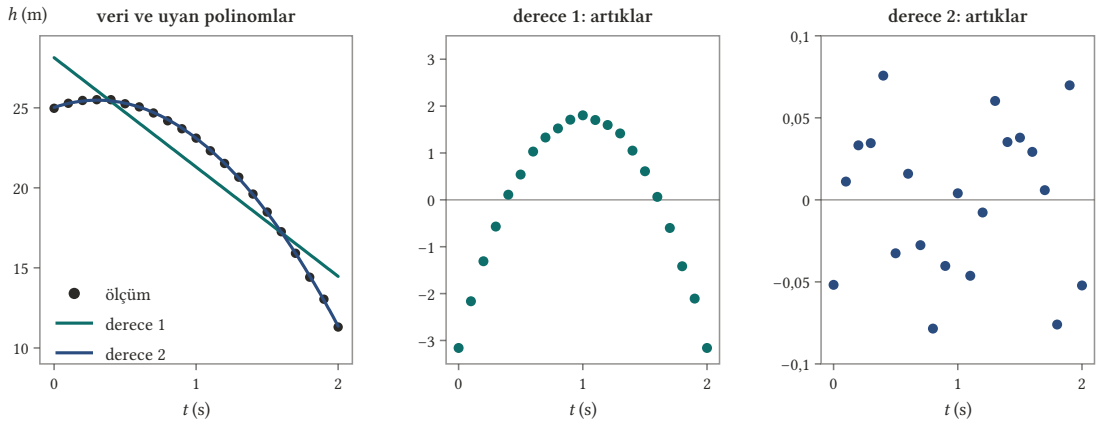
rng = np.random.default_rng(16)
t = np.linspace(0, 2, 21)
h = 25 + 3 * t - 4.905 * t**2 + rng.normal(0, 0.05, t.size)

tt = np.linspace(0, 2, 201)
fig, axes = plt.subplots(1, 3, figsize=(12, 3.5))
axes[0].plot(t, h, "o", color="black", label="ölçüm")
for deg in (1, 2):
    p = Polynomial.fit(t, h, deg)
    r = h - p(t)                                  # artıklar
    axes[0].plot(tt, p(tt), label=f"derece {deg}")
    axes[deg].plot(t, r, "o")
    axes[deg].axhline(0, color="gray", lw=0.8)
    axes[deg].set_title(f"derece {deg}: artıklar")
    print(f"derece {deg}: en büyük |artık| {np.max(np.abs(r)):.3f} m")
axes[0].set_xlabel("t (s)")
axes[0].set_ylabel("h (m)")
axes[0].legend()
fig.savefig("artiklar.png", dpi=150)
```

Çıktı:

```
derece 1: en büyük |artık| 3.161 m
derece 2: en büyük |artık| 0.078 m
```

Kod en büyük artıkları yazdırır ve grafiği `artiklar.png` dosyasına kaydeder. Aşağıdaki şekil aynı grafiğin aynı verilerle çizilmiş hâlidir:



Şekil 12.7: Yukarı atılan cismin 21 yükseklik ölçümüne birinci ve ikinci dereceden polinom uydurulması (sol) ve iki modelin artıkları; yukarıdaki Matplotlib kodunun çizdiği grafiğin aynı verilerle çizilmiş hâli. Doğrunun artıkları belirgin bir kemer çizer: model, verideki eğriliği yakalayamamıştır. Parabolün artıkları ise sıfırın iki yanına düzensizce dağılır ve 0,08 m'yi aşmaz (eksen ölçeklerinin farkına dikkat).

Doğrunun artıkları düzgün bir kemer çiziyor: ortada pozitif, uçlarda negatif. Bu, modelin verideki eğriliği yakalayamadığını söyler. Parabolün artıkları ise sıfırın iki yanına düzensizce dağılıyor ve büyüklükleri, gürültünün 0,05 m olan standart sapmasıyla uyumlu.

Modelin katsayılarının fiziksel bir anlamı da var. $h(t) = h_0 + v_0 t - \tfrac{g}{2}t^2$ olduğundan t^2'nin katsayısının -2 katı yerçekimi ivmesinin bir tahminidir. `np.polyfit`, `cov=True` seçeneğiyle katsayıların kovaryans matrisini de verir; bu matrisin köşegenindeki sayıların karekökleri katsayıların standart hatalarıdır. `polyfit`'in katsayıları en yüksek dereceden başlattığına dikkat edin:

 CC BY-NC-SA 4.0

```
import numpy as np

rng = np.random.default_rng(16)
t = np.linspace(0, 2, 21)
h = 25 + 3 * t - 4.905 * t**2 + rng.normal(0, 0.05, t.size)

c, C = np.polyfit(t, h, 2, cov=True)   # c = [c2, c1, c0]
se = np.sqrt(np.diag(C))                # katsayıların standart hataları
g, se_g = -2 * c[0], 2 * se[0]          # h = h0 + v0 t - (g/2) t^2
print(np.round(c, 4))
print(np.round(se, 4))
print(f"g ≈ {g:.3f} ± {se_g:.3f} m/s^2")
```

Çıktı:

```
[-4.9084  2.9842 25.022 ]
[0.0328 0.068  0.0293]
g ≈ 9.817 ± 0.066 m/s^2
```

Tahmin $g \approx 9{,}817 \pm 0{,}066$ m/s² oldu; gerçek 9,81 değeri tahminin bir standart hata yakınında. NumPy bu standart hataları artıkların büyüklüğünden kestirir: gürültünün varyansını, artık kareleri toplamını serbestlik derecesine, yani $21 - 3 = 18$'e bölerek tahmin eder.

⚠ Derece artırmak artıkları her zaman küçültür

Derecesi en fazla d olan polinomlar, derecesi en fazla $d + 1$ olanların arasındadır. Bu yüzden en küçük kareler uydurmasında artık kareleri toplamı dereceyle hiçbir zaman artmaz:

```
import numpy as np
from numpy.polynomial import Polynomial

rng = np.random.default_rng(16)
t = np.linspace(0, 2, 21)
h = 25 + 3 * t - 4.905 * t**2 + rng.normal(0, 0.05, t.size)

for deg in range(0, 9):
    p = Polynomial.fit(t, h, deg)
    rss = np.sum((h - p(t))**2)
    print(f"derece {deg}: artık kareleri toplamı {rss:.5f}")
```

Çıktı:

```
derece 0: artık kareleri toplamı 413.55893
derece 1: artık kareleri toplamı 54.08900
derece 2: artık kareleri toplamı 0.04348
derece 3: artık kareleri toplamı 0.04281
derece 4: artık kareleri toplamı 0.03167
derece 5: artık kareleri toplamı 0.02490
derece 6: artık kareleri toplamı 0.02473
derece 7: artık kareleri toplamı 0.02216
derece 8: artık kareleri toplamı 0.01933
```

İkinci dereceden sonraki düşüşler çok küçüktür. Bunlar modelin iyileştiğini değil, polinomun gürültüye uymaya başladığını gösterir; buna **aşırı uyum** (overfitting) denir. Dereceyi artık kareleri toplamının küçüklüğüne bakarak değil, artık grafiğine ve modelin anlamına bakarak seçin. Uydurmada kullanılmayan test noktaları da iyi bir ölçüttür (Alıştırma 12.10).

12.9 Doğrusal Olmayan Modeller

Her model katsayılarına göre doğrusal değildir ve böyle modeller için `polyfit` yetmez.

 CC BY-NC-SA 4.0

Tanım 12.11 (Parametrelerine Göre Doğrusal Model). $\varphi_1, \ldots, \varphi_m$ bilinen fonksiyonlar ve $c_1, \ldots, c_m$ parametreler olmak üzere

$$y = c_1\varphi_1(x) + c_2\varphi_2(x) + \ldots + c_m\varphi_m(x)$$

biçimindeki modele **parametrelerine göre doğrusal model** denir. Bu biçimde yazılamayan modellere **doğrusal olmayan model** denir.

Yani "doğrusal" sözü x'e göre değil, parametrelere göredir. $y = c_0 + c_1 x + c_2 x^2$ parabolü ya da $y = c_1 \sin x + c_2 e^x$ doğrusal modellerdir ve bunların en küçük kareler problemi bir lineer sistemin en küçük kareler çözümüne indirgenir. $y = ae^{-kx}$ ya da $y = A\sin \omega t$ ise k ve ω'ya göre doğrusal değildir. Doğrusal olmayan bir modelde artık kareleri toplamı parametrelerin genel bir fonksiyonudur; en küçük değeri, bir başlangıç tahmininden yola çıkan yinelemeli bir yöntemle aranır. Bu bir optimizasyon problemidir (Kök Bulma ve Optimizasyon) ve `scipy.optimize.curve_fit` fonksiyonu onu varsayılan olarak Levenberg–Marquardt yöntemiyle çözer.

Doğrusal olmayan bir modeli üç adımda uydurmak

1. Modeli, ilk argümanı bağımsız değişken, ötekileri parametreler olan bir Python fonksiyonu olarak yazın: `def model(t, a, b, c)`.
2. `popt, pcov = curve_fit(model, t, y, p0=...)` ile uydurun. `p0`, parametreler için verinin grafiğinden okunan makul bir başlangıç tahminidir.
3. `np.sqrt(np.diag(pcov))` ile parametrelerin standart hatalarını hesaplayın ve artıklara bakın.

Örnek 12.4 (Newton Soğuma Yasası). Ortam sıcaklığı T_a olan bir odada soğuyan bir sıvının sıcaklığı

$$T(t) = T_a + (T_0 - T_a)\, e^{-kt}$$

yasasına uyar. $t = 0, 2, \ldots, 30$ dakikalarında ölçülen sıcaklıklar, $T_a = 22$, $T_0 = 90$, $k = 0{,}12$ değerlerine standart sapması $0{,}5\,^\circ\mathrm{C}$ olan gürültü eklenerek üretiliyor. T_a, T_0 ve k parametrelerini `curve_fit` ile tahmin ediniz.

Çözüm.
Model k'ye göre doğrusal değildir, bu yüzden tarifi uygularız. Başlangıç tahminini verinin kendisinden okuruz. Kod önce ilk ve son ölçümü yazdırıyor: son ölçüm $23\,^\circ\mathrm{C}$ civarında olduğundan ortam sıcaklığı bunun biraz altında olmalı ve T_a için 20 deneriz. İlk ölçüm T_0'ın 90 civarında olduğunu gösteriyor; kaba bir tahmin olarak 80 yazmak yeter. k için 0,1 deneriz.

```python
import numpy as np
from scipy.optimize import curve_fit


def model(t, Ta, T0, k):
    """Newton soğuma yasası: ortam Ta, başlangıç T0, hız sabiti k."""
    return Ta + (T0 - Ta) * np.exp(-k * t)


rng = np.random.default_rng(5)
t = np.arange(0, 31, 2.0)                          # dakika
T = model(t, 22, 90, 0.12) + rng.normal(0, 0.5, t.size)   # °C

print("ilk ve son ölçüm:", np.round(T[[0, -1]], 2))
popt, pcov = curve_fit(model, t, T, p0=(20, 80, 0.1))
perr = np.sqrt(np.diag(pcov))
for name, v, e in zip(("Ta", "T0", "k"), popt, perr):
    print(f"{name} = {v:.4f} ± {e:.4f}")
r = T - model(t, *popt)
print(f"artık kareleri toplamı: {np.sum(r**2):.3f}")
```

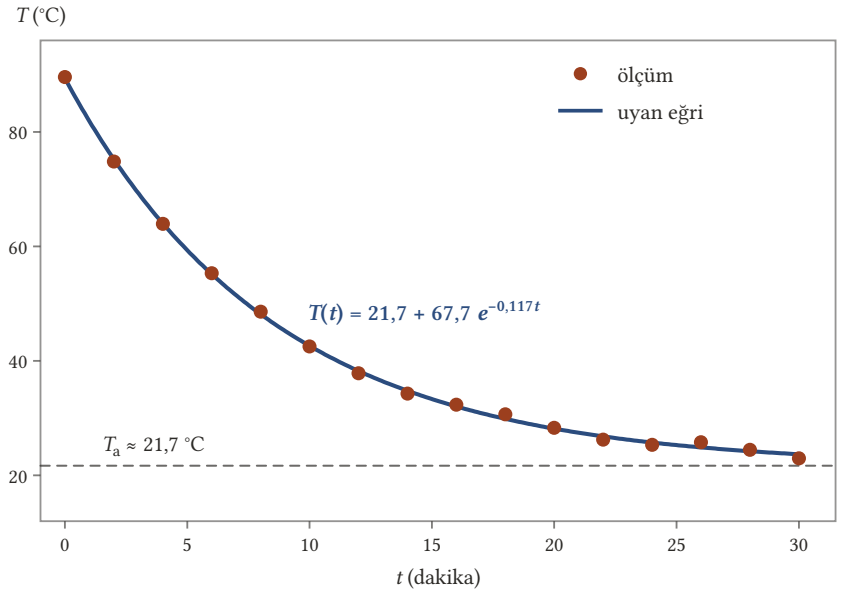
 CC BY-NC-SA 4.0

Çıktı:

```
ilk ve son ölçüm: [89.6  22.99]
Ta = 21.6914 ± 0.3530
T0 = 89.3752 ± 0.4294
k = 0.1173 ± 0.0022
artık kareleri toplamı: 3.439
```

Tahminler $T_a \approx 21{,}69 \pm 0{,}35$, $T_0 \approx 89{,}38 \pm 0{,}43$ ve $k \approx 0{,}1173 \pm 0{,}0022$ oldu. Verinin üretildiği 22, 90 ve 0,12 değerlerinin hepsi tahminlerden iki standart hatadan daha az uzakta. Artık kareleri toplamı 3,44'tür. Bunu serbestlik derecesine, yani $16 - 3 = 13$'e bölünce 0,26 çıkar; bu, gürültünün 0,25 olan varyansına çok yakındır. ■

Uydurulan eğri ölçümlerin arasından geçiyor ve zamanla tahmin edilen ortam sıcaklığına yaklaşıyor:



Şekil 12.8: Soğuyan bir sıvının iki dakikada bir ölçülen sıcaklıkları ve `curve_fit` ile uydurulan Newton soğuma eğrisi. Kesikli çizgi, tahmin edilen ortam sıcaklığıdır; eğri zamanla ona yaklaşır.

 CC BY-NC-SA 4.0

⚠ curve_fit başlangıç tahminine duyarlıdır

Yinelemeli yöntem, başlangıç tahmininin yakınındaki bir yerel minimuma gider. $y = A\sin\omega t$ modelinde frekans için kötü bir başlangıç değeri yanlış bir sonuç verir. Aşağıdaki veri $A = 2$, $\omega = 3$ ile üretildi. Kodda ω, `w` adıyla yazılıyor; artık kareleri toplamı (residual sum of squares) da kısaca `RSS` diye yazdırılıyor:

```
import numpy as np
from scipy.optimize import curve_fit


def model(t, A, w):
    return A * np.sin(w * t)


rng = np.random.default_rng(4)
t = np.linspace(0, 6, 40)
y = model(t, 2.0, 3.0) + rng.normal(0, 0.2, t.size)
for w0 in (1.0, 2.5, 4.0):
    (A, w), _ = curve_fit(model, t, y, p0=(1.0, w0))
    rss = np.sum((y - model(t, A, w))**2)
    print(f"w0 = {w0}:  A = {A:.4f}  w = {w:.4f}  RSS = {rss:.2f}")
```

Çıktı:

```
w0 = 1.0:  A = 0.2805  w = 1.7477  RSS = 78.95
w0 = 2.5:  A = 1.9694  w = 2.9947  RSS = 1.75
w0 = 4.0:  A = 0.2980  w = 4.2162  RSS = 78.73
```

Başlangıç frekansı 1 ve 4 iken yöntem yanlış yerel minimumlarda durdu ve artık kareleri toplamı 79 civarında kaldı. Başlangıç frekansı 2,5 iken gerçek değerlere ulaştı ve toplam 1,75'e indi. Başlangıç tahminini verinin grafiğinden okuyun (burada ardışık iki tepe arasındaki uzaklık $2\pi/\omega$'dır) ve sonucu her zaman artıklarla denetleyin.

12.10 Alıştırmalar

Aşağıdaki alıştırmaları önce kendiniz çözmeye çalışın; her çözümde kod ve çıktısı verilmiştir.

Alıştırma 12.1 (Bir Polinomun Yerel Ekstremumları). $p(x) = 3x^4 - 4x^3 - 12x^2 + 5$ polinomunun yerel ekstremumlarını `Polynomial` sınıfının `deriv` ve `roots` metodlarıyla bulunuz ve ikinci türev testiyle sınıflandırınız.

Çözüm.
Adım 1. Bir polinom her yerde türevlenebilir olduğundan yerel ekstremumlar ancak $p'(x) = 0$ denkleminin köklerinde, yani kritik noktalarda olabilir. Bir kritik noktada $p'' > 0$ ise yerel minimum, $p'' < 0$ ise yerel maksimum vardır (bkz. Analiz 2).
Adım 2. `deriv()` birinci, `deriv(2)` ikinci türevi verir; kritik noktalar birinci türevin kökleridir:

```
from numpy.polynomial import Polynomial

p = Polynomial([5, 0, -12, -4, 3])       # 3x^4 - 4x^3 - 12x^2 + 5
dp, d2p = p.deriv(), p.deriv(2)
print(dp)
for c in dp.roots():                     # kritik noktalar
    kind = "yerel minimum" if d2p(c) > 0 else "yerel maksimum"
    print(f"x = {c:5.2f}   p(x) = {p(c):6.2f}   p''(x) = {d2p(c):5.1f}"
          f"   {kind}")
```

Çıktı:

```
0.0 - 24.0 x - 12.0 x**2 + 12.0 x**3
x = -1.00   p(x) =   0.00   p''(x) =  36.0   yerel minimum
x =  0.00   p(x) =   5.00   p''(x) = -24.0   yerel maksimum
x =  2.00   p(x) = -27.00   p''(x) =  72.0   yerel minimum
```

Adım 3. Türevin çarpanlarına ayrılmış hâli

$$p'(x) = 12x^3 - 12x^2 - 24x = 12x(x+1)(x-2)$$

olduğundan kritik noktalar -1, 0 ve 2'dir. $x=-1$ ve $x=2$'de yerel minimum ($p(-1)=0$, $p(2)=-27$), $x=0$'da yerel maksimum ($p(0)=5$) vardır. ■

Alıştırma 12.2 (Horner Yöntemiyle Sentetik Bölme). Horner yöntemindeki ara değerlerin, $p(x)$ polinomu $x-r$'ye bölündüğünde bölümün katsayıları olduğunu gösteriniz. Bunu kullanarak bölümü ve kalanı veren bir `synthetic_division(coef, r)` fonksiyonu yazınız ve $(x-1)(x-2)(x-3)$ polinomunu $x-4$'e bölüp sonucu `divmod` ile karşılaştırınız.

Çözüm.
Adım 1. Horner yinelemesini $x=r$ noktasında yapalım: $b_n = a_n$ ve $b_k = a_k + r\,b_{k+1}$. $q(x) = b_1 + b_2 x + \dots + b_n x^{n-1}$ diyelim. Çarpımı açarsak

$$(x-r)\,q(x) = b_n x^n + \sum_{k=1}^{n-1} (b_k - r\,b_{k+1})\,x^k - r\,b_1$$

bulunur. Yineleme gereği $b_k - r\,b_{k+1} = a_k$ ve $b_n = a_n$'dir. Ayrıca $b_0 - r\,b_1 = a_0$ olduğundan $-r\,b_1 = a_0 - b_0$'dır. Dolayısıyla $(x-r)\,q(x) = p(x) - b_0$, yani $p(x) = (x-r)\,q(x) + b_0$ olur. Bölüm q, kalan da $b_0 = p(r)$'dir.
Adım 2. Fonksiyon ara değerleri bir listede biriktirir; son değer kalandır, öncekiler bölümün katsayılarıdır:

```
from numpy.polynomial import Polynomial


def synthetic_division(coef, r):
    """p(x) = (x - r) q(x) + p(r) ayrışması; katsayılar artan sırada."""
    b = [coef[-1]]
    for a in reversed(coef[:-1]):
        b.append(a + r * b[-1])          # Horner adımı
    rem = b.pop()                        # son değer p(r)
    return b[::-1], rem


coef = [-6, 11, -6, 1]                   # (x - 1)(x - 2)(x - 3)
q, rem = synthetic_division(coef, 4)
print(q, rem)
quot, rem2 = divmod(Polynomial(coef), Polynomial([-4, 1]))
print(quot, "|", rem2)
```

Çıktı:

```
[3, -2, 1] 6
3.0 - 2.0 x + 1.0 x**2 | 6.0
```

Adım 3. Bölüm $3 - 2x + x^2$, kalan $6 = p(4) = 3\cdot 2\cdot 1$'dir; `divmod` aynı sonucu verdi. Fonksiyon bölümün katsayılarını artan sırada döndürür, bu yüzden liste `[3, -2, 1]` biçimindedir. ■

 CC BY-NC-SA 4.0

Alıştırma 12.3 (Beş Katlı Bir Kökün Hesabı). $(x-1)^5$ polinomunun köklerini `roots` metoduyla hesaplayınız. Bulunan köklerin 1'e uzaklıklarını açıklayınız.

Çözüm.
Adım 1. Polinomu kökünden kurup köklerini ve köklerin 1'e uzaklıklarını yazdıralım; karşılaştırma için makine hassasiyeti ε'un beşinci kökünü de hesaplayalım:

```
import numpy as np
from numpy.polynomial import Polynomial

p = Polynomial.fromroots([1, 1, 1, 1, 1])    # (x - 1)^5
print(p)
r = p.roots()
print(np.round(r, 5))
print(np.abs(r - 1))
print(np.finfo(float).eps ** (1 / 5))
```

Çıktı:

```
-1.0 + 5.0 x - 10.0 x**2 + 10.0 x**3 - 5.0 x**4 + 1.0 x**5
[0.99877+0.j      0.99962-0.00117j 0.99962+0.00117j 1.00099-0.00072j
 1.00099+0.00072j]
[0.0012277  0.00122816 0.00122816 0.0012289  0.0012289 ]
0.000740095979741405
```

Adım 2. Kökler 1 çevresinde, yarıçapı yaklaşık $1{,}23 \cdot 10^{-3}$ olan bir çemberin üzerinde, düzgün bir beşgenin köşelerinde duruyor. Hesaplanan kökler, katsayıları yuvarlama hatası düzeyinde değişmiş bir polinomun kökleridir. $(x-1)^5 = \delta$ denkleminin kökleri 1'in çevresinde, yarıçapı $|\,\delta\,|^{1/5}$ olan bir çemberin üzerinde eşit aralıklarla dizilir. $|\,\delta\,|^{1/5} \approx 1{,}23 \cdot 10^{-3}$ ise $|\,\delta\,| \approx 2{,}8 \cdot 10^{-15}$'tir; bu, makine hassasiyetinin yaklaşık 13 katıdır.
Adım 3. Demek ki 10^{-15} mertebesindeki bir değişiklik, beş katlı kökü 10^{-3} mertebesinde oynatır. Genel olarak m katlı bir kök, katsayılardaki ε büyüklüğündeki bir değişiklikle $\varepsilon^{1/m}$ mertebesinde yer değiştirir; son satırdaki $\varepsilon^{1/5} \approx 7{,}4 \cdot 10^{-4}$ bu ölçeği gösteriyor. ■

Alıştırma 12.4 (Bölünmüş Farklarla Newton Biçimi). $(0,1), (1,3), (2,2), (3,5)$ verisinin bölünmüş farklarını hesaplayan bir fonksiyon yazınız. İnterpolasyon polinomunun Newton biçimini `Polynomial` nesneleriyle kurup Vandermonde yoluyla bulduğumuz polinomla karşılaştırınız.

Çözüm.
Adım 1. Newton biçimi

$$p(x) = f[x_0] + f[x_0,x_1](x-x_0) + \ldots + f[x_0,\ldots,x_n](x-x_0)\cdots(x-x_{n-1})$$

yazılışıdır (bkz. Nümerik Analiz). Bölünmüş farklar $f[x_i] = y_i$ ve

$$f[x_i,\ldots,x_{i+j}] = \frac{f[x_{i+1},\ldots,x_{i+j}] - f[x_i,\ldots,x_{i+j-1}]}{x_{i+j} - x_i}$$

yinelemesiyle hesaplanır.
Adım 2. Fark tablosunu tek bir dizi üzerinde sütun sütun güncelleriz: j'inci adımdan sonra `c[i]` ($i \geq j$) elemanı $f[x_{i-j},\ldots,x_i]$'yi tutar. Atamanın sağ tarafı eski değerlerle hesaplandığından güncelleme vektörel yapılabilir. Newton biçimini kurarken $(x-x_0)\cdots(x-x_{k-1})$ çarpımını `basis` adlı bir `Polynomial` nesnesinde biriktiririz:

```
import numpy as np
from numpy.polynomial import Polynomial
```

 CC BY-NC-SA 4.0

```
def divided_differences(x, y):
    """Newton biçiminin katsayıları: f[x0], f[x0,x1], ..., f[x0,...,xn]."""
    c = np.array(y, dtype=float)
    for j in range(1, len(x)):
        # c[i] <- (c[i] - c[i-1]) / (x[i] - x[i-j]),  i = j, ..., n
        c[j:] = (c[j:] - c[j - 1:-1]) / (x[j:] - x[:-j])
    return c


x = np.array([0.0, 1.0, 2.0, 3.0])
y = np.array([1.0, 3.0, 2.0, 5.0])
c = divided_differences(x, y)
print(c)
p = Polynomial([0.0])
basis = Polynomial([1.0])               # (x - x0)...(x - x_{k-1})
for ck, xk in zip(c, x):
    p = p + ck * basis
    basis = basis * Polynomial([-xk, 1.0])
print(p)
```

Çıktı:

```
[ 1.          2.         -1.5         1.16666667]
1.0 + 5.83333333 x - 5.0 x**2 + 1.16666667 x**3
```

Adım 3. Bölünmüş farklar $1, 2, -1{,}5$ ve $\frac{7}{6}$'dır, yani

$$p(x) = 1 + 2x - \frac{3}{2}\,x(x-1) + \frac{7}{6}\,x(x-1)(x-2).$$

Açılmış hâli Vandermonde yoluyla bulduğumuz $1 + \frac{35}{6}x - 5x^2 + \frac{7}{6}x^3$ polinomunun aynısıdır. Son bölünmüş fark, beklendiği gibi baş katsayıya eşittir (bkz. Nümerik Analiz). ■

Alıştırma 12.5 (Sinüs İnterpolasyonunun Hata Sınırı). $[0, \pi]$ aralığındaki 5 eşit aralıklı düğümde $\sin x$ fonksiyonunun interpolasyon polinomunu kurunuz. Gerçek en büyük hatayı interpolasyon hatası teoreminin verdiği sınırla karşılaştırınız.

Çözüm.
Adım 1. 5 düğüm $n = 4$ demektir. $|\sin^{(5)} x| = |\cos x| \le 1$ olduğundan teorem (Teorem 12.2) her $x \in [0, \pi]$ için

$$|\sin x - p(x)| \le \frac{\max|\omega|}{5!}$$

sınırını verir.
Adım 2. Beş noktaya dördüncü dereceden `Polynomial.fit` interpolasyon polinomunu verir. $\max|\omega|$'yı sık bir ızgarada ölçeriz:

```
import math

import numpy as np
from numpy.polynomial import Polynomial

xn = np.linspace(0, np.pi, 5)               # n = 4, beş düğüm
p = Polynomial.fit(xn, np.sin(xn), 4)
xx = np.linspace(0, np.pi, 4001)
```

 CC BY-NC-SA 4.0

```
err = np.max(np.abs(p(xx) - np.sin(xx)))
omega = np.max(np.abs(np.prod(xx[:, None] - xn, axis=1)))
bound = omega / math.factorial(5)            # max |sin^(5)| = 1
print(f"gerçek hata: {err:.2e}")
print(f"max |omega|: {omega:.4f}   sınır: {bound:.2e}")
```

Çıktı:

```
gerçek hata: 1.81e-03
max |omega|: 1.0852   sınır: 9.04e-03
```

Adım 3. Gerçek hata $1{,}81 \cdot 10^{-3}$, sınır $9{,}04 \cdot 10^{-3}$'tür; sınır sağlanıyor ve gerçek hatanın yaklaşık 5 katı. Sınırın kötümser olmasının nedeni, hata formülündeki $|\sin^{(5)}\xi| = |\cos\xi|$ çarpanının yerine en kötü değeri olan 1'i koymasıdır. Teoremdeki ξ noktası x'e bağlıdır ama x'in yakınında olmak zorunda değildir; bu yüzden $|\cos x|$'in 1'e yakın olduğu uçlarda bile $|\cos\xi|$ küçük olabilir. Gerçek hatadan geri hesaplanan $|\cos\xi(x)|$ değeri, yani $5!\,|\sin x - p(x)| \,/\, |\omega(x)|$ oranı, düğümler dışındaki her noktada $0{,}24$'ün altında kalıyor. ■

Alıştırma 12.6 (Runge'un Özgün Örneği). Runge örneğini $[-5,5]$ aralığında $g(x) = \frac{1}{1+x^2}$ fonksiyonuyla vermişti. 11 eşit aralıklı düğümle ve $[-5,5]$ aralığına taşınmış 11 Çebişev düğümüyle en büyük interpolasyon hatalarını hesaplayınız. Sonuçların bölümdeki $1{,}9156$ ve $0{,}1092$ değerleriyle aynı çıkmasını açıklayınız.

Çözüm.
Adım 1. Çebişev düğümlerini $[-1,1]$ aralığında hesaplayıp $t \mapsto \frac{a+b}{2} + \frac{b-a}{2}\,t$ dönüşümüyle $[a,b] = [-5,5]$ aralığına taşırız (Tanım 12.5). `Chebyshev.interpolate` aynı işi `domain` seçeneğiyle yapar.
Adım 2. Hataları sık bir ızgarada ölçelim:

```
import numpy as np
from numpy.polynomial import Chebyshev
from scipy.interpolate import BarycentricInterpolator


def f(x):
    return 1 / (1 + x**2)


a, b, n = -5.0, 5.0, 10
xx = np.linspace(a, b, 4001)
xe = np.linspace(a, b, n + 1)
k = np.arange(n + 1)
t = np.cos((2 * k + 1) * np.pi / (2 * n + 2))   # [-1, 1] düğümleri
xc = (a + b) / 2 + (b - a) / 2 * t               # [a, b] aralığına taşı
for name, xn in (("eşit aralıklı", xe), ("Çebişev", xc)):
    p = BarycentricInterpolator(xn, f(xn))
    print(f"{name:13s}: {np.max(np.abs(p(xx) - f(xx))):.4f}")
c = Chebyshev.interpolate(f, n, domain=[a, b])
print(f"{'interpolate':13s}: {np.max(np.abs(c(xx) - f(xx))):.4f}")
```

Çıktı:

```
eşit aralıklı: 1.9156
Çebişev      : 0.1092
interpolate  : 0.1092
```

 CC BY-NC-SA 4.0

Adım 3. Hatalar bölümdeki $f(x) = \frac{1}{1+25x^2}$ için bulunanlarla aynı. Nedeni, $x = 5u$ dönüşümüyle $g(5u) = \frac{1}{1+25u^2} = f(u)$ olmasıdır. Bu dönüşüm $[-1,1]$'deki eşit aralıklı düğümleri $[-5,5]$'teki eşit aralıklı düğümlere, Çebişev düğümlerini de Çebişev düğümlerine taşır. g'nin interpolasyon polinomu P ise $P(5u)$, u'nun derecesi en fazla 10 olan ve f'yi u-düğümlerinde interpolasyonla yaklaşan bir polinomdur; teklik gereği f'nin interpolasyon polinomudur. Dolayısıyla iki problemin hataları nokta nokta aynıdır. ■

Alıştırma 12.7 (Hız Ölçümlerinden Yol ve İvme). Bir aracın hızı $t = 0, 1, \ldots, 6$ saniyelerinde ölçülüyor; ölçümler $v(t) = 10\,(1 - e^{-t/2})$ m/s fonksiyonundan alınıyor. Kübik spline ile $[0,6]$ aralığında alınan yolu ve $t = 3$'teki ivmeyi tahmin edip gerçek değerlerle karşılaştırınız.

Çözüm.
Adım 1. Yol hızın integrali, ivme hızın türevidir. Gerçek değerler

$$\int_0^6 v(t)\,dt = 10\left[\,t + 2e^{-t/2}\right]_0^6 = 40 + 20e^{-3}, \qquad v'(3) = 5e^{-3/2}$$

olur.
Adım 2. Spline'ı kurup `integrate` ve birinci türevle hesaplayalım:

```
import numpy as np
from scipy.interpolate import CubicSpline

t = np.arange(0.0, 7.0)                    # 0, 1, ..., 6 saniye
v = 10 * (1 - np.exp(-t / 2))              # m/s, ölçülen hızlar
S = CubicSpline(t, v)
print(np.round(v, 4))
print(f"yol: {S.integrate(0, 6):.4f}   gerçek: {40 + 20 * np.exp(-3):.4f}")
print(f"ivme: {S(3, 1):.4f}   gerçek: {5 * np.exp(-1.5):.4f}")
```

Çıktı:

```
[0.     3.9347 6.3212 7.7687 8.6466 9.1792 9.5021]
yol: 40.9908   gerçek: 40.9957
ivme: 1.1164   gerçek: 1.1157
```

Adım 3. Yalnız 7 ölçümle yol 40,9908 m bulundu; gerçek değer 40,9957 m, bağıl hata yaklaşık 10^{-4}. İvme 1,1164 m/s², gerçeği 1,1157 m/s²'dir. ■

Alıştırma 12.8 (Doğal Spline'ın Uçlardaki Hatası). $[0,1]$ aralığında e^x fonksiyonunu $n = 10, 20, 40, 80$ eşit alt aralıkla doğal spline ve düğüm olmama koşullu spline ile yaklaşınız. Hataların azalma hızlarını karşılaştırıp farkı açıklayınız.

Çözüm.
Adım 1. İki spline'ı her n için kurup en büyük hatalarını ölçelim:

```
import numpy as np
from scipy.interpolate import CubicSpline

xx = np.linspace(0, 1, 20001)
for n in (10, 20, 40, 80):
    xn = np.linspace(0, 1, n + 1)
    errs = []
    for bc in ("natural", "not-a-knot"):
        S = CubicSpline(xn, np.exp(xn), bc_type=bc)
```

 CC BY-NC-SA 4.0

```
        errs.append(np.max(np.abs(S(xx) - np.exp(xx))))
    print(f"n = {n:2d}   doğal: {errs[0]:8.2e}   not-a-knot: {errs[1]:8.2e}")
```

Çıktı:

```
n = 10   doğal: 1.33e-03   not-a-knot: 6.93e-06
n = 20   doğal: 3.34e-04   not-a-knot: 4.56e-07
n = 40   doğal: 8.34e-05   not-a-knot: 2.92e-08
n = 80   doğal: 2.09e-05   not-a-knot: 1.85e-09
```

Adım 2. n iki katına çıkınca doğal spline'ın hatası 4 kat ($1{,}33 \cdot 10^{-3} / 3{,}34 \cdot 10^{-4} \approx 4{,}0$), düğüm olmama koşullu spline'ınki yaklaşık 16 kat ($4{,}56 \cdot 10^{-7} / 2{,}92 \cdot 10^{-8} \approx 15{,}6$) küçülüyor. Doğal spline yalnız h^2, öteki h^4 hızıyla yakınsıyor.
Adım 3. Nedeni uç koşuludur. $(e^x)'' = e^x$, uçlarda 1 ve e değerlerini alır; doğal spline ise uçlarda $S'' = 0$'ı zorlar. Bu yanlış bilgi uçlara yakın parçalarda h^2 mertebesinde bir hata doğurur. Gerçek f'' uçlarda sıfır olmadıkça doğal spline'ın bu zayıflığı vardır. ■

Alıştırma 12.9 (Doğal Spline'ın Koşullarını Denetlemek). $(0; 1)$, $(1; 2)$, $(2{,}5; 0)$, $(3; 1)$ ve $(4; 3)$ noktalarından geçen doğal spline'ı kurunuz. `S.c` katsayılarını kullanarak S'''nün iç düğümlerde sürekli ve uçlarda sıfır olduğunu doğrulayınız.

Çözüm.
Adım 1. i'inci parçada $u = x - x_i$ ile $S(x) = c_0 u^3 + c_1 u^2 + c_2 u + c_3$'tür, burada c_0, c_1 sırasıyla `S.c[0, i]` ve `S.c[1, i]`'dir. İkinci türev $S'' = 6c_0 u + 2c_1$ olur. Parçanın uzunluğu h_i ise S'' parçanın sağ ucunda $6c_0 h_i + 2c_1$, sol ucunda $2c_1$ değerini alır. Bir iç düğümde, soldaki parçanın sağ uç değeri ile sağdaki parçanın sol uç değeri eşit olmalıdır.
Adım 2. Bu değerleri bütün parçalar için birden hesaplayalım; kodda c_0 ve c_1, `c3` ve `c2` adlarıyla tutuluyor:

```
import numpy as np
from scipy.interpolate import CubicSpline

x = np.array([0.0, 1.0, 2.5, 3.0, 4.0])
y = np.array([1.0, 2.0, 0.0, 1.0, 3.0])
S = CubicSpline(x, y, bc_type="natural")
h = np.diff(x)
c3, c2 = S.c[0], S.c[1]          # i'inci parça: c3 u^3 + c2 u^2 + ...
left = 6 * c3 * h + 2 * c2       # S'' parçanın sağ ucunda (soldan)
right = 2 * c2                   # S'' parçanın sol ucunda (sağdan)
print(np.round(left[:-1], 10))   # iç düğümlerde soldan
print(np.round(right[1:], 10))   # iç düğümlerde sağdan
print(S(x[0], 2), S(x[-1], 2))   # uç noktalarda S''
```

Çıktı:

```
[-4.89423077  6.98076923 -1.16346154]
[-4.89423077  6.98076923 -1.16346154]
0.0 0.0
```

Adım 3. İç düğümlerde ($x = 1; 2{,}5; 3$) soldan ve sağdan hesaplanan S'' değerleri on basamağa kadar aynı ve uç noktalarda $S'' = 0$. Spline, tanımdaki süreklilik koşulunu ve doğal spline'ın uç koşulunu sağlıyor. ■

Alıştırma 12.10 (Test Noktalarıyla Derece Seçimi). $[0, 1]$ aralığından rastgele seçilmiş 40 noktada $\sin 2\pi x$ değerlerine standart sapması 0,2 olan gürültü eklenerek bir veri üretiliyor. Her dördüncü noktayı

 CC BY-NC-SA 4.0

test için ayırıp kalan 30 noktaya 1'den 10'a kadar her dereceden polinom uydurunuz. Uydurmada kullanılan noktalardaki artık kareleri toplamını ve test noktalarındaki hatanın karesel ortalamasını karşılaştırarak bir derece seçiniz.

Çözüm.
Adım 1. `test` adlı mantıksal dizi, test noktalarını işaretler; `~test` onun değilidir (Vektörizasyon ve Broadcasting). Uydurma yalnız `~test` noktalarıyla yapılır. Test hatası olarak

$$\mathrm{RMS} = \sqrt{\frac{1}{10}\sum_{\text{test}} \big(\, y_i - p(x_i)\big)^2}$$

karesel ortalamasını kullanırız.
Adım 2. Bütün dereceleri bir döngüde deneyelim:

```
import numpy as np
from numpy.polynomial import Polynomial

rng = np.random.default_rng(3)
x = np.sort(rng.uniform(0, 1, 40))
y = np.sin(2 * np.pi * x) + rng.normal(0, 0.2, x.size)
test = np.zeros(x.size, dtype=bool)
test[::4] = True                          # her dördüncü nokta test için

for deg in range(1, 11):
    p = Polynomial.fit(x[~test], y[~test], deg)
    rss = np.sum((y[~test] - p(x[~test]))**2)
    rms = np.sqrt(np.mean((y[test] - p(x[test]))**2))
    print(f"derece {deg:2d}   eğitim RSS: {rss:6.3f}   test RMS: {rms:.3f}")
```

Çıktı:

```
derece  1   eğitim RSS:  8.859   test RMS: 0.565
derece  2   eğitim RSS:  8.511   test RMS: 0.652
derece  3   eğitim RSS:  1.294   test RMS: 0.184
derece  4   eğitim RSS:  1.212   test RMS: 0.204
derece  5   eğitim RSS:  0.927   test RMS: 0.284
derece  6   eğitim RSS:  0.927   test RMS: 0.285
derece  7   eğitim RSS:  0.902   test RMS: 0.325
derece  8   eğitim RSS:  0.899   test RMS: 0.303
derece  9   eğitim RSS:  0.895   test RMS: 0.281
derece 10   eğitim RSS:  0.889   test RMS: 0.251
```

Adım 3. Eğitim noktalarındaki artık kareleri toplamı dereceyle hep azalıyor, uyarıda gördüğümüz gibi. Test hatası ise 3'üncü derecede en küçük değerini (0,184) alıyor ve daha yüksek derecelerde hep bundan büyük kalıyor; polinom gürültüye uymaya başlıyor. En küçük test hatasının gürültünün 0,2 olan standart sapmasına yakın olması, modelin açıklanabilen yapıyı yakaladığını düşündürür. Seçilecek derece 3'tür. ■

Alıştırma 12.11 (Üstel Bir Modeli İki Yoldan Uydurmak). $x \in [0, 4]$ aralığında eşit aralıklı 17 noktada $y = 2e^{0{,}8x}$ değerlerine standart sapması 0,5 olan gürültü ekleniyor. $y = ae^{bx}$ modelini önce $\ln y = \ln a + bx$ doğrusuna `polyfit` uydurarak, sonra `curve_fit` ile uydurunuz. İki sonucu artık kareleri toplamıyla karşılaştırınız.

Çözüm.
Adım 1. Logaritma almak modeli $\ln a$ ve b parametrelerine göre doğrusal yapar; ama bunun için bütün ölçümlerin pozitif olması gerekir. `curve_fit` için başlangıç tahminini logaritma yönteminin sonucundan alırız.
Adım 2. İki uydurmayı yapıp artık kareleri toplamlarını özgün ölçekte hesaplayalım:

 CC BY-NC-SA 4.0

```python
import numpy as np
from scipy.optimize import curve_fit


def model(x, a, b):
    return a * np.exp(b * x)


rng = np.random.default_rng(5)
x = np.linspace(0, 4, 17)
y = model(x, 2.0, 0.8) + rng.normal(0, 0.5, x.size)
print(f"en küçük ölçüm: {y.min():.3f}")          # log için y > 0 gerekir

b_log, ln_a = np.polyfit(x, np.log(y), 1)        # ln y = ln a + b x
a_log = np.exp(ln_a)
(a_cf, b_cf), _ = curve_fit(model, x, y, p0=(a_log, b_log))
for name, a, b in (("log + polyfit", a_log, b_log),
                   ("curve_fit", a_cf, b_cf)):
    rss = np.sum((y - model(x, a, b))**2)
    print(f"{name:13s}  a = {a:.4f}  b = {b:.4f}  RSS = {rss:.2f}")
```

Çıktı:

```
en küçük ölçüm: 1.599
log + polyfit  a = 1.8214  b = 0.8343  RSS = 14.61
curve_fit      a = 2.0259  b = 0.7954  RSS = 3.93
```

Adım 3. Logaritma yöntemi $a = 1{,}82$, $b = 0{,}834$ ve 14,61 artık kareleri toplamı verdi; `curve_fit` ise $a = 2{,}026$, $b = 0{,}795$ ve 3,93 verdi. Fark, iki yöntemin farklı şeyleri en küçük yapmasından gelir: logaritma yöntemi $\sum(\ln y_i - \ln \hat{y}_i)^2$'yi, `curve_fit` ise $\sum(y_i - \hat{y}_i)^2$'yi en küçük yapar. Gürültü her noktada aynı büyüklükte olduğundan küçük y değerlerinde göreli olarak büyüktür ($x = 0$ civarında 2'nin yanında 0,5). Logaritma bu noktaların sapmalarını büyütür ve uydurmayı onlara doğru çeker. Asıl ölçekteki hatayı en küçük yapmak istiyorsak `curve_fit` kullanılmalıdır; logaritma yöntemi yine de iyi bir başlangıç tahmini verir. ■

Bu bölümde polinomlarla NumPy'da hesap yapmayı, köklerin katsayılara duyarlılığını, interpolasyonu ve Runge olayı gibi tuzaklarını, spline'larla parça parça yaklaşımı ve ölçüm verisine model uydurmayı gördük. Bu fikirler bir sonraki bölümde de iş görecek: adi diferansiyel denklemler için kullanılan sayısal yöntemler, çözümü küçük adımlarda polinomlarla yaklaşır ve SciPy'ın çözücüleri adımlar arasındaki değerleri de interpolasyonla verir. Diferansiyel Denklemlerin Sayısal Çözümü bölümünde Euler ve Runge–Kutta yöntemlerini kendimiz yazıp `solve_ivp` ile tanışacağız.

 CC BY-NC-SA 4.0

13. Diferansiyel Denklemlerin Sayısal Çözümü

Bir nüfusun büyümesi, bir sarkacın salınımı, iki türün birbirini beslemesi birer diferansiyel denklemle anlatılır. Diferansiyel Denklemler notlarında ayrılabilen, homojen ve tam denklemleri kapalı biçimde çözdük; SymPy ile Sembolik Hesap bölümünde de `dsolve` aynı işi bilgisayara yaptırdı. Ne var ki çözümü bir formülle yazılabilen denklemler azınlıktadır. $y' = t - y^2$ gibi masum görünen bir denklemin çözümü elementer fonksiyonlarla yazılamaz; sarkacın $\theta'' = -\sin\theta$ denkleminin çözümü ise eliptik fonksiyonlar adı verilen özel fonksiyonlar gerektirir.

Böyle durumlarda çözümün formülünü değil, sonlu sayıda noktadaki değerlerini hesaplarız. Bu bölümde önce bu fikrin en basit iki uygulamasını, Euler ve dördüncü mertebe Runge–Kutta yöntemlerini, birkaç satırlık Python fonksiyonları olarak kendimiz yazacak ve hatalarının adım uzunluğuyla nasıl küçüldüğünü ölçeceğiz. Ardından aynı işi çok daha akıllıca yapan SciPy fonksiyonu `solve_ivp`'yi, katı denklemleri, denklem sistemlerini ve faz düzlemini göreceğiz. Bölüm, Lotka–Volterra av-avcı modeli ve sarkaçla bitiyor. Bağımsız değişkeni SciPy'nin alışkanlığına uyarak t ile göstereceğiz ve onu zaman olarak düşüneceğiz; Diferansiyel Denklemler notlarındaki x'in yerini tutar. Kodlar NumPy, SciPy ve bir yerde SymPy kullanır; grafik kodları Matplotlib ile Grafik Çizimi bölümündeki nesne yönelimli yazımı izler.

13.1 Başlangıç Değer Problemi ve Yön Alanı

Sayısal yöntemlerin çözdüğü problem, bir diferansiyel denklemle bir başlangıç koşulunun ikilisidir. Birinci mertebeden bir başlangıç değer problemi (bkz. Diferansiyel Denklemler)

$$y' = f(t, y), \qquad y(t_0) = y_0$$

biçimindedir. f ve $\partial f/\partial y$, (t_0, y_0) noktasını içeren bir bölgede sürekliyse problemin t_0'ın bir komşuluğunda tek bir çözümü vardır (bkz. Diferansiyel Denklemler). Sayısal yöntemler bu çözümü bulmaz, ona yaklaşır; varlık ve teklik teoremi de yaklaştığımız şeyin gerçekten var ve tek olduğunu güvenceye alır. Bu bölümdeki denklemlerin sağ tarafları polinom, üstel ve trigonometrik fonksiyonlardan kurulu olduğundan bu koşullar her yerde sağlanır.

Denklemi çözmeden önce bile f'nin bize bir şey söylediğini fark edebiliriz: her noktada çözümün eğimini.

Tanım 13.1 (Yön Alanı). $y' = f(t, y)$ denklemi verilsin. ty-düzleminin her (t, y) noktasına, o noktadan geçen ve eğimi $f(t, y)$ olan kısa bir doğru parçası iliştirelim. Bu doğru parçalarının oluşturduğu resme denklemin **yön alanı** (direction field) denir.

Yani (t, y) noktasından geçen çözüm eğrisi o noktada tam olarak $f(t, y)$ eğimiyle ilerler; çözüm eğrileri bu yüzden yön alanındaki doğru parçalarına her noktada teğettir. Başlangıç değer problemini çözmek, (t_0, y_0) noktasından çıkıp bu doğru parçalarının gösterdiği yönü izlemek demektir.

Yön alanını çizmek için f'yi bir ızgaranın her noktasında hesaplamak yeter. Vektörizasyon ve Broadcasting bölümündeki `np.meshgrid` bunu tek satırda yapar. $y' = t - y^2$ denkleminin eğimlerini küçük bir ızgarada hesaplayalım:

```
import numpy as np


def f(t, y):
    return t - y**2


t = np.linspace(0, 2, 5)          # 0; 0.5; 1; 1.5; 2
y = np.linspace(1, -1, 5)         # yukarıdan aşağıya: 1; 0.5; ...; -1
```

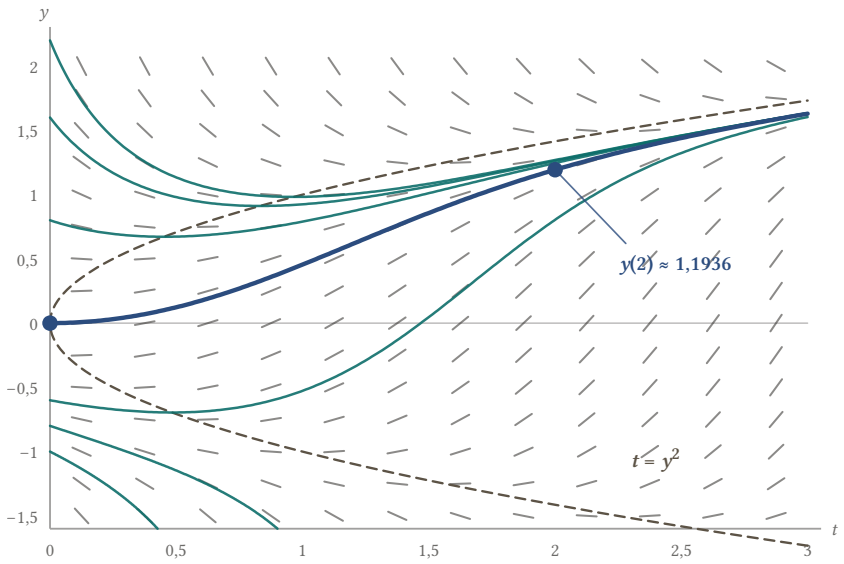
 CC BY-NC-SA 4.0

```
T, Y = np.meshgrid(t, y)
S = f(T, Y)                          # her ızgara noktasındaki eğim
print("t      :", t)
for yk, row in zip(y, S):
    print(f"y = {yk:4.1f}:", row)
```

Çıktı:

```
t      : [0.  0.5 1.  1.5 2. ]
y =  1.0: [-1.  -0.5  0.   0.5  1. ]
y =  0.5: [-0.25  0.25  0.75  1.25  1.75]
y =  0.0: [0.  0.5 1.  1.5 2. ]
y = -0.5: [-0.25  0.25  0.75  1.25  1.75]
y = -1.0: [-1.  -0.5  0.   0.5  1. ]
```

Her satır sabit bir y değerine, her sütun sabit bir t değerine karşılık gelir; satırları, şekildeki gibi yukarıdan aşağıya dizdik. Eğim y'ye yalnız y^2 üzerinden bağlı olduğundan $y = 1$ ile $y = -1$ satırları ve $y = 0{,}5$ ile $y = -0{,}5$ satırları aynıdır. $t = y^2$ olan noktalarda, örneğin $(1, 1)$ ve $(1, -1)$'de, eğim sıfırdır. Bu parabolün iç tarafında ($t > y^2$) eğim pozitiftir ve çözümler artar; dış tarafında çözümler azalır. Aşağıdaki şekil aynı hesabı daha sık bir ızgarada yapıp her noktaya eğimi kadar eğik bir çizgi koyar:



Şekil 13.1: $y' = t - y^2$ denkleminin yön alanı: her kısa çizgi, bulunduğu noktadaki eğimi gösterir. Kalın eğri $y(0) = 0$ koşulunu sağlayan çözümdür; ince eğriler başka başlangıç değerlerinden çıkan çözümlerdir. Kesikli $t = y^2$ parabolü üzerinde eğim sıfırdır: parabolün içinde çözümler artar, dışında azalır.

Şekil, denklemi çözmeden çok şey söylüyor. $y(0) \geq -0{,}6$ olan çözümler t büyüdükçe birbirine yaklaşıp parabolün üst kolunun hemen altında ilerliyor. $y(0) \leq -0{,}8$ olanlar ise parabolün alt kolunun dışına düşüyor ve orada eğim gittikçe dikleşen negatif değerler aldığından hızla aşağı iniyor. Kalın eğri $y(0) = 0$ koşulunu sağlayan çözümdür; onun $t = 2$'deki değerini bu bölümde birkaç yöntemle hesaplayacağız.

13.2 Euler Yöntemi

Yön alanını izlemenin en doğrudan yolu, bulunduğumuz noktadaki eğimle kısa bir adım atmak ve vardığımız yerde eğimi yeniden okumaktır.

Tanım 13.2 (Euler Yöntemi). $y' = f(t, y)$, $y(t_0) = y_0$ problemi ve bir $h > 0$ **adım uzunluğu** (step size) verilsin. $t_i = t_0 + ih$ noktalarında çözüme yaklaşan y_i sayıları

$$y_{i+1} = y_i + h\, f(t_i, y_i), \qquad i = 0, 1, 2, \ldots$$

kuralıyla hesaplanır. Bu kurala **Euler yöntemi** denir.

Yani her adımda çözümü bulunduğumuz noktadaki teğet doğrusuyla değiştiririz: (t_i, y_i) noktasından eğimi $f(t_i, y_i)$ olan doğru boyunca h kadar ilerleyip (t_{i+1}, y_{i+1}) noktasına varırız. Formül Taylor açılımından da okunabilir:

$$y(t+h) = y(t) + h\,y'(t) + \frac{h^2}{2}\,y''(\xi)$$

eşitliğinde son terimi atıp $y'(t) = f\,\big(\,t, y(t)\,\big)$ yazarsak Euler adımı çıkar (bkz. Nümerik Analiz). Atılan $\frac{h^2}{2}\,y''(\xi)$ terimi, bir adımda yapılan hatadır.
Yöntemi bir Python fonksiyonu olarak yazalım. Fonksiyon f'yi, başlangıç noktasını, adım uzunluğunu ve adım sayısını alır; t_i ve y_i değerlerini iki NumPy dizisi olarak döndürür. İlk denemeyi, çözümü $y = e^t$ olan $y' = y$, $y(0) = 1$ problemiyle $h = 0{,}25$ alarak yapalım:

```
import numpy as np


def euler(f, t0, y0, h, n):
    """y' = f(t, y), y(t0) = y0 problemini n Euler adımıyla çözer."""
    t = t0 + h * np.arange(n + 1)     # t_0, t_1, ..., t_n
    y = np.zeros(n + 1)
    y[0] = y0
    for i in range(n):
        y[i + 1] = y[i] + h * f(t[i], y[i])
    return t, y

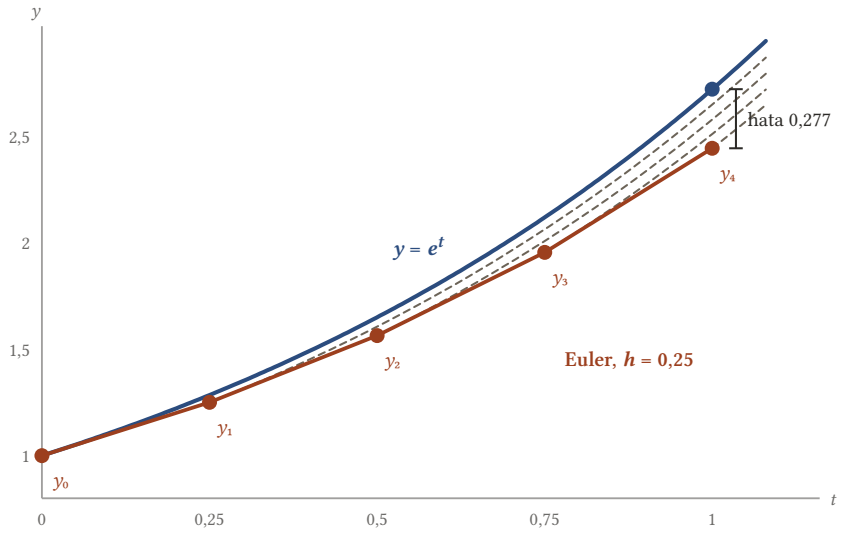

t, y = euler(lambda t, y: y, 0.0, 1.0, 0.25, 4)
print("  t      y_n        e^t        hata")
for tk, yk in zip(t, y):
    print(f"{tk:.2f}  {yk:.6f}  {np.exp(tk):.6f}  {np.exp(tk) - yk:.6f}")
```

Çıktı:

```
  t      y_n        e^t        hata
0.00  1.000000  1.000000  0.000000
0.25  1.250000  1.284025  0.034025
0.50  1.562500  1.648721  0.086221
0.75  1.953125  2.117000  0.163875
1.00  2.441406  2.718282  0.276876
```

Hesap elle de kolayca izlenir. $f(t, y) = y$ olduğundan her adım $y_{i+1} = y_i + 0{,}25\,y_i = 1{,}25\,y_i$ olur, yani $y_i = 1{,}25^i$'dir. $t = 1$'de $1{,}25^4 = 2{,}441406\ldots$ değeri $e = 2{,}718281\ldots$'den $0{,}277$ kadar küçük kalır ve hata her adımda büyür. Nedenini şekil gösteriyor:

 CC BY-NC-SA 4.0



Şekil 13.2: $y' = y$, $y(0) = 1$ problemine $h = 0{,}25$ ile dört Euler adımı. Her adım, bulunduğu noktadan geçen çözümün teğeti boyunca ilerler; kesikli eğriler bu çözümlerdir ($y = y_i e^{t - t_i}$). Yöntem her adımda bir alttaki çözüme atlar ve hatalar birikir: $t = 1$'de 2,4414 değeri $e \approx 2{,}7183$'ün 0,277 altında kalır.

e^t konveks olduğundan teğet doğruları eğrinin altında kalır. Her Euler adımı bu yüzden bir alttaki çözüm eğrisine iner ve sonraki adım artık o eğrinin teğetini izler; küçük hatalar böylece birikir.

Euler yöntemini dört adımda uygulamak

1. Denklemin sağ tarafını `f(t, y)` biçiminde bir Python fonksiyonu olarak yazın.
2. Başlangıç noktası (t_0, y_0)'ı, varmak istediğiniz T noktasını ve adım sayısı n'yi seçin; adım uzunluğu $h = (T - t_0)/n$ olur.
3. `euler(f, t0, y0, h, n)` çağrısıyla $y_1, \ldots, y_n$ değerlerini hesaplayın.
4. Sonucu denetleyin: tam çözüm biliniyorsa onunla karşılaştırın; bilinmiyorsa n'yi ikiye katlayıp sonucun ne kadar değiştiğine bakın.

Örnek 13.1 (Tam Çözümü Bilinmeyen Bir Problemde Euler Yöntemi). $y' = t - y^2$, $y(0) = 0$ probleminde $y(2)$'yi Euler yöntemiyle $n = 25, 50, 100, 200, 400$ adımla hesaplayınız. Sonucun kaç basamağına güvenilebileceğini ardışık sonuçlara bakarak kestiriniz.

Çözüm.
Adım 1 ve 2. $f(t, y) = t - y^2$, $t_0 = 0$, $y_0 = 0$, $T = 2$ ve $h = 2/n$'dir.
Adım 3. Yukarıda tanımladığımız `euler` fonksiyonunu kullanıyoruz; kod bir önceki kodun devamıdır:

```
def f(t, y):
    return t - y**2


ends = []
for n in [25, 50, 100, 200, 400]:
    t, y = euler(f, 0.0, 0.0, 2 / n, n)
    ends.append(y[-1])
    print(f"n = {n:3d}   h = {2 / n:.3f}   y(2) ≈ {y[-1]:.6f}")
d = np.diff(ends)
print("ardışık farklar:", np.round(d, 6))
print("farkların oranı:", np.round(d[:-1] / d[1:], 3))
```

Çıktı:

```
n =  25   h = 0.080   y(2) ≈ 1.194340
n =  50   h = 0.040   y(2) ≈ 1.193908
n = 100   h = 0.020   y(2) ≈ 1.193731
n = 200   h = 0.010   y(2) ≈ 1.193651
n = 400   h = 0.005   y(2) ≈ 1.193613
ardışık farklar: [-4.31e-04 -1.78e-04 -8.00e-05 -3.80e-05]
farkların oranı: [2.429 2.214 2.107]
```

Adım 4. Tam çözüm bilinmediğinden ardışık sonuçların farklarına bakıyoruz. Adım yarıya indikçe fark da kabaca yarıya iniyor; farkların oranı 2'ye yaklaşıyor. Hata adımla orantılıysa, yani $y_h \approx y(2) + Ch$ ise

$$y_h - y_{h/2} \approx Ch - C\,\frac{h}{2} = C\,\frac{h}{2} \approx y_{h/2} - y(2)$$

olur: son farkın büyüklüğü, son sonucun hatası için iyi bir kestirimdir. $h = 0{,}005$ ile bulunan $1{,}193613$ değerinin hatası bu yüzden yaklaşık $3{,}8 \cdot 10^{-5}$'tir ve güvenebileceğimiz kısım $1{,}1936$'dır. Hatta kestirilen hatayı sonuçtan çıkarırsak çok daha iyi bir değer elde ederiz: $1{,}193613 - 0{,}000038 = 1{,}193575$. Bu sayıları aşağıda `solve_ivp` ile çok daha doğru bir hesapla karşılaştıracağız.

■

Euler yönteminin hatasının adımla orantılı olduğunu tam çözümü bilinen bir problemde kesin olarak görebiliriz.

Örnek 13.2 (Euler Yöntemi ve e Sayısı). $y' = y$, $y(0) = 1$ problemine $[0, 1]$ aralığında n adımlık Euler yöntemi uygulandığında $y_n = \left(1 + \frac{1}{n}\right)^n$ olduğunu gösteriniz ve bunu $n = 10, 100, 1000, 10000$ için sayısal olarak doğrulayınız.

Çözüm.
Formül. $h = 1/n$ ve $f(t, y) = y$ için Euler adımı

$$y_{i+1} = y_i + \frac{1}{n}\,y_i = \left(1 + \frac{1}{n}\right) y_i$$

olur. $y_0 = 1$ olduğundan tümevarımla $y_i = \left(1 + \frac{1}{n}\right)^i$, özel olarak $y_n = \left(1 + \frac{1}{n}\right)^n$ bulunur. Bu dizinin limiti e'dir (bkz. Analiz 1); yani Euler yöntemi $h \to 0$ iken doğru değere yakınsar.
Sayısal doğrulama. Yine `euler` fonksiyonunu kullanıyoruz; kod önceki kodların devamıdır:

```
print("     n   Euler y(1)    (1 + 1/n)^n   n·(e - y(1))")
for n in [10, 100, 1000, 10000]:
    t, y = euler(lambda t, y: y, 0.0, 1.0, 1 / n, n)
    print(f"{n:6d}  {y[-1]:.10f}  {(1 + 1 / n)**n:.10f}  "
          f"{n * (np.e - y[-1]):.6f}")
print("e / 2 =", np.e / 2)
```

Çıktı:

```
     n   Euler y(1)    (1 + 1/n)^n   n·(e - y(1))
    10  2.5937424601  2.5937424601  1.245394
   100  2.7048138294  2.7048138294  1.346800
  1000  2.7169239322  2.7169239322  1.357896
 10000  2.7181459268  2.7181459268  1.359016
e / 2 = 1.3591409142295225
```

İlk iki sütun aynıdır. Son sütundaki $n\,(e - y_n)$ çarpımı $e/2 \approx 1{,}3591$'e yaklaşıyor; yani hata yaklaşık $\frac{e}{2n} = \frac{e}{2}\,h$'dir ve adım uzunluğuyla orantılıdır. Aynı yaklaşıklığı Python ile İlk Adımlar bölümündeki örnekte de görmüştük.

■

 CC BY-NC-SA 4.0

13.3 Dördüncü Mertebe Runge–Kutta Yöntemi

Euler yönteminin zayıflığı, eğimi yalnız adımın başında okumasıdır; adım boyunca eğim değiştikçe hata birikir. Runge–Kutta yöntemleri eğimi adımın içinde birkaç noktada ölçer ve bu ölçümlerin ağırlıklı ortalamasıyla ilerler.

Tanım 13.3 (Dördüncü Mertebe Runge–Kutta Yöntemi). $y' = f(t,y)$, $y(t_0) = y_0$ problemi ve $h > 0$ verilsin. $t_i = t_0 + ih$ olmak üzere her adımda

$$\begin{aligned} k_1 &= f(t_i, y_i),\\ k_2 &= f\!\left(t_i + \frac{h}{2},\, y_i + \frac{h}{2}\,k_1\right),\\ k_3 &= f\!\left(t_i + \frac{h}{2},\, y_i + \frac{h}{2}\,k_2\right),\\ k_4 &= f(t_i + h,\, y_i + h\,k_3) \end{aligned}$$

eğimleri hesaplanır ve

$$y_{i+1} = y_i + \frac{h}{6}\left(\, k_1 + 2k_2 + 2k_3 + k_4 \,\right)$$

alınır. Bu kurala **dördüncü mertebe Runge–Kutta yöntemi** ya da kısaca **RK4** denir.

Yani RK4 eğimi dört kez ölçer: adımın başında (k_1), ortasında iki kez (k_2 ve k_3) ve sonunda (k_4). Ortadaki ve sondaki ölçümler için gereken y değerleri, bir önceki eğimle atılan kısa Euler adımlarıdır. Sonra bu eğimlerin $1, 2, 2, 1$ ağırlıklı ortalamasıyla tek bir adım atılır. Ağırlıkların kaynağı Simpson kuralıdır: f yalnız t'ye bağlıysa $k_2 = k_3$ olur ve adım

$$y_{i+1} - y_i = \frac{h}{6}\left(f(t_i) + 4f\!\left(t_i + \frac{h}{2}\right) + f(t_i + h)\right)$$

biçimini alır. Bu, $\int_{t_i}^{t_i+h} f(t)\,dt$ integrali için Simpson kuralıdır (bkz. Nümerik Analiz); kuralın Python'daki kullanımı Sayısal Türev ve İntegral bölümündedir.
Tek bir adımı elle izleyelim. $y' = y$, $y(0) = 1$ için $h = 1$ alırsak $f(t,y) = y$ olduğundan

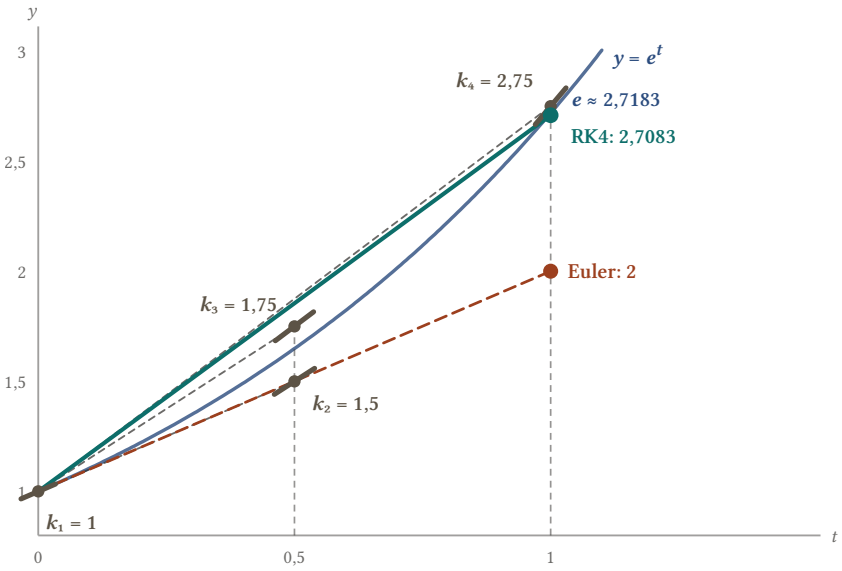
$$\begin{aligned} k_1 &= f(0;\,1) = 1,\\ k_2 &= f(0{,}5;\,1 + 0{,}5\cdot 1) = 1{,}5,\\ k_3 &= f(0{,}5;\,1 + 0{,}5\cdot 1{,}5) = 1{,}75,\\ k_4 &= f(1;\,1 + 1\cdot 1{,}75) = 2{,}75 \end{aligned}$$

ve

$$y_1 = 1 + \frac{1}{6}\,(1 + 3 + 3{,}5 + 2{,}75) = 1 + \frac{10{,}25}{6} = 2{,}708\overline{3}$$

bulunur. Tek adımda Euler yöntemi 2'ye varırken RK4, $e = 2{,}71828\ldots$ sayısına $0{,}01$'den daha yakın bir değer verir:

 CC BY-NC-SA 4.0



Şekil 13.3: $y' = y$, $y(0) = 1$ için tek bir RK4 adımı ($h = 1$). Eğim dört noktada ölçülür: başta ($k_1 = 1$), ortada iki kez ($k_2 = 1{,}5$ ve $k_3 = 1{,}75$) ve sonda ($k_4 = 2{,}75$); kesikli doğrular bu noktaların nasıl bulunduğunu gösterir. Ağırlıklı ortalama eğimle atılan adım 2,7083'e varır ve $e \approx 2{,}7183$'e çok yaklaşır; aynı adımı Euler yöntemi 2'de bitirir.

RK4'ü de bir fonksiyon olarak yazalım. Gövde Euler'inkiyle aynıdır; yalnız döngünün içi dört eğim hesaplar. Aynı problemi $h = 0{,}25$ ile çözelim:

```
import numpy as np


def rk4(f, t0, y0, h, n):
    """y' = f(t, y), y(t0) = y0 problemini n RK4 adımıyla çözer."""
    t = t0 + h * np.arange(n + 1)
    y = np.zeros(n + 1)
    y[0] = y0
    for i in range(n):
        k1 = f(t[i], y[i])
        k2 = f(t[i] + h / 2, y[i] + h / 2 * k1)
        k3 = f(t[i] + h / 2, y[i] + h / 2 * k2)
        k4 = f(t[i] + h, y[i] + h * k3)
        y[i + 1] = y[i] + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)
    return t, y


t, y = rk4(lambda t, y: y, 0.0, 1.0, 0.25, 4)
print("  t      y_n          e^t          hata")
for tk, yk in zip(t, y):
    print(f"{tk:.2f}  {yk:.8f}  {np.exp(tk):.8f}  {np.exp(tk) - yk:.2e}")
```

Çıktı:

```
  t      y_n          e^t          hata
0.00  1.00000000  1.00000000  0.00e+00
0.25  1.28401693  1.28402542  8.49e-06
0.50  1.64869947  1.64872127  2.18e-05
0.75  2.11695803  2.11700002  4.20e-05
1.00  2.71820994  2.71828183  7.19e-05
```

 CC BY-NC-SA 4.0

Aynı dört adımda Euler'in 0,277 olan hatası RK4'te $7{,}19 \cdot 10^{-5}$'e iner; yaklaşık 3850 kat küçüktür. RK4 adım başına f'yi dört kez hesapladığından adil bir karşılaştırma için Euler'e dört kat, yani 16 adım vermek gerekir. O zaman bile Euler'in hatası $e - \left(\frac{17}{16}\right)^{16} \approx 0{,}080$ olur.
Bu farkın nedenini görmek için tek bir RK4 adımını SymPy ile Sembolik Hesap bölümündeki gibi sembolik olarak açalım:

```
import sympy as sp

h, y = sp.symbols("h y")


def f(t, y):
    return y


k1 = f(0, y)
k2 = f(h / 2, y + h / 2 * k1)
k3 = f(h / 2, y + h / 2 * k2)
k4 = f(h, y + h * k3)
step = sp.expand(y + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4))
print(sp.collect(step, y))
print(sp.series(sp.exp(h), h, 0, 6))
```

Çıktı:

```
y*(h**4/24 + h**3/6 + h**2/2 + h + 1)
1 + h + h**2/2 + h**3/6 + h**4/24 + h**5/120 + O(h**6)
```

$y' = y$ denkleminde bir RK4 adımı y_i'yi $1 + h + \frac{h^2}{2} + \frac{h^3}{6} + \frac{h^4}{24}$ ile çarpar. Tam çözüm ise bir adımda e^h ile çarpılır; RK4'ün çarpanı tam olarak e^h'nin dördüncü dereceden Taylor polinomudur. Adım başına hata bu yüzden $\frac{h^5}{120}$ mertebesindedir. Euler'in çarpanı $1 + h$ ise yalnız birinci dereceden Taylor polinomudur ve adım başına $\frac{h^2}{2}$ mertebesinde hata yapar.

13.4 Adım Uzunluğu ve Hata

İki yöntemi tek bir adım uzunluğunda karşılaştırmak yetmez; adım küçüldükçe hatanın nasıl küçüldüğünü ölçmek gerekir. Bundan sonra iki fonksiyonu sık kullanacağımız için önce onları bir modülde toplayalım. Sınıflar, Hata Yakalama ve Dosyalar bölümünde gördüğümüz gibi aşağıdaki kodu `odetools.py` adıyla kaydedelim. Fonksiyonlarda küçük bir değişiklik var: `np.zeros(n + 1)` yerine `np.zeros((n + 1,) + np.shape(y0))` yazdık. `y0` bir sayıysa `np.shape(y0)` boş demettir ve hiçbir şey değişmez; `y0` m elemanlı bir diziyse `y` tablosu $(n+1) \times m$ boyutlu olur. Böylece aynı fonksiyonlar ileride denklem sistemlerinde de çalışacak.

```
"""Başlangıç değer problemleri için Euler ve RK4 yöntemleri."""
import numpy as np


def euler(f, t0, y0, h, n):
    """y' = f(t, y), y(t0) = y0 problemini n Euler adımıyla çözer."""
    t = t0 + h * np.arange(n + 1)
    y = np.zeros((n + 1,) + np.shape(y0))
    y[0] = y0
    for i in range(n):
        y[i + 1] = y[i] + h * f(t[i], y[i])
    return t, y


def rk4(f, t0, y0, h, n):
    """y' = f(t, y), y(t0) = y0 problemini n RK4 adımıyla çözer."""
    t = t0 + h * np.arange(n + 1)
    y = np.zeros((n + 1,) + np.shape(y0))
```

 CC BY-NC-SA 4.0

```
    y[0] = y0
    for i in range(n):
        k1 = f(t[i], y[i])
        k2 = f(t[i] + h / 2, y[i] + h / 2 * k1)
        k3 = f(t[i] + h / 2, y[i] + h / 2 * k2)
        k4 = f(t[i] + h, y[i] + h * k3)
        y[i + 1] = y[i] + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)
    return t, y


if __name__ == "__main__":
    # sınama: y' = y, y(0) = 1 probleminde y(1) = e olmalı
    for method in (euler, rk4):
        t, y = method(lambda t, y: y, 0.0, 1.0, 0.25, 4)
        print(f"{method.__name__:5s}  y(1) ≈ {y[-1]:.8f}"
              f"   hata = {np.e - y[-1]:.2e}")
```

Dosya doğrudan çalıştırıldığında (`python odetools.py`) sınama bloğu şunu basar:
Çıktı:

```
euler  y(1) ≈ 2.44140625   hata = 2.77e-01
rk4    y(1) ≈ 2.71820994   hata = 7.19e-05
```

Bundan sonra bu fonksiyonları kullanan kodlar `from odetools import euler, rk4` satırını içerir; `odetools.py` dosyası çalıştırılan programla aynı klasörde olmalıdır. Şimdi hatayı adım uzunluğuyla ilişkilendiren kavramları tanımlayalım.

Tanım 13.4 (Global Hata ve Yöntemin Mertebesi). Bir yöntem $[t_0, T]$ aralığında n adımla, yani $h = (T - t_0)/n$ adım uzunluğuyla uygulansın. $E(h) = \mid y(T) - y_n \mid$ farkına yöntemin T'deki **global hatası** denir. f yeterince düzgün olduğunda h'den bağımsız bir C sabiti ve yeterince küçük her h için $E(h) \le Ch^p$ oluyorsa, yönteme ***p*-inci mertebeden** denir.

Yani mertebe, adımı küçültmenin ne kadar kazandırdığını söyler: p-inci mertebeden bir yöntemde adımı yarıya indirmek hatayı kabaca 2^p'ye böler. Global hata, her adımda yapılan küçük hataların birikmiş sonucudur. Adım başına hata h^{p+1} mertebesindeyse ve $(T - t_0)/h$ adım atılıyorsa, toplam hata h^p mertebesinde kalır. Euler'de adım başına hata h^2, RK4'te h^5 mertebesinde olduğundan şu sonuç beklenir:

Teorem 13.1 (Euler ve RK4 Yöntemlerinin Mertebesi). *f fonksiyonu, çözüm eğrisini içeren bir bölgede dördüncü mertebeye kadar sürekli kısmi türevlere sahip olsun. Bu durumda $[t_0, T]$ aralığında Euler yöntemi birinci mertebeden, RK4 dördüncü mertebedendir: bir C sabiti ve yeterince küçük her h için sırasıyla $E(h) \le Ch$ ve $E(h) \le Ch^4$ olur.*

İspat, adım başına yapılan hatayı Taylor teoremiyle kestirmeye ve bu hataların sonraki adımlarda en fazla ne kadar büyüyebileceğini sınırlamaya dayanır; Euler'in adım başına hatasının $\frac{h^2}{2}\,y''(\xi)$ olduğunu yukarıda gördük. Biz teoremi sayısal olarak doğrulayacağız. Bunun için tam çözümü bilinen, doğrusal olmayan bir problem seçelim: lojistik denklem $y' = y(1 - y)$, $y(0) = 0{,}1$. Bu ayrılabilen denklemin çözümü (bkz. Diferansiyel Denklemler)

$$y(t) = \frac{1}{1 + 9e^{-t}}$$

dir. $T = 4$'teki hatayı $n = 8, 16, \ldots, 1024$ adımla hesaplayalım:

```
import numpy as np

from odetools import euler, rk4


def f(t, y):
    return y * (1 - y)
```

 CC BY-NC-SA 4.0

```
def exact(t):
    return 1 / (1 + 9 * np.exp(-t))


T = 4.0
ns = 2 ** np.arange(3, 11)            # 8, 16, ..., 1024
hs = T / ns
err_e = np.array([abs(euler(f, 0, 0.1, T / n, n)[1][-1] - exact(T))
                  for n in ns])
err_r = np.array([abs(rk4(f, 0, 0.1, T / n, n)[1][-1] - exact(T))
                  for n in ns])
print("    n        h    Euler hata   oran    RK4 hata   oran")
for k in range(len(ns)):
    q_e = f"{err_e[k - 1] / err_e[k]:5.2f}" if k else "    -"
    q_r = f"{err_r[k - 1] / err_r[k]:5.2f}" if k else "    -"
    print(f"{ns[k]:5d}  {hs[k]:.5f}  {err_e[k]:.3e}  {q_e}"
          f"   {err_r[k]:.3e}  {q_r}")
```

Çıktı:

```
    n        h    Euler hata   oran    RK4 hata   oran
    8  0.50000  1.349e-02      -   5.828e-05      -
   16  0.25000  5.489e-03   2.46   3.730e-06  15.62
   32  0.12500  2.496e-03   2.20   2.365e-07  15.77
   64  0.06250  1.192e-03   2.09   1.490e-08  15.88
  128  0.03125  5.825e-04   2.05   9.348e-10  15.94
  256  0.01562  2.880e-04   2.02   5.854e-11  15.97
  512  0.00781  1.432e-04   2.01   3.663e-12  15.98
 1024  0.00391  7.138e-05   2.01   2.286e-13  16.03
```

"oran" sütunu, bir önceki satırın hatasının bu satırın hatasına bölümüdür. Euler'de oran 2'ye, RK4'te $16 = 2^4$'e yaklaşıyor; teoremin söylediği tam olarak budur. $n = 1024$'te RK4'ün hatası $2{,}3 \cdot 10^{-13}$'tür. Adım daha da küçültülürse hata, kayan noktalı sayılarla hesap yapmanın getirdiği yuvarlama hatası düzeyine iner ve oradan aşağı inmez (bkz. Nümerik Analiz).

Mertebe en iyi log-log grafikte görünür. $E \approx Ch^p$ ise $\log E \approx \log C + p \log h$ olur ve noktalar eğimi p olan bir doğru üzerinde dizilir. Eğimi en küçük kareler doğrusuyla (`np.polyfit`, bkz. Polinomlar, İnterpolasyon ve Eğri Uydurma) ölçelim ve grafiği `loglog` ile çizelim. Kod bir önceki kodun devamıdır:

```
import matplotlib.pyplot as plt

p_e = np.polyfit(np.log(hs), np.log(err_e), 1)[0]
p_r = np.polyfit(np.log(hs), np.log(err_r), 1)[0]
print(f"Euler doğrusunun eğimi: {p_e:.3f}")
print(f"RK4 doğrusunun eğimi  : {p_r:.3f}")

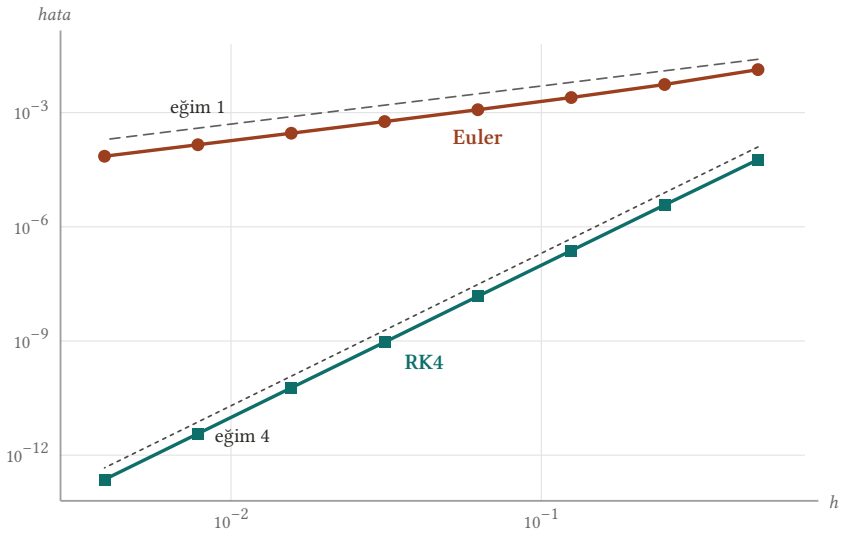
fig, ax = plt.subplots(figsize=(6, 4.5))
ax.loglog(hs, err_e, "o-", label="Euler")
ax.loglog(hs, err_r, "s-", label="RK4")
ax.loglog(hs, 0.05 * hs, "k--", lw=0.8, label="eğim 1")
ax.loglog(hs, 0.002 * hs**4, "k:", lw=0.8, label="eğim 4")
ax.set_xlabel("adım uzunluğu h")
ax.set_ylabel("t = 4'teki hata")
ax.legend()
fig.savefig("hata-adim.png", dpi=150)
```

Çıktı:

```
Euler doğrusunun eğimi: 1.067
RK4 doğrusunun eğimi  : 3.990
```

 CC BY-NC-SA 4.0

Kod grafiği `hata-adim.png` dosyasına kaydeder; aşağıdaki şekil aynı grafiğin aynı verilerle çizilmiş hâlidir:



Şekil 13.4: $y' = y(1-y)$, $y(0) = 0{,}1$ probleminde $t = 4$'teki hatanın adım uzunluğuna göre değişimi; Matplotlib kodunun çizdiği log-log grafiğin aynı verilerle çizilmiş hâli ($h = 4/n$, $n = 8, 16, \ldots, 1024$). Euler'in noktaları eğimi 1, RK4'ünkiler eğimi 4 olan kesikli ve noktalı doğrulara paraleldir: adımı yarıya indirmek Euler'in hatasını yarıya, RK4'ün hatasını on altıda bire indirir.

RK4'ün eğimi 3,99 ile tam beklendiği gibi. Euler'in eğimi 1'den biraz büyük çıktı, çünkü büyük adımlarda hatanın h^2'li terimi de hissediliyor; tablodaki oranların küçük adımlarda 2'ye oturması bunu doğruluyor.

13.5 SciPy ile Çözüm: solve_ivp

Kendi yazdığımız fonksiyonlar yöntemleri anlamak için idealdir, ama gerçek işte iki eksikleri var: adım uzunluğunu bizim seçmemiz gerekir ve sonucun ne kadar doğru olduğunu söylemezler. SciPy'nin `scipy.integrate.solve_ivp` fonksiyonu ikisini de kendisi halleder.

Tanım 13.5 (Uyarlamalı Adım Kontrolü). Her adımda o adımın hatasını kestirip adım uzunluğunu bu kestirime göre ayarlayan yöntemlere **uyarlamalı adımlı** (adaptive step size) yöntemler denir. Kestirilen hata istenen hata payının altındaysa adım kabul edilir ve sonraki adım büyütülebilir; üstündeyse adım küçültülerek yeniden atılır.

Yani yöntem, çözümün yavaş değiştiği yerlerde uzun, hızlı değiştiği yerlerde kısa adımlar atar. `solve_ivp`'nin varsayılan yöntemi `RK45`, her adımda aynı f değerlerinden hem dördüncü hem beşinci mertebeden bir sonuç üretir; ikisinin farkı adımın hatasının kestirimidir. Hata payı `rtol` (bağıl) ve `atol` (mutlak) parametreleriyle verilir: her bileşende adım başına hata kabaca `atol + rtol * abs(y)` değerinin altında tutulur. Varsayılan değerler `rtol=1e-3` ve `atol=1e-6`'dır.

solve_ivp ile bir başlangıç değer problemini dört adımda çözmek

1. Sağ tarafı `fun(t, y)` biçiminde yazın: ilk argüman zaman, ikincisi bilinmeyenlerin dizisidir. Fonksiyon y' değerlerini bir liste ya da dizi olarak döndürür.
2. Aralığı `t_span=(t0, T)`, başlangıç değerini `y0=[...]` listesi olarak hazırlayın; tek bir denklemde bile `y0` tek elemanlı bir listedir.
3. `sol = solve_ivp(fun, t_span, y0, rtol=..., atol=...)` çağrısını yapın. Belirli anlardaki değerler gerekiyorsa `t_eval` ya da `dense_output=True` ekleyin.
4. `sol.success` değerini denetleyin. Sonuç `sol.t` (zamanlar) ve `sol.y` (her satırı bir bilinmeyen) dizilerindedir.

 CC BY-NC-SA 4.0

Örnek 13.3 (SciPy ile Tanjant Fonksiyonu). $y' = 1 + y^2$, $y(0) = 0$ problemini $[0; 1{,}5]$ aralığında `solve_ivp` ile varsayılan ayarlarla çözünüz ve sonucu tam çözüm $y = \tan t$ ile karşılaştırınız.

Çözüm.
Adım 1 ve 2. Tam çözümün $\tan t$ olduğunu Diferansiyel Denklemler notlarında görmüştük (bkz. Diferansiyel Denklemler); çözüm $t \to \pi/2 \approx 1{,}5708$ iken sonsuza gider. Sağ taraf `f(t, y)` fonksiyonudur, `t_span=(0, 1.5)` ve `y0=[0.0]`'dır.
Adım 3. Hata paylarını vermeden çağırıyoruz:

```
import numpy as np
from scipy.integrate import solve_ivp


def f(t, y):
    return 1 + y**2


sol = solve_ivp(f, (0, 1.5), [0.0])
print(sol.success, sol.message)
print("y'nin şekli:", sol.y.shape, "  f çağrısı:", sol.nfev)
for tk, yk in zip(sol.t, sol.y[0]):
    print(f"t = {tk:.6f}   y = {yk:10.6f}   tan t = {np.tan(tk):10.6f}")
```

Çıktı:

```
True The solver successfully reached the end of the integration interval.
y'nin şekli: (1, 10)   f çağrısı: 74
t = 0.000000   y =   0.000000   tan t =   0.000000
t = 0.000100   y =   0.000100   tan t =   0.000100
t = 0.001100   y =   0.001100   tan t =   0.001100
t = 0.011100   y =   0.011100   tan t =   0.011100
t = 0.111100   y =   0.111559   tan t =   0.111559
t = 0.822584   y =   1.077292   tan t =   1.077281
t = 1.083655   y =   1.887806   tan t =   1.887783
t = 1.344726   y =   4.348962   tan t =   4.347783
t = 1.447468   y =   8.071618   tan t =   8.067294
t = 1.500000   y =  14.114791   tan t =  14.101420
```

Adım 4. `sol.success` değeri `True`'dur. `sol.y`'nin şekli $(1, 10)$'dur: tek bilinmeyen ve 10 zaman noktası, yani 9 adım. İlk adımlar 0,0001, 0,001, 0,01 ve 0,1 uzunluğundadır; yöntem önce ölçeği yoklar, sonra adımı büyütür. Sonuç ise beklenenden kaba: $t = 1{,}5$'te bulunan $14{,}114791$ değeri $\tan 1{,}5 = 14{,}101420$'den $0{,}013$ kadar sapıyor. Bağıl hata $0{,}013/14{,}1 \approx 10^{-3}$'tür, yani varsayılan `rtol=1e-3` kadardır.
■

⚠ Varsayılan hata payları ve argüman sırası

`solve_ivp`'nin varsayılan `rtol=1e-3` değeri yalnız üç basamaklık bir doğruluk hedefler; sonucun kaç basamağına ihtiyacınız varsa `rtol` ve `atol`'u açıkça verin. Argüman sırasına da dikkat edin: `solve_ivp` sağ tarafı `fun(t, y)` biçiminde ister. Eski `scipy.integrate.odeint` fonksiyonu ise `fun(y, t)` sırasını kullanır; `odeint` için yazılmış bir fonksiyon `solve_ivp`'ye verilirse t ile y yer değiştirir ve sonuç sessizce yanlış çıkar.

Hata payını sıkılaştırınca ne olduğunu görelim. Aynı problemi varsayılan `RK45` ve sekizinci mertebeden `DOP853` yöntemleriyle, `atol=1e-12` sabit tutup farklı `rtol` değerleriyle çözüyoruz:

 CC BY-NC-SA 4.0

```
import numpy as np
from scipy.integrate import solve_ivp

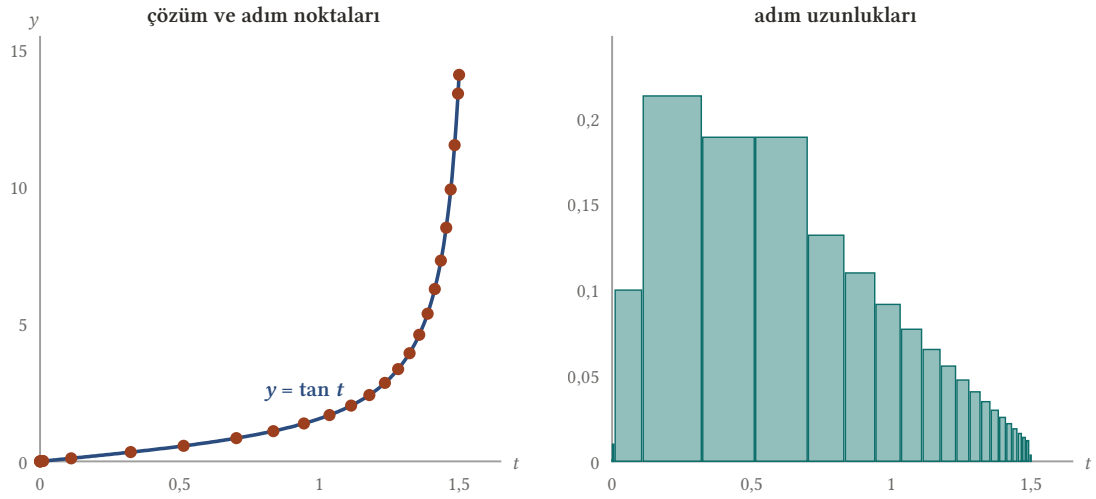

def f(t, y):
    return 1 + y**2


print("yöntem   rtol    adım  f çağrısı   y(1.5) hatası")
for method in ["RK45", "DOP853"]:
    for rtol in [1e-3, 1e-6, 1e-9, 1e-12]:
        sol = solve_ivp(f, (0, 1.5), [0.0], method=method,
                        rtol=rtol, atol=1e-12)
        err = abs(sol.y[0, -1] - np.tan(1.5))
        print(f"{method:6s}  {rtol:.0e}  {sol.t.size - 1:4d}"
              f"  {sol.nfev:9d}   {err:.2e}")
```

Çıktı:

```
yöntem   rtol    adım  f çağrısı   y(1.5) hatası
RK45    1e-03     9         74   1.33e-02
RK45    1e-06    24        248   6.20e-05
RK45    1e-09    80        494   7.14e-08
RK45    1e-12   289       1748   5.75e-11
DOP853  1e-03     8        134   3.36e-03
DOP853  1e-06    12        218   2.55e-06
DOP853  1e-09    24        506   4.74e-09
DOP853  1e-12    46        566   3.03e-12
```

`rtol` bin kat küçüldükçe hata da kabaca 200 ila 1600 kat küçülüyor; bedeli daha çok adım ve daha çok f çağrısıdır. Yüksek doğrulukta sekizinci mertebeden `DOP853` çok daha ekonomiktir: `rtol=1e-12` için `RK45` 1748, `DOP853` ise 566 çağrıyla ve daha küçük hatayla bitirir. `RK45`'in `rtol=1e-6` ile attığı 24 adım şöyle dağılıyor:



Şekil 13.5: `solve_ivp`'nin $y' = 1 + y^2$, $y(0) = 0$ problemini `rtol=1e-6` ile çözerken attığı 24 adım. Solda adım noktaları $y = \tan t$ eğrisi üzerinde, sağda her adımın uzunluğu (sütunun genişliği adımın kapladığı aralık, yüksekliği adım uzunluğu). İlk dört adım, yöntemin ölçeği yoklarken attığı 0,0001, 0,001, 0,01 ve 0,1 uzunluğundaki adımlardır; sonra adımlar en fazla 0,21'e çıkar ve çözüm dikleştikçe küçülür.

Çözüm $t = \pi/2$'deki dikey asimptotuna yaklaştıkça dikleşiyor; yöntem de adımlarını kısaltarak buna uyuyor. Sabit adımlı bir yöntem aynı doğruluğa ulaşmak için çözümün en dik yerinde gereken küçük

 CC BY-NC-SA 4.0

adımı baştan sona kullanmak zorunda kalırdı. Sıkı bir hata payıyla `solve_ivp`, tam çözümü bilinmeyen problemlerde bir referans değer üretmek için de kullanılır.

Örnek 13.4 (Euler Kestirimini Doğrulamak). $y' = t - y^2, y(0) = 0$ probleminde $y(2)$'yi `solve_ivp` ile sıkı bir hata payıyla hesaplayınız ve daha önce bulduğumuz Euler sonuçlarının (bkz. Örnek 13.1) gerçek hatalarını bulunuz.

Çözüm.
`rtol=1e-12` ve `atol=1e-14` ile bulunan değeri doğru kabul edip Euler sonuçlarından çıkarıyoruz:

```
from scipy.integrate import solve_ivp

from odetools import euler


def f(t, y):
    return t - y**2


ref = solve_ivp(f, (0, 2), [0.0], rtol=1e-12, atol=1e-14).y[0, -1]
print(f"solve_ivp: y(2) ≈ {ref:.10f}")
for n in [100, 200, 400]:
    t, y = euler(f, 0.0, 0.0, 2 / n, n)
    print(f"Euler, n = {n}:  hata = {y[-1] - ref:.2e}")
```

Çıktı:

```
solve_ivp: y(2) ≈ 1.1935759753
Euler, n = 100:  hata = 1.55e-04
Euler, n = 200:  hata = 7.48e-05
Euler, n = 400:  hata = 3.67e-05
```

Referans değer $y(2) \approx 1{,}1935759753$'tür. $n = 400$ için gerçek hata $3{,}67 \cdot 10^{-5}$'tir; ardışık farklardan kestirdiğimiz $3{,}8 \cdot 10^{-5}$ buna çok yakındır. Kestirilen hatayı çıkararak bulduğumuz $1{,}193575$ değeri ise referanstan yalnızca yaklaşık 10^{-6} kadar farklıdır. Hatalar her satırda yarıya iniyor: Euler yöntemi birinci mertebedendir.
■

Çözümün adım noktaları dışındaki değerleri de çoğu zaman gerekir. `t_eval=[...]` parametresi sonucu istenen anlarda verir. `dense_output=True` ise `sol.sol` adında, her t'de çağrılabilen bir fonksiyon döndürür; bu fonksiyon her adımın içinde yöntemin kendi ara değer polinomunu kullanır. Bir de `events` parametresi var.

Tanım 13.6 (Olay Fonksiyonu). `solve_ivp`'ye `events` parametresiyle verilen ve `g(t, y)` biçiminde çağrılan fonksiyona **olay fonksiyonu** (event function) denir. Çözüm boyunca $g\left(t, y(t)\right)$ işaret değiştirdiğinde `solve_ivp` sıfır yerini bulup `sol.t_events` listesine yazar. Fonksiyonun `terminal` niteliği `True` ise çözüm ilk olayda durdurulur; `direction` niteliği $+1$ ya da -1 ise yalnız artarak ya da azalarak sıfırdan geçişler sayılır.

Yani olay fonksiyonu "çözüm şu koşulu ne zaman sağlıyor?" sorusunu yanıtlar: bir topun yere düştüğü an, bir sarkacın en alttan geçtiği an ya da bir nüfusun belli bir sayıya ulaştığı an. Sıfır yeri, adımın içindeki ara değer polinomu üzerinde Kök Bulma ve Optimizasyon bölümündekilere benzer bir kök bulma yöntemiyle bulunur. `events` parametresine birden çok olay fonksiyonu da verilebildiği için `sol.t_events`, her olay fonksiyonu için bir dizi içeren bir listedir. Tek olay fonksiyonunda olay anları `sol.t_events[0]` dizisindedir ve ilk olay anı `sol.t_events[0][0]`'dır. Olay anlarındaki çözüm vektörleri de aynı düzende `sol.y_events` listesinde durur: `sol.y_events[0]` dizisinin her satırı, bir olay anındaki çözüm vektörüdür. Tanjant probleminde çözümün 10'a ulaştığı anı bulalım ve ara noktalardaki değerleri yoğun çıktıdan okuyalım:

 CC BY-NC-SA 4.0

```
import numpy as np
from scipy.integrate import solve_ivp


def f(t, y):
    return 1 + y**2


def reach10(t, y):
    return y[0] - 10          # y = 10 olduğunda sıfır


reach10.terminal = True       # olay olunca çözümü durdur

sol = solve_ivp(f, (0, 1.5), [0.0], rtol=1e-10, atol=1e-12,
                dense_output=True, events=reach10)
print(f"olay anı : {sol.t_events[0][0]:.12f}")
print(f"arctan 10: {np.arctan(10):.12f}")
print(f"son nokta: t = {sol.t[-1]:.12f}, y = {sol.y[0, -1]:.12f}")
t = np.array([0.25, 0.5, 0.75, 1.0, 1.25])
err = sol.sol(t)[0] - np.tan(t)
print(f"ara noktalardaki en büyük hata: {np.max(np.abs(err)):.1e}")
```

Çıktı:

```
olay anı : 1.471127674330
arctan 10: 1.471127674304
son nokta: t = 1.471127674330, y = 10.000000000000
ara noktalardaki en büyük hata: 2.9e-10
```

$\tan t = 10$ denkleminin çözümü $\arctan 10$'dur ve olay anı onunla on ondalık basamakta uyuşuyor. `terminal=True` olduğundan çözüm $t = 1{,}5$'e gitmeden olay anında durdu. Ara noktalardaki en büyük hata $2{,}9 \cdot 10^{-10}$'dur: yoğun çıktı, istenen hata payına uygun doğruluktadır.

Örnek 13.5 (Sonlu Zamanda Patlayan Bir Çözüm). $y' = y^2$, $y(0) = 1$ problemini $[0, 2]$ aralığında `solve_ivp` ile çözmeye çalışınız ve ne olduğunu tam çözümle açıklayınız.

Çözüm.
Tam çözüm. Denklem ayrılabilir: $y^{-2}\,dy = dt$ integrallenince $-1/y = t + c$ bulunur ve $y(0) = 1$ koşulu $c = -1$ verir. Çözüm

$$y(t) = \frac{1}{1 - t}$$

dir ve $t \to 1^-$ iken sonsuza gider; $t = 1$'in ötesine uzatılamaz.
Sayısal çözüm.

```
from scipy.integrate import solve_ivp


def f(t, y):
    return y**2


sol = solve_ivp(f, (0, 2), [1.0])
print(sol.status, sol.success)
print(sol.message)
print("son t:", sol.t[-1], "  son y:", sol.y[0, -1])
```

Çıktı:

 CC BY-NC-SA 4.0

```
-1 False
Required step size is less than spacing between numbers.
son t: 0.9999286400563746   son y: 480848617754195.6
```

`status` değeri -1 ve `success` değeri `False`'tur: çözüm başarısız oldu. Yöntem $t = 1$'e yaklaştıkça adımı küçülttü ve gereken adım, t'nin yakınındaki iki ardışık kayan noktalı sayı arasındaki aralıktan bile küçük olunca durdu. Son y değeri güvenilir değildir; o anda $\frac{1}{1-t} \approx 14\,000$ olmalıydı. Bu çıktıdan öğrendiğimiz asıl şey, çözümün $t = 1$ civarında patladığıdır.
■

> ⚠ solve_ivp başarısız olunca hata vermez
>
> `solve_ivp` çözemediği bir problemde istisna fırlatmaz; o ana kadar hesapladığı değerleri döndürür ve durumu `sol.status`, `sol.success` ve `sol.message` alanlarına yazar. Sonucu kullanmadan önce bu alanları mutlaka denetleyin.

13.6 Katı Denklemler

Bazı denklemlerde adım uzunluğunu doğruluk değil kararlılık belirler. $\lambda > 0$ büyük bir sayı olmak üzere

$$y' = -\lambda\,(y - \cos t), \qquad y(0) = 0$$

problemini düşünelim. Bu birinci mertebeden doğrusal denklemin çözümü, integrasyon çarpanıyla (bkz. Diferansiyel Denklemler)

$$y(t) = \frac{\lambda^2}{\lambda^2 + 1}\left(\cos t + \frac{\sin t}{\lambda} - e^{-\lambda t}\right)$$

bulunur. $e^{-\lambda t}$ terimi çok hızlı söner ve çözüm kısa sürede $\cos t$'ye yapışır; sonrasında çözüm yavaş ve düzgündür. Euler yöntemini $\lambda = 50$ için $[0, 3]$ aralığında farklı adımlarla çalıştıralım:

```
import numpy as np

from odetools import euler

lam = 50.0


def f(t, y):
    return -lam * (y - np.cos(t))


def exact(t):
    a = lam**2 / (lam**2 + 1)
    return a * (np.cos(t) + np.sin(t) / lam - np.exp(-lam * t))


for h in [0.01, 0.03, 0.0375, 0.04, 0.05]:
    n = round(3 / h)
    t, y = euler(f, 0.0, 0.0, h, n)
    err = abs(y[-1] - exact(t[-1]))
    print(f"h = {h:6.4f}   h·λ = {h * lam:5.2f}   y(3) hatası = {err:.3e}")
```

Çıktı:

```
h = 0.0100   h·λ =  0.50   y(3) hatası = 9.839e-05
h = 0.0300   h·λ =  1.50   y(3) hatası = 2.956e-04
```

 CC BY-NC-SA 4.0

```
h = 0.0375   h·λ =  1.88   y(3) hatası = 3.926e-04
h = 0.0400   h·λ =  2.00   y(3) hatası = 9.996e-01
h = 0.0500   h·λ =  2.50   y(3) hatası = 3.677e+10
```

$h\lambda < 2$ olduğu sürece hata küçüktür ve kabaca h ile orantılıdır. $h\lambda = 2$'de hata 1 civarında takılır, $h\lambda > 2$'de ise sayılar patlar. Oysa tam çözüm $|\,y\,| \le 1$ aralığında kalan zararsız bir fonksiyondur. Nedeni şu önermedir:

Önerme 13.1 (Euler Yönteminin Kararlılık Koşulu). *$\lambda > 0$ olsun. $y' = -\lambda y$, $y(0) = y_0 \neq 0$ problemine h adımlı Euler yöntemi uygulandığında $y_i = (1 - h\lambda)^i\, y_0$ olur. Bu dizi ancak ve ancak $0 < h < 2/\lambda$ ise sıfıra yakınsar; $h > 2/\lambda$ ise $|\,y_i\,| \to \infty$ olur.*

> ***İspat.***
> *Euler adımı $y_{i+1} = y_i - h\lambda\, y_i$, yani $y_{i+1} = (1 - h\lambda)\, y_i$'dir; tümevarımla $y_i = (1 - h\lambda)^i\, y_0$ bulunur. Bir q^i geometrik dizisi ancak ve ancak $|\,q\,| < 1$ ise sıfıra gider ve $|\,q\,| > 1$ ise mutlak değerce sonsuza gider. $q = 1 - h\lambda$ için*
>
> $$|\,q\,| < 1 \Leftrightarrow -1 < 1 - h\lambda < 1 \Leftrightarrow 0 < h\lambda < 2$$
>
> *olur. $h > 2/\lambda$ ise $q < -1$, yani $|\,q\,| > 1$'dir.*
> ■

Tam çözüm $y_0 e^{-\lambda t}$ her $\lambda > 0$ için sönerken, Euler dizisi ancak $h < 2/\lambda$ ise söner. Yukarıdaki denklem de doğrusal olduğundan, iki Euler dizisinin farkı her adımda tam olarak $1 - h\lambda$ ile çarpılır. $h\lambda > 2$ iken bu çarpanın mutlak değeri 1'den büyüktür ve her adımda yapılan küçük hata katlanarak büyür; $\lambda = 50$ için sınır $h = 0{,}04$'tür. Euler, RK4 ve `RK45` gibi, yeni y_{i+1} değerini bilinen değerlerden doğrudan hesaplayan yöntemlere **açık** (explicit) yöntem denir.

Tanım 13.7 (Katı Denklem). Çözümünde çok hızlı sönen bir bileşenle yavaş değişen bir bileşenin bir arada bulunduğu ve bu yüzden açık yöntemlerin adım uzunluğunu doğruluğun değil kararlılığın sınırladığı denklemlere **katı** (stiff) denklem denir.

Yani katı bir denklemde hızlı bileşen kısa sürede söner ve sonrasında çözüm yavaş değişir; yine de açık yöntemler, sönmüş bileşen kararsızlaşmasın diye baştan sona küçük adım atmak zorundadır. Bu tür denklemler için **kapalı** (implicit) yöntemler geliştirilmiştir: her adımda yeni değeri bir denklemin çözümü olarak bulurlar ve büyük adımlarda da kararlı kalırlar. `solve_ivp`'de bunlar `Radau` ve `BDF` yöntemleridir; `LSODA` ise denklemin katı olup olmadığını kendisi sezip yöntem değiştirir. $\lambda = 1000$ için dört yöntemi karşılaştıralım:

```python
import numpy as np
from scipy.integrate import solve_ivp

lam = 1000.0


def f(t, y):
    return -lam * (y - np.cos(t))


def exact(t):
    a = lam**2 / (lam**2 + 1)
    return a * (np.cos(t) + np.sin(t) / lam - np.exp(-lam * t))


print("yöntem   adım  f çağrısı   y(10) hatası")
for method in ["RK45", "Radau", "BDF", "LSODA"]:
    sol = solve_ivp(f, (0, 10), [0.0], method=method,
                    rtol=1e-6, atol=1e-9)
```

 CC BY-NC-SA 4.0

```
    err = abs(sol.y[0, -1] - exact(10))
    print(f"{method:6s}  {sol.t.size - 1:5d}  {sol.nfev:9d}   {err:.1e}")
```

Çıktı:

```
yöntem   adım  f çağrısı   y(10) hatası
RK45     3587      21548   4.5e-07
Radau      81        624   2.1e-08
BDF       209        478   2.1e-09
LSODA     272        542   5.1e-09
```

Çözüm ilk birkaç binde birlik süreden sonra $\cos t$ kadar yavaş değişmesine karşın `RK45` 3587 adım atıyor; ortalama adım $10/3587 \approx 0{,}0028$'dir. Adımı doğruluk değil kararlılık sınırlıyor. `Radau` 81, `BDF` 209 adımla ve daha küçük hatayla bitiriyor. Kapalı yöntemler her adımda f'nin Jacobi matrisine (bkz. Kök Bulma ve Optimizasyon) ihtiyaç duyar; buradaki gibi tek bir denklemde bu matris yalnızca $\partial f/\partial y$ türevidir. `jac` parametresiyle verilmezse yöntemler onu sonlu farklarla kendileri hesaplar.

13.7 Denklem Sistemleri ve Yüksek Mertebeden Denklemler

Gerçek modellerin çoğunda birden çok bilinmeyen birbirini etkiler; bu yüzden tek bir denklem yerine bir denklem sistemi çözeriz.

Tanım 13.8 (Birinci Mertebeden Denklem Sistemi). $\mathbf{y}(t) = \big(y_1(t), \dots, y_m(t) \big)$ bilinmeyen bir vektör fonksiyonu ve $\mathbf{F} : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ verilmiş bir fonksiyon olsun.

$$\mathbf{y}' = \mathbf{F}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

problemine **birinci mertebeden bir denklem sistemi** için başlangıç değer problemi denir.

Yani tek denklemdeki her şey aynen geçerlidir; yalnız y bir sayı değil bir vektör, f de vektör değerli bir fonksiyondur. Euler ve RK4 formülleri hiç değişmeden vektörler için de anlamlıdır ve `solve_ivp` zaten vektörlerle çalışır. Yüksek mertebeden bir denklem de böyle bir sisteme çevrilir.

Yüksek mertebeden bir denklemi üç adımda sisteme çevirmek

1. $y^{(m)} = g\big(t, y, y', \dots, y^{(m-1)}\big)$ denkleminde yeni bilinmeyenler tanımlayın: $u_1 = y$, $u_2 = y'$, …, $u_m = y^{(m-1)}$.
2. Sistemi yazın: $u_1' = u_2$, …, $u_{m-1}' = u_m$ ve $u_m' = g(t, u_1, \dots, u_m)$.
3. Başlangıç koşullarını bir vektörde toplayın: $\mathbf{u}(t_0)$'ın bileşenleri $y(t_0), y'(t_0), \dots, y^{(m-1)}(t_0)$'dır.

Örnek 13.6 (Harmonik Salınıcı). $y'' + y = 0$, $y(0) = 1$, $y'(0) = 0$ problemini birinci mertebeden bir sisteme çevirip `solve_ivp` ile $[0, 2\pi]$ aralığında çözünüz; sonucu tam çözüm $y = \cos t$ ile karşılaştırınız.

Çözüm.
Adım 1. $u_1 = y$ ve $u_2 = y'$ diyelim.
Adım 2. $u_1' = y' = u_2$ ve $u_2' = y'' = -y = -u_1$ olur; yani $\mathbf{u}' = (u_2, -u_1)$.
Adım 3. $\mathbf{u}(0) = (1, 0)$'dır.
Kodda vektörü `u`, bileşenlerini okunaklı olsun diye `y, v` adlarıyla açıyoruz. Sonucu çeyrek periyotlarda istiyoruz:

```
import numpy as np
from scipy.integrate import solve_ivp


def oscillator(t, u):
    y, v = u                  # u = (y, y')
    return [v, -y]
```

 CC BY-NC-SA 4.0

```
t = np.pi / 2 * np.arange(5)          # 0, π/2, π, 3π/2, 2π
sol = solve_ivp(oscillator, (0, 2 * np.pi), [1.0, 0.0], t_eval=t,
                rtol=1e-10, atol=1e-12)
print("sol.y şekli:", sol.y.shape)
for tk, yk, vk in zip(sol.t, sol.y[0], sol.y[1]):
    print(f"t = {tk:.4f}   y = {yk:+.10f}   y' = {vk:+.10f}")
print("cos ile en büyük fark:", np.max(np.abs(sol.y[0] - np.cos(t))))
```

Çıktı:

```
sol.y şekli: (2, 5)
t = 0.0000   y = +1.0000000000   y' = +0.0000000000
t = 1.5708   y = -0.0000000000   y' = -1.0000000000
t = 3.1416   y = -1.0000000000   y' = +0.0000000000
t = 4.7124   y = +0.0000000000   y' = +0.9999999999
t = 6.2832   y = +0.9999999999   y' = -0.0000000000
cos ile en büyük fark: 7.59422524865272e-11
```

`sol.y` artık iki satırlıdır: ilk satır y, ikinci satır y' değerleridir. Değerler $\cos t$ ve $-\sin t$ ile 10^{-10} duyarlıkla uyuşuyor; çıktıdaki `-0.0000000000` gibi değerler, sıfıra çok yakın negatif sayılardır.
■

Kendi yazdığımız fonksiyonlar da sistemlerde çalışır; tek koşul, sağ tarafın bir NumPy dizisi döndürmesidir, çünkü bir liste `h / 2` gibi bir kayan noktalı sayıyla (`float`) çarpılamaz. Euler ve RK4'ü harmonik salınıcıda $h = 2\pi/40$ ile bir tam tur çalıştıralım ve faz düzleminde (y, y') noktasının orijine uzaklığını, $r = \sqrt{y^2 + y'^2}$'yi izleyelim. Tam çözümde $r = \sqrt{\cos^2 t + \sin^2 t} = 1$ sabittir:

```
import numpy as np

from odetools import euler, rk4


def oscillator(t, u):
    return np.array([u[1], -u[0]])    # NumPy dizisi döndürmeli


n = 40
h = 2 * np.pi / n                     # bir tam tur: t = 2π
for method in (euler, rk4):
    t, u = method(oscillator, 0.0, np.array([1.0, 0.0]), h, n)
    r = np.hypot(u[:, 0], u[:, 1])    # faz düzleminde orijine uzaklık
    print(f"{method.__name__:5s}  u şekli: {u.shape}   "
          f"y(2π) = {u[-1, 0]:.6f}   r(2π) = {r[-1]:.6f}")
print("(1 + h²)^(n/2) =", (1 + h**2) ** (n / 2))
```

Çıktı:

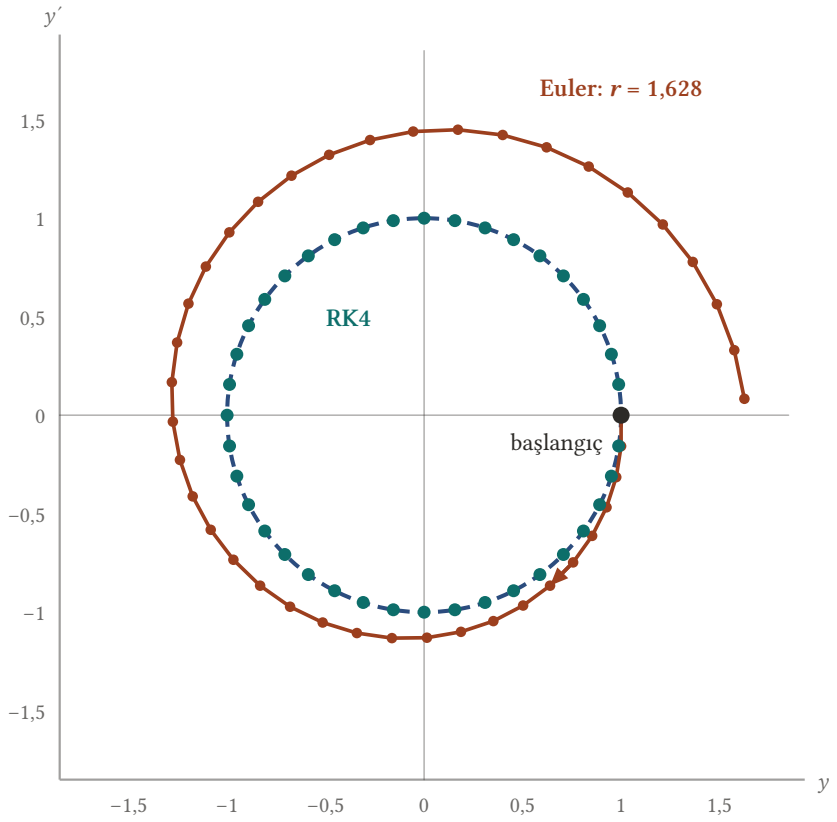
```
euler  u şekli: (41, 2)   y(2π) = 1.626114   r(2π) = 1.628225
rk4    u şekli: (41, 2)   y(2π) = 0.999996   r(2π) = 0.999996
(1 + h²)^(n/2) = 1.6282250240481193
```

RK4 tur sonunda $r = 0{,}999996$ ile neredeyse başladığı yere döner. Euler'in yarıçapı ise $1{,}628$'e çıkmıştır ve bu sayı tam olarak $(1 + h^2)^{n/2}$'dir. Nedeni şudur: $v = y'$ yazarsak Euler adımı

$$\begin{pmatrix} y_{i+1} \\ v_{i+1} \end{pmatrix} = \begin{pmatrix} 1 & h \\ -h & 1 \end{pmatrix} \begin{pmatrix} y_i \\ v_i \end{pmatrix}$$

matris çarpımıdır. Bu matris, $\sqrt{1 + h^2}$ sayısıyla bir dönme matrisinin çarpımıdır; her adım vektörü biraz döndürür ve boyunu $\sqrt{1 + h^2}$ kat uzatır. n adımda boy $(1 + h^2)^{n/2}$ katına çıkar:

 CC BY-NC-SA 4.0



Şekil 13.6: $y'' + y = 0$ sisteminin faz düzleminde $h = 2\pi/40$ ile bir tam tur. Tam çözüm kesikli birim çemberi saat yönünde dolaşır. RK4'ün 41 noktası çemberin üzerinde kalır; Euler'in noktaları her adımda orijinden $\sqrt{(1 + h^2)}$ kat uzaklaşır ve tur sonunda yarıçap 1,628 olur.

Euler yöntemi salınımlı sistemlerde enerjiyi yapay olarak büyütür; adım küçüldükçe bu etki azalır, ama her $h > 0$ için vardır ve uzun sürelerde birikir. RK4 gibi yüksek mertebeden açık yöntemlerde de enerji sürüklenir, yalnızca çok daha yavaş: yukarıda RK4'ün yarıçapı bir turda 0,999996'ya indi. Uzun süreli salınım hesaplarında bu yüzden ya hata payı sıkı tutulan yüksek mertebeden yöntemler ya da enerji hatası zamanla birikmeyecek biçimde kurulmuş simplektik (symplectic) yöntemler kullanılır.

13.8 Faz Düzlemi

İki bilinmeyenli bir sistemin çözümünü, zamanı gizleyip (y_1, y_2) noktasının düzlemde çizdiği eğri olarak da düşünebiliriz. Bu bakış özellikle sağ tarafı zamana bağlı olmayan sistemlerde verimlidir.

Tanım 13.9 (Otonom Sistem). Sağ tarafı t'ye açıkça bağlı olmayan $\mathbf{y}' = \mathbf{F}(\mathbf{y})$ sistemine **otonom** (autonomous) sistem denir.

Yani otonom bir sistemde hareket kuralı zamanla değişmez: aynı noktadan bugün de yarın da aynı biçimde çıkılır. Harmonik salınıcı, aşağıdaki Lotka–Volterra modeli ve sarkaç otonomdur.

Tanım 13.10 (Faz Düzlemi ve Yörünge). İki bilinmeyenli otonom bir $x' = F(x, y)$, $y' = G(x, y)$ sisteminin bir çözümü, t değiştikçe xy-düzleminde bir eğri çizer. Bu eğriye çözümün **yörüngesi** (trajectory), yörüngelerin çizildiği düzleme **faz düzlemi** (phase plane) denir. Tipik yörüngeleri bir arada gösteren resme de **faz portresi** (phase portrait) denir.

Yani faz düzlemi zamanı eksenlerden çıkarıp yalnız durumu gösterir. Harmonik salınıcının yörüngesi bir çemberdir: çemberin her noktası bir an, bir tam tur bir periyottur. F ve G'nin kısmi türevleri sürekli oldu-

 CC BY-NC-SA 4.0

ğunda varlık ve teklik teoremi sayesinde iki farklı yörünge kesişemez; kesişselerdi kesişim noktasından iki farklı çözüm geçerdi.

Tanım 13.11 (Denge Noktası). $\mathbf{F}(\mathbf{y}^*) = \mathbf{0}$ eşitliğini sağlayan $\mathbf{y}^*$ noktasına otonom $\mathbf{y}' = \mathbf{F}(\mathbf{y})$ sisteminin **denge noktası** (equilibrium point) denir.

Yani denge noktasından başlayan çözüm hiç kıpırdamaz: $\mathbf{y}(t) = \mathbf{y}^*$. Bir denge noktasının yakınındaki davranış hakkında $\mathbf{F}$'nin o noktadaki Jacobi matrisi J çok şey söyler (Jacobi matrisini Kök Bulma ve Optimizasyon bölümünde Newton yönteminde kullanmıştık). $\mathbf{y} = \mathbf{y}^* + \mathbf{u}$ yazıp $\mathbf{F}$'yi doğrusallaştırınca küçük $\mathbf{u}$ için $\mathbf{u}' \approx J\mathbf{u}$ olur ve bu doğrusal sistemin davranışını J'nin özdeğerleri belirler (bkz. Lineer Cebir). Özdeğerler $\pm i\omega$ biçiminde saf sanal sayılarsa doğrusal sistemin çözümleri $2\pi/\omega$ periyotla döner. Asıl sistemin denge noktası çevresindeki yörüngeleri kapalıysa, ki aşağıdaki iki örnekte öyledir, noktaya çok yakın yörüngelerin periyodu da $2\pi/\omega$'ya yaklaşır. Harmonik salınıcıda $J = \left(\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right)$'dir, özdeğerler $\pm i$ ve periyot 2π'dir.

13.9 Lotka–Volterra Av-Avcı Modeli

Faz düzleminin en ünlü örneklerinden biri, bir av türüyle onu yiyen bir avcı türün nüfuslarını birlikte modelleyen Lotka–Volterra denklemleridir.

Tanım 13.12 (Lotka–Volterra Modeli). a, b, c, d pozitif sabitler olmak üzere

$$x' = ax - bxy, \qquad y' = -cy + dxy$$

sistemine **Lotka–Volterra modeli** denir. Burada $x(t) > 0$ av, $y(t) > 0$ avcı nüfusudur.

Yani avcı yokken av üstel olarak çoğalır (ax terimi), av yokken avcı üstel olarak azalır ($-cy$ terimi); iki türün karşılaşmaları (xy terimleri) avı azaltıp avcıyı çoğaltır. $x' = x(a - by)$ ve $y' = y(dx - c)$ yazılışından, sistemin $x, y > 0$ bölgesindeki tek denge noktasının $\left(\frac{c}{d}, \frac{a}{b}\right)$ olduğu görülür. Bu bölümde $a = 1$, $b = 0{,}5$, $c = 0{,}75$, $d = 0{,}25$ alacağız; denge noktası $(3, 2)$'dir. Önce denge noktasındaki Jacobi matrisini

$$J = \begin{pmatrix} a - by^* & -bx^* \\ dy^* & -c + dx^* \end{pmatrix} = \begin{pmatrix} 0 & -1{,}5 \\ 0{,}5 & 0 \end{pmatrix}$$

yazıp özdeğerlerini NumPy ile Lineer Cebir bölümündeki gibi hesaplayalım:

```
import numpy as np

a, b, c, d = 1.0, 0.5, 0.75, 0.25
x_eq, y_eq = c / d, a / b                # denge noktası (3, 2)
J = np.array([[a - b * y_eq, -b * x_eq],
              [d * y_eq, -c + d * x_eq]])   # dengedeki Jacobi matrisi
print("denge noktası:", (x_eq, y_eq))
print(J)
print("özdeğerler   :", np.linalg.eigvals(J))
print("2π / √(ac)   :", 2 * np.pi / np.sqrt(a * c))
```

Çıktı:

```
denge noktası: (3.0, 2.0)
[[ 0.  -1.5]
 [ 0.5  0. ]]
özdeğerler   : [0.+0.8660254j 0.-0.8660254j]
2π / √(ac)   : 7.255197456936871
```

Özdeğerler $\pm i\sqrt{ac} \approx \pm 0{,}866\,i$'dir; çünkü J'nin izi sıfır, determinantı $ac = 0{,}75$'tir. Denge noktası çevresindeki küçük salınımların periyodu bu yüzden $2\pi/\sqrt{ac} \approx 7{,}2552$ olmalıdır. Yörüngelerin gerçekten kapalı olduğunu görmek için bir korunan büyüklük kullanacağız.

 CC BY-NC-SA 4.0

Tanım 13.13 (Korunan Büyüklük). Bir $V(\mathbf{y})$ fonksiyonu sistemin her $\mathbf{y}(t)$ çözümü boyunca sabit kalıyorsa, yani $\frac{d}{dt}V\left(\,\mathbf{y}(t)\,\right)=0$ ise, V'ye sistemin **korunan büyüklüğü** (conserved quantity) ya da **ilk integrali** denir.

Yani her yörünge V'nin bir seviye eğrisinin üzerinde kalır. Fizikteki enerjinin korunumu bunun en bilinen örneğidir. Korunan büyüklük sayısal çözüm için de değerli bir denetimdir: V sayısal çözüm boyunca ne kadar değişiyorsa çözüm en az o ölçüde hatalıdır.

Önerme 13.2 (Lotka–Volterra Modelinin Korunan Büyüklüğü). *$x, y > 0$ bölgesinde*

$$V(x,y) = dx - c\ln x + by - a\ln y$$

fonksiyonu Lotka–Volterra modelinin korunan büyüklüğüdür.

> ***İspat.***
> *Zincir kuralıyla*
>
> $$\frac{d}{dt}V\left(\,x(t),y(t)\,\right) = \left(d-\frac{c}{x}\right)x' + \left(b-\frac{a}{y}\right)y'$$
>
> *olur. $x' = x(a-by)$ ve $y' = y(dx-c)$ yerine konursa*
>
> $$\frac{d}{dt}V = (dx-c)(a-by) + (by-a)(dx-c) = 0$$
>
> *bulunur.*
> ■

V, iki konveks tek değişkenli fonksiyonun toplamıdır: $dx - c\ln x$ ifadesi en küçük değerini $x = c/d$'de, $by - a\ln y$ ifadesi $y = a/b$'de alır ve ikisi de 0'a ya da sonsuza giderken sonsuza gider. Bu yüzden V'nin seviye eğrileri $(3,2)$ noktasını çevreleyen kapalı eğrilerdir; her yörünge kapalıdır ve her çözüm periyodiktir.

Sistemi `solve_ivp` ile $(6,2)$ başlangıcından çözelim. Parametreleri fonksiyonun içine gömmek yerine `args` parametresiyle veriyoruz: `solve_ivp, args=(a, b, c, d)` demetini hem sağ tarafa hem olay fonksiyonuna ek argüman olarak geçirir. Avın denge değeri 3'ü yukarıdan aşağıya kestiği anları bir olay fonksiyonuyla yakalayıp periyodu ölçüyor, V'nin çözüm boyunca ne kadar değiştiğine de bakıyoruz:

```
import numpy as np
from scipy.integrate import solve_ivp


def lotka(t, z, a, b, c, d):
    x, y = z
    return [a * x - b * x * y, -c * y + d * x * y]


def V(x, y, a, b, c, d):
    return d * x - c * np.log(x) + b * y - a * np.log(y)


def prey_down(t, z, a, b, c, d):
    return z[0] - c / d       # av, denge değerini yukarıdan aşağı keser


prey_down.direction = -1

p = (1.0, 0.5, 0.75, 0.25)            # a, b, c, d
sol = solve_ivp(lotka, (0, 30), [6.0, 2.0], args=p, events=prey_down,
                rtol=1e-10, atol=1e-12, dense_output=True)
for tk in [0, 2, 4, 6, 8]:
```

 CC BY-NC-SA 4.0

```
    x, y = sol.sol(tk)
    print(f"t = {tk}:   av = {x:6.3f}   avcı = {y:6.3f}")
v = V(sol.y[0], sol.y[1], *p)
print("V'nin en büyük sapması:", np.max(np.abs(v - v[0])))
print("olay anları:", np.round(sol.t_events[0], 6))
print("periyot    :", np.round(np.diff(sol.t_events[0]), 6))
```

Çıktı:

```
t = 0:   av =  6.000   avcı =  2.000
t = 2:   av =  1.817   avcı =  3.303
t = 4:   av =  1.361   avcı =  1.447
t = 6:   av =  3.402   avcı =  0.942
t = 8:   av =  5.591   avcı =  2.695
V'nin en büyük sapması: 3.400024706223803e-11
olay anları: [ 1.350724  8.934098 16.517472 24.100846]
periyot    : [7.583374 7.583374 7.583374]
```

V, 30 birimlik süre boyunca $3{,}4 \cdot 10^{-11}$'den fazla değişmiyor; çözüm hata payı ölçüsünde doğru. Olay anları arasındaki süre, yani periyot, 7,583374'tür. Tablo modelin döngüsünü gösteriyor: av çokken avcı çoğalır ($t = 2$), avcı çoğalınca av azalır, av azalınca avcı da azalır ($t = 4$) ve avcı azalınca av yeniden çoğalır ($t = 6$ ve $t = 8$).

Periyodun genliğe nasıl bağlı olduğunu görmek için birkaç yörüngeyi birlikte çizelim. Kod bir önceki kodun devamıdır; `sol`, `lotka`, `prey_down` ve `p` oradan gelir:

```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
t = np.linspace(0, 30, 601)
x, y = sol.sol(t)
ax1.plot(t, x, label="av x(t)")
ax1.plot(t, y, label="avcı y(t)")
ax1.set_xlabel("t")
ax1.legend()

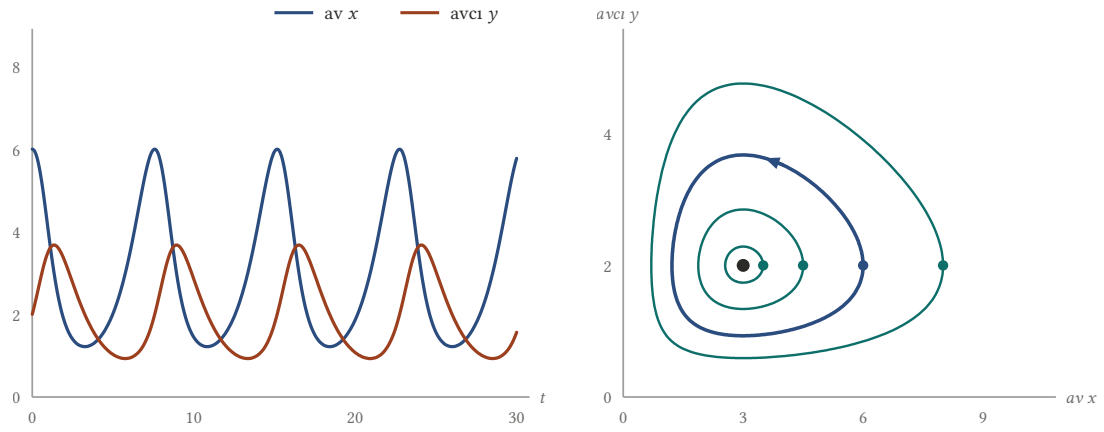
for x0 in [3.5, 4.5, 6.0, 8.0]:
    s = solve_ivp(lotka, (0, 30), [x0, 2.0], args=p, events=prey_down,
                  rtol=1e-10, atol=1e-12, dense_output=True)
    T = s.t_events[0][1] - s.t_events[0][0]
    tt = np.linspace(0, T, 400)
    ax2.plot(*s.sol(tt))
    print(f"x(0) = {x0}:  periyot = {T:.4f}")
ax2.plot(3, 2, "ko")
ax2.set_xlabel("av x")
ax2.set_ylabel("avcı y")
fig.savefig("lotka-volterra.png", dpi=150)
```

Çıktı:

```
x(0) = 3.5:  periyot = 7.2684
x(0) = 4.5:  periyot = 7.3556
x(0) = 6.0:  periyot = 7.5834
x(0) = 8.0:  periyot = 7.9976
```

Kod iki grafiği `lotka-volterra.png` dosyasına kaydeder; aşağıdaki şekil aynı grafiklerin aynı verilerle çizilmiş hâlidir:

 CC BY-NC-SA 4.0



Şekil 13.7: Lotka–Volterra modeli, $a = 1$, $b = 0{,}5$, $c = 0{,}75$, $d = 0{,}25$; Matplotlib kodunun çizdiği iki grafiğin aynı verilerle çizilmiş hâli. Solda (6; 2) başlangıcından çıkan çözümün zamana göre değişimi: avcı sayısının tepeleri av sayısınınkinden sonra gelir. Sağda $x(0) = 3{,}5$; 4,5; 6; 8 ve $y(0) = 2$ başlangıçlarından çıkan kapalı yörüngeler; hepsi (3; 2) denge noktasının çevresinde saat yönünün tersine dolanır.

Yörünge denge noktasına yaklaştıkça periyot küçülüyor ve doğrusallaştırmanın verdiği $7{,}2552$ değerine yaklaşıyor: $x(0) = 3{,}5$ için periyot $7{,}2684$'tür.

13.10 Sarkaç

Bölümün son örneği, uzunluğu L olan ağırlıksız bir çubuğun ucundaki noktasal bir kütleden oluşan basit sarkaçtır. Sarkacın düşeyle yaptığı açı $\theta(t)$ ise Newton'un ikinci yasası

$$\theta'' = -\frac{g}{L}\sin\theta$$

denklemini verir; g yerçekimi ivmesidir. Zaman birimini $g/L = 1$ olacak biçimde seçersek denklem $\theta'' = -\sin\theta$ olur. Açısal hızı $\omega = \theta'$ ile gösterip denklemi sisteme çevirirsek

$$\theta' = \omega, \qquad \omega' = -\sin\theta$$

otonom sistemi elde edilir. Küçük açılarda $\sin\theta \approx \theta$ olduğundan denklem harmonik salınıcıya, $\theta'' = -\theta$ denklemine dönüşür ve periyot 2π olur. Büyük açılarda ise durum değişir. Sistemin korunan büyüklüğü enerjidir:

$$E(\theta, \omega) = \frac{\omega^2}{2} - \cos\theta.$$

Gerçekten

$$\frac{d}{dt}E = \omega\,\omega' + \sin\theta\,\theta' = -\omega\sin\theta + \omega\sin\theta = 0$$

olur. $\frac{\omega^2}{2}$ terimi kinetik, $-\cos\theta$ terimi potansiyel enerjiye karşılık gelir. Enerjinin korunumu, periyot için kapalı bir formül verir.

Teorem 13.2 (Sarkacın Periyodu). *$0 < \theta_0 < \pi$ olsun. $\theta'' = -\sin\theta$, $\theta(0) = \theta_0$, $\theta'(0) = 0$ sarkacının periyodu*

$$T(\theta_0) = 4K(k), \qquad k = \sin\frac{\theta_0}{2}$$

dir. Burada

 CC BY-NC-SA 4.0

$$K(k) = \int_0^{\pi/2} \frac{d\varphi}{\sqrt{1 - k^2 \sin^2 \varphi}}$$

birinci türden tam eliptik integraldir.

> ***İspat.***
> *Enerjinin korunumundan* $\frac{\omega^2}{2} - \cos\theta = -\cos\theta_0$, *yani* $\omega^2 = 2(\cos\theta - \cos\theta_0)$*'dır. Sarkaç* θ_0*'dan bırakılınca* $\omega < 0$ *olur ve sarkaç* $\theta = 0$*'a varana kadar böyle kalır. Denklem* $\theta \mapsto -\theta$ *ve* $t \mapsto -t$ *dönüşümleri altında değişmediğinden hareketin dört çeyreği eşit sürer;* θ_0*'dan* 0*'a iniş bunlardan biridir.* $dt = d\theta / \mid \omega \mid$ *olduğundan*
>
> $$\frac{T}{4} = \int_0^{\theta_0} \frac{d\theta}{\sqrt{2(\cos\theta - \cos\theta_0)}}$$
>
> *bulunur.* $\cos\theta = 1 - 2\sin^2\frac{\theta}{2}$ *özdeşliğiyle* $2(\cos\theta - \cos\theta_0) = 4\left(k^2 - \sin^2\frac{\theta}{2}\right)$ *olur.* $\sin\frac{\theta}{2} = k\sin\varphi$ *değişken değiştirmesini yapalım:* θ*,* 0*'dan* θ_0*'a giderken* φ*,* 0*'dan* $\pi/2$*'ye gider. Ayrıca*
>
> $$\frac{1}{2}\cos\frac{\theta}{2}\,d\theta = k\cos\varphi\,d\varphi, \qquad \cos\frac{\theta}{2} = \sqrt{1 - k^2\sin^2\varphi}$$
>
> *ve* $\sqrt{4(k^2 - k^2\sin^2\varphi)} = 2k\cos\varphi$*'dir. Bunlar yerine konunca*
>
> $$\begin{aligned}\frac{T}{4} &= \int_0^{\pi/2} \frac{1}{2k\cos\varphi} \cdot \frac{2k\cos\varphi\,d\varphi}{\sqrt{1 - k^2\sin^2\varphi}} \\ &= \int_0^{\pi/2} \frac{d\varphi}{\sqrt{1 - k^2\sin^2\varphi}} = K(k)\end{aligned}$$
>
> *elde edilir. İlk integralin* $\theta = \theta_0$*'daki tekilliği değişken değiştirmeyle ortadan kalktı.*
> ■

Yani $\theta_0 \to 0$ iken $k \to 0$, $K(k) \to \pi/2$ ve $T \to 2\pi$ olur: harmonik salınıcının periyodu. $\theta_0 \to \pi$ iken ise $k \to 1$ ve $K(k) \to \infty$'dur: tepeye yakın bırakılan sarkacın periyodu sınırsız büyür. Teoremi sayısal olarak sınayalım. Periyodu, θ'nın aşağıdan yukarıya sıfırdan geçtiği ardışık iki anın farkı olarak ölçüyoruz; K için SciPy'nin `scipy.special.ellipk` fonksiyonunu kullanıyoruz:

```
import numpy as np
from scipy.integrate import solve_ivp
from scipy.special import ellipk


def pendulum(t, z):
    theta, omega = z
    return [omega, -np.sin(theta)]


def upward(t, z):
    return z[0]              # θ = 0 geçişleri


upward.direction = 1         # yalnız aşağıdan yukarıya geçişler

print(" θ0    T (sayısal)    4K(sin²(θ0/2))   T / 2π")
for theta0 in [0.1, 1.0, 2.0, 3.0]:
    sol = solve_ivp(pendulum, (0, 60), [theta0, 0.0], events=upward,
                    rtol=1e-10, atol=1e-12)
    te = sol.t_events[0]
    T_num = te[1] - te[0]
    T_ell = 4 * ellipk(np.sin(theta0 / 2) ** 2)
```

```
    print(f"{theta0:3.1f}  {T_num:13.10f}  {T_ell:13.10f}"
          f"     {T_num / (2 * np.pi):.4f}")
```

Çıktı:

```
 θ0    T (sayısal)    4K(sin²(θ0/2))   T / 2π
0.1   6.2871145493   6.2871145493     1.0006
1.0   6.6999756642   6.6999756644     1.0663
2.0   8.3497529262   8.3497529269     1.3289
3.0  16.1555393359  16.1555393724     2.5712
```

İki sütun en az sekiz anlamlı basamakta uyuşuyor. $\theta_0 = 0{,}1$ radyanda (yaklaşık $5{,}7°$) periyot 2π'den yalnız binde $0{,}6$ uzundur ve harmonik yaklaşım çok iyidir. $\theta_0 = 3$ radyanda (yaklaşık $172°$) ise periyot 2π'nin $2{,}57$ katıdır.

> ⚠ ellipk fonksiyonunun argümanı
>
> `scipy.special.ellipk(m)` fonksiyonu argüman olarak k'yi değil $m = k^2$'yi alır. Yukarıdaki kodda bu yüzden `ellipk(np.sin(theta0 / 2) ** 2)` yazdık; `ellipk(np.sin(theta0 / 2))` yazmak sessizce yanlış bir periyot verirdi. Özel fonksiyonların hangi gösterimi kullandığını belgelerinden denetleyin.

Son olarak sarkacın faz portresini çizelim. Bir periyodu 17'den kısa olan beş salınımı ve sarkacın tepeyi aşarak döndüğü iki hareketi çözüp her birinin enerjisini yazdırıyoruz. Dönen yörüngeleri $\theta = -\pi$'den başlatıp $\theta = \pi$'ye varınca bir olay fonksiyonuyla durduruyoruz; ters yönde dönen yörüngeler, denklemin simetrisi sayesinde $(\theta, \omega) \mapsto (-\theta, -\omega)$ yansımasıyla elde ediliyor:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp


def pendulum(t, z):
    return [z[1], -np.sin(z[0])]


def energy(theta, omega):
    return omega**2 / 2 - np.cos(theta)


def at_right(t, z):
    return z[0] - np.pi          # sağ kenar: θ = π


at_right.terminal = True
opts = dict(rtol=1e-10, atol=1e-12, dense_output=True)

fig, ax = plt.subplots(figsize=(7, 4.5))
starts = [(th, 0.0) for th in [0.5, 1.0, 1.5, 2.0, 2.5]]
starts += [(-np.pi, om) for om in [0.8, 1.6]]
for z0 in starts:
    if z0[1] == 0.0:             # salınım: t = 17 bir tam periyottan uzun
        s = solve_ivp(pendulum, (0, 17), z0, **opts)
    else:                        # dönme: θ = π olana kadar
        s = solve_ivp(pendulum, (0, 20), z0, events=at_right, **opts)
    th, om = s.sol(np.linspace(0, s.t[-1], 400))
    ax.plot(th, om, "C0" if z0[1] == 0.0 else "C1")
    if z0[1] != 0.0:
        ax.plot(-th, -om, "C1")  # ters yönde dönen simetrik yörünge
    E = energy(th, om)
    print(f"başlangıç ({z0[0]:+.3f}, {z0[1]:.1f}):  E = {E[0]:+.4f}"
```

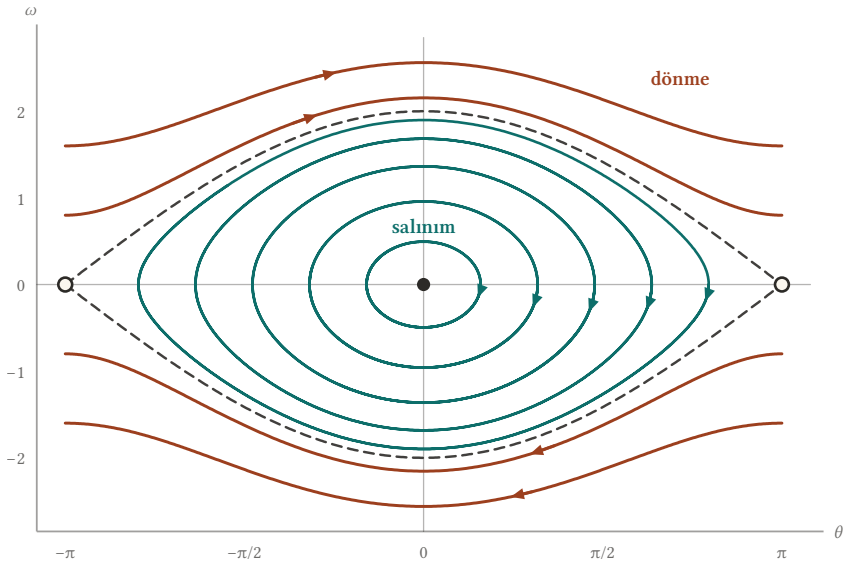
 CC BY-NC-SA 4.0

```
          f"   sapma = {np.max(np.abs(E - E[0])):.1e}")
th = np.linspace(-np.pi, np.pi, 400)
ax.plot(th, 2 * np.cos(th / 2), "k--", th, -2 * np.cos(th / 2), "k--")
ax.set_xlabel("θ")
ax.set_ylabel("ω = θ'")
fig.savefig("sarkac-faz.png", dpi=150)
```

Çıktı:

```
başlangıç (+0.500, 0.0):  E = -0.8776   sapma = 5.5e-11
başlangıç (+1.000, 0.0):  E = -0.5403   sapma = 2.6e-10
başlangıç (+1.500, 0.0):  E = -0.0707   sapma = 8.8e-10
başlangıç (+2.000, 0.0):  E = +0.4161   sapma = 5.0e-10
başlangıç (+2.500, 0.0):  E = +0.8011   sapma = 5.6e-10
başlangıç (-3.142, 0.8):  E = +1.3200   sapma = 8.4e-10
başlangıç (-3.142, 1.6):  E = +2.2800   sapma = 1.7e-09
```

Her yörüngede enerji sabit kalıyor; en büyük sapma $1{,}7 \cdot 10^{-9}$'dur. Kod grafiği `sarkac-faz.png` dosyasına kaydeder; aşağıdaki şekil aynı grafiğin aynı verilerle çizilmiş hâlidir:



Şekil 13.8: Sarkacın faz portresi; Matplotlib kodunun çizdiği grafiğin aynı verilerle çizilmiş hâli. Kapalı eğriler θ_0 = 0,5; 1; 1,5; 2; 2,5 genlikli salınımlardır; dalgalı eğriler, sarkacın tepeden aşarak döndüğü hareketlerdir. Kesikli ayırıcı eğri (E = 1) iki tür hareketi ayırır. Dolu nokta kararlı denge (0; 0), içi boş noktalar tepedeki kararsız denge (±π; 0).

Faz portresinde iki tür hareket görülüyor. $E < 1$ ise sarkaç tepeye ulaşamaz ve orijin çevresindeki kapalı yörüngelerde salınır. $E > 1$ ise sarkaç tepeyi aşarak hep aynı yönde döner; θ sürekli artar ya da azalır ve şekilde bu yörüngeler $\theta = \pm\pi$ kenarlarından girip çıkar. İki türü $E = 1$ seviye eğrisi ayırır: $\frac{\omega^2}{2} - \cos\theta = 1$ eşitliği $\omega^2 = 2(1 + \cos\theta) = 4\cos^2\frac{\theta}{2}$ verir, yani bu **ayırıcı eğri** (separatrix) $\omega = \pm 2\cos\frac{\theta}{2}$'dir. $(0, 0)$ kararlı bir denge noktasıdır: Jacobi matrisinin özdeğerleri $\pm i$'dir ve yakınındaki yörüngeler onu çevreler. $(\pm\pi, 0)$ ise sarkacın tam tepede durduğu kararsız dengedir: oradaki Jacobi matrisi $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$'nin özdeğerleri ± 1'dir ve yörüngelerin bir kısmı noktaya yaklaşırken bir kısmı ondan uzaklaşır. $\theta = \pi$ ile $\theta = -\pi$ aynı fiziksel konumdur.

13.11 Alıştırmalar

Aşağıdaki alıştırmalarda bölümün yöntemlerini yeni problemlerde deneyeceğiz. Çözümlerdeki bazı kodlar `odetools.py` modülünü kullanır; modül, çalıştırılan programla aynı klasörde olmalıdır.

Alıştırma 13.1 (Doğrusal Bir Denklemde Euler Yöntemi). $y' = t - y$, $y(0) = 1$ probleminin tam çözümü $y = t - 1 + 2e^{-t}$'dir. $y(1)$'i Euler yöntemiyle $h = 0{,}1$ ve $h = 0{,}05$ alarak hesaplayınız ve iki hatanın oranını bulunuz.

Çözüm.
Adım 1. $f(t, y) = t - y$'dir. Tam değer $y(1) = 1 - 1 + 2e^{-1}$, yani yaklaşık $0{,}73575888$'dir.
Adım 2. $h = 1/n$ olmak üzere $n = 10$ ve $n = 20$ adımla `euler` fonksiyonunu çağırıyoruz:

```
import numpy as np

from odetools import euler


def f(t, y):
    return t - y


exact = 2 * np.exp(-1.0)          # y(1) = 1 - 1 + 2e^(-1)
errs = []
for n in [10, 20]:
    t, y = euler(f, 0.0, 1.0, 1 / n, n)
    errs.append(abs(y[-1] - exact))
    print(f"h = {1 / n:.2f}   y(1) ≈ {y[-1]:.8f}   hata = {errs[-1]:.3e}")
print(f"tam değer: {exact:.8f}   hata oranı: {errs[0] / errs[1]:.3f}")
```

Çıktı:

```
h = 0.10   y(1) ≈ 0.69735688   hata = 3.840e-02
h = 0.05   y(1) ≈ 0.71697184   hata = 1.879e-02
tam değer: 0.73575888   hata oranı: 2.044
```

Adım 3. Hatalar $0{,}0384$ ve $0{,}0188$'dir; oranları $2{,}04 \approx 2$'dir. Adım yarıya inince hata da yarıya iniyor: Euler yöntemi birinci mertebedendir.
Denetim. Bu doğrusal denklemde Euler dizisi kapalı biçimde de yazılabilir: $y_{i+1} = (1 - h)\,y_i + h\,t_i$ adımından tümevarımla $y_i = t_i - 1 + 2(1 - h)^i$ bulunur. $h = 0{,}1$ için $y_{10} = 2 \cdot 0{,}9^{10} \approx 0{,}69735688$ çıktıyla aynıdır. Tam çözümdeki e^{-t} çarpanının yerini $(1 - h)^{t/h}$ almıştır ve $h \to 0$ iken $(1 - h)^{t/h} \to e^{-t}$'dir.
■

Alıştırma 13.2 (Doğrusal Bir Denklemde RK4). $y' = t - y$, $y(0) = 1$ probleminde $y(1)$'i RK4 ile $h = 0{,}1$ ve $h = 0{,}05$ alarak hesaplayınız. Hataların oranı, RK4'ün dördüncü mertebeden olmasıyla uyumlu mudur?

Çözüm.
Adım 1. Tam değer yine $y(1) = 2e^{-1}$'dir. Dördüncü mertebeden bir yöntemde adımı yarıya indirmek hatayı kabaca $2^4 = 16$'ya bölmelidir.
Adım 2. Bu kez `rk4` fonksiyonunu çağırıyoruz:

```
import numpy as np

from odetools import rk4


def f(t, y):
    return t - y
```

```
exact = 2 * np.exp(-1.0)
errs = []
for n in [10, 20]:
    t, y = rk4(f, 0.0, 1.0, 1 / n, n)
    errs.append(abs(y[-1] - exact))
    print(f"h = {1 / n:.2f}   y(1) ≈ {y[-1]:.10f}   hata = {errs[-1]:.3e}")
print(f"tam değer: {exact:.10f}   hata oranı: {errs[0] / errs[1]:.2f}")
```

Çıktı:

```
h = 0.10   y(1) ≈ 0.7357595488   hata = 6.665e-07
h = 0.05   y(1) ≈ 0.7357589223   hata = 3.995e-08
tam değer: 0.7357588823   hata oranı: 16.68
```

Adım 3. Hatalar $6{,}7 \cdot 10^{-7}$ ve $4{,}0 \cdot 10^{-8}$'dir; oran $16{,}68$'dir ve 16'ya yakındır. Yanıt evettir. Aynı adım sayısıyla Euler'in hatası $0{,}038$ idi; RK4 ünki yaklaşık $58\,000$ kat küçüktür.
■

Alıştırma 13.3 (Heun Yöntemi). Heun yöntemi önce bir Euler adımıyla $\tilde{y} = y_i + h\,f(t_i, y_i)$ tahminini yapar, sonra iki uçtaki eğimin ortalamasıyla ilerler:

$$y_{i+1} = y_i + \frac{h}{2}\left(f(t_i, y_i) + f(t_{i+1}, \tilde{y}) \right).$$

Yöntemi bir `heun` fonksiyonu olarak yazınız ve lojistik problemde ($y' = y(1-y)$, $y(0) = 0{,}1$, $T = 4$) mertebesini adımı yarıya indirerek ölçünüz.

Çözüm.
Adım 1. Fonksiyon `euler`'in döngüsüne ikinci bir eğim ekler. f yalnız t'ye bağlıysa adım, $\int_{t_i}^{t_{i+1}} f\,dt$ için yamuk kuralı olur (bkz. Nümerik Analiz); bu yüzden yöntemin ikinci mertebeden olmasını bekleriz.
Adım 2. Tam değer $y(4) = 1/(1 + 9e^{-4})$'tür. $n = 16, 32, \ldots, 256$ adımla hataları ve ardışık oranları hesaplıyoruz:

```
import numpy as np


def heun(f, t0, y0, h, n):
    """y' = f(t, y), y(t0) = y0 problemini n Heun adımıyla çözer."""
    t = t0 + h * np.arange(n + 1)
    y = np.zeros(n + 1)
    y[0] = y0
    for i in range(n):
        k1 = f(t[i], y[i])                    # baştaki eğim
        k2 = f(t[i] + h, y[i] + h * k1)       # Euler tahminindeki eğim
        y[i + 1] = y[i] + h / 2 * (k1 + k2)
    return t, y


def f(t, y):
    return y * (1 - y)


exact = 1 / (1 + 9 * np.exp(-4.0))
prev = None
for n in [16, 32, 64, 128, 256]:
    t, y = heun(f, 0.0, 0.1, 4 / n, n)
    err = abs(y[-1] - exact)
```

```
    ratio = f"{prev / err:.3f}" if prev else "-"
    print(f"n = {n:3d}   hata = {err:.3e}   oran = {ratio}")
    prev = err
```

Çıktı:

```
n =  16   hata = 2.150e-03   oran = -
n =  32   hata = 5.402e-04   oran = 3.980
n =  64   hata = 1.356e-04   oran = 3.983
n = 128   hata = 3.400e-05   oran = 3.989
n = 256   hata = 8.512e-06   oran = 3.994
```

Adım 3. Oranlar $4 = 2^2$'ye yaklaşıyor: Heun yöntemi ikinci mertebedendir. Adım başına iki f çağrısıyla Euler (1) ile RK4 (4) arasında durur; doğruluğu da öyle: $n = 256$'da hatası $8{,}5 \cdot 10^{-6}$'dır. Aynı n için Euler'in hatası $2{,}9 \cdot 10^{-4}$, RK4'ünki $5{,}9 \cdot 10^{-11}$'dir.
■

Alıştırma 13.4 (Aynı Doğruluk İçin Kaç Adım?). $y' = y$, $y(0) = 1$ probleminde $y(1) = e$ değerini 10^{-6}'dan küçük bir hatayla bulmak için Euler ve RK4 yöntemlerinin kaçar adım ve kaçar f çağrısı gerektirdiğini karşılaştırınız.

Çözüm.
Adım 1. RK4 için n'yi birer birer artırıp hatanın 10^{-6}'nın altına ilk düştüğü adım sayısını arıyoruz.
Adım 2. Euler için $e - y_n \approx \frac{e}{2n}$ yaklaşıklığını kullanıyoruz (bkz. Örnek 13.2): $\frac{e}{2n} < 10^{-6}$ koşulu $n > \frac{e}{2} \cdot 10^6 \approx 1\,359\,141$ verir. Kestirimi, bu sayının iki yanındaki $n = 1\,350\,000$ ve $n = 1\,360\,000$ değerleriyle sınıyoruz:

```
import numpy as np

from odetools import euler, rk4


def f(t, y):
    return y


n = 1
while abs(np.e - rk4(f, 0.0, 1.0, 1 / n, n)[1][-1]) >= 1e-6:
    n += 1
err = abs(np.e - rk4(f, 0.0, 1.0, 1 / n, n)[1][-1])
print(f"RK4  : n = {n}   hata = {err:.2e}   f çağrısı = {4 * n}")

for n in [1_350_000, 1_360_000]:
    err = abs(np.e - euler(f, 0.0, 1.0, 1 / n, n)[1][-1])
    print(f"Euler: n = {n}   hata = {err:.4e}   f çağrısı = {n}")
```

Çıktı:

```
RK4  : n = 13   hata = 7.44e-07   f çağrısı = 52
Euler: n = 1350000   hata = 1.0068e-06   f çağrısı = 1350000
Euler: n = 1360000   hata = 9.9937e-07   f çağrısı = 1360000
```

Adım 3. RK4'e 13 adım, yani 52 çağrı yetiyor. Euler'de 1 350 000 adım yetmiyor, 1 360 000 adım yetiyor; kestirim doğru. Euler yaklaşık 1,36 milyon çağrı ister, yani RK4'ün yaklaşık 26 000 katı. Yüksek mertebenin değeri budur.
■

 CC BY-NC-SA 4.0

Alıştırma 13.5 (Lojistik Büyümede Yarı Doygunluk Anı). $y' = 0{,}5\,y\big(1 - \frac{y}{100}\big)$, $y(0) = 5$ lojistik denkleminin çözümünün 50 değerine ulaştığı anı `solve_ivp`'nin olay fonksiyonuyla bulunuz ve tam çözümden elde edilen değerle karşılaştırınız.

Çözüm.
Adım 1: tam çözüm. $y' = ry(1 - y/K)$, $y(0) = y_0$ denkleminin çözümü

$$y(t) = \frac{K}{1 + \left(\frac{K}{y_0} - 1\right)e^{-rt}}$$

dir. $r = 0{,}5$, $K = 100$, $y_0 = 5$ için $y(t) = \frac{100}{1+19e^{-t/2}}$ olur. $y = 50$ koşulu $19e^{-t/2} = 1$, yani $t = 2\ln 19 \approx 5{,}8889$ verir.
Adım 2: olay fonksiyonu. $y - 50$ fonksiyonu aranan anda sıfır olur; `terminal=True` ile çözümü orada durduruyoruz:

```
import numpy as np
from scipy.integrate import solve_ivp

r, K = 0.5, 100.0


def logistic(t, y):
    return r * y * (1 - y / K)


def half(t, y):
    return y[0] - K / 2


half.terminal = True

sol = solve_ivp(logistic, (0, 50), [5.0], events=half,
                rtol=1e-10, atol=1e-10)
print(f"sayısal : t = {sol.t_events[0][0]:.10f}")
print(f"2 ln 19 : t = {2 * np.log(19):.10f}")
```

Çıktı:

```
sayısal : t = 5.8888779596
2 ln 19 : t = 5.8888779583
```

Adım 3. İki değer sekiz ondalık basamakta uyuşuyor; aradaki $1{,}3 \cdot 10^{-9}$'luk fark verdiğimiz hata payı düzeyindedir. Nüfus, doygunluk değerinin yarısına $t \approx 5{,}889$'da ulaşır; büyüme hızı da bu anda en büyüktür.
■

Alıştırma 13.6 (Sönümlü Salınıcı). $y'' + 2y' + 5y = 0$, $y(0) = 1$, $y'(0) = 0$ problemini birinci mertebeden bir sisteme çevirip `solve_ivp` ile çözünüz. $t = 1, 2, 3$'teki değerleri tam çözüm $y = e^{-t}\big(\cos 2t + \frac{1}{2}\sin 2t\big)$ ile karşılaştırınız.

Çözüm.
Adım 1. $u_1 = y$, $u_2 = y'$ dersek $u_1' = u_2$ ve $u_2' = y'' = -2u_2 - 5u_1$ olur; başlangıç vektörü $(1, 0)$'dır.
Adım 2. Tam çözümü denetleyelim. Karakteristik denklem $r^2 + 2r + 5 = 0$'ın kökleri $-1 \pm 2i$'dir, dolayısıyla $y = e^{-t}(A\cos 2t + B\sin 2t)$ olur. $y(0) = A = 1$ ve $y'(0) = -A + 2B = 0$ koşulları $B = \frac{1}{2}$ verir.
Adım 3. Sistemi `t_eval` ile istenen anlarda çözüyoruz:

 CC BY-NC-SA 4.0

```
import numpy as np
from scipy.integrate import solve_ivp


def damped(t, u):
    y, v = u
    return [v, -2 * v - 5 * y]        # y'' = -2y' - 5y


def exact(t):
    return np.exp(-t) * (np.cos(2 * t) + 0.5 * np.sin(2 * t))


t = np.array([1.0, 2.0, 3.0])
sol = solve_ivp(damped, (0, 3), [1.0, 0.0], t_eval=t,
                rtol=1e-10, atol=1e-12)
for tk, yk in zip(sol.t, sol.y[0]):
    print(f"t = {tk:.0f}   sayısal = {yk:+.10f}   tam = {exact(tk):+.10f}")
```

Çıktı:

```
t = 1   sayısal = +0.0141640490   tam = +0.0141640489
t = 2   sayısal = -0.1396720846   tam = -0.1396720846
t = 3   sayısal = +0.0408484245   tam = +0.0408484245
```

Adım 4. Değerler on ondalık basamakta (ilk satırda dokuz basamakta) uyuşuyor. Çözüm e^{-t} zarfı içinde salınarak sıfıra iner.
■

Alıştırma 13.7 (Doğrusal Sistem ve Matris Üsteli). $\mathbf{x}' = A\mathbf{x}$, $\mathbf{x}(0) = (1, 0)$ sisteminde $A = \begin{pmatrix} 0 & 1 \\ -2 & -3 \end{pmatrix}$'tür. Kare bir A matrisinin üsteli (matrix exponential), e^{x}'in kuvvet serisinde x yerine tA yazılarak $e^{tA} = \sum_{k \geq 0} \frac{t^k A^k}{k!}$ biçiminde tanımlanır. Çözümün $t = 1$'deki değerini `solve_ivp` ile hesaplayınız ve $e^{A}\mathbf{x}(0)$ ile karşılaştırınız. Matris üsteli için `scipy.linalg.expm` fonksiyonunu kullanınız.

Çözüm.
Adım 1. e^{tA} serisi her t için yakınsar ve terim terim türevlenebilir:

$$\left(e^{tA}\right)' = A + tA^2 + \frac{t^2}{2!}A^3 + \cdots = A\,e^{tA}.$$

$e^{0A} = I$ olduğundan $\mathbf{x}(t) = e^{tA}\mathbf{x}(0)$ fonksiyonu hem $\mathbf{x}' = A\mathbf{x}$ denklemini hem de başlangıç koşulunu sağlar; yani sistemin çözümüdür. Bu yüzden $t = 1$'de $e^{A}\mathbf{x}(0)$'ı karşılaştırmalıyız. A'nın özdeğerleri -1 ve -2'dir. Sistemin ilk bileşeni $y'' + 3y' + 2y = 0$ denklemini sağlar ve $y(0) = 1$, $y'(0) = 0$ koşullarıyla $y = 2e^{-t} - e^{-2t}$, $y' = -2e^{-t} + 2e^{-2t}$ bulunur.
Adım 2. Sağ tarafı `A @ x` ile tek satırlık bir `lambda` olarak yazıyoruz:

```
import numpy as np
from scipy.integrate import solve_ivp
from scipy.linalg import expm

A = np.array([[0.0, 1.0],
              [-2.0, -3.0]])
x0 = np.array([1.0, 0.0])
print("özdeğerler:", np.linalg.eigvals(A))

sol = solve_ivp(lambda t, x: A @ x, (0, 1), x0, rtol=1e-10, atol=1e-12)
```

```
print("solve_ivp :", sol.y[:, -1])
print("expm(A)x0 :", expm(A) @ x0)
e1, e2 = np.exp(-1.0), np.exp(-2.0)
print("formül    :", np.array([2 * e1 - e2, -2 * e1 + 2 * e2]))
print("en büyük fark:", np.max(np.abs(sol.y[:, -1] - expm(A) @ x0)))
```

Çıktı:

```
özdeğerler: [-1.+0.j -2.+0.j]
solve_ivp : [ 0.6004236  -0.46508832]
expm(A)x0 : [ 0.6004236  -0.46508832]
formül    : [ 0.6004236  -0.46508832]
en büyük fark: 1.0809686479262837e-11
```

Adım 3. NumPy özdeğerleri karmaşık sayı türünde döndürdü; sanal kısımları sıfırdır ve özdeğerler -1 ile -2'dir. Üç yol aynı vektörü veriyor; `solve_ivp` ile `expm` arasındaki fark $1{,}1 \cdot 10^{-11}$'dir. İki özdeğer de negatif olduğundan çözüm sıfıra söner.
■

Alıştırma 13.8 (Van der Pol Salınıcısının Limit Çevrimi). $y'' - (1 - y^2)\,y' + y = 0$ van der Pol denkleminin çözümleri, başlangıç koşulundan bağımsız olarak aynı periyodik harekete yaklaşır. Denklemi $(y, y') = (0{,}5;\, 0)$ ve $(3;\, 0)$ başlangıçlarından $[0, 100]$ aralığında çözüp son tam turdaki periyodu ve genliği, yani y'nin en büyük değerini, ölçünüz.

Çözüm.
Adım 1. $u_1 = y$, $u_2 = y'$ ile sistem $u_1' = u_2$, $u_2' = (1 - u_1^2)\,u_2 - u_1$ olur. Kodda katsayıyı `mu = 1.0` adıyla genel bıraktık.
Adım 2. Periyodu, y'nin aşağıdan yukarıya sıfırdan geçtiği son iki anın farkı olarak ölçüyoruz; genliği de bu son tur boyunca yoğun çıktıdan okuyoruz:

```
import numpy as np
from scipy.integrate import solve_ivp

mu = 1.0


def vdp(t, u):
    y, v = u
    return [v, mu * (1 - y**2) * v - y]


def upward(t, u):
    return u[0]


upward.direction = 1

for u0 in [[0.5, 0.0], [3.0, 0.0]]:
    sol = solve_ivp(vdp, (0, 100), u0, events=upward,
                    rtol=1e-10, atol=1e-12, dense_output=True)
    te = sol.t_events[0]
    T = te[-1] - te[-2]                       # son tam tur
    tt = np.linspace(te[-2], te[-1], 2001)
    amp = np.max(sol.sol(tt)[0])
    print(f"başlangıç {u0}:  periyot = {T:.6f}   genlik = {amp:.6f}")
```

Çıktı:

```
başlangıç [0.5, 0.0]:  periyot = 6.663287   genlik = 2.008619
başlangıç [3.0, 0.0]:  periyot = 6.663287   genlik = 2.008619
```

Adım 3. İki başlangıç da periyodu 6,663287, genliği 2,008619 olan aynı harekete varıyor. Faz düzleminde bu hareket, öteki yörüngeleri kendine çeken yalıtılmış bir kapalı yörüngedir ve **limit çevrim** (limit cycle) adını alır. Lotka–Volterra modelinden farkı buradadır: orada her başlangıç kendi kapalı yörüngesinde kalıyordu. Van der Pol denkleminde $|\,y\,|<1$ iken $-(1-y^2)\,y'$ terimi harekete enerji katar, $|\,y\,|>1$ iken enerji çeker; küçük salınımlar büyür, büyükler söner.
■

Alıştırma 13.9 (Lotka–Volterra Modelinde Ortalama Nüfuslar). Lotka–Volterra modelinde ($a=1$, $b=0{,}5$, $c=0{,}75$, $d=0{,}25$) bir periyot boyunca av ve avcı nüfuslarının zaman ortalamalarının yörüngeden bağımsız olarak denge değerlerine, $c/d=3$ ve $a/b=2$'ye eşit olduğunu gösteriniz. Sonucu $y(0)=2$ ve $x(0)=4{,}5;\,6;\,8$ yörüngelerinde sayısal olarak doğrulayınız.

Çözüm.
Adım 1: ispat. T periyot olsun. İkinci denklem $\frac{y'}{y}=-c+dx$, yani $(\ln y)'=-c+dx$ biçiminde yazılabilir. $[0,T]$ üzerinde integralleyince, $y(T)=y(0)$ olduğundan

$$0=\ln y(T)-\ln y(0)=-cT+d\int_0^T x\,dt$$

bulunur; buradan $\frac{1}{T}\int_0^T x\,dt=\frac{c}{d}$ çıkar. Aynı biçimde $(\ln x)'=a-by$ eşitliği $\frac{1}{T}\int_0^T y\,dt=\frac{a}{b}$ verir.
Adım 2: sayısal doğrulama. Periyodu olay fonksiyonuyla ölçüp ortalamaları yoğun çıktı üzerinde `scipy.integrate.quad` ile (bkz. Sayısal Türev ve İntegral) hesaplıyoruz:

```
import numpy as np
from scipy.integrate import solve_ivp, quad


def lotka(t, z, a, b, c, d):
    x, y = z
    return [a * x - b * x * y, -c * y + d * x * y]


def prey_down(t, z, a, b, c, d):
    return z[0] - c / d


prey_down.direction = -1

p = (1.0, 0.5, 0.75, 0.25)
for x0 in [4.5, 6.0, 8.0]:
    sol = solve_ivp(lotka, (0, 30), [x0, 2.0], args=p, events=prey_down,
                    rtol=1e-10, atol=1e-12, dense_output=True)
    t1, t2 = sol.t_events[0][:2]              # bir tam periyot
    T = t2 - t1
    mx = quad(lambda t: sol.sol(t)[0], t1, t2)[0] / T
    my = quad(lambda t: sol.sol(t)[1], t1, t2)[0] / T
    print(f"x(0) = {x0}:  T = {T:.4f}   ort. x = {mx:.8f}"
          f"   ort. y = {my:.8f}")
```

Çıktı:

```
x(0) = 4.5:  T = 7.3556   ort. x = 3.00000000   ort. y = 2.00000000
x(0) = 6.0:  T = 7.5834   ort. x = 3.00000000   ort. y = 2.00000000
x(0) = 8.0:  T = 7.9976   ort. x = 3.00000000   ort. y = 2.00000000
```

Adım 3. Periyotlar farklı olsa da ortalamalar sekiz ondalık basamakta 3 ve 2'dir. Nüfuslar dengeden uzakta salınsa bile ortalamaları hep denge değerleridir.
■

Alıştırma 13.10 (Sönümlü Sarkaç Kaç Tur Atar?). Sürtünmeli bir sarkacın denklemi $\theta'' = -\sin\theta - 0{,}2\,\theta'$ olsun. Sarkaç en alttan ($\theta(0) = 0$) $\theta'(0) = 4$ hızıyla itiliyor. Sarkaç tepe noktasından kaç kez geçer ve sonunda hangi açıda durulur?

Çözüm.
Adım 1. Tepe noktası $\theta = \pi$ konumudur; sarkaç döndükçe θ artmaya devam eder, bu yüzden tepeden geçişler $\theta = \pi, 3\pi, 5\pi, \ldots$ anlarıdır. Bu açıların hepsinde $\cos\frac{\theta}{2} = 0$ olduğundan tek bir olay fonksiyonu, `cos(theta / 2)`, hepsini yakalar.
Adım 2. Başlangıç enerjisi $E = \frac{4^2}{2} - \cos 0 = 7$'dir ve tepeyi aşmak için $E > 1$ gerekir. Sürtünme enerjiyi yavaş yavaş tüketir; denklemi uzun bir süre boyunca çözüp geçişleri sayıyoruz. Geçiş anlarındaki açıları `sol.y_events[0]` dizisinden okuyoruz; bu dizinin her satırı bir geçiş anındaki (θ, θ') vektörü olduğundan açılar ilk sütundadır:

```
import numpy as np
from scipy.integrate import solve_ivp


def damped_pendulum(t, z):
    theta, omega = z
    return [omega, -np.sin(theta) - 0.2 * omega]


def top(t, z):
    return np.cos(z[0] / 2)       # θ = π, 3π, 5π, ... iken sıfır


sol = solve_ivp(damped_pendulum, (0, 100), [0.0, 4.0], events=top,
                rtol=1e-10, atol=1e-12)
print("tepeden geçiş anları:", np.round(sol.t_events[0], 4))
print("o anlardaki θ / π   :", np.round(sol.y_events[0][:, 0] / np.pi, 6))
theta_end, omega_end = sol.y[:, -1]
print(f"θ(100) / 2π = {theta_end / (2 * np.pi):.6f}")
print(f"ω(100)      = {omega_end:.1e}")
```

Çıktı:

```
tepeden geçiş anları: [0.936  3.6575]
o anlardaki θ / π   : [1. 3.]
θ(100) / 2π = 1.999975
ω(100)      = -3.0e-05
```

Adım 3. Sarkaç tepeden iki kez geçiyor: $t \approx 0{,}94$'te $\theta = \pi$'den, $t \approx 3{,}66$'da $\theta = 3\pi$'den. Sonra enerjisi 1'in altına düşer ve $\theta = 4\pi$ çevresinde sönümlü salınım yaparak durulur: $\theta(100)/2\pi \approx 2$'dir. Sarkaç iki tam tur atıp yine en alttaki konumda durulmuştur; 4π açısı fiziksel olarak 0 ile aynı konumdur.
■

Bu bölümde başlangıç değer problemlerini sayısal olarak çözmeyi öğrendik: Euler ve RK4 yöntemlerini kendimiz yazdık, mertebelerini log-log grafikte ölçtük, `solve_ivp` ile uyarlamalı adımlı ve katı denklem-

 CC BY-NC-SA 4.0

lere uygun yöntemleri kullandık, olay fonksiyonlarıyla belirli anları yakaladık ve faz düzleminde Lotka–Volterra modelini ve sarkacı inceledik. Buraya kadarki bütün modeller deterministikti: aynı başlangıç her seferinde aynı sonucu verdi. Bir sonraki bölüm Rastgele Sayılar ve Monte Carlo Yöntemleri, rastlantıyı hesaba katıyor: rastgele sayı üretmeyi, olasılık yasalarını simülasyonla görmeyi ve integralleri rastgele örneklemle hesaplamayı anlatıyor.

 CC BY-NC-SA 4.0

14. Rastgele Sayılar ve Monte Carlo Yöntemleri

İki zar atıldığında toplamın 7 gelme olasılığı kaçtır? Sayma teknikleriyle (bkz. Olasılık Teorisi) eşit olasılıklı 36 sonucun 6'sının toplamı 7 olduğunu görür ve $1/6$ buluruz. Aynı soruyu bilgisayara deneyi yüz bin kez yaptırarak da sorabiliriz:

```
import numpy as np

rng = np.random.default_rng(1)
n = 100_000
d1 = rng.integers(1, 7, size=n)     # birinci zar: 1, 2, ..., 6
d2 = rng.integers(1, 7, size=n)     # ikinci zar
seven = d1 + d2 == 7                # toplam 7 mi?
print(d1[:8])
print(d2[:8])
print(seven.sum(), seven.mean(), 1 / 6)
```

Çıktı:

```
[3 4 5 6 1 1 5 6]
[5 4 6 3 5 4 1 3]
16450 0.1645 0.16666666666666666
```

`rng.integers(1, 7, size=n)` 1 ile 6 arasında n tane tamsayı üretir; üst sınır olan 7 dahil değildir. `d1 + d2 == 7` bir boolean dizidir: `sum` doğru olan elemanları sayar, `mean` ise doğruların oranını verir, çünkü hesapta `True` 1, `False` 0 sayılır. Yüz bin denemenin 16450'sinde toplam 7 geldi; göreli sıklık $0{,}1645$, gerçek olasılık ise $1/6 \approx 0{,}1667$'dir.

Bir olasılığı, bir beklenen değeri ya da bir integrali, ilgili rastgele deneyi çok kez tekrarlayıp sonuçların ortalamasını alarak kestirme yöntemine **Monte Carlo yöntemi** denir; adını kumarhaneleriyle ünlü Monte Carlo şehrinden alır. Vektörizasyon ve Broadcasting bölümünde π sayısını rastgele noktalarla kestirirken bu yöntemi bir kez kullanmıştık. Kestirim tam değildir: kodu başka bir tohumla çalıştırınca başka bir sayı çıkar. Bu bölüm iki soruyu birlikte ele alıyor: rastgele sayıları bilgisayarda nasıl üretiriz ve onlarla yaptığımız kestirimlerin hatası ne kadardır? Önce NumPy'ın üretecini ve temel dağılımları tanıyacağız. Sonra büyük sayılar yasasını ve merkezi limit teoremini simülasyonla gözleyeceğiz; bu iki teorem Monte Carlo yönteminin neden çalıştığını ve hatasının neden $1/\sqrt{N}$ hızıyla küçüldüğünü açıklar. Bölüm Monte Carlo integrali ve rastgele yürüyüşle bitiyor.

14.1 Sözde Rastgele Sayılar

Bir bilgisayar programı aynı girdiyle her zaman aynı işi yapar; bu yüzden gerçekten rastgele bir sayı üretemez. Kullandığımız "rastgele" sayılar, rastgele görünen ama tamamen belirlenmiş bir diziden gelir.

Tanım 14.1 (Sözde Rastgele Sayı Üreteci). Bir **sözde rastgele sayı üreteci** (pseudorandom number generator) sonlu bir S durum kümesinden, bir $T : S \to S$ geçiş fonksiyonundan ve bir $G : S \to [0, 1)$ çıktı fonksiyonundan oluşur. Başlangıç durumu $s_0 \in S$ **tohum** (seed) ile belirlenir ve üreteç

$$s_k = T(s_{k-1}), \qquad u_k = G(s_k), \qquad k = 1, 2, \ldots$$

sayılarını üretir.

Yani bir üreteç, deterministik bir dizidir: aynı tohum her seferinde aynı $u_1, u_2, \ldots$ sayılarını verir. İyi bir üretecin sayıları, bağımsız ve $U(0, 1)$ dağılımlı (bkz. Olasılık Teorisi) sayılardan istatistiksel testlerle ayırt

edilemez. S sonlu olduğundan dizi bir yerden sonra kendini tekrarlar; tekrarlanan parçanın uzunluğuna üretecin **periyodu** denir.

Örnek 14.1 (Doğrusal Kongrüans Üreteci). En eski üreteçlerden biri olan **doğrusal kongrüans üretecinde** (linear congruential generator) durumlar $\{0, 1, \ldots, m-1\}$ kümesindedir, geçiş $x_{k+1} = (a x_k + c) \bmod m$ kuralıyla yapılır ve çıktı $u_k = x_k / m$'dir. $m = 16$, $a = 5$, $c = 3$ ve $x_0 = 7$ için dizinin ilk 20 terimini üretin ve periyodunu bulun.

Çözüm.
Kuralı bir döngüyle uyguluyoruz. Periyodu bulmak için ilk terimin listede ikinci kez göründüğü yeri arıyoruz. Karşılaştırma için aynı üreteci bir de $c = 2$ ile çalıştırıyoruz.

```
def lcg(seed, a=5, c=3, m=16, n=20):
    """x_{k+1} = (a*x_k + c) mod m dizisinin ilk n terimi."""
    x = seed
    xs = []
    for _ in range(n):
        x = (a * x + c) % m
        xs.append(x)
    return xs


xs = lcg(7)
print(xs)
print([x / 16 for x in xs[:5]])            # [0, 1) aralığına
print("periyot:", xs[1:].index(xs[0]) + 1)

ys = lcg(7, c=2)                           # c çift: kötü seçim
print(ys)
print("periyot:", ys[1:].index(ys[0]) + 1)
```

Çıktı:

```
[6, 1, 8, 11, 10, 5, 12, 15, 14, 9, 0, 3, 2, 13, 4, 7, 6, 1, 8, 11]
[0.375, 0.0625, 0.5, 0.6875, 0.625]
periyot: 16
[5, 11, 9, 15, 13, 3, 1, 7, 5, 11, 9, 15, 13, 3, 1, 7, 5, 11, 9, 15]
periyot: 8
```

İlk 16 terim $0, 1, \ldots, 15$ sayılarının her birini tam bir kez içerir ve 17. terimden itibaren dizi baştan tekrarlanır: periyot 16'dır, yani m kadar olabilecek en uzun değerdir. Bu bir rastlantı değildir. Sayılar teorisinde bilinen bir sonuca (Hull–Dobell teoremi) göre $c \neq 0$ iken periyot ancak ve ancak şu üç koşul sağlanınca m olur: c ile m aralarında asaldır, $a - 1$ sayısı m'nin her asal bölenine bölünür ve $4 \mid m$ ise $a - 1$ sayısı 4'e de bölünür. Burada $\gcd(3, 16) = 1$, m'nin tek asal böleni 2'dir ve $a - 1 = 4$ olduğundan üç koşul da sağlanır. $c = 2$ seçince ilk koşul bozulur ve dizi yalnız 8 sayı arasında döner. ■

Böyle küçük bir üretecin sayıları rastgele görünmez; gerçek üreteçler aynı fikri çok daha büyük durum kümeleri ve daha karmaşık T ve G fonksiyonlarıyla uygular. NumPy'ın varsayılan üreteci **PCG64**'tür: 128 bitlik bir durumla çalışır ve periyodu $2^{128} \approx 3{,}4 \cdot 10^{38}$'dir. Saniyede bir milyar sayı üretseniz bile bu periyodu evrenin yaşı boyunca tüketemezsiniz.

Tanım 14.2 (Generator Nesnesi). `np.random.default_rng(seed)` çağrısı, PCG64 üretecini `seed` tamsayısını tohum alarak başlatır ve bir `Generator` nesnesi döndürür. Bu nesnenin `random`, `integers`, `normal` gibi metotları üretecin durumunu ilerleterek istenen dağılımdan sayılar üretir; metotların `size` argümanı çıktının şeklidir. `seed` verilmezse tohum işletim sisteminden alınan öngörülemez bir sayıdır.

 CC BY-NC-SA 4.0

Yani bütün rastgelelik tek bir nesnede, genellikle `rng` adını verdiğimiz üreteçte toplanır. Her çağrı yeni sayılar verir, çünkü durum ilerler; aynı sayıları yeniden görmek için üreteci aynı tohumla yeniden kurmak yeterlidir:

```
import numpy as np

rng = np.random.default_rng(42)
print(rng.random(3))
print(rng.random(3))               # durum ilerledi: yeni sayılar

rng = np.random.default_rng(42)    # aynı tohumla baştan
print(rng.random(3))
print(type(rng).__name__, type(rng.bit_generator).__name__)
```

Çıktı:

```
[0.77395605 0.43887844 0.85859792]
[0.69736803 0.09417735 0.97562235]
[0.77395605 0.43887844 0.85859792]
Generator PCG64
```

İkinci `rng.random(3)` çağrısı birincinin devamıdır ve başka sayılar verir. Üreteci 42 tohumuyla yeniden kurunca ilk üç sayı aynen geri gelir. Bu notlardaki her kod sabit bir tohum kullanır; böylece kodu çalıştıran herkes aynı çıktıyı alır ve bir sonucu tekrarlamak mümkün olur.

⚠ Üreteci bir kez kurun

Üreteci bir döngünün içinde aynı tohumla yeniden kurmak, her turda aynı sayıları üretir:

```
import numpy as np

for trial in range(3):
    rng = np.random.default_rng(0)     # YANLIŞ: her turda aynı tohum
    print(rng.integers(1, 7, size=5))

rng = np.random.default_rng(0)         # DOĞRU: üreteç bir kez kurulur
for trial in range(3):
    print(rng.integers(1, 7, size=5))
```

Çıktı:

```
[6 4 4 2 2]
[6 4 4 2 2]
[6 4 4 2 2]
[6 4 4 2 2]
[1 1 1 2 5]
[4 6 4 4 6]
```

Üreteç programın başında bir kez kurulur ve sonra hep o kullanılır. Eski kodlarda sık görülen `np.random.seed(0)` ve `np.random.rand(5)` yazılışı ise bütün programın paylaştığı gizli, ortak bir üreteci kullanır. NumPy bu arayüzü yalnız eski kodlar çalışmaya devam etsin diye tutar; yeni kodda `default_rng` kullanılır.

14.2 Temel Dağılımlardan Örneklem

`Generator` nesnesinin metotları, Olasılık Teorisi notlarında tanıdığımız dağılımların hepsinden örneklem üretir. En sık kullanılanlar şunlardır:

Metot	Dağılım
`rng.random(size)`	$U(0,1)$, değerler $[0,1)$ aralığında

 CC BY-NC-SA 4.0

Metot	Dağılım
`rng.uniform(a, b, size)`	$U(a, b)$
`rng.integers(lo, hi, size)`	$\{lo, lo+1, \ldots, hi-1\}$ üzerinde kesikli düzgün
`rng.normal(mu, sigma, size)`	$N(\mu, \sigma^2)$
`rng.exponential(scale, size)`	Üstel (λ), `scale` $= 1/\lambda$
`rng.binomial(n, p, size)`	$B(n, p)$
`rng.poisson(lam, size)`	Poisson (λ)
`rng.geometric(p, size)`	Geo (p)
`rng.choice(a, size, p=p)`	`a`'nın elemanları arasından `p` olasılıklarıyla seçim
`rng.permutation(x)`	`x`'in elemanlarının rastgele bir sıralaması

```
import numpy as np

rng = np.random.default_rng(7)
print(rng.uniform(-1, 1, size=3))
print(rng.integers(1, 7, size=10))        # 7 dahil değil
print(rng.normal(10, 2, size=3))          # ortalama 10, std 2
print(rng.exponential(0.5, size=3))       # ölçek 1/lambda = 0.5
print(rng.binomial(10, 0.3, size=10))
print(rng.poisson(4, size=10))
print(rng.choice(["yazı", "tura"], size=5))
print(rng.permutation(8))
print(rng.random((2, 3)))                 # size bir şekil de olabilir
```

Çıktı:

```
[0.25019093 0.7944276  0.55137138]
[6 2 1 2 2 6 6 1 3 5]
[ 9.01558696  8.7590502  10.9796841 ]
[0.15607281 0.44988508 0.53685033]
[3 3 7 4 3 6 2 2 3 1]
[2 5 3 0 4 5 3 3 6 3]
['yazı' 'tura' 'tura' 'yazı' 'yazı']
[7 6 0 3 5 1 2 4]
[[0.84507432 0.94494817 0.90391679]
 [0.56971915 0.14545995 0.19246349]]
```

`rng.choice` bir listeden eleman seçer; `p` verilmezse bütün elemanlar eşit olasılıklıdır, `replace=False` ile aynı eleman ikinci kez seçilmez. `rng.permutation(8)` ise $0, 1, \ldots, 7$ sayılarını karıştırır. Bir diziyi yerinde karıştırmak için `rng.shuffle(x)` de vardır.

> ⚠ Parametrelerin anlamı
>
> Üç parametre sık karışır. `rng.integers(lo, hi)` üst sınırı **dahil etmez**: zar için `integers(1, 7)` yazılır (ya da `integers(1, 6, endpoint=True)`). `rng.normal(mu, sigma)` ikinci parametre olarak varyansı değil **standart sapmayı** alır: $N(1, 4)$ için `normal(1, 2)` yazılır. `rng.exponential(scale)` oran parametresi λ'yı değil **ortalama** $1/\lambda$'yı alır: Üstel (2) için `exponential(0.5)` yazılır.

Bir örneklemin dağılımına uyup uymadığını görmenin ilk yolu, örneklem ortalamasını ve örneklem varyansını (bkz. Matematiksel İstatistik) kuramsal değerlerle karşılaştırmaktır. `x.mean()` ortalamayı, `x.var(ddof=1)` ise $n-1$'e bölen örneklem varyansını verir; `ddof` yazılmazsa NumPy n'ye böler.

```
import numpy as np

rng = np.random.default_rng(2025)
```

 CC BY-NC-SA 4.0

```
n = 10**6
samples = {                       # örneklem, E(X), Var(X)
    "B(10, 0.3)": (rng.binomial(10, 0.3, n), 3.0, 2.1),
    "Poisson(4)": (rng.poisson(4, n), 4.0, 4.0),
    "Üstel(2)": (rng.exponential(0.5, n), 0.5, 0.25),
    "N(1, 4)": (rng.normal(1, 2, n), 1.0, 4.0),
    "U(0, 3)": (rng.uniform(0, 3, n), 1.5, 0.75),
}
print(f"{'dağılım':11}{'ort.':>9}{'E(X)':>7}{'varyans':>10}{'Var':>7}")
for name, (x, mean, var) in samples.items():
    print(f"{name:11}{x.mean():9.4f}{mean:7.2f}"
          f"{x.var(ddof=1):10.4f}{var:7.2f}")
```

Çıktı:

```
dağılım         ort.   E(X)   varyans    Var
B(10, 0.3)    3.0014   3.00    2.1064   2.10
Poisson(4)    3.9993   4.00    4.0023   4.00
Üstel(2)      0.4995   0.50    0.2493   0.25
N(1, 4)       1.0002   1.00    3.9952   4.00
U(0, 3)       1.5013   1.50    0.7509   0.75
```

Kuramsal değerler Olasılık Teorisi'nin moment formüllerinden gelir: binom için np ve $np(1-p)$ (teorem), Poisson için λ ve λ (teorem), üstel dağılım için $1/\lambda$ ve $1/\lambda^2$ (teorem), normal dağılım için μ ve σ^2 (teorem), $U(a,b)$ için $(a+b)/2$ ve $(b-a)^2/12$ (teorem). Bir milyon elemanlı örneklemlerde farklar binde birkaç mertebesindedir. Farkın neden tam bu büyüklükte olduğunu merkezi limit teoremi açıklayacak.

14.3 Histogram ve Ters Dönüşüm Yöntemi

Ortalama ve varyans, bir dağılımın yalnız iki sayısal özetidir. Dağılımın biçimini görmek için örneklemin histogramına bakarız.
Yoğunluk ölçeğinde histogramı Matplotlib ile Grafik Çizimi bölümünde tanımlamıştık: $x_1, \ldots, x_n$ örneklemlerinde $[a_{j-1}, a_j)$ kutusuna c_j eleman düşüyorsa histogram o kutuda

$$h(x) = \frac{c_j}{n\,(a_j - a_{j-1})}, \qquad a_{j-1} \le x < a_j$$

değerini alan basamak fonksiyonudur.
Böylece her çubuğun alanı, o kutuya düşen elemanların oranıdır; bütün elemanlar kutulara düşüyorsa çubukların toplam alanı 1'dir. Örneklem f yoğunluklu bir dağılımdan geliyorsa c_j/n oranı $P(a_{j-1} \le X < a_j) = \int_{a_{j-1}}^{a_j} f(x)\,dx$ olasılığına yakındır. Bu yüzden yoğunluk histogramı, büyük n ve dar kutular için yoğunluk fonksiyonunun grafiğine benzer. NumPy'da sayımı `np.histogram` yapar:

```
import math
import numpy as np

rng = np.random.default_rng(3)
n = 10_000
x = rng.normal(0, 1, size=n)
counts, edges = np.histogram(x, bins=8, range=(-4, 4))
print(counts, counts.sum())
print(edges)

width = np.diff(edges)                # kutu genişlikleri
dens = counts / (n * width)           # yoğunluk histogramı
print(dens.round(4))
print((dens * width).sum())           # çubukların toplam alanı

# kuramsal kutu olasılıkları: Phi(a_j) - Phi(a_{j-1})
```

 CC BY-NC-SA 4.0

```
Phi = [0.5 * (1 + math.erf(a / math.sqrt(2))) for a in edges]
print(np.diff(Phi).round(4))
```

Çıktı:

```
[   8  216 1361 3401 3433 1364  206   10] 9999
[-4. -3. -2. -1.  0.  1.  2.  3.  4.]
[0.0008 0.0216 0.1361 0.3401 0.3433 0.1364 0.0206 0.001 ]
0.9999
[0.0013 0.0214 0.1359 0.3413 0.3413 0.1359 0.0214 0.0013]
```

`np.histogram(x, bins=8, range=(-4, 4))` aralığı 8 eşit kutuya böler ve iki dizi döndürür: kutulardaki eleman sayıları ve 9 kutu sınırı. Son kutu dışındaki kutular sağdan açıktır; son kutu $[3, 4]$ kapalıdır. Aralığın dışında kalan değerler sayılmaz: on bin elemandan biri $[-4, 4]$ dışına düştüğü için sayıların toplamı 9999, çubukların toplam alanı da 0,9999'dur. Son satırda kuramsal kutu olasılıkları $\Phi(a_j) - \Phi(a_{j-1})$ var; Φ standart normal dağılım fonksiyonudur ve `math.erf` hata fonksiyonuyla $\Phi(z) = \frac{1}{2}\left(1 + \operatorname{erf}\left(z/\sqrt{2}\right)\right)$ biçiminde hesaplanır. Histogramın değerleri bu olasılıklarla iki üç basamak uyuşuyor. Matplotlib'de aynı histogramı `ax.hist(x, bins=8, range=(-4, 4), density=True)` çizer (bkz. Matplotlib ile Grafik Çizimi); yalnız `density=True`, alanı aralığın içine düşen eleman sayısına göre 1 yapar.
Düzgün dağılımlı sayılar üretebiliyorsak başka her dağılımdan da örneklem üretebiliriz. Bunu sağlayan olgu şudur.

Önerme 14.1 (Ters Dönüşüm Yöntemi). *F bir dağılım fonksiyonu, $0 < u < 1$ için $F^{-1}(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\}$ ve $U \sim U(0,1)$ olsun. O zaman $X = F^{-1}(U)$ rastgele değişkeninin dağılım fonksiyonu F'dir.*

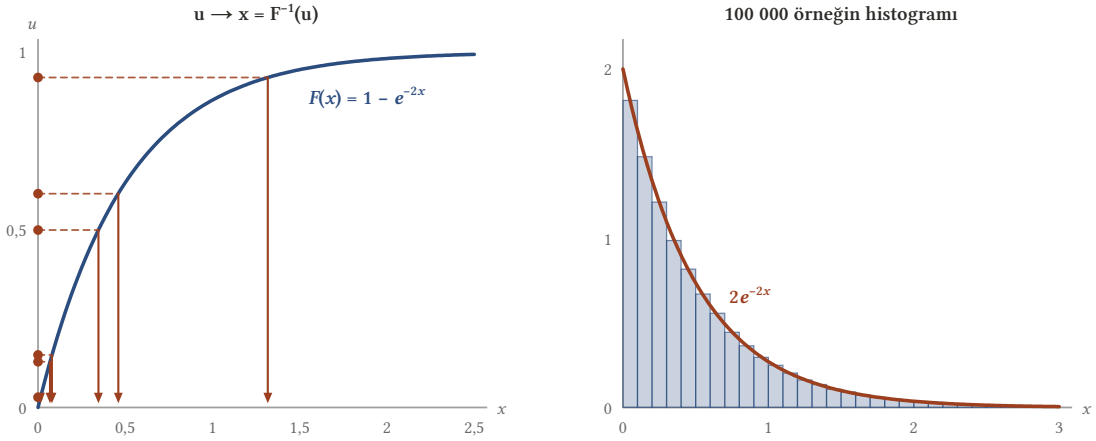
İspatı için bkz. Olasılık Teorisi. F sürekli ve kesin artan olduğunda F^{-1} sıradan ters fonksiyondur ve $u = F(x)$ denklemini x için çözerek bulunur. Önerme, `rng.random` ile ürettiğimiz sayıları istediğimiz dağılıma taşımanın bir yolunu verir.

Ters dönüşümle örneklem üç adımda

1. İstenen dağılımın F dağılım fonksiyonunu yazın.
2. $u = F(x)$ denklemini x için çözerek $F^{-1}(u)$'yu bulun.
3. `u = rng.random(n)` ile düzgün sayılar üretip `x = Finv(u)` hesaplayın.

Örnek 14.2 (Üstel Dağılımdan Ters Dönüşümle Örneklem). $\lambda = 2$ için Üstel (λ) dağılımından ters dönüşüm yöntemiyle 10^5 elemanlı bir örneklem üretin. Örneklemin ortalamasını, varyansını ve $P(X > 1)$ için verdiği kestirimi kuramsal değerlerle karşılaştırın.
Aşağıdaki şekil yöntemin işleyişini ve ürettiği örneklemin histogramını gösteriyor.

 CC BY-NC-SA 4.0



Şekil 14.1: Ters dönüşüm örneğindeki kodun verisi. Solda yöntemin kendisi: `default_rng(11)` üretecinin ilk altı u sayısı dikey eksenden $F(x) = 1 - e^{-2x}$ eğrisine, oradan yatay eksendeki $x = -\ln(1-u)/2$ noktasına taşınır. Eğri sıfırın yakınında dik olduğundan u ekseninin geniş bir parçası küçük x'lerin dar bir aralığına taşınır; x'ler bu yüzden sıfırın yakınında sıklaşır. Sağda 100 000 değerin 30 kutulu yoğunluk histogramı ve Üstel(2) yoğunluğu $2e^{-2x}$.

Çözüm.
Üç adımı uyguluyoruz.

1. Üstel (λ) dağılımının yoğunluğu $x > 0$ için $\lambda e^{-\lambda x}$'tir (bkz. Olasılık Teorisi); integralini alarak $x \geq 0$ için $F(x) = 1 - e^{-\lambda x}$ buluruz.
2. $u = 1 - e^{-\lambda x}$ denkleminden $e^{-\lambda x} = 1 - u$, yani $F^{-1}(u) = -\ln(1-u)/\lambda$ olur.
3. Kodda `u` dizisine bu formülü uyguluyoruz.

```
import numpy as np

lam = 2.0
n = 100_000
rng = np.random.default_rng(11)
u = rng.random(n)
x = -np.log(1 - u) / lam            # x = F^{-1}(u)
print(u[:4].round(4))
print(x[:4].round(4))
print(x.mean(), x.var(ddof=1))      # teori: 0.5 ve 0.25
print((x > 1).mean(), np.exp(-lam))  # P(X > 1) = e^(-2)

counts, edges = np.histogram(x, bins=30, range=(0, 3))
dens = counts / (n * np.diff(edges))
mid = (edges[:-1] + edges[1:]) / 2
print(np.abs(dens - lam * np.exp(-lam * mid)).max())
```

Çıktı:

```
[0.1286 0.4993 0.6015 0.0287]
[0.0688 0.3459 0.46   0.0146]
0.5001468651191905 0.2495862780343443
0.1359 0.1353352832366127
0.011336413931975042
```

Kuramsal değerler $E(X) = 1/\lambda = 0{,}5$, $\mathrm{Var}\,(X) = 1/\lambda^2 = 0{,}25$ ve $P(X > 1) = e^{-2} \approx 0{,}1353$'tür; örneklem bunları $0{,}5001$, $0{,}2496$ ve $0{,}1359$ olarak veriyor. Son satır, 30 kutulu yoğunluk histogramının

yüksekliklerinin kutu ortalarındaki $2e^{-2x}$ değerlerinden en çok 0,011 saptığını gösteriyor. $1-U$ de $U(0,1)$ dağılımlı olduğundan `-np.log(u) / lam` da aynı dağılımı verir. NumPy'ın `rng.exponential` metodu ise daha hızlı olan başka bir yöntem kullanır. ■

14.4 Olasılıkları ve Beklenen Değerleri Kestirmek

Giriş örneğinde bir olasılığı, deneyi tekrarlayıp göreli sıklığa bakarak kestirmiştik. Aynı fikir her beklenen değer için çalışır.

Tanım 14.3 (Monte Carlo Kestiricisi). X bir rastgele değişken ya da rastgele vektör, g bir fonksiyon ve $\theta = E[g(X)]$ sonlu olsun. $X_1, \ldots, X_N$, X ile aynı dağılımlı ve bağımsız ise

$$\hat{\theta}_N = \frac{1}{N}\sum_{i=1}^{N} g(X_i)$$

sayısına θ'nın N örneklemli **Monte Carlo kestiricisi** denir.

Yani beklenen değeri, deneyi N kez yapıp g'nin aldığı değerlerin ortalamasıyla kestiriyoruz. Bir A olayı için g, A gerçekleşince 1, gerçekleşmeyince 0 değerini alan gösterge fonksiyonu olursa $E[g(X)] = P(A)$ olur ve $\hat{\theta}_N$, A'nın N denemedeki göreli sıklığıdır. NumPy'da X_i'ler bir dizinin satırları, g vektörel bir ifade, $\hat{\theta}_N$ de bir `mean` çağrısıdır.

> Bir olasılığı simülasyonla kestirmek üç adımda
>
> 1. N denemenin hepsini tek bir diziyle üretin: her satır bir deneme olsun, örneğin `size=(N, k)`.
> 2. Olayı her satır için bir boolean değere çevirin; karşılaştırmalar ve `axis=1` ile `any`, `all`, `sum` bunun araçlarıdır.
> 3. Bu boolean dizinin `mean` değerini alın.

Örnek 14.3 (Doğum Günü Problemi). Bir yıl 365 gün olsun ve herkesin doğum günü bu günlerden birine eşit olasılıkla, birbirinden bağımsız olarak düşsün. 23 kişilik bir grupta en az iki kişinin doğum gününün aynı olma olasılığını 10^5 denemeyle kestirin ve tam değerle karşılaştırın.

Çözüm.
Üç adımı uyguluyoruz. Her deneme 23 kişinin doğum günleridir, yani `(N, 23)` şekilli bir dizinin bir satırı. Bir satırda aynı gün iki kez geçiyor mu sorusunu yanıtlamak için satırı sıralarız: sıralı bir listede eşit elemanlar yan yana gelir, dolayısıyla ardışık farklardan biri 0 olur.

```
import numpy as np

rng = np.random.default_rng(23)
N, k = 100_000, 23
days = rng.integers(0, 365, size=(N, k))     # her satır bir grup
days.sort(axis=1)                            # satırları sırala
shared = (np.diff(days, axis=1) == 0).any(axis=1)
print(shared[:6])
print(shared.mean())

exact = 1 - np.prod((365 - np.arange(k)) / 365)
print(exact)
```

Çıktı:

```
[ True  True False  True  True  True]
0.50956
0.5072972343239857
```

 CC BY-NC-SA 4.0

`days.sort(axis=1)` her satırı yerinde sıralar, `np.diff(days, axis=1)` satır içindeki ardışık farkları alır, `.any(axis=1)` her satırda sıfır fark olup olmadığını söyler. Simülasyonun kestirimi $0{,}5096$'dır. Tam değer, hiç çakışma olmaması olasılığının tümleyenidir:

$$P(\text{çakışma}) = 1 - \frac{365 \cdot 364 \cdots 343}{365^{23}} = 1 - \prod_{k=0}^{22} \frac{365-k}{365} \approx 0{,}5073.$$

Bu formülün sayma ile çıkarılışı için bkz. Olasılık Teorisi. Kodun son satırı çarpımı `np.prod` ile hesaplıyor. Kestirim ile tam değer arasındaki fark $0{,}002$ civarındadır; bu büyüklükteki farkların neden beklendiğini merkezi limit teoremi gösterecek. ■

14.5 Büyük Sayılar Yasası

Monte Carlo kestiricisinin doğru değere gittiğini garanti eden teorem büyük sayılar yasasıdır. Önce bir zarı on bin kez atıp ortalamanın nasıl değiştiğini izleyelim. Bir dizinin ilk n elemanının ortalamasına, n ilerledikçe güncellendiği için **koşu ortalaması** (running mean) denir.

```
import numpy as np

rng = np.random.default_rng(5)
n = 10_000
rolls = rng.integers(1, 7, size=(5, n))     # beş ayrı deney
k = np.arange(1, n + 1)
means = rolls.cumsum(axis=1) / k            # satır satır X̄_1, ..., X̄_n
for m in [10, 100, 1000, 10_000]:
    print(m, means[:, m - 1].round(3))
```

Çıktı:

```
10 [3.6 3.3 3.2 3.3 2.9]
100 [3.3  3.59 3.27 3.7  3.47]
1000 [3.451 3.422 3.41  3.477 3.523]
10000 [3.5   3.49  3.497 3.528 3.512]
```

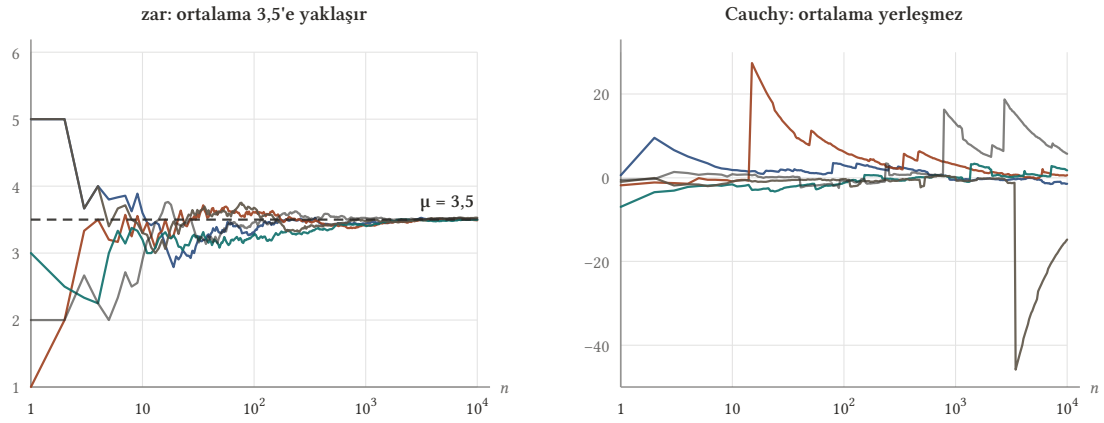
`rolls.cumsum(axis=1)` her satırın kısmi toplamlarını, yani $X_1 + \cdots + X_n$ değerlerini verir. Bunu broadcasting ile `k = [1, 2, ..., n]` dizisine bölünce bütün koşu ortalamaları tek ifadeyle hesaplanır. On atışta ortalamalar $2{,}9$ ile $3{,}6$ arasında dağınıktır; on bin atışta hepsi $3{,}5$'e yüzde birden daha yakındır. Bir zarın beklenen değeri $(1+2+\cdots+6)/6 = 3{,}5$'tir.

Teorem 14.1 (Güçlü Büyük Sayılar Yasası). *$X_1, X_2, \ldots$ bağımsız, aynı dağılımlı ve $E \mid X_1 \mid < \infty$ olan rastgele değişkenler, $\mu = E(X_1)$ olsun. O zaman olasılığı 1 olan bir olay üzerinde*

$$\bar{X}_n = \frac{X_1 + X_2 + \cdots + X_n}{n} \to \mu \qquad (n \to \infty)$$

olur.
Aşağıdaki şeklin solunda yukarıdaki kodun beş zar deneyi, sağında ise teoremin hipotezini sağlamayan bir dağılımın beş yolu var.

 CC BY-NC-SA 4.0



Şekil 14.2: Koşu ortalamaları, yatay eksen logaritmik. Solda büyük sayılar yasası örneğindeki beş zar deneyi: $\bar{X}_n$ her yolda 3,5'e yerleşir. Sağda uyarı kutusundaki Cauchy kodunun beş yolu: arada bir gelen çok büyük bir değer ortalamayı sıçratır (bir yolda −46'ya kadar) ve yollar hiçbir sayıya yerleşmez; Cauchy dağılımının beklenen değeri yoktur.

Yani bir deneyi yeterince çok tekrarlarsak sonuçların ortalaması neredeyse kesinlikle beklenen değere yaklaşır. Monte Carlo kestiricisi $\hat{\theta}_N$, $g(X_i)$ değerlerinin ortalaması olduğundan $E \mid g(X) \mid < \infty$ ise $\hat{\theta}_N \to \theta$ olur. Teoremin zayıf biçimi ve ispatı için bkz. Matematiksel İstatistik, güçlü biçimi için aynı bölümdeki teorem.

⚠ Beklenen değeri olmayan dağılımlar

Teoremdeki $E \mid X_1 \mid < \infty$ koşulu bir ayrıntı değildir. Yoğunluğu $\frac{1}{\pi(1+x^2)}$ olan standart Cauchy dağılımında $\int \mid x \mid /(\pi(1+x^2))\,dx$ integrali ıraksar, yani beklenen değer yoktur:

```
import numpy as np

rng = np.random.default_rng(6)
n = 10_000
x = rng.standard_cauchy(size=(5, n))
means = x.cumsum(axis=1) / np.arange(1, n + 1)
for m in [10, 100, 1000, 10_000]:
    print(m, means[:, m - 1].round(2))
print(np.abs(x).max().round(1))
```

Çıktı:

```
10 [ 1.87 -0.9  -1.58 -0.85  0.87]
100 [ 2.96  6.21 -1.11 -0.34 -0.69]
1000 [-0.48  3.14  0.08 -0.72 13.03]
10000 [ -1.41   0.62   1.78 -14.78   5.75]
154047.4
```

Koşu ortalamaları hiçbir sayıya yerleşmez. Arada bir gelen çok büyük bir değer (beş yoldaki elli bin sayının mutlak değerce en büyüğü yüz elli binden büyük) ortalamayı birden sıçratır. Hatta n tane bağımsız standart Cauchy değişkenin ortalaması yine standart Cauchy dağılımlıdır: ortalama almak hiçbir şeyi iyileştirmez. Bir Monte Carlo hesabında ortalamalar yerleşmiyorsa, kestirilen beklenen değerin sonlu olup olmadığını kontrol edin.

14.6 Merkezi Limit Teoremi

Büyük sayılar yasası ortalamanın μ'ye gittiğini söyler ama ne kadar hızlı gittiğini söylemez. Bu soruyu merkezi limit teoremi yanıtlar: $\bar{X}_n - \mu$ farkı $\sigma/\sqrt{n}$ mertebesindedir ve yaklaşık normal dağılımlıdır.

 CC BY-NC-SA 4.0

Bunu çok çarpık bir dağılımla, Üstel (1) ile deneyelim; bu dağılımda $\mu = \sigma = 1$'dir. Her n için 10^5 kez n elemanlı bir örneklem alıp ortalamasını $Z = (\bar{X}_n - \mu)/(\sigma/\sqrt{n})$ biçiminde standartlaştırıyor ve Z'lerin histogramını standart normal yoğunlukla karşılaştırıyoruz:

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(8)
reps = 100_000                       # her n için tekrar sayısı
t = np.linspace(-4, 4, 201)
phi = np.exp(-t**2 / 2) / np.sqrt(2 * np.pi)

fig, axs = plt.subplots(1, 3, figsize=(12, 3.5), sharey=True,
                        layout="constrained")
for ax, n in zip(axs, [1, 4, 32]):
    x = rng.exponential(1.0, size=(reps, n))     # mu = sigma = 1
    z = (x.mean(axis=1) - 1.0) * np.sqrt(n)      # standartlaştır
    ax.hist(z, bins=40, range=(-4, 4), density=True, alpha=0.6)
    ax.plot(t, phi, "k")
    ax.set_title(f"n = {n}")
    p1, p0 = np.mean(z <= -1), np.mean(z <= 0)
    print(f"n = {n:2d}: P(Z <= -1) ~ {p1:.4f}, P(Z <= 0) ~ {p0:.4f}")
fig.savefig("merkezi_limit.png", dpi=150)
```

Çıktı:

```
n =  1: P(Z <= -1) ~ 0.0000, P(Z <= 0) ~ 0.6308
n =  4: P(Z <= -1) ~ 0.1427, P(Z <= 0) ~ 0.5657
n = 32: P(Z <= -1) ~ 0.1558, P(Z <= 0) ~ 0.5205
```

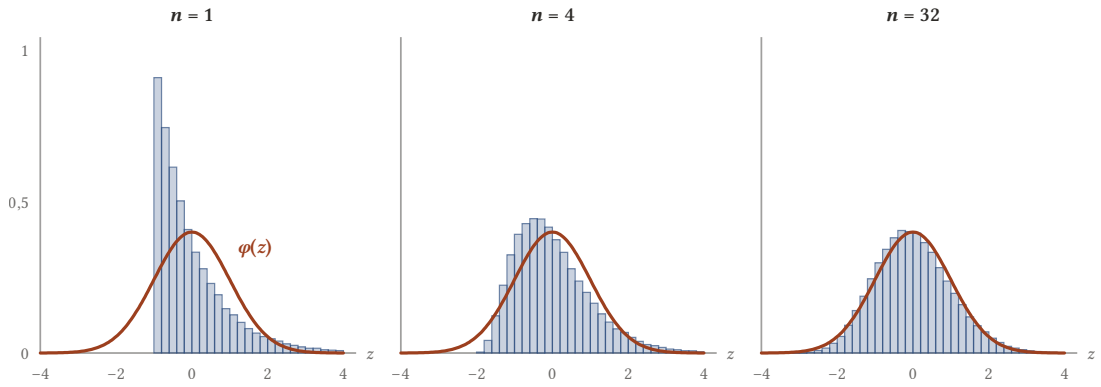
`rng.exponential(1.0, size=(reps, n))` her satırı bir örneklem olan bir dizi üretir ve `x.mean(axis=1)` satır ortalamalarını alır. Standart normal dağılımda $\Phi(-1) \approx 0{,}1587$ ve $\Phi(0) = 0{,}5$'tir. $n = 1$'de $Z = X - 1 \geq -1$ olduğundan $P(Z \leq -1) = 0$'dır ve $P(Z \leq 0) = 1 - e^{-1} \approx 0{,}632$'dir. n büyüdükçe iki olasılık da normal değerlerine yaklaşıyor.

Teorem 14.2 (Merkezi Limit Teoremi). *$X_1, X_2, \ldots$ bağımsız, aynı dağılımlı, beklenen değeri μ ve varyansı $0 < \sigma^2 < \infty$ olan rastgele değişkenler ve Φ standart normal dağılım fonksiyonu olsun. O zaman her $z \in \mathbb{R}$ için*

$$\lim_{n \to \infty} P\left(\frac{\bar{X}_n - \mu}{\sigma / \sqrt{n}} \leq z \right) = \Phi(z)$$

olur.
Aşağıdaki şekil yukarıdaki kodun çizdiği üç histogramdır.

 CC BY-NC-SA 4.0



Şekil 14.3: Merkezi limit teoremi kodunun çizdiği üç histogram: Üstel(1) dağılımından alınan n *elemanlı örneklemlerin standartlaştırılmış ortalaması* $Z = \sqrt{n}\,(\bar{X}_n - 1)$*, her* n *için 100 000 tekrar.* n = *1'de histogram üstel yoğunluğun kaydırılmışıdır ve çok çarpıktır;* n = *4'te çarpıklık azalır;* n = *32'de histogram standart normal yoğunluk* φ(z) *eğrisine neredeyse oturur.*

Yani ortalamanın μ'den sapması, n büyüdükçe $\sigma/\sqrt{n}$ ölçekli bir normal dağılıma benzer; başlangıçtaki dağılım ne kadar çarpık olursa olsun. Özel olarak

$$P\left(\mid \bar{X}_n - \mu \mid \leq 1{,}96\,\frac{\sigma}{\sqrt{n}}\right) \approx \Phi(1{,}96) - \Phi(-1{,}96) \approx 0{,}95$$

olur. Moment üreten fonksiyonun var olduğu durumda ispat için bkz. Matematiksel İstatistik. Önceki bölümlerdeki farklar bu yüzden beklenen büyüklükteydi. Örneğin doğum günü probleminde bir denemenin sonucu olasılığı $p \approx 0{,}507$ olan bir Bernoulli değişkenidir, yani $\sigma = \sqrt{p(1-p)} \approx 0{,}5$'tir. $N = 10^5$ için $\sigma/\sqrt{N} \approx 0{,}0016$ olur ve bulduğumuz 0,002'lik fark bu ölçekte sıradandır.

Örnek 14.4 (On İki Düzgün Sayıdan Yaklaşık Normal Sayı). $U_1, \ldots, U_{12}$ bağımsız ve $U(0,1)$ dağılımlı olmak üzere $Z = U_1 + \cdots + U_{12} - 6$ olsun. 10^6 tane Z üretin. $z = 0{,}5$, 1 ve 2 için $P(Z \leq z)$ kestirimlerini $\Phi(z)$ ile, $P(\mid Z \mid > 3)$ kestirimini de standart normal dağılımın değeriyle karşılaştırın.

Çözüm.
Önce Z'nin ortalama ve varyansını bulalım. $E(U_i) = 1/2$ ve $\mathrm{Var}\,(U_i) = 1/12$ olduğundan beklenen değerin lineerliği ile $E(Z) = 12 \cdot \frac{1}{2} - 6 = 0$, bağımsız değişkenlerin varyansları toplandığı için de $\mathrm{Var}\,(Z) = 12 \cdot \frac{1}{12} = 1$ olur. Merkezi limit teoremine göre Z yaklaşık $N(0,1)$ dağılımlıdır. Φ'yi `math.erf` ile hesaplıyoruz.

```
import math
import numpy as np


def Phi(z):
    """Standart normal dağılım fonksiyonu."""
    return 0.5 * (1 + math.erf(z / math.sqrt(2)))


rng = np.random.default_rng(12)
N = 10**6
Z = rng.random((N, 12)).sum(axis=1) - 6     # her satır 12 düzgün sayı
print(Z.mean(), Z.var(ddof=1))
for z in [0.5, 1.0, 2.0]:
    print(z, np.mean(Z <= z), Phi(z))
print(np.mean(np.abs(Z) > 3), 2 * (1 - Phi(3)))
print(np.abs(Z).max())
```

Çıktı:

```
0.0008496661665800563 1.002624404897434
0.5 0.688608 0.691462461274013
1.0 0.838893 0.8413447460685428
2.0 0.977393 0.9772498680518209
0.002056 0.002699796063260207
4.557258791575404
```

Örneklemin ortalaması ve varyansı 0 ve 1'e yakındır. $P(Z \le z)$ kestirimleri $\Phi(z)$ ile iki basamak uyuşuyor: dağılımın orta kısmı için on iki terim yeterlidir. Kuyrukta durum farklıdır: $P(|Z| > 3)$ için kestirim $0{,}0021$, normal değer $0{,}0027$'dir. Bu fark simülasyon hatası değildir; toplamın tam dağılımından hesaplanan değer de $0{,}0020$ civarındadır. Z hiçbir zaman $[-6, 6]$ aralığının dışına çıkamaz, oysa normal dağılım her değeri alabilir; bir milyon Z arasında en büyük mutlak değer yalnız $4{,}56$'dır. Merkezi limit teoremi dağılımın merkezini iyi açıklar, uç olayların olasılıklarını ise ancak büyük n için. ■

14.7 Monte Carlo ile İntegral

Bir integral de bir beklenen değer olarak yazılabilir; bu gözlem Monte Carlo yöntemini integral hesabının bir aracına çevirir. $U \sim U(a,b)$ ise U'nun yoğunluğu $[a,b]$ üzerinde $1/(b-a)$ olduğundan

$$E[f(U)] = \int_a^b f(x)\,\frac{dx}{b-a}, \qquad \text{yani} \qquad \int_a^b f(x)\,dx = (b-a)\,E[f(U)]$$

olur. Aynı akıl yürütme birden çok boyutta da geçerlidir.

Tanım 14.4 (Monte Carlo İntegrali). $D \subset \mathbb{R}^d$ hacmi $|D|$ olan sınırlı bir bölge ve f, D üzerinde integrallenebilir bir fonksiyon olsun. $U_1, \dots, U_N$, D içinde bağımsız ve düzgün dağılımlı noktalar ise

$$\hat{I}_N = \frac{|D|}{N} \sum_{i=1}^{N} f(U_i)$$

sayısına $I = \int_D f$ integralinin N noktalı **Monte Carlo kestirimi** denir.

Yani integral, bölgenin hacmiyle f'nin D üzerindeki ortalama değerinin (bkz. İntegral Calculus) çarpımıdır ve ortalama değeri rastgele noktalardaki değerlerin ortalamasıyla kestiriyoruz. Bu, Monte Carlo kestiricisinin (Tanım 14.3) $g = |D|\,f$ için özel hâlidir. İlk örnek olarak e^{-x^2}'nin $[0,1]$ üzerindeki integralini alalım. Bu fonksiyonun ilkeli elemanter fonksiyonlarla yazılamaz; integralin değeri hata fonksiyonuyla $\frac{\sqrt{\pi}}{2}\,\mathrm{erf}\,(1) \approx 0{,}746824$'tür.

```
import math
import numpy as np


def f(x):
    return np.exp(-x**2)

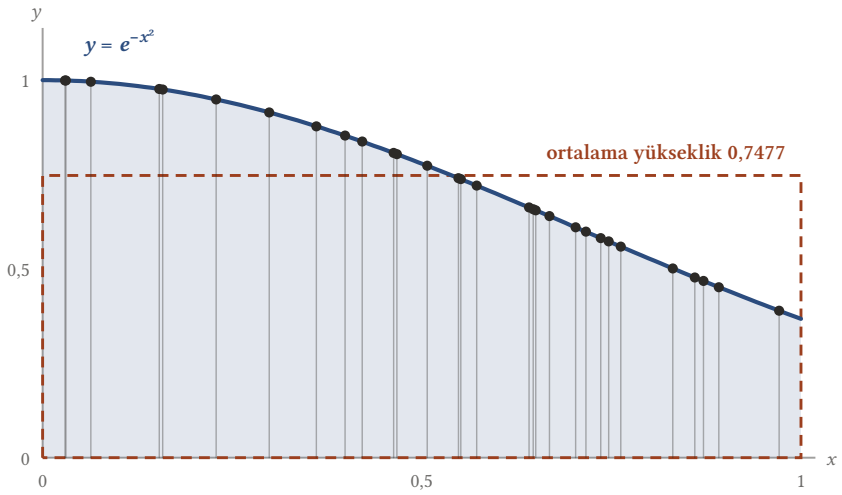

rng = np.random.default_rng(14)
N = 10_000
u = rng.random(N)                  # [0, 1) aralığında düzgün noktalar
y = f(u)
I_hat = y.mean()                   # (b - a) = 1 olduğu için
print(u[:4].round(4))
print(y[:4].round(4))
exact = math.sqrt(math.pi) / 2 * math.erf(1)
print(I_hat, exact, abs(I_hat - exact))
```

Çıktı:

 CC BY-NC-SA 4.0

```
[0.831  0.3609 0.7027 0.8601]
[0.5013 0.8778 0.6103 0.4772]
0.7476505933702069 0.746824132812427 0.0008264605577799067
```

$b - a = 1$ olduğundan kestirim, f'nin rastgele noktalardaki değerlerinin ortalamasıdır. On bin noktayla bulunan 0,74765, gerçek değerden yaklaşık $8 \cdot 10^{-4}$ uzaktadır. Aşağıdaki şekil bu hesabı geometrik olarak gösteriyor.



Şekil 14.4: Monte Carlo integral kodunun verisi. Noktalar, `default_rng(14)` ile üretilen ilk 30 u sayısında e^{-u^2} değerleridir. Kesikli dikdörtgenin yüksekliği, 10 000 değerin ortalaması olan 0,7477 sayısıdır; dikdörtgenin alanı, eğrinin altında kalan taralı alanın (0,7468) kestirimidir. Eğrinin dikdörtgenin üstüne taşan parçasıyla dikdörtgenin içinde eğrinin üstünde kalan boşluğun alanları yaklaşık eşittir.

Kestirimin hatasını kesin olarak bilemeyiz, çünkü bilseydik integrali de bilirdik. Ama hatanın tipik büyüklüğünü kestirebiliriz.

Önerme 14.2 (Monte Carlo Kestiriminin Hatası). *$\hat{I}_N, I = \int_D f$ integralinin bir Monte Carlo kestirimi (Tanım 14.4) ve $0 < \sigma^2 = \mathrm{Var}\left(\, f(U_1) \,\right) < \infty$ olsun. O zaman*

$$E(\hat{I}_N) = I, \qquad \mathrm{Var}\,(\hat{I}_N) = \frac{|\, D \,|^2\, \sigma^2}{N}$$

olur. Ayrıca $N \to \infty$ iken

$$P\left(|\, \hat{I}_N - I \,| \le 1{,}96\, \frac{|\, D \,|\, \sigma}{\sqrt{N}} \right) \to \Phi(1{,}96) - \Phi(-1{,}96) \approx 0{,}95$$

olur.

İspat.
$Y_i = |\, D \,|\, f(U_i)$ diyelim. U_i'ler bağımsız ve aynı dağılımlı olduğundan Y_i'ler de öyledir ve $\hat{I}_N = \bar{Y}_N$ olur. U_i'nin yoğunluğu D üzerinde $1/|\, D \,|$ olduğundan

$$E(Y_i) = |\, D \,| \int_D f(x)\, \frac{dx}{|\, D \,|} = I, \qquad \mathrm{Var}\,(Y_i) = |\, D \,|^2\, \sigma^2$$

olur. Bağımsız ve aynı ortalamalı, aynı varyanslı gözlemlerin ortalaması için $E(\bar{Y}_N) = E(Y_1)$ ve $\mathrm{Var}\,(\bar{Y}_N) = \mathrm{Var}\,(Y_1)/N$'dir (bkz. Matematiksel İstatistik); bu ilk iki eşitliği verir.
Son iddia için $Z_N = \sqrt{N}\,(\bar{Y}_N - I)/(|\, D \,|\, \sigma)$ diyelim; iddiadaki olay $|\, Z_N \,| \le 1{,}96$ olayıdır. Y_i'lerin beklenen değeri I, standart sapması $|\, D \,|\, \sigma$ olduğundan merkezi limit teoremi (Teorem 14.2) her z için $P(Z_N \le z) \to \Phi(z)$ verir. Φ sürekli olduğu için $P(Z_N < z) \to \Phi(z)$ de geçerlidir. Böylece

 CC BY-NC-SA 4.0

$$P(|\ Z_N\ | \le 1{,}96) = P(Z_N \le 1{,}96) - P(Z_N < -1{,}96)$$
$$\to \Phi(1{,}96) - \Phi(-1{,}96)$$

olur. Bu limitin değeri $0{,}950004\ldots$*, yani yaklaşık* $0{,}95$*'tir.* ■

Yani kestirimin standart hatası (bkz. Matematiksel İstatistik) $|\ D\ |\sigma/\sqrt{N}$'dir: hata N ile değil $\sqrt{N}$ ile küçülür. Hatayı 10 kat küçültmek için 100 kat fazla nokta gerekir. σ'yı bilmediğimiz için onu, zaten hesapladığımız $f(U_i)$ değerlerinin örneklem standart sapması s ile kestiririz. Böylece

$$\hat{I}_N \pm 1{,}96\,\frac{|\ D\ |\,s}{\sqrt{N}}$$

aralığı, I için yaklaşık %95 düzeyli bir güven aralığı (bkz. Matematiksel İstatistik) olur. Önceki hesaba standart hatayı ekleyelim:

```
import math
import numpy as np


def f(x):
    return np.exp(-x**2)


rng = np.random.default_rng(14)
N = 10_000
y = f(rng.random(N))
I_hat = y.mean()
s = y.std(ddof=1)                  # sigma'nın kestirimi
se = s / math.sqrt(N)              # standart hata
print(s, se)
print(f"I = {I_hat:.4f} ± {1.96 * se:.4f}")
print(I_hat - 1.96 * se, I_hat + 1.96 * se)
```

Çıktı:

```
0.2013487113344417 0.002013487113344417
I = 0.7477 ± 0.0039
0.7437041586280518 0.751597028112362
```

Standart hata $0{,}0020$'dir ve %95 güven aralığı $[0{,}7437;\ 0{,}7516]$ gerçek değer $0{,}7468$'i içerir. Gerçek hata $0{,}0008$, standart hatadan küçüktür. Güven aralığının ne demek olduğunu bir deneyle görelim: aynı hesabı $N = 1000$ noktayla 20 kez bağımsız olarak yapıp her seferinde aralığın gerçek değeri içerip içermediğine bakalım.

```
import math
import numpy as np


def f(x):
    return np.exp(-x**2)


exact = math.sqrt(math.pi) / 2 * math.erf(1)
rng = np.random.default_rng(16)
N, runs = 1000, 20
hits = 0
for k in range(runs):
    y = f(rng.random(N))
    I_hat = y.mean()
    h = 1.96 * y.std(ddof=1) / math.sqrt(N)    # yarı genişlik
```

 CC BY-NC-SA 4.0

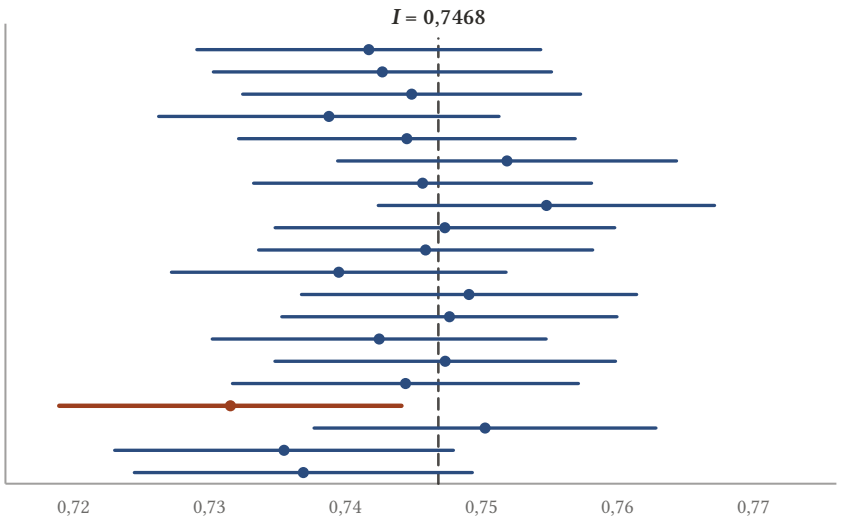
```
    ok = I_hat - h <= exact <= I_hat + h
    hits += ok
    if k < 3:
        print(f"[{I_hat - h:.4f}, {I_hat + h:.4f}]", ok)
print(hits, "/", runs)

# 10 000 tekrar: tek bir (10000, N) dizisiyle
Y = f(rng.random((10_000, N)))
I_hats = Y.mean(axis=1)
hs = 1.96 * Y.std(axis=1, ddof=1) / math.sqrt(N)
print(np.mean(np.abs(I_hats - exact) <= hs))
```

Çıktı:

```
[0.7291, 0.7544] True
[0.7303, 0.7551] True
[0.7324, 0.7573] True
19 / 20
0.9492
```

Yirmi aralıktan 19'u gerçek değeri içeriyor. İkinci kısımda aynı deneyi tek bir `(10000, 1000)` şekilli diziyle on bin kez yapıyoruz; aralıkların %94,9'u gerçek değeri yakalıyor. "%95 güven" sözü tek bir aralık hakkında değil, aralığı üreten yöntem hakkındadır.



Şekil 14.5: Güven aralığı kodunun ilk döngüsündeki 20 bağımsız tekrar: her satırda $N = 1000$ noktalı kestirim (nokta) ve %95 güven aralığı. Kesikli çizgi gerçek değerdir. 19 aralık onu içerir, biri (kırmızı) ıskalar. Aralıklar tekrardan tekrara yer değiştirir; uzun vadede yaklaşık %95'i gerçek değeri yakalar.

Monte Carlo integrali dört adımda

1. Bölgeyi hacmi bilinen bir kutunun içine alın; f'yi kutunun bölge dışında kalan kısmında 0 kabul edin. Böylece integral kutu üzerinde bir integral olur.
2. Kutuda `rng.uniform` ile N tane düzgün dağılımlı nokta üretin.
3. Fonksiyon değerlerini `vals` dizisine yazıp kestirimi `vol * vals.mean()` ile hesaplayın.
4. Standart hatayı `vol * vals.std(ddof=1) / np.sqrt(N)` ile bulun ve sonucu kestirim $\pm 1{,}96 \cdot$ standart hata aralığıyla bildirin.

 CC BY-NC-SA 4.0

Örnek 14.5 (Yarım Halka Üzerinde İki Katlı İntegral). R, üst yarı düzlemde $x^2+y^2=1$ ve $x^2+y^2=4$ çemberleri arasında kalan bölge olsun. $\iint_R (3x+4y^2)\,dA$ integralini 10^6 noktalı Monte Carlo yöntemiyle, %95 güven aralığıyla kestirin ve kutupsal koordinatlarla bulunan tam değerle karşılaştırın.

Çözüm.
Dört adımı uyguluyoruz.

1. R, $[-2,2]\times[0,2]$ kutusunun içindedir; kutunun alanı 8'dir. Kutu üzerinde R içinde $3x+4y^2$, dışında 0 olan fonksiyonun integralini alacağız.
2. Kutuda 10^6 nokta üretiyoruz; x ve y koordinatları ayrı ayrı düzgün dağılımlıdır.
3. `np.where` noktanın R'de olup olmadığına göre fonksiyon değerini ya da 0'ı seçer.
4. Standart hatayı ve güven aralığını hesaplıyoruz.

```
import numpy as np

rng = np.random.default_rng(18)
N = 10**6
x = rng.uniform(-2, 2, N)              # kutu: [-2, 2] x [0, 2]
y = rng.uniform(0, 2, N)
r2 = x**2 + y**2
in_R = (1 <= r2) & (r2 <= 4)           # nokta yarım halkada mı?
vals = np.where(in_R, 3 * x + 4 * y**2, 0.0)
vol = 4 * 2                            # kutunun alanı
I_hat = vol * vals.mean()
se = vol * vals.std(ddof=1) / np.sqrt(N)
print(in_R.mean(), 3 * np.pi / 16)     # R'nin kutudaki payı
print(I_hat, se)
print(I_hat - 1.96 * se, I_hat + 1.96 * se)
print(15 * np.pi / 2)
```

Çıktı:

```
0.589537 0.5890486225480862
23.623947891599368 0.038170669799821234
23.54913337879172 23.698762404407017
23.561944901923447
```

İlk satır bir kontrol: noktaların R'ye düşme oranı $0{,}5895$'tir ve R'nin alanının kutunun alanına oranı olan $\frac{(4\pi-\pi)/2}{8}=\frac{3\pi}{16}\approx 0{,}5890$'a yakındır. Kestirim $23{,}62$, standart hata $0{,}038$'dir. Kutupsal koordinatlarla integral

$$\int_0^{2\pi}\!\!\int_1^2 (3r\cos\theta + 4r^2\sin^2\theta)\, r\,dr\,d\theta = 0 + 4\cdot\frac{\pi}{2}\cdot\frac{15}{4} = \frac{15\pi}{2}\approx 23{,}562$$

olarak bulunur (ayrıntılı hesap için bkz. İntegral Calculus). Tam değer $[23{,}549;\ 23{,}699]$ güven aralığının içindedir. ■

14.8 Monte Carlo Ne Zaman Kazanır?

$1/\sqrt{N}$ hızı yavaş bir hızdır. Bir boyutlu integrallerde Nümerik Analiz notlarındaki yamuk kuralı çok daha iyi sonuç verir. Aşağıdaki kod iki yöntemi aynı sayıda fonksiyon değeriyle karşılaştırıyor. Monte Carlo hatası rastgele olduğu için her N değerinde hesabı 400 kez tekrarlayıp hataların karelerinin ortalamasının karekökünü alıyoruz. Bunu $\sigma/\sqrt{N}$ ile karşılaştırıyoruz; $\sigma^2 = E[f(U)^2]-I^2$ ve $E[f(U)^2]=\int_0^1 e^{-2x^2}\,dx$ yine hata fonksiyonuyla hesaplanır. Yamuk kuralı için Sayısal Türev ve İntegral bölümünde gördüğümüz `np.trapezoid` fonksiyonunu kullanıyoruz.

 CC BY-NC-SA 4.0

```
import math
import numpy as np


def f(x):
    return np.exp(-x**2)

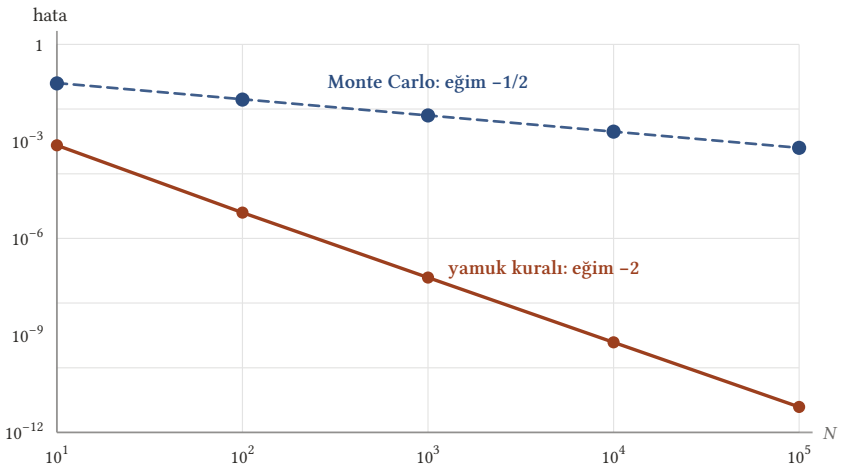

exact = math.sqrt(math.pi) / 2 * math.erf(1)
Ef2 = math.sqrt(math.pi / 8) * math.erf(math.sqrt(2))   # E[f(U)^2]
sigma = math.sqrt(Ef2 - exact**2)
print("sigma =", sigma)

rng = np.random.default_rng(17)
reps = 400                          # her N için tekrar sayısı
print(f"{'N':>7}{'MC hatası':>11}{'sigma/kök N':>13}{'yamuk':>10}")
for N in [10, 100, 1000, 10_000, 100_000]:
    est = np.array([f(rng.random(N)).mean() for _ in range(reps)])
    rms = np.sqrt(np.mean((est - exact) ** 2))
    x = np.linspace(0, 1, N)        # yamuk kuralı da N nokta kullanır
    trap = np.trapezoid(f(x), x)
    print(f"{N:7d}{rms:11.2e}{sigma / math.sqrt(N):13.2e}"
          f"{abs(trap - exact):10.2e}")
```

Çıktı:

```
sigma = 0.2009918438899211
      N  MC hatası  sigma/kök N     yamuk
     10   6.20e-02     6.36e-02  7.57e-04
    100   1.96e-02     2.01e-02  6.26e-06
   1000   6.37e-03     6.36e-03  6.14e-08
  10000   2.01e-03     2.01e-03  6.13e-10
 100000   6.33e-04     6.36e-04  6.13e-12
```

Monte Carlo sütunu $\sigma/\sqrt{N}$ sütunuyla neredeyse aynıdır: Önerme 14.2 tam olarak gözleniyor. N her 10 katına çıktığında Monte Carlo hatası yalnız $\sqrt{10} \approx 3{,}16$ kat küçülür, yamuk kuralının hatası ise 100 kat küçülür; çünkü yamuk kuralının hatası h^2 ile, yani N^{-2} ile orantılıdır. Bir boyutta Monte Carlo açık ara kaybeder.



Şekil 14.6: Hata hızı kodunun tablosu, iki ekseni de logaritmik ölçekte. Mavi: N noktalı Monte Carlo kestiriminin 400 tekrar üzerinden karekök ortalama kare hatası (noktalar); kesikli çizgi kuramsal $\sigma/\sqrt{N}$ değeridir ve noktalar onun üstüne oturur. Kırmızı: aynı sayıda nokta kullanan yamuk kuralının hatası. Bir boyutta yamuk kuralı çok daha hızlıdır; Monte Carlo'da bir basamak kazanmak için nokta sayısını 100 ile çarpmak gerekir.

 CC BY-NC-SA 4.0

Tablo yüksek boyutta tersine döner. d boyutlu bir küpte her eksende m nokta kullanan bir ızgara $N = m^d$ fonksiyon değeri gerektirir ve yamuk kuralının hatası $h^2 \sim m^{-2} = N^{-2/d}$ ile küçülür. $d = 4$'te bu Monte Carlo'nun $N^{-1/2}$ hızına eşittir, daha yüksek boyutta ondan yavaştır. Monte Carlo hatasının hızı ise boyuttan bağımsızdır: standart hata formülünde (Önerme 14.2) d hiç geçmez. Izgara yöntemlerinin boyut arttıkça çaresiz kalmasına **boyutun laneti** (curse of dimensionality) denir.

Örnek 14.6 (On Boyutlu Birim Topun Hacmi). $\mathbb{R}^{10}$'daki birim topun hacmi $\pi^5/120 \approx 2{,}5502$'dir. Bu hacmi $[-1,1]^{10}$ küpünde 10^6 noktalı Monte Carlo yöntemiyle kestirin ve sonucu, yaklaşık aynı sayıda (4^{10}) nokta kullanan orta nokta ızgarasının verdiği değerle karşılaştırın.

Çözüm.
Topun hacmi, topun içinde 1, dışında 0 olan fonksiyonun küp üzerindeki integralidir. Küpün hacmi $2^{10} = 1024$ olduğundan Monte Carlo kestirimi, topa düşen noktaların oranının 1024 katıdır. Izgarada ise her eksen $[-1,1]$ aralığının dört eşit parçasının orta noktalarına, yani $\pm 0{,}25$ ve $\pm 0{,}75$'e bölünür. Her ızgara noktası hacmi $(2/4)^{10} = 1/1024$ olan bir küçük küpü temsil eder.

```
import math
import numpy as np

d, N = 10, 10**6
rng = np.random.default_rng(19)
P = rng.uniform(-1, 1, size=(N, d))       # [-1, 1]^10 küpü, hacmi 2^10
inside = (P**2).sum(axis=1) <= 1
V_hat = 2**d * inside.mean()
se = 2**d * inside.std(ddof=1) / math.sqrt(N)
print(inside.sum(), V_hat, se)
print(math.pi**5 / 120)                   # tam değer

# Orta nokta kuralı: her eksende m = 4 nokta, toplam 4^10 nokta
m = 4
c = -1 + (np.arange(m) + 0.5) * 2 / m     # -0.75, -0.25, 0.25, 0.75
G = np.stack(np.meshgrid(*[c] * d, indexing="ij"), axis=-1)
G = G.reshape(-1, d)
grid_in = (G**2).sum(axis=1) <= 1
print(len(G), grid_in.sum(), grid_in.sum() * (2 / m) ** d)
```

Çıktı:

```
2480 2.53952 0.05093154142760173
2.550164039877345
1048576 1024 1.0
```

Izgarayı kuran satırda `[c] * d`, `c` dizisinin 10 kopyasından oluşan bir listedir. Başındaki `*` bu listeyi açar ve 10 elemanını `meshgrid`'e ayrı ayrı argüman olarak verir; yani `np.meshgrid(c, c, ..., c)` yazmış oluruz. `meshgrid` her biri $(4, 4, \ldots, 4)$ şekilli 10 koordinat dizisi döndürür. `np.stack(..., axis=-1)` bunları yeni bir son eksen boyunca yığar ve $(4, \ldots, 4, 10)$ şekilli bir dizi kurar. `reshape(-1, d)` ise bu diziyi $(4^{10}, 10)$ şekline getirir: her satır bir ızgara noktasının 10 koordinatıdır.
Bir milyon noktanın yalnız 2480'i topun içine düştü; topun küpteki payı $2{,}55/1024 \approx 0{,}0025$ kadardır. Kestirim 2,540, standart hatası 0,051'dir ve tam değer 2,550 güven aralığının içindedir. Izgara ise 1,0 veriyor, yani gerçek değerin yarısından az. Nedeni basittir: bir ızgara noktasının koordinatlarının karelerinin toplamı en az $10 \cdot 0{,}25^2 = 0{,}625$'tir ve tek bir koordinat $\pm 0{,}75$ olursa toplam $0{,}625 + 0{,}5 = 1{,}125 > 1$ olur. Topun içine yalnız bütün koordinatları $\pm 0{,}25$ olan $2^{10} = 1024$ nokta düşer. Her eksende yalnız dört nokta kullanabilen bir ızgara, topun biçimini göremez. Daha iyi bir ızgara için her eksende 10 nokta bile 10^{10} fonksiyon değeri demektir. Topun hacmi için genel formül $\pi^{d/2}/\Gamma(d/2+1)$'dir; $d = 10$ için $\pi^5/5! = \pi^5/120$ olur. ■

 CC BY-NC-SA 4.0

14.9 Rastgele Yürüyüş

Şimdiye kadar bağımsız rastgele sayıları tek tek kullandık. Onları art arda toplayınca zamanla değişen rastgele bir büyüklük elde ederiz. Bunun en basit örneği rastgele yürüyüştür.

Tanım 14.5 (Basit Simetrik Rastgele Yürüyüş). $X_1, X_2, \ldots$ bağımsız ve her biri 1 ve -1 değerlerini $\frac{1}{2}$'şer olasılıkla alan rastgele değişkenler olsun. $S_0 = 0$ ve $n \geq 1$ için $S_n = X_1 + \cdots + X_n$ ile tanımlanan (S_n) dizisine **basit simetrik rastgele yürüyüş** (simple symmetric random walk) denir.

Yani sayı doğrusunda 0'dan başlayan bir yürüyücü her adımda yazı tura atar; tura gelirse bir birim sağa, yazı gelirse bir birim sola gider. S_n, n adım sonraki konumdur. Bir sonraki konum yalnız şimdiki konuma ve yeni adıma bağlı olduğundan (S_n) bir Markov zinciridir (bkz. Raslantı Süreçleri). Beş bin yürüyüşü 400'er adımla tek bir diziyle simüle edip ilk sekizini çizelim:

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(20)
m, n = 5000, 400                        # 5000 yürüyüş, her biri 400 adım
steps = rng.choice([-1, 1], size=(m, n))
S = np.zeros((m, n + 1), dtype=int)
S[:, 1:] = steps.cumsum(axis=1)         # S[:, 0] = 0 başlangıç
k = np.arange(n + 1)

fig, ax = plt.subplots(figsize=(8, 4.5), layout="constrained")
for path in S[:8]:                      # ilk sekiz yürüyüş
    ax.plot(k, path, linewidth=1)
for c in (1, 2):
    ax.plot(k, c * np.sqrt(k), "k--", k, -c * np.sqrt(k), "k--")
ax.set_xlabel("n")
ax.set_ylabel("$S_n$")
fig.savefig("yuruyus.png", dpi=150)

print(S[0, :11])
end = S[:, n]                           # 400. adımdaki konumlar
print(end.mean(), (end**2).mean())      # teori: 0 ve 400
print(np.mean(np.abs(end) <= 2 * np.sqrt(n)))
```

Çıktı:

```
[ 0  1  0 -1 -2 -1 -2 -3 -2 -3 -4]
0.0916 395.6552
0.9614
```

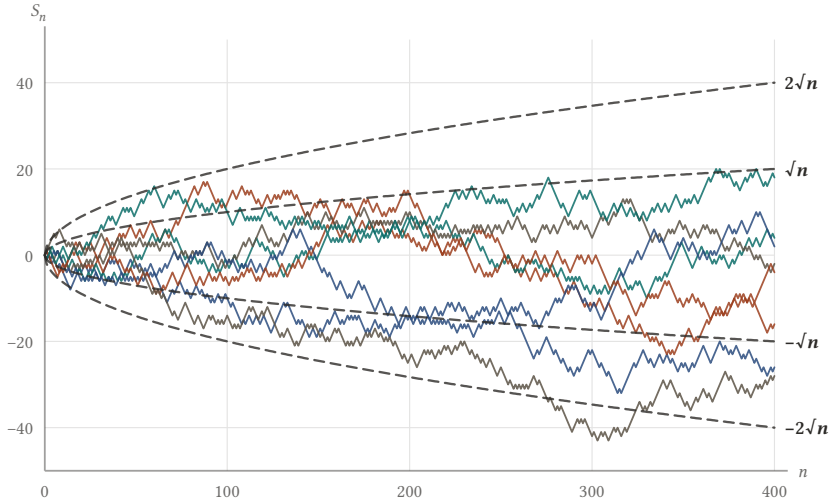
`rng.choice([-1, 1], size=(m, n))` her satırı bir yürüyüşün adımları olan bir dizi üretir; `cumsum(axis=1)` adımları satır boyunca toplayarak konumları verir. `S` dizisinin ilk sütunu başlangıç konumu $S_0 = 0$'dır. Son üç satır 400. adımdaki konumların ortalamasını, karelerinin ortalamasını ve $|S_{400}| \leq 40$ olan yürüyüşlerin oranını basıyor.

Önerme 14.3 (Yürüyüşün Ortalaması ve Varyansı). *Basit simetrik rastgele yürüyüşte her n için*

$$E(S_n) = 0, \qquad \operatorname{Var}(S_n) = E(S_n^2) = n$$

olur ve $(S_n + n)/2$ rastgele değişkeni $B(n, \frac{1}{2})$ dağılımlıdır.
Aşağıdaki şekil yukarıdaki kodun çizdiği sekiz yürüyüştür; kesikli eğriler $\pm\sqrt{n}$ ve $\pm 2\sqrt{n}$'dir.

 CC BY-NC-SA 4.0



Şekil 14.7: Rastgele yürüyüş kodunun çizdiği ilk sekiz yol, her biri 400 adım. Kesikli eğriler $\pm\sqrt{n}$ ve $\pm 2\sqrt{n}$'dir. Yollar sıfırın etrafında dolaşır ama uzaklaştıkları mesafe n ile değil $\sqrt{n}$ ile büyür: 400 adımda tipik uzaklık 20 civarıdır ve 5000 yolun %96'sı ± 40 bandında biter.

> ***İspat.***
> *Her adım için* $E(X_i) = \frac{1}{2}\cdot 1 + \frac{1}{2}\cdot(-1) = 0$ *ve* $E(X_i^2) = 1$*, dolayısıyla* $\operatorname{Var}(X_i) = 1$*'dir. Beklenen değerin lineerliğinden* $E(S_n) = 0$ *olur. Bağımsız değişkenlerin toplamının varyansı varyansların toplamı olduğundan (bkz. Olasılık Teorisi; iki terimli eşitlik tümevarımla n terime genişler)* $\operatorname{Var}(S_n) = n$*'dir.* $E(S_n) = 0$ *olduğundan* $E(S_n^2) = \operatorname{Var}(S_n) = n$ *olur.*
> $B_i = (X_i + 1)/2$ *değişkenleri bağımsızdır ve* 1 *ile* 0 *değerlerini* $\frac{1}{2}$*'şer olasılıkla alır, yani* $B(1, \frac{1}{2})$ *dağılımlıdır. Toplamları* $\sum_{i=1}^{n} B_i = (S_n + n)/2$ *olduğundan bağımsız Bernoulli değişkenlerinin toplamı hakkındaki önerme (bkz. Olasılık Teorisi)* $(S_n + n)/2 \sim B(n, \frac{1}{2})$ *verir.* ■

Yani yürüyücü ortalamada başladığı yerdedir, ama başlangıçtan tipik uzaklığı $\sqrt{E(S_n^2)} = \sqrt{n}$'dir: uzaklık adım sayısıyla değil, onun kareköküyle büyür. Simülasyonda 400. adımdaki konumların ortalaması 0,09, karelerinin ortalaması 395,7'dir; kuramsal değerler 0 ve 400'dür. Merkezi limit teoremine göre $S_n/\sqrt{n}$ yaklaşık $N(0,1)$ dağılımlıdır, bu yüzden yürüyüşlerin yaklaşık %95'i $\pm 2\sqrt{n}$ bandında biter. Önermenin binom kısmından tam değer de hesaplanabilir: $|\,S_{400}\,| \le 40$ olması $180 \le (S_{400}+400)/2 \le 220$ demektir ve bu olasılık 0,9598'dir. Simülasyon 0,9614 buldu.

Örnek 14.7 (Kumarbazın İflası). Bir oyuncu 3 lirayla başlıyor ve her turda adil bir yazı tura oyununda 1 lira kazanıyor ya da kaybediyor. Parası 0'a düşünce ya da 10 liraya ulaşınca oyun bitiyor. 20 000 oyun simüle ederek oyuncunun 10 liraya ulaşma olasılığını kestirin.

> **Çözüm.**
> Oyuncunun parası, 3'ten başlayan ve 0 ya da 10'a değince duran bir rastgele yürüyüştür. Oyunlar farklı sürelerde bittiği için sabit uzunlukta bir dizi kullanamayız. Bunun yerine bütün oyunları birlikte ilerletiyor ve hâlâ süren oyunları bir boolean maskeyle izliyoruz: her turda yalnız maskenin doğru olduğu oyunlara bir adım ekleniyor, maske de yeniden hesaplanıyor. Döngü bütün oyunlar bitince duruyor.

```
import numpy as np

rng = np.random.default_rng(21)
games, k, N = 20_000, 3, 10
pos = np.full(games, k)                 # her oyunda başlangıç sermayesi
steps = np.zeros(games, dtype=int)      # oyunların süresi
active = np.ones(games, dtype=bool)     # hâlâ süren oyunlar
```

```
while active.any():
    pos[active] += rng.choice([-1, 1], size=active.sum())
    steps[active] += 1
    active = (0 < pos) & (pos < N)
print((pos == N).mean(), k / N)
print(steps.mean(), k * (N - k))
print(steps.max())
```

Çıktı:

```
0.2997 0.3
21.1472 21
265
```

Oyunların %29,97'si 10 lirayla bitti. Tam değer $k/N = 3/10$'dur. Bunu görmek için p_k, k lirayla başlayan oyuncunun 10'a ulaşma olasılığı olsun. İlk atışa göre koşullayınca $0 < k < 10$ için

$$p_k = \frac{1}{2}\,p_{k-1} + \frac{1}{2}\,p_{k+1}, \qquad p_0 = 0, \quad p_{10} = 1$$

olur. Bu, $p_{k+1} - p_k = p_k - p_{k-1}$ demektir: ardışık farklar sabittir, p_k doğrusal büyür ve sınır koşulları $p_k = k/10$ verir. Aynı simülasyon oyunların süresini de veriyor: ortalama süre 21,15 tur, en uzun oyun 265 tur. Ortalama süre d_k için aynı koşullama $d_k = 1 + \tfrac{1}{2}(d_{k-1} + d_{k+1})$, $d_0 = d_{10} = 0$ denklemlerini verir; çözümü $d_k = k(10-k)$'dir ve $k = 3$ için 21 olur. Bu iki sonuç, durumu yutucu bir Markov zinciri olarak yazıp temel matrisi hesaplayarak da bulunabilir (bkz. Raslantı Süreçleri). ■

14.10 Alıştırmalar

Aşağıdaki alıştırmalarda her kod sabit bir tohumla çalışır; kendi denemelerinizde tohumu değiştirip sonuçların ne kadar oynadığına da bakın.

Alıştırma 14.1 (İki Zarın Toplamının Dağılımı). `default_rng(31)` üreteciyle iki zarı 360 000 kez atın. Toplamın $2, 3, \ldots, 12$ değerlerini alma sıklıklarını `np.bincount` ile bulun ve $P(T = t) = \frac{6 - |\,t-7\,|}{36}$ olasılıklarıyla karşılaştırın.

Çözüm.
Önce formülü doğrulayalım: toplamı t olan (a, b) çiftleri $2 \le t \le 7$ için $(1, t-1), \ldots, (t-1, 1)$, yani $t-1$ tanedir; $7 \le t \le 12$ için $13 - t$ tanedir. İki durumda da sayı $6 - |\,t-7\,|$'dir ve 36 eşit olasılıklı sonuca bölünür.
`np.bincount(s, minlength=13)` dizisinin j. elemanı, `s` içinde j değerinin kaç kez geçtiğidir. 0 ve 1 toplamları imkânsız olduğundan `[2:]` ile atıyoruz.

```
import numpy as np

rng = np.random.default_rng(31)
N = 360_000
s = rng.integers(1, 7, N) + rng.integers(1, 7, N)
freq = np.bincount(s, minlength=13)[2:] / N    # toplam 2, 3, ..., 12
exact = (6 - np.abs(np.arange(2, 13) - 7)) / 36
for total, fr, ex in zip(range(2, 13), freq, exact):
    print(f"{total:2d}  {fr:.4f}  {ex:.4f}")
print(np.abs(freq - exact).max())
```

Çıktı:

```
 2  0.0275  0.0278
 3  0.0553  0.0556
 4  0.0837  0.0833
 5  0.1115  0.1111
 6  0.1391  0.1389
 7  0.1656  0.1667
 8  0.1390  0.1389
 9  0.1107  0.1111
10  0.0837  0.0833
11  0.0560  0.0556
12  0.0280  0.0278
0.001063888888888892
```

Sıklıklar olasılıklardan en çok $0{,}0011$ sapıyor. Olasılığı $p = 1/6$ olan bir olayın $N = 360\,000$ denemedeki göreli sıklığının standart hatası $\sqrt{p(1-p)/N} \approx 0{,}0006$'dır; sapmalar bu ölçektedir. ■

Alıştırma 14.2 (Hileli Zarı Kümülatif Toplamla Atmak). Bir hileli zarda $1, 2, 3, 4, 5$ yüzlerinin her biri $0{,}1$, 6 yüzü ise $0{,}5$ olasılıkla geliyor. `default_rng(32)` ile ürettiğiniz 10^5 düzgün sayıyı, olasılıkların kümülatif toplamı ve `np.searchsorted` yardımıyla zar atışlarına çevirin; yüzlerin sıklıklarını ve ortalamayı kontrol edin.

Çözüm.
Bu, kesikli bir dağılım için ters dönüşüm yöntemidir (Önerme 14.1). Kümülatif olasılıklar $0{,}1; 0{,}2; 0{,}3; 0{,}4; 0{,}5; 1$'dir. u sayısı $[0; 0{,}1)$ aralığındaysa 1, $[0{,}1; 0{,}2)$ aralığındaysa 2, ..., $[0{,}5; 1)$ aralığındaysa 6 yüzünü seçeriz. Bu aralıkların uzunlukları tam olarak yüzlerin olasılıklarıdır. `np.searchsorted(cdf, u, side="right")` her u için $u < \text{cdf}[j]$ olan ilk j indeksini döndürür; bu indeks de `faces` dizisinde yüzü seçer. Karşılaştırma için hazır yol olan `rng.choice(..., p=p)` sonucunu da yazdırıyoruz.

```
import numpy as np

faces = np.arange(1, 7)
p = np.array([0.1, 0.1, 0.1, 0.1, 0.1, 0.5])
cdf = np.cumsum(p)
print(cdf)

rng = np.random.default_rng(32)
N = 100_000
u = rng.random(N)
idx = np.searchsorted(cdf, u, side="right")   # u < cdf[j] olan ilk j
x = faces[idx]
print(np.bincount(x, minlength=7)[1:] / N)
print(x.mean(), (faces * p).sum())

y = rng.choice(faces, size=N, p=p)            # hazır yol
print(np.bincount(y, minlength=7)[1:] / N)
```

Çıktı:

```
[0.1 0.2 0.3 0.4 0.5 1. ]
[0.09994 0.09821 0.09993 0.10118 0.10079 0.49995]
4.50452 4.5
[0.09925 0.09895 0.10062 0.09792 0.09869 0.50457]
```

Sıklıklar olasılıklara yakındır ve ortalama $4{,}5045$, kuramsal değer $0{,}1 \cdot 15 + 0{,}5 \cdot 6 = 4{,}5$'tir. `rng.choice` aynı dağılımı verir. ■

 CC BY-NC-SA 4.0

Alıştırma 14.3 (Ters Dönüşümle Doğrusal Bir Yoğunluktan Örneklem). $[0,1]$ aralığında yoğunluğu $f(x) = 2x$ olan dağılımdan, `default_rng(33)` ile ters dönüşüm yöntemini kullanarak 10^5 elemanlı bir örneklem üretin. Örneklemin ortalamasını, varyansını ve $P(X \le 1/2)$ kestirimini tam değerlerle karşılaştırın.

Çözüm.
Üç adımı uyguluyoruz.

1. $0 \le x \le 1$ için $F(x) = \int_0^x 2t\,dt = x^2$'dir.
2. $u = x^2$ denkleminin $[0,1]$'deki çözümü $x = \sqrt{u}$'dur, yani $F^{-1}(u) = \sqrt{u}$.
3. `np.sqrt(rng.random(n))` örneklemi verir.

Tam değerler: $E(X) = \int_0^1 2x^2\,dx = \frac{2}{3}$, $E(X^2) = \int_0^1 2x^3\,dx = \frac{1}{2}$, dolayısıyla $\mathrm{Var}\,(X) = \frac{1}{2} - \frac{4}{9} = \frac{1}{18}$; ayrıca $P(X \le \frac{1}{2}) = F(\frac{1}{2}) = \frac{1}{4}$.

```
import numpy as np

rng = np.random.default_rng(33)
n = 100_000
x = np.sqrt(rng.random(n))        # F^{-1}(u) = kök u
print(x.mean(), 2 / 3)
print(x.var(ddof=1), 1 / 18)
print(np.mean(x <= 0.5), 0.25)
```

Çıktı:

```
0.666972682598439 0.6666666666666666
0.05534262966614402 0.05555555555555555
0.25007 0.25
```

Üç kestirim de tam değerlerle üç basamak uyuşuyor. ■

Alıştırma 14.4 (İlk Altıya Kadar Atış Sayısı). Bir zar ilk kez 6 gelene kadar atılıyor ve T atış sayısı olsun. Bu deneyi `default_rng(34)` ile bir `while` döngüsü kullanarak 10^5 kez simüle edin; T'nin ortalamasını, varyansını ve $P(T > 10)$ olasılığını kestirip tam değerlerle karşılaştırın.

Çözüm.
Tek bir deneyi bir fonksiyon yapıyor: zar 6 gelmediği sürece sayacı artırıyor. T, başarı olasılığı $p = 1/6$ olan bağımsız denemelerde ilk başarıya kadar yapılan deneme sayısıdır, yani $\mathrm{Geo}\,(1/6)$ dağılımlıdır. Bu dağılımda $E(T) = 1/p = 6$, $\mathrm{Var}\,(T) = q/p^2 = 30$ ve $P(T > k) = q^k$ olur, burada $q = 5/6$'dır (bkz. Olasılık Teorisi). Son iki satırda aynı dağılımdan hazır `rng.geometric` metoduyla da örneklem alıyoruz.

```
import numpy as np


def rolls_until_six(rng):
    """İlk 6 gelene kadar atılan zar sayısı."""
    count = 1
    while rng.integers(1, 7) != 6:
        count += 1
    return count


rng = np.random.default_rng(34)
T = np.array([rolls_until_six(rng) for _ in range(100_000)])
print(T[:10])
print(T.mean(), T.var(ddof=1))          # teori: 6 ve 30
```

```
print(np.mean(T > 10), (5 / 6) ** 10)   # P(T > 10) = (5/6)^10
G = rng.geometric(1 / 6, size=100_000)  # hazır geometrik dağılım
print(G.mean(), G.var(ddof=1))
```

Çıktı:

```
[ 4 10  2  4  2  7  1  2  1  1]
6.00385 30.22769745447455
0.16042 0.1615055828898458
6.00523 30.212884775947757
```

Ortalama 6,004, varyans 30,23 ve $P(T>10)$ kestirimi 0,1604'tür; tam değerler 6, 30 ve $(5/6)^{10}\approx 0{,}1615$'tir. Döngülü simülasyon yavaş ama anlaşılırdır; `rng.geometric` aynı işi tek çağrıda yapar. ■

Alıştırma 14.5 (Monty Hall Problemi). Bir yarışmada üç kapıdan birinin arkasında araba, ikisinin arkasında keçi vardır. Yarışmacı bir kapı seçer. Arabanın yerini bilen sunucu, kalan iki kapıdan arkasında keçi olan birini açar; böyle iki kapı varsa birini rastgele seçer. Yarışmacı kapısını kapalı kalan diğer kapıyla değiştirirse arabayı kazanma olasılığı nedir? `default_rng(35)` ile 10^5 oyun simüle ederek kestirin.

Çözüm.
Kapıları $0,1,2$ diye numaralayalım. Her oyunda arabanın kapısını ve yarışmacının ilk seçimini düzgün dağılımla seçiyoruz. Sunucunun açabileceği kapılar, ne seçilen ne de arabalı olan kapılardır; bunlardan birini `rng.choice` ile seçiyoruz. Üç kapının numaralarının toplamı 3 olduğundan kapalı kalan diğer kapı `3 - pick - host`'tur.

```
import numpy as np

rng = np.random.default_rng(35)
N = 100_000
car = rng.integers(0, 3, N)          # arabanın arkasındaki kapı
pick = rng.integers(0, 3, N)         # yarışmacının ilk seçimi
host = np.empty(N, dtype=int)        # sunucunun açtığı kapı
for i in range(N):
    options = [d for d in range(3) if d != pick[i] and d != car[i]]
    host[i] = rng.choice(options)
other = 3 - pick - host              # kalan kapı (0 + 1 + 2 = 3)
print(car[:6], pick[:6], host[:6], other[:6])
print(np.mean(pick == car), np.mean(other == car))
```

Çıktı:

```
[0 1 2 1 2 2] [0 1 0 2 1 0] [2 2 1 0 0 1] [1 0 2 1 2 2]
0.33427 0.66573
```

Kapıyı değiştirmeyen yarışmacı oyunların %33,4'ünü, değiştiren %66,6'sını kazanıyor. Tam değerler $1/3$ ve $2/3$'tür. Nedeni şudur: değiştiren yarışmacı, ilk seçimi yanlışsa her zaman kazanır, çünkü sunucu öteki keçiyi açmıştır ve kalan kapı arabalıdır. İlk seçimin yanlış olma olasılığı $2/3$'tür. ■

Alıştırma 14.6 (Binoma Normal Yaklaşım). $X\sim B(100,\tfrac12)$ için $P(X\le 55)$ olasılığını binom toplamıyla tam olarak hesaplayın. Bu değeri `default_rng(36)` ile 10^6 simülasyonun kestirimiyle ve süreklilik düzeltmeli normal yaklaşımla karşılaştırın.

Çözüm.
Tam değer $P(X\le 55)=\sum_{j=0}^{55}\binom{100}{j}2^{-100}$'dür; bunu `math.comb` ile kesin tamsayılarla hesaplayabiliriz. X, 100 bağımsız Bernoulli değişkeninin toplamı olduğundan merkezi limit teoremine göre

 CC BY-NC-SA 4.0

yaklaşık $N(\mu, \sigma^2)$ dağılımlıdır; burada $\mu = np = 50$ ve $\sigma = \sqrt{np(1-p)} = 5$'tir. X tamsayı değerli olduğundan $P(X \le 55) = P(X < 55{,}5)$'tir; normal yaklaşımda 55 yerine 55,5 kullanmaya **süreklilik düzeltmesi** denir (bkz. Matematiksel İstatistik).

```
import math
import numpy as np


def Phi(z):
    """Standart normal dağılım fonksiyonu."""
    return 0.5 * (1 + math.erf(z / math.sqrt(2)))


n, p = 100, 0.5
exact = sum(math.comb(n, j) for j in range(56)) / 2**n   # P(X <= 55)
rng = np.random.default_rng(36)
X = rng.binomial(n, p, size=10**6)
mu, sd = n * p, math.sqrt(n * p * (1 - p))
print(exact)
print(np.mean(X <= 55))
print(Phi((55 - mu) / sd), Phi((55.5 - mu) / sd))
```

Çıktı:

```
0.8643734879630827
0.864361
0.8413447460685428 0.8643339390536173
```

Tam değer 0,86437, simülasyon 0,86436'dır. Düzeltmesiz normal yaklaşım $\Phi(1) \approx 0{,}8413$ değeriyle yaklaşık 0,023 yanılıyor; düzeltmeli yaklaşım $\Phi(1{,}1) \approx 0{,}86433$ ise dört basamak doğrudur. ■

Alıştırma 14.7 (Bir İntegralle Pi Sayısını Kestirmek). $\int_0^1 \frac{4}{1+x^2}\,dx = \pi$'dir. Bu integrali Monte Carlo yöntemiyle 10^{-4} standart hatayla hesaplamak için kaç nokta gerekir? `default_rng(37)` ile 10^5 noktalı bir ön deneme yapıp σ'yı kestirerek bulun.

Çözüm.
İntegralin değeri $4\arctan 1 = \pi$'dir. $f(x) = 4/(1+x^2)$ ve $[0,1]$ aralığının uzunluğu 1 olduğundan Önerme 14.2 standart hatanın $\sigma/\sqrt{N}$ olduğunu söyler. Ön denemenin örneklem standart sapması s ile σ'yı kestiririz; $s/\sqrt{N} \le 10^{-4}$ koşulu $N \ge (s/10^{-4})^2$ demektir.

```
import math
import numpy as np

rng = np.random.default_rng(37)
N = 10**5
y = 4 / (1 + rng.random(N) ** 2)
I_hat = y.mean()
s = y.std(ddof=1)
se = s / math.sqrt(N)
print(I_hat, se, abs(I_hat - math.pi))
print(s, math.ceil((s / 1e-4) ** 2))   # std. hata <= 1e-4 için gereken N
```

Çıktı:

```
3.143196965868777 0.002028128191954597 0.0016043122789839437
0.6413504473375709 41133040
```

 CC BY-NC-SA 4.0

Ön denemenin kestirimi 3,14320, standart hatası 0,0020'dir; gerçek hata 0,0016 bu ölçektedir. $s \approx 0{,}641$ olduğundan yaklaşık $4{,}1 \cdot 10^7$ nokta gerekir. Bunu standart hatanın davranışından da görebiliriz: standart hatayı 0,002'den 0,0001'e, yani 20 kat küçültmek için nokta sayısını $20^2 = 400$ katına çıkarmak gerekir. Kesin değer

$$\sigma^2 = \int_0^1 \frac{16}{(1+x^2)^2}\,dx - \pi^2 = 2\pi + 4 - \pi^2$$

olduğundan $\sigma \approx 0{,}643$'tür. ■

Alıştırma 14.8 (Karşıt Değişkenlerle Varyansı Azaltmak). $\int_0^1 e^x\,dx = e - 1$ integralini `default_rng(38)` ile iki yolla kestirin: $N = 10^4$ noktalı sıradan Monte Carlo ile ve $N/2$ tane düzgün V_j sayısı için $\frac{1}{2}\left(e^{V_j} + e^{1-V_j}\right)$ değerlerinin ortalamasıyla. İki yol da f'yi 10^4 kez hesaplar; hangisinin standart hatası daha küçüktür ve neden?

Çözüm.
$1 - V$ de $U(0,1)$ dağılımlı olduğundan $E\left[\frac{1}{2}(e^V + e^{1-V})\right] = e - 1$'dir; ikinci kestirim de doğru değeri hedefler. İkinci yöntemde birlikte kullanılan e^V ve e^{1-V} değerlerine **karşıt değişkenler** (antithetic variates) denir.

```
import math
import numpy as np

rng = np.random.default_rng(38)
N = 10_000                              # f'nin hesaplanma sayısı
plain = np.exp(rng.random(N))           # sıradan Monte Carlo
v = rng.random(N // 2)                  # N/2 nokta, her biri iki kez
anti = (np.exp(v) + np.exp(1 - v)) / 2  # karşıt çiftlerin ortalaması
for name, y in [("sıradan", plain), ("karşıt", anti)]:
    se = y.std(ddof=1) / math.sqrt(len(y))
    print(f"{name:8}{y.mean():.5f}   std. hata = {se:.2e}")
print(math.e - 1)
```

Çıktı:

```
sıradan 1.71762   std. hata = 4.93e-03
karşıt  1.71803   std. hata = 8.84e-04
1.718281828459045
```

Karşıt değişkenlerin standart hatası 5,6 kat küçüktür. Nedeni, e^V büyükken e^{1-V}'nin küçük olmasıdır: iki değerin kovaryansı negatiftir ve ortalamada hatalar birbirini kısmen götürür. Tam hesap:

$$\operatorname{Var}\left(e^V\right) = \frac{e^2 - 1}{2} - (e-1)^2 \approx 0{,}2420,$$
$$\operatorname{Cov}\left(e^V, e^{1-V}\right) = E[e^V e^{1-V}] - (e-1)^2 = e - (e-1)^2 \approx -0{,}2342.$$

Bir çiftin ortalamasının varyansı $\frac{1}{2}(\operatorname{Var} + \operatorname{Cov})$, yani $\frac{1}{2}(0{,}2420 - 0{,}2342) \approx 0{,}0039$'dur. Sıradan yöntemde 10^4 değerin ortalamasının varyansı $0{,}2420/10^4$, karşıt yöntemde 5000 çiftin ortalamasının varyansı $0{,}0039/5000$'dir. Oran yaklaşık 31'dir; standart hataların oranı da $\sqrt{31} \approx 5{,}6$ olur. Aynı doğruluğa sıradan Monte Carlo ancak yaklaşık 31 kat fazla fonksiyon değeriyle ulaşır. ■

Alıştırma 14.9 (Diskte Orijine Ortalama Uzaklık). Birim diskte düzgün dağılımlı noktaları, `default_rng(39)` ile $[-1,1]^2$ karesinde ürettiğiniz 10^6 noktadan diskin dışında kalanları atarak elde edin. Bu noktalarla bir noktanın orijine ortalama uzaklığını kestirin ve tam değerle karşılaştırın.

Çözüm.

 CC BY-NC-SA 4.0

Karede düzgün dağılımlı bir noktanın, diske düştüğü bilindiğinde, diskte düzgün dağılımlı olduğunu kullanıyoruz: diskin içindeki her bölgenin koşullu olasılığı alanıyla orantılıdır. Bu yüzden diske düşmeyen noktaları atmak yeterlidir; bu yönteme **reddetme yöntemi** (rejection sampling) denir. Noktaların yaklaşık $\pi/4$'ü kalır.

```
import numpy as np

rng = np.random.default_rng(39)
N = 10**6
P = rng.uniform(-1, 1, size=(N, 2))     # [-1, 1]^2 karesinde noktalar
r = np.sqrt((P**2).sum(axis=1))
r = r[r <= 1]                           # diskin dışındakileri at
print(len(r), len(r) / N, np.pi / 4)
print(r.mean(), 2 / 3)
print(r.std(ddof=1) / np.sqrt(len(r)))
```

Çıktı:

```
785471 0.785471 0.7853981633974483
0.6664194817138088 0.6666666666666666
0.0002660426055334039
```

$785\,471$ nokta kaldı. Ortalama uzaklık kestirimi $0{,}66642$, standart hatası $0{,}00027$'dir. Tam değer, $\sqrt{x^2+y^2}$'nin diskteki ortalama değeridir: kutupsal koordinatlarla $\frac{1}{\pi}\int_0^{2\pi}\int_0^1 r \cdot r\,dr\,d\theta = \frac{2}{3}$ (bkz. İntegral Calculus). Fark $0{,}00025$, bir standart hatadan küçüktür. ■

Alıştırma 14.10 (Kumarbazın İflası İçin Temel Matris). Kumarbazın İflası örneğindeki oyunu (Örnek 14.7) $0, 1, \ldots, 10$ durumlu ve 0 ile 10 durumları yutucu olan bir Markov zinciri olarak yazın. Q ve R matrislerini kurup $(I-Q)\,t = \mathbf{1}$ ve $(I-Q)\,B = R$ sistemlerini `np.linalg.solve` ile çözerek, simülasyonun 3 liralık başlangıç için bulduğu $0{,}2997$ ve $21{,}15$ değerlerini tam değerlerle karşılaştırın.

Çözüm.
Geçici durumlar $1, \ldots, 9$, yutucu durumlar 0 ve 10'dur. Q, geçici durumlar arasındaki geçiş olasılıklarının 9×9 matrisidir: i durumundan $i-1$ ve $i+1$ durumlarına $\frac{1}{2}$ olasılıkla gidilir. R, geçici durumlardan yutucu durumlara geçişlerin 9×2 matrisidir; yalnız $1 \to 0$ ve $9 \to 10$ geçişleri sıfırdan farklıdır. Temel matris $N = (I-Q)^{-1}$'in (i,j) elemanı, i'den başlayan zincirin yutulmadan önce j durumunda geçirdiği ortalama adım sayısıdır. Satır toplamları $t = N\mathbf{1}$ ortalama süreleri, $B = NR$ ise yutulma olasılıklarını verir (bkz. Raslantı Süreçleri). Ters matrisi açıkça hesaplamak yerine aynı sonuçları `solve` ile buluyoruz (bkz. NumPy ile Lineer Cebir).

```
import numpy as np

goal = 10                               # durumlar 0, 1, ..., 10
T = goal - 1                            # geçici durumlar 1, ..., 9
Q = np.zeros((T, T))                    # i. satır: i + 1 durumu
for i in range(T - 1):
    Q[i, i + 1] = Q[i + 1, i] = 0.5     # bir adım sağa ya da sola
R = np.zeros((T, 2))                    # sütunlar: 0 ve 10 durumları
R[0, 0] = R[-1, 1] = 0.5
A = np.eye(T) - Q                       # I - Q
t = np.linalg.solve(A, np.ones(T))      # (I - Q)^{-1} 1: beklenen süre
B = np.linalg.solve(A, R)               # (I - Q)^{-1} R: yutulma
print(t)
print(B[:, 1])
print(t[2], B[2, 1])                    # başlangıç sermayesi 3
```

Çıktı:

```
[ 9. 16. 21. 24. 25. 24. 21. 16.  9.]
[0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
21.0 0.2999999999999999
```

Ortalama süreler $t_k = k(10-k)$, 10'a ulaşma olasılıkları $k/10$ çıkıyor; Kumarbazın İflası örneğinde fark denklemleriyle bulduğumuz formüllerin aynısı. $k=3$ için tam değerler 21 ve 0,3'tür (son basamaktaki 0,2999999999999999 yuvarlama hatasıdır). Simülasyonun 21,15 ve 0,2997 değerleri bunlara yakındır. ■

Alıştırma 14.11 (Düzlemde Rastgele Yürüyüş). Düzlemde orijinden başlayan bir yürüyücü her adımda sağa, sola, yukarı ya da aşağı, eşit olasılıkla bir birim gidiyor. `default_rng(41)` ile 10^4 yürüyüşü 100'er adımla simüle edin ve 100 adım sonra orijine uzaklığın karesinin ortalamasını kuramsal değeriyle karşılaştırın.

Çözüm.
Dört yönü `dirs` dizisinin satırlarına yazıyoruz. `rng.integers(0, 4, size=(m, n))` her adım için bir yön indeksi seçer ve `dirs[idx]` gelişmiş indeksleme ile `(m, n, 2)` şekilli adım dizisini kurar (bkz. NumPy Dizileri). Adımları `axis=1` boyunca toplayınca son konumları buluruz.
Kuramsal değer şöyle bulunur. Adım vektörleri $X_1, \dots, X_n$ ise $\mid S_n \mid^2 = \sum_i \sum_j X_i \cdot X_j$'dir. $i \neq j$ için X_i ile X_j bağımsızdır ve her birinin beklenen değeri sıfır vektörüdür; bu yüzden $E(X_i \cdot X_j) = E(X_i) \cdot E(X_j) = 0$ olur. $i = j$ için $X_i \cdot X_i = 1$'dir. Dolayısıyla $E \mid S_n \mid^2 = n = 100$'dür.

```
import math
import numpy as np

rng = np.random.default_rng(41)
m, n = 10_000, 100
dirs = np.array([[1, 0], [-1, 0], [0, 1], [0, -1]])
idx = rng.integers(0, 4, size=(m, n))   # her adımda bir yön
steps = dirs[idx]                       # şekil (m, n, 2)
pos = steps.sum(axis=1)                 # n adım sonraki konum
d2 = (pos**2).sum(axis=1)               # uzaklığın karesi
print(steps.shape, pos.shape)
print(d2.mean(), n)
print(np.mean(d2 == 0), (math.comb(n, n // 2) / 2**n) ** 2)
```

Çıktı:

```
(10000, 100, 2) (10000, 2)
100.7924 100
0.0058 0.006334446707872695
```

Kestirim 100,79'dur. Son satır, yürüyüşlerin 100. adımda orijine dönme oranını $(\binom{100}{50}2^{-100})^2 \approx 0{,}0063$ tam değeriyle karşılaştırıyor. Bu formül şöyle çıkar: $u = x + y$ ve $v = x - y$ koordinatlarında her adım u'yu ve v'yi ± 1 değiştirir ve dört yön dört işaret çiftine karşılık gelir. Dolayısıyla u ve v iki bağımsız basit simetrik yürüyüştür ve orijine dönmek ikisinin de 0'da olması demektir. Tek boyutta $P(S_{100} = 0) = \binom{100}{50}2^{-100}$ olduğu ise yürüyüşün binom dağılımıyla ilişkisinden (Önerme 14.3) gelir. ■

Bu bölümde sözde rastgele sayı üreteçlerini, `default_rng` ile tekrarlanabilir simülasyon yazmayı ve temel dağılımlardan, gerekirse ters dönüşümle, örneklem üretmeyi gördük. Büyük sayılar yasası Monte Carlo kestirimlerinin doğru değere gittiğini, merkezi limit teoremi ise hatalarının $\sigma/\sqrt{N}$ ölçekli ve yaklaşık normal olduğunu söyledi; bu da her kestirime bir standart hata ve bir güven aralığı eklememizi sağladı. Sıradaki Pandas ile Veri Analizi bölümünde, simülasyonların ve ölçümlerin ürettiği tablo biçimindeki verileri okumayı, düzenlemeyi, gruplamayı ve özetlemeyi öğreneceğiz.

 CC BY-NC-SA 4.0

15. Pandas ile Veri Analizi

Şimdiye kadar verilerimiz hep aynı türden sayılardan oluşan dizilerdi: bir fonksiyonun ızgaradaki değerleri, bir matrisin elemanları, bir simülasyonun sonuçları. Gerçek veriler çoğu zaman böyle gelmez. Bir hava istasyonunun ölçüm kaydında her satır bir gündür ve satırda bir tarih, istasyonun adı, sıcaklık ve yağış yan yana durur. Sütunların türleri farklıdır, bazı hücreler boştur ve sorduğumuz sorular "Erzurum'un ortalama sıcaklığı nedir?" ya da "hangi ay en yağışlıdır?" gibi gruplara ve zamana bağlı sorulardır. NumPy dizisi bütün elemanların aynı türden olmasını ister ve satırlara yalnız $0, 1, 2, \ldots$ konumlarıyla ulaşır; böyle bir tabloyu taşımak için tek başına elverişli değildir.

pandas bu boşluğu dolduran kütüphanedir. NumPy dizilerinin üzerine iki yapı kurar: etiketli bir boyutlu dizi olan **Series** ve sütunları adlandırılmış iki boyutlu tablo olan **DataFrame**. Satırlara ve sütunlara adlarıyla ulaşılır, eksik değerler tanınır, gruplama, birleştirme ve zamana göre özetleme gibi işlemler tek satırda yapılır. Kütüphane geleneksel kısaltmasıyla `import pandas as pd` diye yüklenir.

Bu bölümde önce iki temel yapıyı tanıyacak, sonra bölümün içinde ürettiğimiz bir veri kümesini bir CSV dosyasına yazıp geri okuyacağız: Ankara, İzmir ve Erzurum'daki üç istasyonun 2025 yılına ait günlük sıcaklık ve yağış değerleri. Bu sayılar gerçek ölçüm değildir; bir kosinüs dalgasına rastgele gürültü eklenerek üretilmiştir, böylece her sonucu bildiğimiz modelle karşılaştırabiliriz. Ortalama, medyan, varyans ve çeyreklik gibi kavramların kuramı Matematiksel İstatistik notlarındadır; burada onları pandas ile hesaplamayı öğreniyoruz. Önce kısa bir tadımlık:

```
import pandas as pd

df = pd.DataFrame({
    "istasyon": ["Ankara", "İzmir", "Ankara", "Erzurum", "İzmir",
                 "Erzurum"],
    "ay": ["Ocak", "Ocak", "Temmuz", "Ocak", "Temmuz", "Temmuz"],
    "sicaklik": [0.4, 9.1, 23.6, -8.9, 27.8, 19.7],
})
print(df)
print(df.groupby("istasyon")["sicaklik"].mean())
```

Çıktı:

```
  istasyon      ay  sicaklik
0   Ankara    Ocak       0.4
1    İzmir    Ocak       9.1
2   Ankara  Temmuz      23.6
3  Erzurum    Ocak      -8.9
4    İzmir  Temmuz      27.8
5  Erzurum  Temmuz      19.7
istasyon
Ankara     12.00
Erzurum     5.40
İzmir      18.45
Name: sicaklik, dtype: float64
```

Altı satırlık tabloda her istasyonun ortalamasını tek satırla bulduk. Aynı işi NumPy ile yapmak için istasyon adlarını ayrı bir dizide tutmak, her istasyon için bir maske kurmak ve sonuçları elle bir araya getirmek gerekirdi. Bu satırın nasıl çalıştığını gruplama başlığında ayrıntılı göreceğiz.

15.1 Series: Etiketli Bir Boyutlu Dizi

pandas'ın en basit yapısı, her elemanına bir etiket iliştirilmiş bir boyutlu dizidir. Asal sayı sayma fonksiyonu $\pi(x)$'in, yani x'e kadar olan asal sayıların sayısının birkaç değeriyle başlayalım:

 CC BY-NC-SA 4.0

```
import pandas as pd

# pi(x): x'e kadar olan asal sayıların sayısı
pi_x = pd.Series([4, 25, 168, 1229, 9592, 78498],
                 index=[10, 100, 10**3, 10**4, 10**5, 10**6],
                 name="pi(x)")
print(pi_x)
print(pi_x.index)
print(pi_x.to_numpy(), pi_x.dtype)
```

Çıktı:

```
10            4
100          25
1000        168
10000      1229
100000     9592
1000000   78498
Name: pi(x), dtype: int64
Index([10, 100, 1000, 10000, 100000, 1000000], dtype='int64')
[    4    25   168  1229  9592 78498] int64
```

Çıktının solundaki sütun etiketleri, sağdaki sütun değerleri gösterir. `index` özelliği etiketleri, `to_numpy()` yöntemi değerleri bir NumPy dizisi olarak verir; değerlerin türü `int64`'tür. `name` ile verdiğimiz ad çıktının en altında görünür.

Tanım 15.1 (Series). Bir **Series**, aynı türden n değer $v_0, v_1, \ldots, v_{n-1}$ ile bu değerlere birer **etiket** veren bir **indeksin** (index) $\ell_0, \ell_1, \ldots, \ell_{n-1}$ birleşimidir. k numaralı **konumdaki** değer v_k, onun etiketi ℓ_k'dir. Değerlerin ortak türüne Series'in `dtype`'ı denir; Series'e isteğe bağlı bir ad (`name`) da verilebilir.

Yani bir Series, bir NumPy dizisinin yanına bir etiketler sütunu eklenmiş hâlidir. Etiketler birbirinden farklıysa Series, etiket kümesinden değerlere giden $\ell_k \mapsto v_k$ fonksiyonu gibi okunabilir. Her değere iki yoldan ulaşılır: k konumuyla ya da ℓ_k etiketiyle. Etiket verilmezse pandas $0, 1, \ldots, n-1$ etiketlerini kendisi koyar; o zaman konum ile etiket aynı sayıdır.
Bir sözlükten kurulan Series'te anahtarlar etiket, değerler değer olur:

```
import pandas as pd

squares = pd.Series({"bir": 1, "iki": 4, "üç": 9, "dört": 16})
print(squares)
print(squares.index.tolist(), squares.size)
```

Çıktı:

```
bir      1
iki      4
üç       9
dört    16
dtype: int64
['bir', 'iki', 'üç', 'dört'] 4
```

Konumla ve etiketle ulaşmanın iki ayrı aracı vardır: `loc` etiketle, `iloc` konumla seçer. İkisi dilimlerde de farklı davranır:

```
import pandas as pd

pi_x = pd.Series([4, 25, 168, 1229, 9592, 78498],
                 index=[10, 100, 10**3, 10**4, 10**5, 10**6])
print(pi_x.loc[1000])          # etiketi 1000 olan eleman
print(pi_x.iloc[2])            # 2 numaralı konumdaki eleman
print(pi_x.iloc[-1])           # son eleman
```

 CC BY-NC-SA 4.0

```
print(pi_x.loc[100:10**4])      # etiket dilimi: iki uç da dahil
print(pi_x.iloc[1:3])           # konum dilimi: sağ uç hariç
```

Çıktı:

```
168
168
78498
100        25
1000      168
10000    1229
dtype: int64
100      25
1000    168
dtype: int64
```

`pi_x.loc[1000]` etiketi 1000 olan elemanı, `pi_x.iloc[2]` ise sıfırdan saymaya başlayınca 2 numaralı konumdaki elemanı verir; burada ikisi de 168'dir. Dilimlerdeki fark önemlidir: etiket dilimi `loc[100:10**4]` iki ucu da içerir, konum dilimi `iloc[1:3]` ise Python'daki bütün dilimler gibi sağ ucu dışarıda bırakır. Etiketler sıralı olmak zorunda olmadığından "son etiketten bir önceki etiket" diye bir kavram yoktur; bu yüzden etiket dilimleri iki ucu da kapsar.

Series üzerindeki aritmetik, NumPy dizilerinde olduğu gibi eleman eleman yapılır ve etiketler korunur. Asal sayı teoremi, $\pi(x)$'in $x/\ln x$ ile aynı hızda büyüdüğünü, yani $\pi(x)\ln x/x$ oranının $x \to \infty$ iken 1'e gittiğini söyler. Oranı tablomuzdaki değerler için hesaplayalım:

```
import numpy as np
import pandas as pd

pi_x = pd.Series([4, 25, 168, 1229, 9592, 78498],
                 index=[10, 100, 10**3, 10**4, 10**5, 10**6])
x = pi_x.index.to_numpy()
ratio = pi_x / (x / np.log(x))       # asal sayı teoremindeki oran
print(ratio.round(4))
print(ratio[ratio > 1.12])           # boolean maskeyle seçim
print(ratio.idxmax(), round(ratio.max(), 4))
```

Çıktı:

```
10         0.9210
100        1.1513
1000       1.1605
10000      1.1320
100000     1.1043
1000000    1.0845
dtype: float64
100      1.151293
1000     1.160503
10000    1.131951
dtype: float64
1000 1.1605
```

`ratio` yine bir Series'tir ve etiketleri hâlâ x değerleridir. `ratio > 1.12` bir boolean Series verir; onu köşeli parantez içine yazınca yalnız `True` olan satırlar kalır. `idxmax` en büyük değerin konumunu değil **etiketini** döndürür. Tablodaki noktalar arasında oran en büyük değerine (1,16) $x = 10^3$'te ulaşır, sonra yavaş yavaş 1'e doğru iner: $x = 10^6$'da bile 1,08'dir. Teoremdeki yakınsama oldukça yavaştır.

İki Series arasındaki işlemlerde pandas'ın NumPy'dan en önemli farkı ortaya çıkar: değerler konumlarına göre değil, etiketlerine göre eşlenir.

```
import pandas as pd

a = pd.Series([1, 2, 3], index=["x", "y", "z"])
b = pd.Series([10, 20, 30], index=["y", "z", "w"])
```

 CC BY-NC-SA 4.0

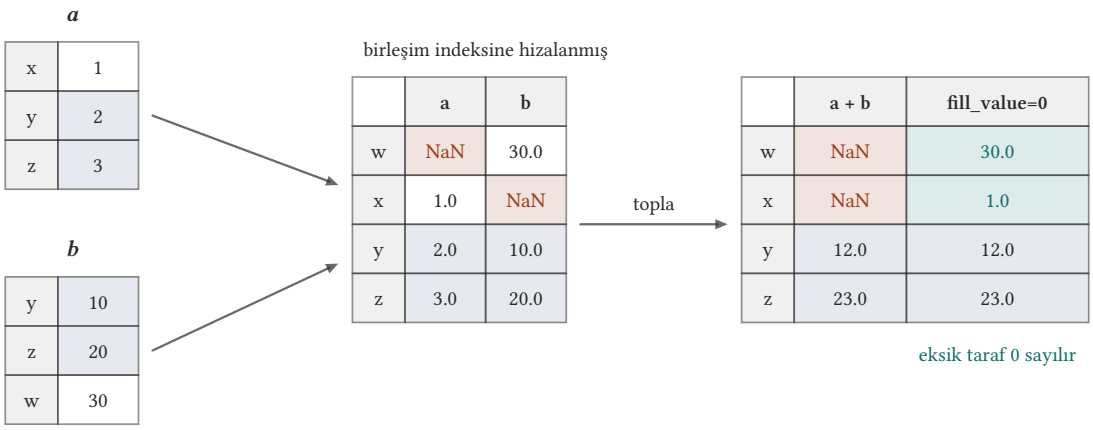
```
print(a + b)
print(a.add(b, fill_value=0))
```

Çıktı:

```
w     NaN
x     NaN
y    12.0
z    23.0
dtype: float64
w    30.0
x     1.0
y    12.0
z    23.0
dtype: float64
```

Tanım 15.2 (İndeks Hizalaması). a ve b iki Series olsun. $a + b$ toplamının (ve öteki aritmetik işlemlerin) indeksi, iki indeksin birleşimidir. Bu birleşimdeki her ℓ etiketi için ℓ hem a'da hem b'de varsa sonuç $a[\ell] + b[\ell]$ olur; ℓ yalnız birinde varsa sonuç **eksik değer** `NaN` olur.

Yani pandas iki Series'i önce ortak bir etiket kümesine **hizalar**, sonra işlemi satır satır yapar. Etiketi olmayan tarafa `NaN` ("not a number") yazılır ve `NaN` içeren her toplam yine `NaN` olur. Eksik tarafın 0 sayılmasını istiyorsak `+` yerine `add` yöntemini `fill_value=0` parametresiyle çağırırız; `sub`, `mul` ve `div` yöntemleri de aynı parametreyi alır. Sonucun etiketleri sıralanmıştır ve değerleri `float64` türündedir, çünkü `NaN` bir ondalıklı sayı değeridir.



Şekil 15.1: `a + b` işleminin iki adımı. Önce iki Series etiketlerin birleşimi olan {w, x, y, z} indeksine hizalanır; etiketi olmayan taraf `NaN` ile doldurulur. Sonra değerler satır satır toplanır. Yalnız iki tarafta da bulunan y ve z etiketlerinde sayı çıkar. `a.add(b, fill_value=0)` ise eksik tarafı 0 sayar ve w ile x satırlarında da sonuç verir.

Hizalama, aynı etiketin aynı şeyi anlattığı verilerde büyük kolaylıktır: farklı sırada kaydedilmiş iki ölçüm listesini toplarken satırların karışmasından korkmayız. Beklenmedik `NaN` değerleri görürseniz ilk bakmanız gereken yer, iki tarafın etiketleridir.

 CC BY-NC-SA 4.0

⚠ Köşeli parantez konum değil etiket arar

Bir Series'te `s[...]` yazımı etiketle seçer. Etiketler tamsayıysa bu kafa karıştırır: `pi_x[0]` "ilk eleman" değil "etiketi 0 olan eleman" demektir. pandas 3.0'dan itibaren metin etiketli bir Series'te de `s[0]` konum olarak yorumlanmaz:

```
import pandas as pd

pi_x = pd.Series([4, 25, 168], index=[10, 100, 1000])
print(pi_x[100])                # köşeli parantez: etiket 100
try:
    pi_x[0]                     # 0 diye bir etiket yok
except KeyError as err:
    print("KeyError:", err)
s = pd.Series([3, 5, 7], index=["a", "b", "c"])
try:
    s[0]                        # metin indekste de konum aranmaz
except KeyError as err:
    print("KeyError:", err)
print(s.iloc[0], pi_x.iloc[0])
```

Çıktı:

```
25
KeyError: 0
KeyError: 0
3 4
```

Kural basittir: etiketle seçerken `loc`, konumla seçerken `iloc` yazın.

Bir Series'teki farklı değerlerin kaçar kez geçtiğini `value_counts` sayar. Sonuç, etiketleri bu farklı değerler olan bir Series'tir ve en sık değerden başlayarak sıralanır. `sort_index()` sonucu etiketlere göre sıralar, `normalize=True` parametresi ise sayımlar yerine oranları verir.

Örnek 15.1 (Asal Sayıların Son Basamakları). 10^6'dan küçük asal sayıları Eratosthenes kalburunun NumPy sürümüyle bulun ve son basamaklarının dağılımını `value_counts` ile çıkarın.

Çözüm.
Vektörizasyon ve Broadcasting bölümündeki maskeli kalburu kullanırız: $0, 1, \ldots, N$ sayıları için bir boolean dizi tutulur ve her asal p'nin p^2'den başlayan katları tek bir dilim atamasıyla silinir. `np.flatnonzero` kalan `True` değerlerinin konumlarını, yani asalları verir. Bunları bir Series'e koyup 10'a bölümden kalanlarını sayarız. Son satırdaki `to_dict()` bir Series'i etiketlerden değerlere giden bir sözlüğe çevirir; kısa sonuçları tek satırda yazdırmak için bölüm boyunca bunu kullanacağız.

```
import numpy as np
import pandas as pd

N = 10**6
is_prime = np.ones(N + 1, dtype=bool)
is_prime[:2] = False
for p in range(2, int(N**0.5) + 1):
    if is_prime[p]:
        is_prime[p * p::p] = False
primes = pd.Series(np.flatnonzero(is_prime))
last = primes % 10                    # son basamaklar
counts = last.value_counts().sort_index()
print(len(primes))
print(counts)
print((counts / len(primes)).round(4).to_dict())
```

Çıktı:

 CC BY-NC-SA 4.0

```
78498
1    19617
2        1
3    19665
5        1
7    19621
9    19593
Name: count, dtype: int64
{1: 0.2499, 2: 0.0, 3: 0.2505, 5: 0.0, 7: 0.25, 9: 0.2496}
```

78498 asalın son basamağı yalnız bir kez 2 (asal 2) ve bir kez 5'tir (asal 5). Geri kalan her asal 10 ile aralarında asal olduğundan son basamağı 1, 3, 7 ya da 9'dur ve bu dört basamağın her biri asalların yaklaşık dörtte birinde görülür. Bu bir rastlantı değildir: asal sayı teoreminin aritmetik dizilerdeki karşılığına göre, $\gcd(a, 10) = 1$ olan her a için son basamağı a olan asalların payı $x \to \infty$ iken $1/4$'e yaklaşır. ■

15.2 DataFrame: Sütunları Adlandırılmış Tablo

Series tek bir sütundur; birden çok sütunu aynı satır etiketleriyle yan yana koyunca bir DataFrame elde ederiz. İlk yedi asal sayıyı, ikilik gösterimlerini, kareköklerini ve $p + 2$'nin de asal olup olmadığını bir tabloda toplayalım:

```
import math
import pandas as pd


def is_prime(m):
    return m > 1 and all(m % d for d in range(2, math.isqrt(m) + 1))


p = [2, 3, 5, 7, 11, 13, 17]
df = pd.DataFrame({
    "p": p,
    "ikilik": [bin(q)[2:] for q in p],         # ikilik gösterim
    "kok": [math.sqrt(q) for q in p],
    "ikiz": [is_prime(q + 2) for q in p],      # p + 2 de asal mı?
})
print(df)
print(df.shape)
print(df.columns.tolist())
print(df.dtypes)
```

Çıktı:

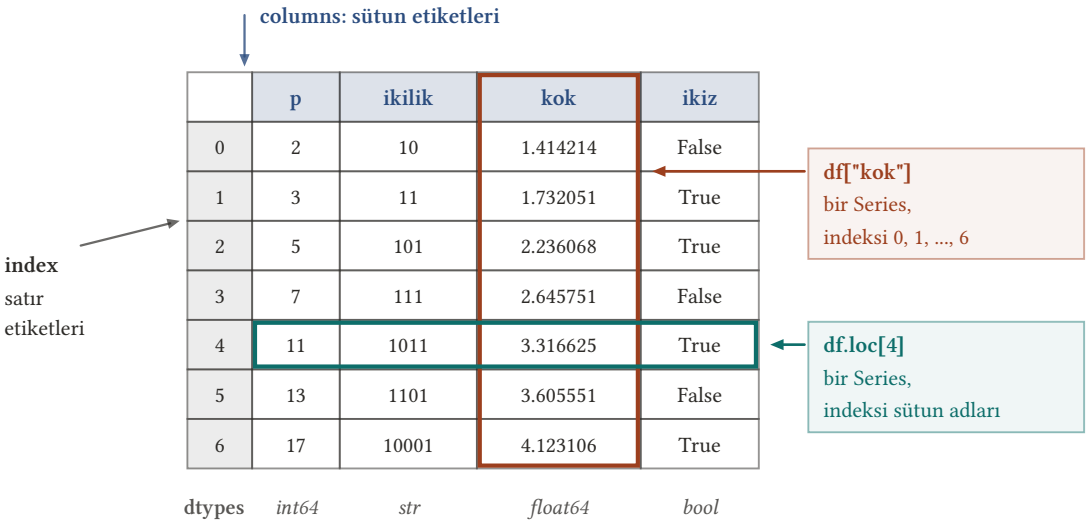
```
    p ikilik       kok   ikiz
0   2     10  1.414214  False
1   3     11  1.732051   True
2   5    101  2.236068   True
3   7    111  2.645751  False
4  11   1011  3.316625   True
5  13   1101  3.605551  False
6  17  10001  4.123106   True
(7, 4)
['p', 'ikilik', 'kok', 'ikiz']
p           int64
ikilik        str
kok       float64
ikiz         bool
dtype: object
```

 CC BY-NC-SA 4.0

`pd.DataFrame`'e verdiğimiz sözlüğün her anahtarı bir sütunun adı, her değeri o sütunun değerleridir. `shape` satır ve sütun sayısını, `columns` sütun adlarını, `dtypes` her sütunun türünü verir. Dört sütunun dört farklı türü vardır: tamsayı (`int64`), metin (`str`), ondalıklı sayı (`float64`) ve mantıksal değer (`bool`). Bir NumPy dizisinde bu mümkün değildir. Sütun adlarında Türkçe harf kullanmak serbesttir; biz bu bölümde yazması kolay olsun diye ASCII harflerle yetiniyoruz.

Tanım 15.3 (DataFrame). Bir **DataFrame**, aynı indeksi paylaşan Series'lerin sütun sütun yan yana konmasıyla oluşan iki boyutlu bir tablodur. Satır etiketlerine **indeks** (`index`), sütun etiketlerine **sütunlar** (`columns`) denir ve her sütunun kendi `dtype`'ı vardır.

Yani DataFrame'in her sütunu türdeş bir dizidir, ama tablonun bütünü türdeş olmak zorunda değildir. Bir satır etiketi ve bir sütun adı verildiğinde hücre tek türlü belirlenir; bu yüzden DataFrame, satır etiketleri ile sütun adlarının çarpım kümesinden değerlere giden bir fonksiyon gibi düşünülebilir. Bir sütunu seçmek bir Series verir; bir satırı seçmek de bir Series verir, bu kez indeksi sütun adlarıdır.



Şekil 15.2: Asal sayılar tablosu `df`'nin parçaları. Soldaki gri sütun satır etiketleri (index), üstteki satır sütun etiketleridir (columns). Her sütun kendi türünde bir Series'tir; türler en altta `df.dtypes` satırında yazılıdır.

Çerçeveli sütun `df["kok"]`, çerçeveli satır `df.loc[4]` seçimidir; ikisi de birer Series olarak döner.

```
import math
import pandas as pd

p = [2, 3, 5, 7, 11, 13, 17]
df = pd.DataFrame({"p": p, "kok": [math.sqrt(q) for q in p]})
col = df["kok"]                     # bir sütun: Series
print(type(col).__name__, col.name)
row = df.loc[4]                     # bir satır: yine Series
print(row)
df["p2"] = df["p"] ** 2             # yeni sütun, vektörel hesap
df["mod4"] = df["p"] % 4
print(df.head(4))
print(df["p2"].sum(), df["kok"].max())
```

Çıktı:

```
Series kok
p      11.000000
kok     3.316625
Name: 4, dtype: float64
```

 CC BY-NC-SA 4.0

```
   p       kok  p2  mod4
0  2  1.414214   4     2
1  3  1.732051   9     3
2  5  2.236068  25     1
3  7  2.645751  49     3
666 4.123105625617661
```

`df["kok"]`, adı `kok` olan bir Series'tir. `df.loc[4]` ise etiketi 4 olan satırdır. Satırdaki `p` değeri `11.000000` diye yazıldı, çünkü bir Series'in bütün değerleri tek bir türde olmalıdır ve tamsayı ile ondalıklı sayı ortak tür olan `float64`'te buluşur. Yeni bir sütun, sözlüğe yeni bir anahtar ekler gibi atamayla eklenir; sağ taraftaki ifade bütün sütun üzerinde vektörel olarak hesaplanır. `head(n)` ilk n satırı, `tail(n)` son n satırı gösterir. `mod4` sütununda 2'den sonraki asalların 4'e bölümünden kalanların 1 ya da 3 olduğunu görüyoruz.

Örnek 15.2 (Sinüs İçin Taylor Polinomlarının Hata Tablosu). $x = 0; 0{,}25; \ldots; 1{,}5$ noktalarında $\sin x$ değerlerini, $P_3(x) = x - x^3/6$ ve $P_5(x) = P_3(x) + x^5/120$ Taylor polinomlarını ve hatalarını bir DataFrame'in sütunlarında toplayarak P_3'ün hatasının 10^{-2}'yi aştığı en küçük tablo noktasını bulun.

Çözüm.
Önce `x` ve `sin` sütunlarıyla tabloyu kurar, sonra her yeni sütunu var olan sütunlardan hesaplarız. `abs()` mutlak değeri alır. Son satırda `t.loc[maske, "x"]` hatası 10^{-2}'yi aşan satırların `x` değerlerini seçer, `min()` bunların en küçüğünü verir.

```
import numpy as np
import pandas as pd

x = np.arange(0, 1.75, 0.25)
t = pd.DataFrame({"x": x, "sin": np.sin(x)})
t["P3"] = t["x"] - t["x"]**3 / 6
t["P5"] = t["P3"] + t["x"]**5 / 120
t["hata3"] = (t["sin"] - t["P3"]).abs()
t["hata5"] = (t["sin"] - t["P5"]).abs()
print(t.round(6))
print(t.loc[t["hata3"] > 1e-2, "x"].min())
```

Çıktı:

```
      x       sin        P3        P5     hata3     hata5
0  0.00  0.000000  0.000000  0.000000  0.000000  0.000000
1  0.25  0.247404  0.247396  0.247404  0.000008  0.000000
2  0.50  0.479426  0.479167  0.479427  0.000259  0.000002
3  0.75  0.681639  0.679688  0.681665  0.001951  0.000026
4  1.00  0.841471  0.833333  0.841667  0.008138  0.000196
5  1.25  0.948985  0.924479  0.949910  0.024505  0.000926
6  1.50  0.997495  0.937500  1.000781  0.059995  0.003286
1.25
```

P_3'ün hatası $x = 1$'de $0{,}0081$, $x = 1{,}25$'te $0{,}0245$'tir; aranan nokta $x = 1{,}25$'tir. sin fonksiyonunda $P_3 = P_4$ olduğundan Taylor teoreminin kalan terimi (Nümerik Analiz) hatayı $x^5/120$ ile sınırlar. $x = 1{,}25$ için bu sınır $0{,}0254$'tür ve tablodaki hatanın hemen üstündedir. P_5'in hatası aynı noktada $0{,}0009$'a iner. ■

15.3 CSV Dosyaları

Gerçek veri çoğu zaman bir dosyadan gelir ve tablo verisi için en yaygın biçim **CSV**'dir (comma-separated values): her satır bir kayıt, satırdaki alanlar virgülle ayrılmış, ilk satır da sütun adlarıdır. Sınıflar, Hata Yakalama ve Dosyalar bölümünde `csv` modülüyle satır satır okuduğumuz bu dosyaları pandas tek bir fonksiyonla okur ve yazar.

 CC BY-NC-SA 4.0

Bölümün geri kalanında kullanacağımız veri kümesini üretip `olcumler.csv` dosyasına yazalım. Model şudur: $t = 0, 1, \ldots, 364$ yılın günü olmak üzere bir istasyonun sıcaklığı

$$T(t) = m - a\cos\frac{2\pi(t-15)}{365} + \varepsilon_t, \qquad \varepsilon_t \sim N(0;\,2{,}5^2)$$

olsun; burada m yıllık ortalama, a genliktir. En soğuk gün $t = 15$ (16 Ocak), en sıcak gün ise ondan yarım yıl sonrasıdır. Her gün istasyona özgü bir p olasılığıyla yağışlıdır ve yağış miktarı ortalaması r olan üstel dağılımdan gelir. Rastgele seçilen 30 sıcaklık ölçümünü de bozuk bir sensörü taklit etmek için sileriz. Rastgele sayı üretecini Rastgele Sayılar ve Monte Carlo Yöntemleri bölümünde tanımıştık.

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(2025)
days = pd.date_range("2025-01-01", "2025-12-31")     # 365 gün
t = np.arange(days.size)
# istasyon: (yıllık ortalama, genlik, yağış olasılığı, ortalama yağış)
params = {"Ankara": (12.0, 11.5, 0.25, 4.0),
          "İzmir": (18.0, 8.5, 0.20, 8.0),
          "Erzurum": (6.0, 14.0, 0.30, 3.5)}
names, temps, rains = [], [], []
for name, (m, a, p, r) in params.items():
    wave = m - a * np.cos(2 * np.pi * (t - 15) / 365)
    temps.append(wave + rng.normal(0, 2.5, t.size))
    wet = rng.random(t.size) < p                     # yağışlı gün mü?
    rains.append(np.where(wet, rng.exponential(r, t.size), 0.0))
    names += [name] * t.size
temp = np.concatenate(temps).round(1) + 0.0          # -0.0 yerine 0.0
lost = rng.choice(temp.size, size=30, replace=False)
temp[lost] = np.nan                                  # arızalı sensör
df = pd.DataFrame({"tarih": np.tile(days, 3), "istasyon": names,
                   "sicaklik": temp,
                   "yagis": np.concatenate(rains).round(1)})
df.to_csv("olcumler.csv", index=False)
print(df.shape)
with open("olcumler.csv", encoding="utf-8") as f:
    for _ in range(6):                   # dosyanın ilk altı satırı
        print(f.readline(), end="")
```

Çıktı:

```
(1095, 4)
tarih,istasyon,sicaklik,yagis
2025-01-01,Ankara,-4.7,0.0
2025-01-02,Ankara,0.9,0.0
2025-01-03,Ankara,-0.6,0.0
2025-01-04,Ankara,-2.1,1.3
2025-01-05,Ankara,,0.0
```

`pd.date_range` iki tarih arasındaki bütün günleri bir tarih dizisi olarak üretir; zaman serileri başlığında ona döneceğiz. Tablo $3 \cdot 365 = 1095$ satırlıdır: önce Ankara'nın 365 günü, sonra İzmir'in, en son Erzurum'unki. Bu yüzden `np.tile(days, 3)` tarih dizisini uç uca üç kez yazar (her istasyon için bir kopya); `names` listesi de her istasyonun adını 365 kez içerir. `to_csv` tabloyu dosyaya yazar; `index=False` satır etiketlerinin $(0, 1, \ldots, 1094)$ dosyaya ayrı bir sütun olarak yazılmasını engeller. Dosyanın ilk satırlarında 5 Ocak'ın sıcaklık hücresinin boş kaldığını görüyoruz: silinen ölçümlerden biri. Bölümün bundan sonraki kodları bu dosyanın çalışma klasöründe durduğunu varsayar.

Dosyayı geri okumak için `pd.read_csv` yeterlidir. Tarih sütununun metin olarak değil tarih olarak okunması için `parse_dates` parametresini veririz:

```
import pandas as pd
```

 CC BY-NC-SA 4.0

```
df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
print(df.shape)
print(df.dtypes)
print(df.head())
print(df.tail(3))
```

Çıktı:

```
(1095, 4)
tarih       datetime64[us]
istasyon               str
sicaklik           float64
yagis              float64
dtype: object
       tarih istasyon  sicaklik  yagis
0 2025-01-01   Ankara      -4.7    0.0
1 2025-01-02   Ankara       0.9    0.0
2 2025-01-03   Ankara      -0.6    0.0
3 2025-01-04   Ankara      -2.1    1.3
4 2025-01-05   Ankara       NaN    0.0
          tarih istasyon  sicaklik  yagis
1092 2025-12-29  Erzurum      -7.5    0.0
1093 2025-12-30  Erzurum      -4.7    0.0
1094 2025-12-31  Erzurum      -7.9    0.0
```

Boş hücreler kendiliğinden `NaN` olarak okundu. `tarih` sütununun türü `datetime64[us]`'dir (mikrosaniye çözünürlüklü tarih), `istasyon` sütunu metindir. `parse_dates` verilmeseydi tarihler de düz metin (`str`) olarak kalırdı. `read_csv`'nin en çok kullanılan parametreleri şunlardır:

Parametre	Anlamı
`sep`	alan ayırıcı, varsayılanı `","`
`decimal`	ondalık işareti, varsayılanı `"."`
`na_values`	eksik değer sayılacak ek metinler, örneğin `"-"`
`parse_dates`	tarihe çevrilecek sütunlar
`date_format`	tarihlerin biçimi, örneğin `"%d.%m.%Y"`
`index_col`	indeks yapılacak sütun
`usecols`	okunacak sütunlar
`encoding`	karakter kodlaması, varsayılanı `"utf-8"`

`to_csv` de `sep` ve `decimal` parametrelerini alır: `df.to_csv("dosya.csv", sep=";", decimal=",", index=False)` Türkçe ayarlı bir hesap tablosu programının doğrudan açabileceği bir dosya yazar. Elimizdeki metin bir dosyada değil de bir değişkendeyse `read_csv(io.StringIO(metin))` biçiminde okunabilir.

Bir CSV dosyasını üç adımda okumak

1. Dosyanın ilk birkaç satırına bakın: alan ayırıcı (`,` mı, `;` mı), ondalık işareti (`.` mı, `,` mı), eksik değerlerin yazılışı (boş, `NA`, `-`) ve tarihlerin biçimi nedir?
2. `pd.read_csv`'yi bu bilgilere karşılık gelen `sep`, `decimal`, `na_values`, `parse_dates` ve `date_format` parametreleriyle çağırın.
3. Sonucu denetleyin: `shape` beklenen boyutta mı, sayı olması gereken sütunlar sayı türünde mi, `isna().sum()` kaç eksik değer gösteriyor? Sayı olması gereken bir sütun `str` geldiyse bir parametre eksiktir.

Örnek 15.3 (Türkçe Biçimli Bir Dosyadan Yerçekimi İvmesi). Bir sarkaç deneyinde farklı ip uzunlukları L (metre) için ölçülen periyotlar T (saniye), noktalı virgülle ayrılmış, ondalıkları virgüllü ve tarihleri gün.ay.yıl biçiminde olan bir dosyaya yazılmıştır; ölçülemeyen bir periyot `-` ile gösterilmiştir.

 CC BY-NC-SA 4.0

Dosyayı okuyup küçük salınımlarda geçerli $T = 2\pi\sqrt{L/g}$ formülünden her ölçüm için $g = 4\pi^2 L/T^2$ değerini ve bunların ortalamasını hesaplayın.

Çözüm.
Adım 1. Alan ayırıcı `;`, ondalık işareti `,`, eksik değer `-`, tarih biçimi `%d.%m.%Y`'dir.
Adım 2. Bu dört bilgiyi parametre olarak veririz. Kodun kendi başına çalışması için dosyayı da Python ile yazıyoruz.
Adım 3. Türleri ve eksik değer sayılarını yazdırırız.

```
from pathlib import Path

import numpy as np
import pandas as pd

text = """tarih;uzunluk;periyot
03.03.2025;0,25;1,004
03.03.2025;0,50;1,417
04.03.2025;0,75;1,739
04.03.2025;1,00;-
05.03.2025;1,25;2,241
"""
Path("sarkac.csv").write_text(text, encoding="utf-8")

df = pd.read_csv("sarkac.csv", sep=";", decimal=",", na_values="-",
                 parse_dates=["tarih"], date_format="%d.%m.%Y")
print(df.dtypes)
print(df.isna().sum().to_dict())
df["g"] = 4 * np.pi**2 * df["uzunluk"] / df["periyot"]**2
print(df.round({"g": 3}))
print(round(df["g"].mean(), 3))
```

Çıktı:

```
tarih      datetime64[us]
uzunluk           float64
periyot           float64
dtype: object
{'tarih': 0, 'uzunluk': 0, 'periyot': 1}
       tarih  uzunluk  periyot      g
0 2025-03-03     0.25    1.004  9.791
1 2025-03-03     0.50    1.417  9.831
2 2025-03-04     0.75    1.739  9.791
3 2025-03-04     1.00      NaN    NaN
4 2025-03-05     1.25    2.241  9.826
9.81
```

Bütün sütunlar doğru türde geldi; tek eksik değer `periyot` sütunundadır. `g` sütunu vektörel olarak hesaplandı; periyodu eksik satırda g de `NaN` oldu ve `mean` bu satırı atlayarak dört değerin ortalamasını aldı: $g \approx 9{,}81$ m/s². Adım 1'in neden gerekli olduğunu görmek için aynı dosyayı parametresiz okuyalım:

```
import pandas as pd

df = pd.read_csv("sarkac.csv")          # parametresiz okuma
print(df.shape)
print(df.head(2))
```

Çıktı:

 CC BY-NC-SA 4.0

```
(5, 1)
                    tarih;uzunluk;periyot
03.03.2025;0 25;1                     4.0
             50;1                   417.0
```

Varsayılan ayırıcı virgül olduğundan başlık satırının tamamı tek bir sütun adı oldu. Veri satırlarındaki ondalık virgüller ise alan ayırıcı sanıldı; her satırdan fazladan alan çıkınca pandas bunları satır etiketi yaptı ve örneğin `1,004` sayısının `004` parçası `4.0` diye okundu. Hata vermeden okunan ama anlamsız bir tablo: Adım 3'teki denetim bu yüzden gereklidir. ■

15.4 Satır ve Sütun Seçmek

Veri kümemiz elimizde; şimdi ondan istediğimiz parçaları alalım. DataFrame'de seçimin üç aracı vardır: sütun adlarıyla köşeli parantez, etiketlerle `loc` ve konumlarla `iloc`.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
print(df["sicaklik"].head(3))                    # tek sütun: Series
print(df[["istasyon", "sicaklik"]].head(3))      # liste: DataFrame
print(df.loc[365:367, ["tarih", "istasyon"]])    # etiketlerle
print(df.iloc[730, 1:3])                         # konumlarla
```

Çıktı:

```
0   -4.7
1    0.9
2   -0.6
Name: sicaklik, dtype: float64
  istasyon  sicaklik
0   Ankara      -4.7
1   Ankara       0.9
2   Ankara      -0.6
         tarih istasyon
365 2025-01-01    İzmir
366 2025-01-02    İzmir
367 2025-01-03    İzmir
istasyon    Erzurum
sicaklik       -3.4
Name: 730, dtype: object
```

Tek bir sütun adı bir Series, sütun adlarından oluşan bir liste ise bir DataFrame verir. `loc[365:367, ["tarih", "istasyon"]]` etiketi 365'ten 367'ye kadar olan satırları (iki uç dahil) ve iki sütunu seçer; 365 etiketli satır İzmir'in ilk günüdür. `iloc[730, 1:3]` ise 730 numaralı konumdaki satırın 1 ve 2 numaralı sütunlarını verir: Erzurum'un ilk günü.

Tanım 15.4 (loc ve iloc ile Seçim). `df.loc[satırlar, sütunlar]` satırları ve sütunları **etiketleriyle**, `df.iloc[satırlar, sütunlar]` ise **konumlarıyla** seçer. Her iki yere tek bir etiket (ya da konum), bir liste, bir dilim ya da bir boolean dizi yazılabilir; boolean dizinin uzunluğu satır yerinde satır sayısı, sütun yerinde sütun sayısı olmalıdır. `loc` dilimleri iki ucu da içerir, `iloc` dilimleri sağ ucu içermez.

Yani `loc` tabloya "adı şu olan satır ve sütun", `iloc` ise "şu sıradaki satır ve sütun" diye sorar. İki yere de tek bir değer yazılırsa sonuç tek bir hücredir, biri liste ya da dilim olursa bir Series, ikisi de liste ya da dilim olursa bir DataFrame döner. Varsayılan indeksli bir tabloda etiket ile konum aynı sayılardır; ama tablo süzüldükten ya da sıralandıktan sonra etiketler karışır ve iki araç farklı satırları gösterir.
Çoğu zaman satırları konumlarıyla değil bir koşulla seçeriz. Bir sütunu bir sayıyla karşılaştırmak boolean bir Series verir; onu köşeli parantez içine ya da `loc`'un satır yerine yazınca yalnız koşulu sağlayan satırlar kalır. Birden çok koşul, Vektörizasyon ve Broadcasting bölümündeki maskelerde olduğu gibi `&` (ve), `|` (veya), `~` (değil) ile ve her koşul parantez içinde olmak üzere birleştirilir:

 CC BY-NC-SA 4.0

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
cold = df[df["sicaklik"] < -12]
print(len(cold), cold["istasyon"].unique().tolist())
hot_dry = df[(df["sicaklik"] > 30) & (df["yagis"] == 0)]
print(hot_dry["istasyon"].value_counts())
print(df[df["istasyon"].isin(["Ankara", "İzmir"])].shape)
print(df.query("istasyon == 'Ankara' and yagis > 15"))
```

Çıktı:

```
3 ['Erzurum']
istasyon
İzmir     3
Ankara    2
Name: count, dtype: int64
(730, 4)
        tarih istasyon  sicaklik  yagis
38 2025-02-08   Ankara      -1.4   33.3
```

Sıcaklığın -12 °C'nin altına indiği üç günün hepsi Erzurum'dadır. Sıcaklığın 30 °C'yi aştığı ve yağışın olmadığı beş günün üçü İzmir'de, ikisi Ankara'dadır. `isin` bir sütundaki değerin verilen listede olup olmadığını sorar. `query` aynı süzmeyi bir metin ifadesiyle yazar: sütun adları değişken gibi kullanılır, metin sabitleri tırnak içine alınır. Ankara'da 15 mm'den fazla yağışın düştüğü tek gün 8 Şubat'tır.
`sort_values` tabloyu bir sütuna göre sıralar; `nlargest(k, sütun)` ve `nsmallest(k, sütun)` bir sütunun en büyük ya da en küçük k değerini taşıyan satırları verir:

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
print(df.nlargest(3, "sicaklik"))
print(df.sort_values("yagis", ascending=False).head(3))
```

Çıktı:

```
         tarih istasyon  sicaklik  yagis
203 2025-07-23   Ankara      31.5    0.0
598 2025-08-22    İzmir      31.4    0.0
189 2025-07-09   Ankara      31.0    0.0
         tarih istasyon  sicaklik  yagis
521 2025-06-06    İzmir      21.0   33.7
38  2025-02-08   Ankara      -1.4   33.3
629 2025-09-22    İzmir      22.3   24.6
```

Yılın en sıcak üç ölçümünün ikisi Ankara'da, biri İzmir'dedir; en yağışlı üç günün ikisi İzmir'de, biri Ankara'dadır. Soldaki 203, 598, 189 sayıları satırların özgün tablodaki etiketleridir; satırlar sıralanırken etiketlerini yanlarında taşır. Bu küçük tabloda `iloc[0]` 203 etiketli satırı verir, `loc[0]` ise hata verir, çünkü 0 etiketi artık tabloda yoktur.

 CC BY-NC-SA 4.0

⚠ Zincirleme atama tabloyu değiştirmez

Bir koşulu sağlayan satırlarda bir sütunu değiştirmek isterken `df[maske]["y"] = 0` yazmak doğal görünür. Ama `df[maske]` yeni bir tablodur; atama o geçici tabloya yapılır ve `df` değişmez. pandas 3.0 bu durumda `ChainedAssignmentError` uyarısı verir. Doğrusu, satırları ve sütunu tek bir `loc` içinde vermektir:

```
import warnings

import pandas as pd

df = pd.DataFrame({"x": [1, -2, 3, -4], "y": [10, 20, 30, 40]})
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter("always")
    df[df["x"] < 0]["y"] = 0          # zincirleme atama: etkisiz
print(caught[0].category.__name__)
print(df["y"].tolist())
df.loc[df["x"] < 0, "y"] = 0          # doğru yol: tek bir loc
print(df["y"].tolist())
```

Çıktı:

```
ChainedAssignmentError
[10, 20, 30, 40]
[10, 0, 30, 0]
```

Kodda uyarıyı ekrana basmak yerine yakalayıp türünü yazdırdık. pandas 3.0'da bir tablodan seçilen her parça bir kopya gibi davranır; bir tabloyu değiştirmenin yolu ona doğrudan `df.loc[...] = ...` ya da `df[sütun] = ...` biçiminde atama yapmaktır.

Örnek 15.4 (Erzurum'da Don Günlerinin Aylara Dağılımı). `olcumler.csv` dosyasında Erzurum'da sıcaklığın 0 °C'nin altında ölçüldüğü günleri aylara göre sayın.

Çözüm.
Önce Erzurum satırlarını, sonra bunların arasından sıcaklığı negatif olanları süzeriz. Tarih sütunlarının `.dt` erişimcisi tarihlerin parçalarını verir: `.dt.month` ay numarasını, `.dt.day` ayın gününü, `.dt.dayofyear` yılın kaçıncı günü olduğunu. Ay numaralarını `value_counts` ile sayıp aya göre sıralarız.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
erz = df[df["istasyon"] == "Erzurum"]
frost = erz[erz["sicaklik"] < 0]
months = frost["tarih"].dt.month          # tarihten ay numarası
print(len(frost))
print(months.value_counts().sort_index().to_dict())
```

Çıktı:

```
120
{1: 31, 2: 27, 3: 16, 11: 15, 12: 31}
```

Ölçülmüş 120 don günü vardır: ocakta ve aralıkta her gün, şubatta 27, martta 16, kasımda 15 gün. Nisandan ekime kadar hiç don yoktur. Modelde Erzurum'un ocak ortası sıcaklığı $6 - 14 = -8$ °C olduğundan ocakta her günün donlu çıkması beklenir. Sıcaklığı eksik olan günler sayılmadı, çünkü `NaN < 0` karşılaştırması `False` verir. ■

 CC BY-NC-SA 4.0

15.5 Eksik Değerler

Veri kümemizde 30 sıcaklık ölçümü eksik. pandas eksik değerleri ondalıklı sayı sütunlarında `NaN` ile gösterir ve onları bulmak, saymak, atmak ya da doldurmak için araçlar sunar.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
print(df.isna().sum())
print(df["sicaklik"].count(), df["sicaklik"].size)
print(df[df["sicaklik"].isna()].head(4))
```

Çıktı:

```
tarih        0
istasyon     0
sicaklik    30
yagis        0
dtype: int64
1065 1095
        tarih istasyon  sicaklik  yagis
4  2025-01-05   Ankara       NaN    0.0
8  2025-01-09   Ankara       NaN    0.0
45 2025-02-15   Ankara       NaN    0.0
49 2025-02-19   Ankara       NaN    0.0
```

`isna()` her hücre için eksik olup olmadığını söyleyen boolean bir tablo verir; `sum()` her sütundaki `True` değerlerini sayar. `count()` bir sütundaki eksik olmayan değerleri, `size` ise bütün değerleri sayar: $1095 - 30 = 1065$. Eksik satırları görmek için `isna()` maskesini süzmede kullanırız.

Tanım 15.5 (Eksik Değer). Bir hücrede değer bulunmadığını gösteren özel değere **eksik değer** denir. pandas ondalıklı sayı sütunlarında eksik değeri IEEE 754 standardındaki `NaN` ile gösterir. `isna()` eksik değerleri, `notna()` eksik olmayan değerleri `True` ile işaretler. Toplam, ortalama ve varyans gibi **indirgeme** işlemleri eksik değerleri varsayılan olarak atlar; eleman eleman aritmetik ise eksik değeri sonuca taşır.

Yani eksik değerli bir sütunun ortalaması, yalnız mevcut değerlerin ortalamasıdır: toplam, satır sayısına değil `count()` kadar değere bölünür. Buna karşılık `NaN + 1` yine `NaN`'dır; bir değer bilinmiyorsa ona 1 eklenmiş hâli de bilinmez. `NaN` kendisine bile eşit olmadığı için eksik değerler `==` ile değil `isna()` ile aranır:

```
import numpy as np
import pandas as pd

s = pd.Series([1.0, np.nan, 3.0])
print(s.sum(), s.mean(), s.count())     # toplamalarda NaN atlanır
print(s.sum(skipna=False))              # atlanmasın istenirse
print((s + 1).tolist())                 # aritmetikte NaN yayılır
print(np.nan == np.nan, s.isna().tolist())
```

Çıktı:

```
4.0 2.0 2
nan
[2.0, nan, 4.0]
False [False, True, False]
```

Toplam 4, ortalama $4/2 = 2$'dir: `NaN` hem toplamdan hem paydadan çıkarıldı. `skipna=False` verilince toplam da `NaN` olur.

Eksik değerlerle üç şey yapılabilir: satırı atmak (`dropna`), sabit bir değerle doldurmak (`fillna`) ya da değeri komşularından kestirmek. Kestirmenin en basit iki yolu, son bilinen değeri ileri taşımak (`ffill`, forward fill) ve iki komşu bilinen değer arasında doğrusal interpolasyon yapmaktır (`interpolate`). Hangisinin daha iyi olduğunu görmek için doğru cevabı bildiğimiz bir tablo kuralım: $\sin x$'in $[0, \pi]$ aralığındaki dokuz eşit aralıklı değerinden üçünü silelim.

 CC BY-NC-SA 4.0

```
import numpy as np
import pandas as pd

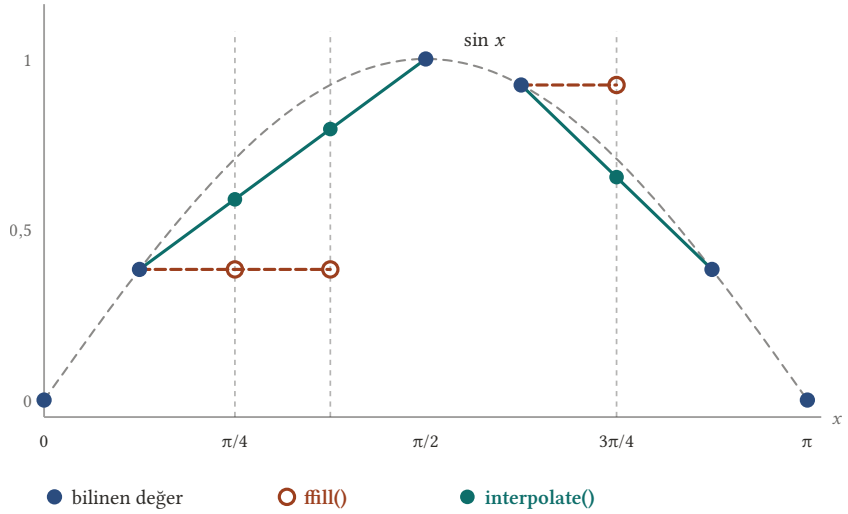
x = np.linspace(0, np.pi, 9)
s = pd.Series(np.sin(x), index=x.round(3))
s.iloc[[2, 3, 6]] = np.nan              # üç değer kayıp
print(s.dropna().size)
filled = pd.DataFrame({"kayipli": s,
                       "ffill": s.ffill(),
                       "ortalama": s.fillna(s.mean()),
                       "dogrusal": s.interpolate()})
print(filled.round(4))
true = pd.Series(np.sin(x), index=s.index)
err = filled.drop(columns="kayipli").sub(true, axis=0).abs().max()
print(err.round(4).to_dict())
```

Çıktı:

```
6
       kayipli   ffill  ortalama  dogrusal
0.000   0.0000  0.0000    0.0000    0.0000
0.393   0.3827  0.3827    0.3827    0.3827
0.785      NaN  0.3827    0.4482    0.5885
1.178      NaN  0.3827    0.4482    0.7942
1.571   1.0000  1.0000    1.0000    1.0000
1.963   0.9239  0.9239    0.9239    0.9239
2.356      NaN  0.9239    0.4482    0.6533
2.749   0.3827  0.3827    0.3827    0.3827
3.142   0.0000  0.0000    0.0000    0.0000
{'ffill': 0.5412, 'ortalama': 0.4757, 'dogrusal': 0.1297}
```

`dropna()` eksikleri atınca 6 değer kaldı. Hataları hesaplayan satırda `drop(columns="kayipli")` kayıplı sütunu tablodan çıkarır. `sub(true, axis=0)` kalan her sütundan `true` serisini çıkarır; `axis=0`, serinin etiketlerinin tablonun satır etiketleriyle eşlenmesini söyler (varsayılan eşleme sütun adlarıyladır). Son olarak `abs().max()` her sütunun en büyük mutlak hatasını verir. Doldurma yöntemlerinin en büyük hataları çıktının son satırındadır: son değeri taşımak $0{,}54$, ortalamayla doldurmak $0{,}48$, doğrusal interpolasyon $0{,}13$ hata yapar. Doğrusal interpolasyonun hatası Nümerik Analiz notlarındaki sınıra göre en çok $MH^2/8$'dir; burada M ikinci türevin mutlak değeri için bir üst sınır, H de iki bilinen komşu arasındaki uzaklıktır. İki ardışık değerin eksik olduğu aralıkta $H = 3\pi/8$ ve $M = 1$ olduğundan sınır $0{,}17$'dir ve bulunan $0{,}13$ bunun altındadır.

 CC BY-NC-SA 4.0



Şekil 15.3: sin x fonksiyonunun dokuz noktalık tablosunda kaybolan üç değer (kesikli dikey çizgiler). `ffill()` son bilinen değeri ileri taşır (boş daireler), en büyük hatası 0,54'tür. `interpolate()` iki komşu bilinen değeri doğru parçasıyla birleştirir (dolu daireler), en büyük hatası 0,13'e iner. Kesikli eğri gerçek sin x'tir.

Hangi yöntemin uygun olduğu verinin doğasına bağlıdır. Sıcaklık gibi zamanla yavaş değişen bir büyüklükte interpolasyon iyi bir tahmindir. Bir günün yağışı gibi günden güne sıçrayan bir büyüklükte ise eksik bir günü komşularından kestirmek anlamsız olabilir; o zaman satırı atmak daha dürüsttür.

Örnek 15.5 (Eksik Ölçümleri Doldurmanın Ortalamaya Etkisi). `olcumler.csv` dosyasında her istasyon için eksik sıcaklıkları doğrusal interpolasyonla doldurmanın yıllık ortalama sıcaklığı ne kadar değiştirdiğini hesaplayın.

Çözüm.
İstasyon adlarını `unique()` ile alıp her biri için o istasyonun sıcaklık sütununu süzeriz. Bir istasyonun satırları tarih sırasında ve ardışık olduğundan `interpolate()` her eksik günü önceki ve sonraki ölçüm arasında doğrusal olarak doldurur.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
for name in df["istasyon"].unique():
    temp = df.loc[df["istasyon"] == name, "sicaklik"]
    filled = temp.interpolate()
    print(f"{name:8} eksik: {temp.isna().sum():2d}   ortalama: "
          f"{temp.mean():.3f} -> {filled.mean():.3f}")
```

Çıktı:

```
Ankara   eksik:  8   ortalama: 12.143 -> 12.016
İzmir    eksik:  9   ortalama: 17.848 -> 17.786
Erzurum  eksik: 13   ortalama: 5.790 -> 5.880
```

Eksik ölçüm sayıları 8, 9 ve 13′tür; toplamları 30′dur. Doldurmak ortalamaları en çok 0,13 °C değiştirir (Ankara: 12,143 → 12,016). Eksikler yılın rastgele günlerine dağıldığı için mevcut değerlerin ortalaması da iyi bir tahmindir ve doldurmak onu pek değiştirmez. Eksikler belirli bir mevsimde toplansaydı, örneğin sensör yalnız kışın bozulsaydı, mevcut değerlerin ortalaması sistematik olarak yukarı kayardı ve interpolasyon bu kaymayı azaltırdı. ■

 CC BY-NC-SA 4.0

⚠ Eksik değer tamsayı sütununu ondalıklı sayıya çevirir

`NaN` bir ondalıklı sayı değeri olduğu için tamsayı bir sütuna eksik değer girince sütunun türü `float64` olur; tabloları birleştirirken bunun bir örneğini göreceğiz. Tamsayı kalması gereken bir sütun için pandas'ın eksik değeri `<NA>` ile gösteren `"Int64"` türü (büyük I ile) kullanılabilir:

```
import pandas as pd

print(pd.Series([1, 2, None]).dtype)        # tamsayılar float64 olur
s = pd.Series([1, 2, None], dtype="Int64")  # tamsayı kalan sütun
print(s.tolist(), s.dtype, s.sum())
```

Çıktı:

```
float64
[1, 2, <NA>] Int64 3
```

15.6 Betimsel İstatistikler

Veri okunup temizlendikten sonra ilk iş onu birkaç sayıyla özetlemektir. Ortalama, medyan, varyans ve çeyreklik gibi **betimsel istatistiklerin** tanımları Matematiksel İstatistik notlarındadır; pandas bunların çoğunu tek bir çağrıyla verir:

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
print(df[["sicaklik", "yagis"]].describe().round(2))
```

Çıktı:

```
       sicaklik    yagis
count   1065.00  1095.00
mean      11.95     1.03
std        9.78     3.01
min      -13.60     0.00
25%        5.00     0.00
50%       13.00     0.00
75%       19.70     0.00
max       31.50    33.70
```

`describe` her sayısal sütun için gözlem sayısını (`count`, eksikler hariç), ortalamayı, standart sapmayı, en küçük değeri, üç çeyrekliği (`25%`, `50%`, `75%`) ve en büyük değeri verir. Yağış sütununun üç çeyrekliği de 0'dır: günlerin dörtte üçünden fazlası kurudur. Aynı sayılar tek tek de alınabilir:

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
t = df["sicaklik"]
print(round(t.mean(), 3), t.median())
print(round(t.var(), 3), round(t.std(), 3))
print(t.max() - t.min(), t.quantile([0.1, 0.9]).round(2).tolist())
print(round((df["yagis"] > 0).mean(), 3))   # yağışlı günlerin oranı
print(df["istasyon"].value_counts().to_dict())
```

Çıktı:

```
11.95 13.0
95.721 9.784
45.1 [-2.96, 23.66]
```

 CC BY-NC-SA 4.0

```
0.223
{'Ankara': 365, 'İzmir': 365, 'Erzurum': 365}
```

Yağışlı günlerin oranını `(df["yagis"] > 0).mean()` ile bulduk: `True` değerleri 1, `False` değerleri 0 sayıldığından boolean bir sütunun ortalaması `True` değerlerinin oranıdır. pandas yöntemlerinin karşılık geldiği istatistikler şunlardır:

Yöntem	İstatistik
`mean()`	aritmetik ortalama
`median()`	medyan
`mode()`	mod
`var()`, `std()`	örneklem varyansı ve standart sapması
`max() - min()`	açıklık
`quantile(p)`	yüzdelik
`cov()`, `corr()`	kovaryans ve korelasyon
`value_counts()`	frekans tablosu

`var` ve `std` örneklem varyansını, yani $n-1$ paydalı s^2'yi ve onun karekökünü hesaplar. NumPy'ın `np.var` ve `np.std` fonksiyonları ise varsayılan olarak n'ye böler.

⚠ pandas ile NumPy'ın standart sapması farklıdır

Aynı veriye `pd.Series(x).std()` ile `np.std(x)` farklı sonuç verir. Fark paydadadır ve `ddof` ("delta degrees of freedom") parametresiyle seçilir: payda $n - \text{ddof}$'tur. pandas'ta varsayılan `ddof=1`, NumPy'da `ddof=0`'dır:

```
import numpy as np
import pandas as pd

x = [2.0, 4.0, 4.0, 4.0, 5.0, 5.0, 7.0, 9.0]
print(pd.Series(x).std(), np.std(x))
print(pd.Series(x).std(ddof=0), np.std(x, ddof=1))
```

Çıktı:

```
2.138089935299395 2.0
2.0 2.138089935299395
```

Bu sekiz sayının ortalaması 5, sapma kareleri toplamı 32'dir: $\sqrt{32/8} = 2$ ve $\sqrt{32/7} \approx 2{,}138$. Hangi paydayı kullandığınızı her zaman belirtin.

Yüzdelikler için de bir kural seçmek gerekir. Matematiksel İstatistik notları P. yüzdeliğin sıralı verideki konumunu $(n+1)P/100$ ile belirler. pandas'ın `quantile` yöntemi ise varsayılan olarak başka bir konum kuralı kullanır.

Tanım 15.6 (Doğrusal Yüzdelik Kuralı). $x_{(1)} \le x_{(2)} \le \cdots \le x_{(n)}$ sıralı veri ve $0 \le p \le 1$ olsun. $h = (n-1)p + 1$ konumunu $h = k + d$ (k tam sayı, $0 \le d < 1$) biçiminde yazalım. pandas'ın `quantile(p)` değeri

$$Q(p) = x_{(k)} + d\,\big(\, x_{(k+1)} - x_{(k)} \,\big)$$

sayısıdır ($d = 0$ ise $Q(p) = x_{(k)}$).

Yani pandas en küçük gözleme $p = 0$, en büyüğüne $p = 1$ karşılığını verir ve aradaki p değerleri için iki komşu sıralı gözlem arasında doğrusal interpolasyon yapar. Matematiksel İstatistik notlarındaki kuralla tek fark konum formülüdür: orada $h = (n+1)p$'dir. NumPy'ın `np.quantile` fonksiyonu

 CC BY-NC-SA 4.0

`method="weibull"` parametresiyle o kuralı uygular. Büyük veride iki kuralın farkı önemsizdir, küçük veride ise görünür.

Örnek 15.6 (İki Yüzdelik Kuralıyla Çeyreklikler). Matematiksel İstatistik notlarındaki $12, 15, 17, 20, 22, 25, 28, 30, 34, 41$ verisinin çeyrekliklerini doğrusal yüzdelik kuralıyla ve $(n+1)p$ kuralıyla hesaplayıp karşılaştırın.

Çözüm.
$n = 10$'dur. Doğrusal kuralda Q_1 için $h = 9 \cdot 0{,}25 + 1 = 3{,}25$, Q_2 için $h = 5{,}5$, Q_3 için $h = 7{,}75$'tir:

$$\begin{aligned} Q_1 &= x_{(3)} + 0{,}25\,(\,x_{(4)} - x_{(3)}\,) = 17 + 0{,}25 \cdot 3 = 17{,}75, \\ Q_2 &= x_{(5)} + 0{,}5\,(\,x_{(6)} - x_{(5)}\,) = 22 + 0{,}5 \cdot 3 = 23{,}5, \\ Q_3 &= x_{(7)} + 0{,}75\,(\,x_{(8)} - x_{(7)}\,) = 28 + 0{,}75 \cdot 2 = 29{,}5. \end{aligned}$$

Kodla doğrulayıp $(n+1)p$ kuralıyla karşılaştıralım:

```
import numpy as np
import pandas as pd

x = pd.Series([12, 15, 17, 20, 22, 25, 28, 30, 34, 41])
q = [0.25, 0.5, 0.75]
print(x.quantile(q).tolist())                       # pandas kuralı
print(np.quantile(x, q, method="weibull").tolist())  # (n + 1)p kuralı
```

Çıktı:

```
[17.75, 23.5, 29.5]
[16.5, 23.5, 31.0]
```

`method="weibull"` ile bulunan 16,5, 23,5 ve 31 değerleri notlardaki sonuçların aynısıdır. Medyan iki kuralda da aynıdır, çeyreklikler ise farklıdır: doğrusal kural çeyreklikleri ortaya doğru, $(n+1)p$ kuralı uçlara doğru yerleştirir. Çeyrekler arası açıklık birinde $29{,}5 - 17{,}75 = 11{,}75$, ötekinde $31 - 16{,}5 = 14{,}5$'tir. Yüzdelik içeren bir sonucu başka bir hesapla karşılaştırırken hangi kuralın kullanıldığını kontrol edin. ■

Bir ortalamanın ne kadar güvenilir olduğunu, onu oluşturan gözlemlerin sayısı belirler. Bölümün geri kalanında birkaç kez kullanacağımız şu sonucu hatırlayalım.

Önerme 15.1 (Bağımsız Gözlemlerin Ortalaması). *$X_1, \ldots, X_n$ bağımsız ve her biri μ beklenen değerli, σ^2 varyanslı rastgele değişkenler olsun. $\bar{X} = (X_1 + \cdots + X_n)/n$ için $E(\bar{X}) = \mu$ ve $\operatorname{Var}(\bar{X}) = \sigma^2/n$'dir; yani $\bar{X}$'in standart sapması $\sigma/\sqrt{n}$'dir. İspatı için bkz. Matematiksel İstatistik.*

Örnek 15.7 (Zar Atışlarının Frekans Tablosu). `default_rng(7)` üreteciyle atılan 6000 zarın göreli frekanslarını, ortalamasını ve varyansını bir zarın kuramsal değerleriyle karşılaştırın.

Çözüm.
`rng.integers(1, 7, size=6000)` $\{1, \ldots, 6\}$ kümesinden eşit olasılıkla seçilmiş 6000 tamsayı verir (üst sınır dahil değildir). Tek bir atış X için $E(X) = (1 + \cdots + 6)/6 = 7/2$ ve $E(X^2) = 91/6$'dır. Buradan varyans $\operatorname{Var}(X) = 91/6 - 49/4$ olarak bulunur; bu sayı $35/12 \approx 2{,}9167$'dir.

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(7)
rolls = pd.Series(rng.integers(1, 7, size=6000))
freq = rolls.value_counts(normalize=True).sort_index()
```

 CC BY-NC-SA 4.0

```
print(freq.round(4))
print(round(rolls.mean(), 4), round(rolls.var(), 4))
print(3.5, round(35 / 12, 4))
```

Çıktı:

```
1    0.1590
2    0.1740
3    0.1685
4    0.1578
5    0.1735
6    0.1672
Name: proportion, dtype: float64
3.5143 2.9023
3.5 2.9167
```

Göreli frekanslar $0{,}158$ ile $0{,}174$ arasında, $1/6 \approx 0{,}1667$ çevresindedir. Ortalama $3{,}5143$, varyans $2{,}9023$'tür. Önerme 15.1 gereği 6000 atışın ortalamasının standart sapması $\sqrt{(35/12)/6000} \approx 0{,}022$ olduğundan ortalamadaki $0{,}014$'lük sapma beklenen büyüklüktedir; aynı soru Matematiksel İstatistik notlarında büyük sayılar yasasıyla ele alınır. ■

15.7 Gruplama: groupby

Bölümün başındaki küçük örnekte istasyonların ortalamasını `groupby` ile bulmuştuk. Şimdi bu işlemin ne yaptığına yakından bakalım.

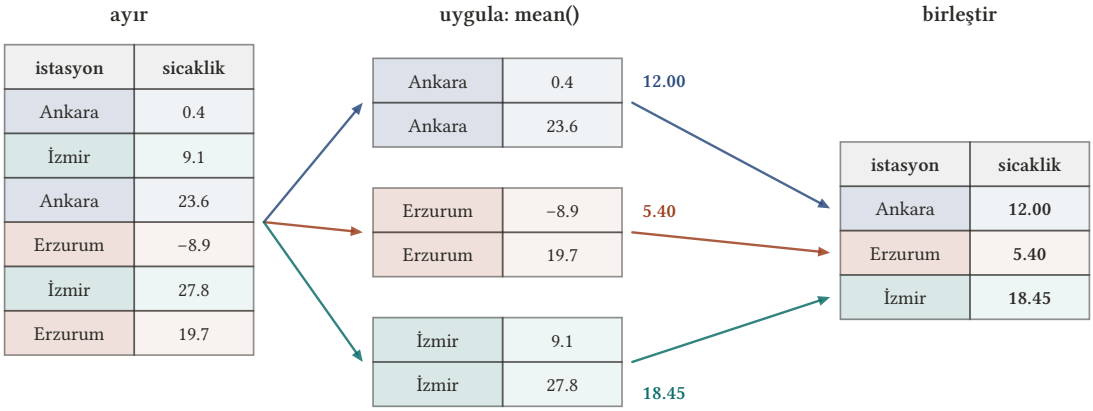
Tanım 15.7 (Ayır, Uygula, Birleştir). Bir DataFrame'in satır etiketleri kümesi I, bir anahtar sütunu da k olsun ve k sütununun aldığı farklı değerler $g_1, \dots, g_m$ olsun. Satırlar

$$G_j = \{\, i \in I : k_i = g_j \,\}, \qquad j = 1, \dots, m$$

gruplarına **ayrılır**. Bir indirgeme fonksiyonu (ortalama, toplam, en büyük değer, ...) her grubun değerlerine **uygulanır** ve m sonuç, etiketleri $g_1, \dots, g_m$ olan bir Series'te **birleştirilir**. `df.groupby(k)[sütun].mean()` gibi bir ifade bu üç adımı yapar.

Yani `groupby`, satırlar kümesini "anahtarı aynı" denklik bağıntısının denklik sınıflarına ayırır ve her sınıfı tek bir sayıya indirger. Sonuçta grup sayısı kadar satır olur ve satırlar anahtarın sıralanmış değerleriyle etiketlenir.

 CC BY-NC-SA 4.0



Şekil 15.4: `df.groupby("istasyon")["sicaklik"].mean()` işleminin üç adımı. Ayır: satırlar istasyon adına göre üç gruba bölünür. Uygula: her grubun sıcaklıklarına `mean` uygulanır, örneğin Ankara için (0,4 + 23,6)/2 = 12. Birleştir: üç sonuç, indeksi grup adları olan tek bir Series'te toplanır.

Şimdi aynı işlemi bütün veri kümesine uygulayalım:

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
g = df.groupby("istasyon")
print(g["sicaklik"].mean().round(2))
print(g.size())
stats = g["sicaklik"].agg(["count", "mean", "std", "min", "max"])
print(stats.round(2))
```

Çıktı:

```
istasyon
Ankara     12.14
Erzurum     5.79
İzmir      17.85
Name: sicaklik, dtype: float64
istasyon
Ankara     365
Erzurum    365
İzmir      365
dtype: int64
          count   mean    std   min   max
istasyon
Ankara      357  12.14   8.40  -4.7  31.5
Erzurum     352   5.79  10.27 -13.6  25.6
İzmir       356  17.85   6.27   6.4  31.4
```

`g = df.groupby("istasyon")` henüz bir şey hesaplamaz, yalnız grupları belirler. `g["sicaklik"].mean()` her grubun ortalama sıcaklığını, `g.size()` her grubun satır sayısını verir. `agg` birden çok indirgemeyi birlikte uygular ve her biri için bir sütun açar. `count` sütunu eksikleri saymaz: Erzurum'un 365 günün 352'sinde ölçümü vardır.

Sonuçlar modelle uyumludur: ortalamalar $m = 12$, 6 ve 18 değerlerine yakındır. Standart sapmalar da önceden kestirilebilir. $a\cos\theta$ dalgasının bir tam periyottaki varyansı $a^2/2$'dir ve gürültü bundan bağımsız olarak $2{,}5^2$ ekler. Ankara için $\sqrt{11{,}5^2/2 + 2{,}5^2} \approx 8{,}51$, Erzurum için 10,21, İzmir için 6,51 çıkar; tablodaki değerler 8,40, 10,27 ve 6,27'dir.

`agg`'ye `yeni_ad=(sütun, fonksiyon)` biçiminde adlandırılmış argümanlar verilirse her sonuç sütununun adını ve hangi sütundan hesaplanacağını biz seçeriz:

 CC BY-NC-SA 4.0

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
summary = df.groupby("istasyon").agg(
    en_dusuk=("sicaklik", "min"),
    en_yuksek=("sicaklik", "max"),
    toplam_yagis=("yagis", "sum"),
)
summary["aciklik"] = summary["en_yuksek"] - summary["en_dusuk"]
print(summary.round(1))
```

Çıktı:

```
          en_dusuk  en_yuksek  toplam_yagis  aciklik
istasyon
Ankara        -4.7       31.5         349.2     36.2
Erzurum      -13.6       25.6         373.7     39.2
İzmir          6.4       31.4         403.9     25.0
```

Erzurum'un yıllık sıcaklık açıklığı 39,2 °C ile en büyük, İzmir'inki 25,0 °C ile en küçüktür.
Gruplama birden çok sütuna göre de yapılabilir. Tarihten ay numarasını yeni bir sütun olarak çıkarıp istasyon ve ay çiftlerine göre gruplayalım:

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
df["ay"] = df["tarih"].dt.month
monthly = df.groupby(["istasyon", "ay"])["sicaklik"].mean()
print(monthly.head(3).round(2))
table = monthly.unstack("istasyon")
print(table.round(1))
pt = df.pivot_table(index="ay", columns="istasyon", values="sicaklik",
                    aggfunc="mean")
print(pt.equals(table))
```

Çıktı:

```
istasyon  ay
Ankara    1     0.51
          2     2.16
          3     6.83
Name: sicaklik, dtype: float64
istasyon  Ankara  Erzurum  İzmir
ay
1           0.5     -8.0    9.9
2           2.2     -6.4   10.1
3           6.8     -1.0   13.9
4          11.0      5.5   17.1
5          17.0     12.6   21.5
6          21.0     17.4   24.7
7          24.2     19.4   26.3
8          21.9     18.0   25.0
9          17.4     13.5   21.7
10         12.1      6.2   17.9
11          6.8     -0.9   13.9
12          2.5     -6.3   11.2
True
```

İki anahtarla gruplamanın sonucu, iki düzeyli bir indeksle (`MultiIndex`) etiketlenmiş 36 elemanlı bir Series'tir. `unstack("istasyon")` istasyon düzeyini sütunlara taşıyıp okunaklı bir 12×3 tablo kurar. Aynı tablo `pivot_table` ile tek adımda da kurulur; `equals` iki tablonun aynı olduğunu doğruluyor. Tabloda mevsim dalgası açıkça görülür: her istasyonda en soğuk ay ocak, en sıcak ay temmuzdur.

 CC BY-NC-SA 4.0

Gruplara `quantile` de uygulanabilir. Her istasyonun beş sayılı özetini, yani en küçük değerini, üç çeyrekliğini ve en büyük değerini çıkaralım:

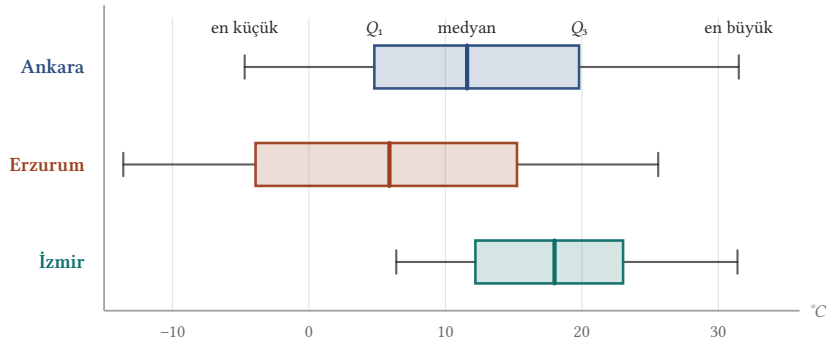
```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
q = [0, 0.25, 0.5, 0.75, 1]
five = df.groupby("istasyon")["sicaklik"].quantile(q).unstack()
print(five)
```

Çıktı:

```
          0.00  0.25  0.50    0.75  1.00
istasyon
Ankara    -4.7   4.8  11.6  19.800  31.5
Erzurum  -13.6  -3.9   5.9  15.250  25.6
İzmir      6.4  12.2  18.0  23.025  31.4
```

Bu beş sayı, Matematiksel İstatistik notlarında tanıtılan kutu grafiğiyle çizilir. Yaklaşık 350 gözlemde iki yüzdelik kuralının konumları yalnız yarım sıra farklıdır; fark ihmal edilebilir.



Şekil 15.5: Üç istasyonun sıcaklıkları için kutu grafiği, `quantile([0, 0.25, 0.5, 0.75, 1])` tablosundan. Kutu birinci çeyreklikten üçüncüye uzanır, kalın çizgi medyandır, bıyıklar en küçük ve en büyük ölçüme gider. İzmir'in kutusu sağda ve dar, Erzurum'unki solda ve geniştir: İzmir hem daha sıcak hem de yıl içinde daha az değişkendir.

`agg` her gruptan tek bir sayı döndürür. Bazen ise her satırın yanına kendi grubunun bir özetini yazmak isteriz; bunu `transform` yapar: grup sonucunu grubun her satırına yayar ve özgün tabloyla aynı uzunlukta bir Series döndürür.

Örnek 15.8 (Aylık Ortalamadan Sapmalar). Bir ölçümün **anomalisi**, ölçümden ait olduğu istasyonun o ayki ortalamasının çıkarılmasıyla bulunur. En büyük pozitif anomalinin hangi gün ve hangi istasyonda görüldüğünü bulun.

Çözüm.
Gruplar istasyon ve ay çiftleridir. `transform("mean")` her satıra kendi grubunun ortalamasını yazar; anomali, ölçüm ile bu ortalamanın farkıdır. `idxmax` en büyük anomalinin satır etiketini verir, `loc` o satırı getirir.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
df["ay"] = df["tarih"].dt.month
keys = ["istasyon", "ay"]
df["normal"] = df.groupby(keys)["sicaklik"].transform("mean")
df["anomali"] = df["sicaklik"] - df["normal"]
cols = ["tarih", "istasyon", "sicaklik", "normal", "anomali"]
```

 CC BY-NC-SA 4.0

```
print(df[cols].head(3).round({"normal": 2, "anomali": 2}))
print(df.loc[df["anomali"].idxmax(), cols])
print(df.groupby("istasyon")["anomali"].std().round(2))
```

Çıktı:

```
       tarih istasyon  sicaklik  normal  anomali
0 2025-01-01   Ankara      -4.7    0.51    -5.21
1 2025-01-02   Ankara       0.9    0.51     0.39
2 2025-01-03   Ankara      -0.6    0.51    -1.11
tarih       2025-10-01 00:00:00
istasyon                 İzmir
sicaklik                  25.4
normal               17.903226
anomali               7.496774
Name: 638, dtype: object
istasyon
Ankara     2.81
Erzurum    2.92
İzmir      2.50
Name: anomali, dtype: float64
```

`normal` sütunu her satıra kendi istasyonunun o ayki ortalamasını yazdı: Ankara'nın ocak ortalaması 0,51'dir. En büyük pozitif anomali 1 Ekim'de İzmir'dedir: o günkü 25,4 °C, İzmir'in ekim ortalamasının 7,5 °C üstündedir. Bu, gürültünün standart sapmasının (2,5) üç katıdır; 1065 ölçüm arasında böyle bir uç değerin bulunması şaşırtıcı değildir. Anomalilerin standart sapması gürültününkine yakındır. Ankara ve Erzurum'da biraz daha büyüktür, çünkü genliği büyük istasyonlarda aylık ortalama, sıcaklığın ay içindeki yükselişini ya da düşüşünü izleyemez ve bu eğilim de anomaliye karışır. ■

`pivot` uzun biçimli bir tabloyu (her satırda bir tarih ve istasyon çifti) geniş biçime (her satırda bir tarih, her istasyon ayrı bir sütunda) çevirir. Geniş tabloda `corr`, sütunlar arasındaki korelasyon katsayılarının tablosunu verir:

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
wide = df.pivot(index="tarih", columns="istasyon", values="sicaklik")
print(wide.head(3))
print(wide.corr().round(3))
```

Çıktı:

```
istasyon    Ankara  Erzurum  İzmir
tarih
2025-01-01    -4.7     -3.4    9.5
2025-01-02     0.9     -5.8   12.1
2025-01-03    -0.6     -7.6   10.1
istasyon  Ankara  Erzurum  İzmir
istasyon
Ankara     1.000    0.920  0.876
Erzurum    0.920    1.000  0.901
İzmir      0.876    0.901  1.000
```

Korelasyonlar 0,88 ile 0,92 arasındadır. Bu yüksek değerler istasyonların birbirini etkilediğini göstermez: üçü de aynı mevsim dalgasını izlediği için sıcaklıkları birlikte yükselip alçalır. Modelde günlük gürültüler bağımsızdır; mevsim etkisi çıkarılınca korelasyonun neredeyse sıfıra indiğini bir alıştırmada göreceğiz. `corr` eksik değerli satırları her sütun çifti için ayrı ayrı atar.

 CC BY-NC-SA 4.0

15.8 Tabloları Birleştirmek

Veriler çoğu zaman birden çok tabloya dağılmıştır: ölçümler bir tabloda, istasyonların bilgileri başka bir tabloda durur. İki tabloyu ortak bir sütun üzerinden yan yana birleştirmek `merge` yönteminin işidir. Aşağıda istasyonların ortalama sıcaklıklarını plaka ve bölge bilgilerini taşıyan küçük bir tabloyla birleştiriyoruz. `groupby`'a verilen `as_index=False`, grup anahtarını indekse taşımak yerine sıradan bir sütun olarak bırakır; böylece `mean_t`, `istasyon` ve `sicaklik` sütunlu bir DataFrame olur ve `merge` anahtarı iki tablonun da `istasyon` sütununda bulur.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
mean_t = df.groupby("istasyon", as_index=False)["sicaklik"].mean()
mean_t["sicaklik"] = mean_t["sicaklik"].round(2)
info = pd.DataFrame({"istasyon": ["Ankara", "İzmir", "Trabzon"],
                     "plaka": [6, 35, 61],
                     "bolge": ["İç Anadolu", "Ege", "Karadeniz"]})
print(mean_t)
print(info)
for how in ["inner", "left", "outer"]:
    print(f"how={how!r}")
    print(mean_t.merge(info, on="istasyon", how=how))
```

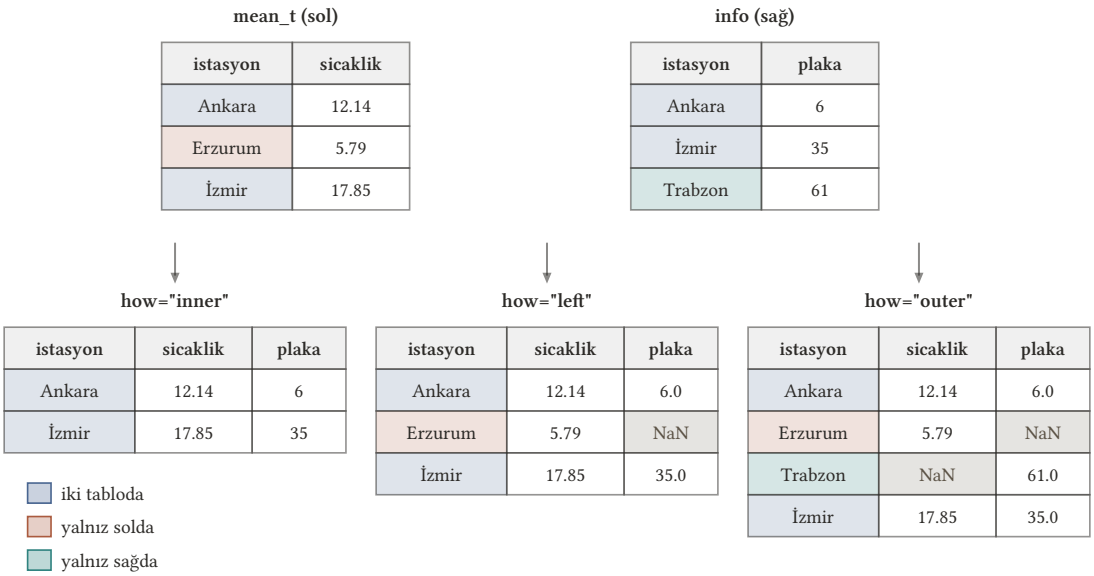
Çıktı:

```
  istasyon  sicaklik
0   Ankara     12.14
1  Erzurum      5.79
2    İzmir     17.85
  istasyon  plaka       bolge
0   Ankara      6  İç Anadolu
1    İzmir     35         Ege
2  Trabzon     61   Karadeniz
how='inner'
  istasyon  sicaklik  plaka       bolge
0   Ankara     12.14      6  İç Anadolu
1    İzmir     17.85     35         Ege
how='left'
  istasyon  sicaklik  plaka       bolge
0   Ankara     12.14    6.0  İç Anadolu
1  Erzurum      5.79    NaN         NaN
2    İzmir     17.85   35.0         Ege
how='outer'
  istasyon  sicaklik  plaka       bolge
0   Ankara     12.14    6.0  İç Anadolu
1  Erzurum      5.79    NaN         NaN
2  Trabzon       NaN   61.0   Karadeniz
3    İzmir     17.85   35.0         Ege
```

Tanım 15.8 (Anahtar Sütunuyla Birleştirme). L ve R iki DataFrame, k ikisinde de bulunan bir sütun olsun. `L.merge(R, on=k)` sonucu, k değerleri eşit olan her $(\ell, r) \in L \times R$ satır çifti için ℓ ile r'nin sütunlarını yan yana koyan bir satırdan oluşur. `how` parametresi karşılığı olmayan anahtarlara ne olacağını belirler: `"inner"` yalnız iki tabloda da bulunan anahtarları tutar; `"left"` L'nin, `"right"` R'nin, `"outer"` iki tablonun bütün anahtarlarını tutar ve karşılığı olmayan hücrelere `NaN` yazar.

Yani birleştirme, iki tablonun satır çiftlerinden anahtarları eşleşenleri seçer; ilişkisel veritabanlarındaki **join** işleminin aynısıdır. Anahtar kümeleri A_L ve A_R ise `inner` sonucunun anahtarları $A_L \cap A_R$, `outer` sonucununkiler $A_L \cup A_R$, `left` sonucununkiler A_L'dir. Örnekte Erzurum yalnız solda, Trabzon yalnız sağdadır.

 CC BY-NC-SA 4.0



Şekil 15.6: `mean_t.merge(info, on="istasyon", how=...)` sonuçları (bölge sütunu yer kazanmak için çizilmedi). `inner` yalnız iki tabloda da bulunan anahtarları, `left` sol tablonun bütün anahtarlarını, `outer` iki tablonun bütün anahtarlarını tutar. Karşılığı olmayan hücreler `NaN` olur; bu yüzden `left` ve `outer` sonuçlarında plaka sütunu ondalıklı sayıya döner.

`left` ve `outer` sonuçlarında `plaka` sütunu `6.0` gibi yazıldı: sütuna `NaN` girdiği için türü `float64` oldu.

⚠ Tekrarlanan anahtar satırları çoğaltır

Bir anahtar sağdaki tabloda iki kez geçiyorsa soldaki tablonun o anahtarlı her satırı iki kez eşleşir ve sonuç sessizce büyür. İstasyon tablosuna Ankara'yı yanlışlıkla iki kez yazdığımızı düşünelim:

```
import pandas as pd
from pandas.errors import MergeError

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
info = pd.DataFrame({"istasyon": ["Ankara", "Ankara", "İzmir", "Erzurum"],
                     "plaka": [6, 6, 35, 25]})   # Ankara iki kez yazılmış
m = df.merge(info, on="istasyon")
print(len(df), len(m))
try:
    df.merge(info, on="istasyon", validate="m:1")
except MergeError as err:
    print("MergeError:", str(err).splitlines()[0])
```

Çıktı:

```
1095 1460
MergeError: Merge keys are not unique in right dataset; not a many-to-one merge
```

1095 ölçüm satırı 1460 satıra çıktı: Ankara'nın 365 günü iki kez sayıldı. `validate="m:1"` ("çoktan bire") sağdaki tablonun anahtarlarının tekrarsız olmasını şart koşar ve koşul bozulursa `MergeError` hatası verir. Bir ölçüm tablosunu bir bilgi tablosuyla birleştirirken bu denetimi eklemek iyi bir alışkanlıktır.

`merge` tabloları yan yana birleştirir. Aynı sütunlara sahip tabloları alt alta eklemek için `pd.concat` kullanılır; `ignore_index=True` satırları 0'dan yeniden numaralar. Eski pandas sürümlerindeki `DataFrame.append` yöntemi kaldırılmıştır.

 CC BY-NC-SA 4.0

```
import pandas as pd

a = pd.DataFrame({"n": [1, 2], "kare": [1, 4]})
b = pd.DataFrame({"n": [3, 4], "kare": [9, 16]})
print(pd.concat([a, b], ignore_index=True))
```

Çıktı:

```
   n  kare
0  1     1
1  2     4
2  3     9
3  4    16
```

Örnek 15.9 (Gözlenen Aylık Ortalamalar ile Modelin Karşılaştırılması). Her istasyon ve ay için gözlenen ortalama sıcaklığı, modelin o ayın 15'indeki değeriyle iki anahtarlı bir `merge` kullanarak karşılaştırın.

Çözüm.
Gözlenen ortalamaları `groupby` ile, model değerlerini bir liste comprehension ile 36 satırlık bir tabloda toplarız. `as_index=False` sayesinde `istasyon` ve `ay` anahtarları `obs` tablosunda sıradan sütunlar olarak kalır; `model` tablosunda da aynı iki sütun vardır. Her ayın 15'inin yılın kaçıncı günü olduğunu ay başlarından buluruz: `freq="MS"` ("month start") ay başlarını üretir, iki tarihin farkının `.days` özelliği aradaki gün sayısıdır. Birleştirmede iki anahtar vardır: `on=["istasyon", "ay"]`. Her istasyon ve ay çifti iki tabloda da bir kez geçtiğinden `validate="1:1"` istenir.

```
import numpy as np
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
df["ay"] = df["tarih"].dt.month
obs = df.groupby(["istasyon", "ay"], as_index=False)["sicaklik"].mean()
# Modelin her ayın 15'indeki değeri: m - a cos(2 pi (t - 15) / 365)
params = {"Ankara": (12.0, 11.5), "İzmir": (18.0, 8.5),
          "Erzurum": (6.0, 14.0)}
starts = pd.date_range("2025-01-01", periods=12, freq="MS")  # ay başları
t = (starts - starts[0]).days.to_numpy() + 14                # ayın 15'i
rows = [(name, k + 1, m - a * np.cos(2 * np.pi * (tk - 15) / 365))
        for name, (m, a) in params.items() for k, tk in enumerate(t)]
model = pd.DataFrame(rows, columns=["istasyon", "ay", "model"])
both = obs.merge(model, on=["istasyon", "ay"], validate="1:1")
both["fark"] = both["sicaklik"] - both["model"]
print(both.shape)
print(both.head(3).round(2))
print(round(both["fark"].abs().max(), 2), round(both["fark"].std(), 2))
```

Çıktı:

```
(36, 5)
  istasyon  ay  sicaklik  model  fark
0   Ankara   1      0.51   0.50  0.01
1   Ankara   2      2.16   2.00  0.16
2   Ankara   3      6.83   5.77  1.06
1.06 0.43
```

36 satırın hepsi eşleşti. En büyük fark 1,06 °C'dir (Ankara, mart), farkların standart sapması 0,43'tür. Bu beklenen büyüklüktür: bir ayın ortalaması yaklaşık 30 bağımsız gürültü teriminin ortalamasını içerir ve Önerme 15.1 gereği bu ortalamanın standart sapması $2{,}5/\sqrt{30} \approx 0{,}46$'dır. ■

 CC BY-NC-SA 4.0

15.9 Zaman Serileri

Ölçümlerimizin her biri bir tarihe bağlıdır. Tarihleri indeks yapınca pandas zamanla ilgili soruları kolaylaştırır: belirli bir dönemi seçmek, günlük veriyi aylığa çevirmek, hareketli ortalama almak.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
ank = df[df["istasyon"] == "Ankara"].set_index("tarih")["sicaklik"]
print(type(ank.index).__name__, ank.index.dtype)
print(ank.index[0], ank.index[-1])
print(ank.loc["2025-07-01":"2025-07-04"])
print(round(ank.loc["2025-02"].mean(), 2))       # yalnız şubat
print(ank.idxmax().date(), ank.max())
```

Çıktı:

```
DatetimeIndex datetime64[us]
2025-01-01 00:00:00 2025-12-31 00:00:00
tarih
2025-07-01    23.9
2025-07-02    22.9
2025-07-03    23.2
2025-07-04    24.6
Name: sicaklik, dtype: float64
2.16
2025-07-23 31.5
```

`set_index("tarih")` tarih sütununu indekse taşır; Ankara satırlarından sıcaklık sütununu seçince indeksi tarihler olan bir Series elde ederiz. Böyle bir indekse `DatetimeIndex` denir. Tarih indeksinde etiket dilimleri tarih metinleriyle yazılabilir ve iki uç dahildir. Yalnız ayı yazmak (`"2025-02"`) o ayın bütün günlerini seçer: Ankara'nın şubat ortalaması 2,16 °C'dir. En sıcak gün 23 Temmuz'dur.
Günlük veriyi aylığa çevirmek de bir gruplamadır: her gün ait olduğu aya göre gruplanır ve her gruba bir indirgeme uygulanır. Zaman indeksli bir Series'te bunu `resample` yapar:

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
ank = df[df["istasyon"] == "Ankara"].set_index("tarih")["sicaklik"]
monthly = ank.resample("ME").mean()
print(monthly.round(2))
```

Çıktı:

```
tarih
2025-01-31     0.51
2025-02-28     2.16
2025-03-31     6.83
2025-04-30    10.97
2025-05-31    17.00
2025-06-30    20.99
2025-07-31    24.15
2025-08-31    21.93
2025-09-30    17.39
2025-10-31    12.12
2025-11-30     6.76
2025-12-31     2.45
Freq: ME, Name: sicaklik, dtype: float64
```

`"ME"` kodu ("month end") günleri aylara göre gruplar ve her ayı son günüyle etiketler. Haftalık gruplama için `"W"`, yıllık için `"YE"` yazılır. Sonuç, gruplama başlığında `groupby` ile bulduğumuz aylık tablonun Ankara sütunuyla aynıdır.

 CC BY-NC-SA 4.0

Aylık ortalama yılı on iki parçaya böler ve her parçayı tek bir sayıyla temsil eder. Her gün için o günün çevresindeki bir pencerenin ortalamasını almak ise günlük çözünürlüğü korur.

Tanım 15.9 (Merkezî Hareketli Ortalama). $x_0, x_1, \ldots, x_{N-1}$ bir zaman serisi ve $w = 2q + 1$ tek bir pencere uzunluğu olsun. $q \le t \le N - 1 - q$ için

$$\bar{x}_t = \frac{1}{w} \sum_{j=-q}^{q} x_{t+j}$$

sayılarına serinin w uzunluklu **merkezî hareketli ortalaması** denir. Pencerenin serinin dışına taştığı ilk ve son q günde $\bar{x}_t$ tanımlı değildir. pandas'ta `s.rolling(w, center=True).mean()` ile hesaplanır.

Yani her gün, kendisinin ve iki yanındaki q'şar günün ortalamasıyla değiştirilir. Günden güne bağımsız olan gürültü bu ortalamada büyük ölçüde birbirini götürür: Önerme 15.1 gereği w bağımsız gürültü teriminin ortalamasının standart sapması $\sigma/\sqrt{w}$'dir ve $w = 31$, $\sigma = 2{,}5$ için bu $0{,}45$'tir. Yavaş değişen mevsim dalgası ise neredeyse olduğu gibi kalır. `center=True` verilmezse pencere $t - w + 1, \ldots, t$ günlerinden, yani yalnız o güne kadarki günlerden oluşur.

⚠ Tek bir NaN bütün pencereyi bozar

`rolling(...).mean()` varsayılan olarak yalnız penceredeki değerlerin hepsi mevcutsa bir sonuç üretir. Ankara serisinde yalnız 8 gün eksik olduğu hâlde 31 günlük hareketli ortalamanın 124 değeri `NaN` çıkar, çünkü her eksik gün kendisini içeren 31 pencerenin hepsini bozar. Önce eksikleri `interpolate()` ile doldurmak ya da `min_periods` parametresiyle daha az değerle yetinmek gerekir:

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
ank = df[df["istasyon"] == "Ankara"].set_index("tarih")["sicaklik"]
r1 = ank.rolling(31, center=True).mean()
r2 = ank.interpolate().rolling(31, center=True).mean()
print(ank.isna().sum(), r1.isna().sum(), r2.isna().sum())
print(r2.loc["2025-07-13":"2025-07-17"].round(2))
print(round((ank - r2).std(), 3))
```

Çıktı:

```
8 124 30
tarih
2025-07-13    24.30
2025-07-14    24.17
2025-07-15    24.17
2025-07-16    24.15
2025-07-17    23.99
Name: sicaklik, dtype: float64
2.635
```

Eksikler doldurulunca yalnız baştaki ve sondaki 15'er gün, toplam 30 değer tanımsız kalır. Günlük değerlerin hareketli ortalamadan sapmalarının standart sapması 2,635'tir ve gürültünün standart sapması olan 2,5'e yakındır.

Zaman serilerini en iyi grafik anlatır. Matplotlib ile Grafik Çizimi bölümündeki nesne yönelimli arayüzle günlük değerleri, hareketli ortalamayı ve aylık ortalamaları aynı eksenlere çizelim. `ax.plot` bir Series'in indeksini ve değerlerini doğrudan kabul eder; tarih ekseni kendiliğinden biçimlenir.

```
import matplotlib.pyplot as plt
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
raw = df[df["istasyon"] == "Ankara"].set_index("tarih")["sicaklik"]
ank = raw.interpolate()
```

 CC BY-NC-SA 4.0

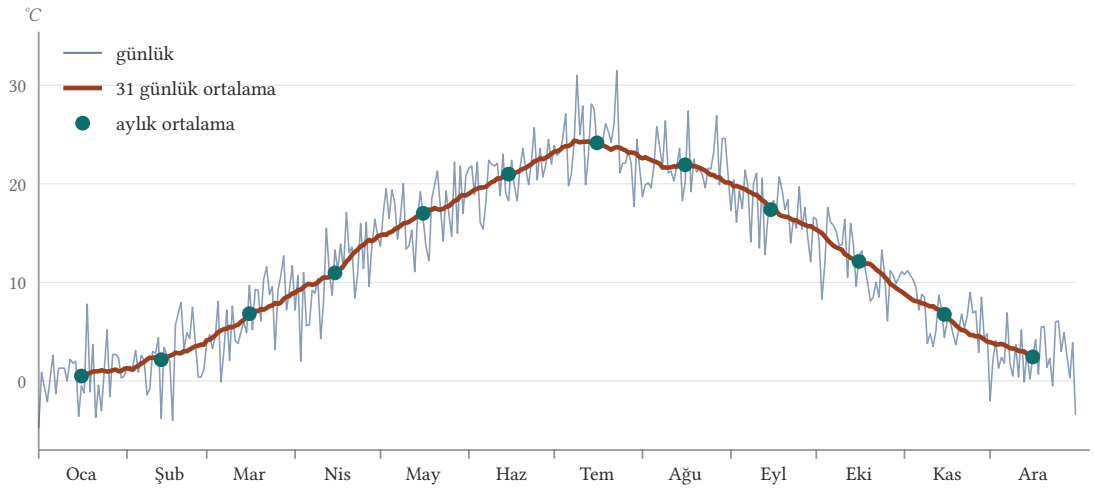
```
smooth = ank.rolling(31, center=True).mean()
monthly = raw.resample("ME").mean()
mid = monthly.index - pd.Timedelta(days=15)          # ay ortaları

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(ank.index, ank, lw=0.7, alpha=0.6, label="günlük")
ax.plot(smooth.index, smooth, lw=2.2, label="31 günlük ortalama")
ax.plot(mid, monthly, "o", label="aylık ortalama")
ax.set_ylabel("sıcaklık (°C)")
ax.legend()
fig.savefig("ankara.png", dpi=150)
print(len(ax.lines), monthly.idxmax().month, monthly.idxmin().month)
```

Çıktı:

```
3 7 1
```

Kod üç çizgi çizip grafiği `ankara.png` dosyasına kaydetti; aylık ortalamaların en büyüğü 7. ayda, en küçüğü 1. aydadır. Aylık ortalamalar ayın son günüyle etiketlendiği için 15 gün geri kaydırılarak ay ortalarına yerleştirildi. Aşağıdaki şekil aynı grafiğin aynı veriden bu sitenin çizim aracıyla yeniden çizilmiş hâlidir. pandas'ın kendi `plot` yöntemi de vardır: `ank.plot(ax=ax)` aynı çizgiyi Matplotlib ile çizer.



Şekil 15.7: Ankara'nın 2025 sıcaklıkları; grafik çizme kodunun çizdiği üç katman. İnce çizgi eksikleri interpolasyonla doldurulmuş günlük değerlerdir. Kalın çizgi 31 günlük merkezî hareketli ortalamadır ve baştaki ile sondaki 15 günde tanımlı değildir. Noktalar `resample("ME").mean()` ile bulunan aylık ortalamalardır ve ay ortalarına yerleştirilmiştir. Hareketli ortalama günlük gürültüyü bastırır, mevsim dalgası kalır.

Zaman serilerinde sık kullanılan birkaç işlem daha vardır: `shift(k)` seriyi k adım kaydırır, yani her güne k gün önceki değeri yazar; `diff()` ardışık farkları $x_t - x_{t-1}$ verir; `cumsum()` birikimli toplamı, `expanding().mean()` ise başlangıçtan o güne kadarki ortalamayı hesaplar. Aşağıdaki örnek `shift` ile ardışık donlu günlerden oluşan dizileri buluyor.

Örnek 15.10 (Erzurum'da En Uzun Don Dizisi). Eksik günleri doğrusal interpolasyonla doldurarak Erzurum'da sıcaklığın üst üste 0 °C'nin altında kaldığı en uzun dönemi bulun.

> **Çözüm.**
> Önce her günün donlu olup olmadığını gösteren boolean bir `frost` serisi kurarız. Bir dizi, `frost` değerinin bir önceki günden farklı olduğu günde başlar; `frost != frost.shift()` bu günlerde `True` olur. İlk gün de `True` olur, çünkü `shift` ilk güne `NaN` yazar. Bu boolean serinin birikimli toplamı, her güne kaçıncı dizide olduğunu gösteren bir numara verir. İki seriyi bir tabloda yan yana koyduğumuzda tarihler indekste durur; `set_index`'in tersi olan `reset_index()` onları yeniden `tarih` sütununa taşır,

 CC BY-NC-SA 4.0

çünkü her dizinin tarihlerini bu sütundan alacağız. Son olarak yalnız donlu günleri alıp dizi numarasına göre gruplar, her dizinin ilk ve son tarihini ve uzunluğunu buluruz.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
erz = df[df["istasyon"] == "Erzurum"].set_index("tarih")["sicaklik"]
frost = erz.interpolate() < 0                  # don günü mü?
run_id = (frost != frost.shift()).cumsum()     # değişince yeni dizi
days = pd.DataFrame({"don": frost, "dizi": run_id}).reset_index()
runs = days[days["don"]].groupby("dizi")["tarih"].agg(
    ["min", "max", "size"])
print(len(runs))
print(runs.nlargest(3, "size"))
```

Çıktı:

```
10
            min        max  size
dizi
1    2025-01-01 2025-03-01    60
19   2025-11-20 2025-12-31    42
3    2025-03-03 2025-03-11     9
```

Yıl içinde 10 ayrı don dizisi vardır. En uzunu 1 Ocak'tan 1 Mart'a kadar $31 + 28 + 1 = 60$ gün sürmüştür. İkinci en uzun dizi 20 Kasım'da başlar ve yılın son gününe kadar 42 gün sürer; veri yıl sonunda kesildiği için bu dizinin gerçek uzunluğunu bilmiyoruz. ■

15.10 Alıştırmalar

Alıştırmaların çoğu, bölümde ürettiğimiz `olcumler.csv` dosyasını kullanır.

Alıştırma 15.1 (İki Zarın Toplamının Dağılımı). `default_rng(12)` üreteciyle iki zarı 36000 kez atın. Toplamın göreli frekanslarını $s = 2, \dots, 12$ için kuramsal olasılıklar $(6 - |\, s - 7\,|)/36$ ile bir DataFrame'de yan yana koyarak en büyük mutlak farkı bulun.

Çözüm.
`rng.integers(1, 7, size=(36000, 2))` her satırı bir atış olan iki sütunlu bir tamsayı dizisi verir; satır toplamları zarların toplamıdır. `value_counts(normalize=True).sort_index()` göreli frekansları 2'den 12'ye sıralar. Kuramsal olasılıkları aynı indeksle bir Series olarak kurarız ki DataFrame'de satırlar etiketlerine göre hizalansın.

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(12)
dice = rng.integers(1, 7, size=(36000, 2))
total = pd.Series(dice.sum(axis=1))
freq = total.value_counts(normalize=True).sort_index()
s = freq.index.to_numpy()
exact = pd.Series((6 - np.abs(s - 7)) / 36, index=freq.index)
table = pd.DataFrame({"goreli": freq, "kuramsal": exact})
table["fark"] = table["goreli"] - table["kuramsal"]
print(table.round(4))
print(round(table["fark"].abs().max(), 4))
```

Çıktı:

 CC BY-NC-SA 4.0

```
    goreli  kuramsal    fark
2   0.0259    0.0278 -0.0019
3   0.0566    0.0556  0.0010
4   0.0826    0.0833 -0.0007
5   0.1130    0.1111  0.0019
6   0.1379    0.1389 -0.0010
7   0.1668    0.1667  0.0002
8   0.1402    0.1389  0.0013
9   0.1127    0.1111  0.0016
10  0.0816    0.0833 -0.0018
11  0.0569    0.0556  0.0014
12  0.0258    0.0278 -0.0020
0.002
```

Göreli frekanslar kuramsal olasılıklardan en çok $0{,}002$ uzaktadır. Örneğin $P(S=7)=6/36\approx 0{,}1667$, gözlenen oran $0{,}1668$'dir. 36000 atışta bir oranın standart sapması en çok $\sqrt{0{,}25/36000}\approx 0{,}0026$ olduğundan farklar beklenen büyüklüktedir. ■

Alıştırma 15.2 (Collatz Adım Sayıları). Kontrol Yapıları ve Fonksiyonlar bölümündeki Collatz dizisinde n'den 1'e ulaşmak için gereken adım sayısını $n=1,\ldots,9999$ için indeksi n olan bir Series'te toplayın ve en çok adım gerektiren üç sayıyı bulun.

Çözüm.
Adım sayısını bir fonksiyonla hesaplayıp sonuçları bir liste comprehension ile toplarız. `index=n` ile etiketler $1,\ldots,9999$ olur; böylece `idxmax` ve `nlargest` doğrudan sayıları verir.

```
import pandas as pd


def collatz_steps(n):
    steps = 0
    while n != 1:
        n = n // 2 if n % 2 == 0 else 3 * n + 1
        steps += 1
    return steps


n = range(1, 10**4)
steps = pd.Series([collatz_steps(k) for k in n], index=n)
print(steps.idxmax(), steps.max())
print(steps.nlargest(3))
print(round(steps.mean(), 2), steps.median())
```

Çıktı:

```
6171 261
6171    261
9257    259
6943    256
dtype: int64
84.97 73.0
```

En çok adımı 6171 gerektirir: 261 adım. Onu 9257 (259 adım) ve 6943 (256 adım) izler. Ortalama adım sayısı yaklaşık 85, medyan 73'tür; ortalamanın medyandan büyük olması, az sayıdaki çok uzun dizinin dağılımı sağa çektiğini gösterir. ■

 CC BY-NC-SA 4.0

Alıştırma 15.3 (Ankara'da Sıcak Temmuz Günleri). `olcumler.csv` dosyasında Ankara'da temmuz ayında sıcaklığın 28 °C'yi aştığı günleri bulun.

Çözüm.
Koşulları art arda uygularız: önce Ankara satırları, sonra bunların arasından ay numarası 7 olanlar, en son sıcaklığı 28'i aşanlar. `count()` temmuzda eksik ölçüm olup olmadığını da gösterir.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
ank = df[df["istasyon"] == "Ankara"]
july = ank[ank["tarih"].dt.month == 7]
hot = july[july["sicaklik"] > 28]
print(len(july), july["sicaklik"].count(), len(hot))
print(hot["tarih"].dt.day.tolist())
print(hot["sicaklik"].tolist())
```

Çıktı:

```
31 31 3
[9, 14, 23]
[31.0, 28.1, 31.5]
```

Temmuzun 31 gününün hepsinde ölçüm vardır. Sıcaklık yalnız 3 günde 28 °C'yi aşmıştır: 9 Temmuz (31,0), 14 Temmuz (28,1) ve 23 Temmuz (31,5). Modelde temmuz ortası sıcaklığı $12 + 11{,}5 = 23{,}5$ °C olduğundan 28 °C, ortalamanın 1,8 standart sapma üstündedir. ■

Alıştırma 15.4 (Doğrusal, Karesel ve Kübik İnterpolasyon). $[0, 2]$ aralığında 0,2 adımlı ızgaradaki e^x değerlerinden 3, 4 ve 8 numaralı konumdakileri silin. Eksikleri `interpolate` yönteminin `"linear"`, `"quadratic"` ve `"cubic"` seçenekleriyle doldurup hataları karşılaştırın.

Çözüm.
`"quadratic"` ve `"cubic"` seçenekleri SciPy'ın ikinci ve üçüncü dereceden spline interpolasyonunu kullanır ve indeksin değerlerini x koordinatı olarak alır. Spline'ları Polinomlar, İnterpolasyon ve Eğri Uydurma bölümünde ayrıntılı ele alıyoruz. `"linear"` değerleri eşit aralıklı sayar; bizim ızgaramız eşit aralıklı olduğu için bu bir şey değiştirmez.

```
import numpy as np
import pandas as pd

x = np.linspace(0, 2, 11)
s = pd.Series(np.exp(x), index=x)
s.iloc[[3, 4, 8]] = np.nan
true = np.exp(x[[3, 4, 8]])
for method in ["linear", "quadratic", "cubic"]:
    filled = s.interpolate(method=method)
    err = np.abs(filled.iloc[[3, 4, 8]].to_numpy() - true)
    print(f"{method:9}", err.round(5), f"{err.max():.1e}")
```

Çıktı:

```
linear    [0.07852 0.08392 0.09939] 9.9e-02
quadratic [0.00103 0.00332 0.00199] 3.3e-03
cubic     [0.00086 0.00087 0.0009 ] 9.0e-04
```

 CC BY-NC-SA 4.0

Doğrusal interpolasyonun en büyük hatası 0,099, karesel spline'ınki 0,0033, kübik spline'ınki 0,0009'dur. e^x dışbükey olduğundan doğru parçaları eğrinin hep üstünde kalır ve hata f'' ile orantılıdır; yüksek dereceli spline'lar eğriliği de izlediği için çok daha isabetlidir. ■

Alıştırma 15.5 (Çeyrekler Arası Açıklık Kuralıyla Aykırı Değerler). Bir veride çeyrekler arası açıklık $\text{ÇAA} = Q_3 - Q_1$ olmak üzere

$$\left[Q_1 - 1{,}5\,\text{ÇAA},\ Q_3 + 1{,}5\,\text{ÇAA}\right]$$

aralığının dışındaki gözlemlere aykırı değer diyelim. Ankara'nın yağışlı günlerindeki yağış miktarlarında doğrusal yüzdelik kuralına göre kaç aykırı değer vardır?

Çözüm.
Yağışlı Ankara günlerini iki koşulla süzeriz. `quantile([0.25, 0.75])` iki değerli bir Series döndürür; onu iki değişkene açabiliriz. Alt sınır negatif çıkacağından (yağış negatif olamaz) aykırı değerlerin hepsi üst taraftadır.

```
import numpy as np
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
wet = (df["istasyon"] == "Ankara") & (df["yagis"] > 0)
rain = df.loc[wet, "yagis"]
q1, q3 = rain.quantile([0.25, 0.75])
iqr = q3 - q1
low, high = q1 - 1.5 * iqr, q3 + 1.5 * iqr
out = rain[(rain < low) | (rain > high)]
print(len(rain), q1, q3, high)
print(len(out), round(len(out) / len(rain), 3))
print(round(np.exp(-(np.log(4) + 1.5 * np.log(3))), 3))
```

Çıktı:

```
88 1.275 5.4 11.5875
3 0.034
0.048
```

88 yağışlı günde $Q_1 = 1{,}275$ mm, $Q_3 = 5{,}4$ mm, çeyrekler arası açıklık $4{,}125$ mm ve üst sınır $11{,}5875$ mm'dir. Bu sınırı 3 gün, yani günlerin %3,4'ü aşmıştır. Veride hiçbir hata olmadığı hâlde kural birkaç günü aykırı saydı. Bunun nedeni dağılımın biçimidir: ortalaması r olan üstel dağılımda $Q_1 = r\ln(4/3)$ ve $Q_3 = r\ln 4$ olduğundan üst sınır $r(\ln 4 + 1{,}5\ln 3)$'tür ve bir gözlemin bu sınırı aşma olasılığı

$$e^{-(\ln 4 + 1{,}5\ln 3)} = \frac{1}{4 \cdot 3^{1{,}5}} \approx 0{,}048$$

olur. Çarpık dağılımlarda bu kural hatalı ölçümlerden çok dağılımın uzun kuyruğunu yakalar. ■

Alıştırma 15.6 (Grup İçinde Standartlaştırma). Her sıcaklık ölçümünü kendi istasyonunun ortalaması $\bar{x}_g$ ve standart sapması s_g ile $z = (x - \bar{x}_g)/s_g$ biçiminde standartlaştırın ve mutlak değerce en büyük z değerinin hangi ölçüme ait olduğunu bulun.

Çözüm.
Vektörizasyon ve Broadcasting bölümünde bir veri matrisini sütun sütun standartlaştırmıştık; burada sütunların yerini istasyonlar alır. `transform("mean")` ve `transform("std")` her satıra kendi istasyonunun ortalamasını ve standart sapmasını yazar, böylece formül bütün sütuna tek ifadeyle uygulanır. Doğrulama için her istasyonda z'nin ortalamasına ve standart sapmasına da bakarız.

 CC BY-NC-SA 4.0

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
g = df.groupby("istasyon")["sicaklik"]
df["z"] = (df["sicaklik"] - g.transform("mean")) / g.transform("std")
check = df.groupby("istasyon")["z"].agg(["mean", "std"])
print(check.abs().round(10))
row = df.loc[df["z"].abs().idxmax()]
print(row["tarih"].date(), row["istasyon"], row["sicaklik"],
      round(row["z"], 3))
```

Çıktı:

```
          mean  std
istasyon
Ankara     0.0  1.0
Erzurum    0.0  1.0
İzmir      0.0  1.0
2025-07-23 Ankara 31.5 2.304
```

Standartlaştırılmış değerlerin ortalaması her istasyonda 0, standart sapması 1'dir (yuvarlama hataları 10^{-10}'un altında). Mutlak değerce en büyük z değeri 2,304'tür ve Ankara'nın 23 Temmuz'daki 31,5 °C ölçümüne aittir. Bir kosinüs dalgası ile küçük bir gürültünün toplamında $|\,z\,|$ değerleri 3'e pek yaklaşmaz, çünkü $a\cos\theta$ dalgasının tepesi ortalamasından yalnız $\sqrt{2}$ standart sapma uzaktadır. ■

Alıştırma 15.7 (Yağışlı Günlerin Oranı ve Ortalaması). Her istasyon için yağışlı günlerin yıl içindeki oranını ve yağışlı günlerdeki ortalama yağışı hesaplayıp veri üretiminde kullanılan p ve r değerleriyle karşılaştırın.

Çözüm.
Önce yağışlı günleri süzer, sonra istasyona göre gruplayıp `agg` ile gün sayısını, ortalamayı ve en büyük değeri alırız. Oran, gün sayısının 365'e bölümüdür.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
wet = df[df["yagis"] > 0]
table = wet.groupby("istasyon")["yagis"].agg(["size", "mean", "max"])
table["oran"] = table["size"] / 365
print(table.round(3))
```

Çıktı:

```
          size   mean   max   oran
istasyon
Ankara      88  3.968  33.3  0.241
Erzurum    101  3.700  22.0  0.277
İzmir       55  7.344  33.7  0.151
```

Oranlar Ankara için 0,241 ($p=0{,}25$), Erzurum için 0,277 ($p=0{,}30$), İzmir için 0,151'dir ($p=0{,}20$). Yağışlı gün sayısı $n=365$ ve p parametreli binom dağılımına uyar. İzmir için beklenen sayı 73 gün, standart sapma $\sqrt{365\cdot 0{,}2\cdot 0{,}8}\approx 7{,}6$ gündür; gözlenen 55 gün beklenenin yaklaşık 2,4 standart sapma altındadır. Bu seyrek ama imkânsız olmayan bir sapmadır. Yağışlı günlerdeki ortalamalar 3,97, 3,70 ve 7,34 mm'dir ve $r=4$, 3,5 ve 8 değerlerine yakındır. ■

 CC BY-NC-SA 4.0

Alıştırma 15.8 (Yalnız Bir Tabloda Geçen Anahtarlar). `a` tablosunda Ankara, Erzurum ve İzmir'in ortalama sıcaklıkları, `b` tablosunda Ankara, İzmir, Trabzon ve Van'ın plaka numaraları var. `outer` birleştirme ve `indicator=True` parametresiyle yalnız bir tabloda geçen istasyonları bulun.

Çözüm.
`indicator=True` sonuca `_merge` adlı bir sütun ekler. Bu sütunun değeri, anahtarın iki tabloda da (`both`), yalnız solda (`left_only`) ya da yalnız sağda (`right_only`) bulunduğunu söyler.

```
import pandas as pd

a = pd.DataFrame({"istasyon": ["Ankara", "Erzurum", "İzmir"],
                  "sicaklik": [12.14, 5.79, 17.85]})
b = pd.DataFrame({"istasyon": ["Ankara", "İzmir", "Trabzon", "Van"],
                  "plaka": [6, 35, 61, 65]})
m = a.merge(b, on="istasyon", how="outer", indicator=True)
print(m)
print(m["_merge"].value_counts().to_dict())
print(m.loc[m["_merge"] != "both", "istasyon"].tolist())
```

Çıktı:

```
  istasyon  sicaklik  plaka      _merge
0   Ankara     12.14    6.0        both
1  Erzurum      5.79    NaN   left_only
2  Trabzon       NaN   61.0  right_only
3      Van       NaN   65.0  right_only
4    İzmir     17.85   35.0        both
{'right_only': 2, 'both': 2, 'left_only': 1}
['Erzurum', 'Trabzon', 'Van']
```

Ankara ve İzmir iki tabloda da vardır; Erzurum yalnız solda, Trabzon ve Van yalnız sağdadır. Anahtar kümeleriyle söylersek aradığımız küme simetrik farktır: $(A_L \cup A_R) \setminus (A_L \cap A_R)$. ■

Alıştırma 15.9 (Aylık Ortalamaların Yarı Açıklığı). `pivot_table` ile kurduğunuz ay ve istasyon tablosundan her istasyonun aylık ortalamalarının yarı açıklığını, yani en büyük ile en küçük aylık ortalamanın farkının yarısını hesaplayıp modeldeki a genliğiyle karşılaştırın.

Çözüm.
Tablonun her sütunu bir istasyondur. `idxmax()` ve `idxmin()` her sütunda en büyük ve en küçük değerin etiketini, yani en sıcak ve en soğuk ayın numarasını verir; `max() - min()` de sütun sütun hesaplanır.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
df["ay"] = df["tarih"].dt.month
pt = df.pivot_table(index="ay", columns="istasyon", values="sicaklik",
                    aggfunc="mean")
print(pt.idxmax().to_dict(), pt.idxmin().to_dict())
print(((pt.max() - pt.min()) / 2).round(2).to_dict())
```

Çıktı:

```
{'Ankara': 7, 'Erzurum': 7, 'İzmir': 7} {'Ankara': 1, 'Erzurum': 1, 'İzmir': 1}
{'Ankara': 11.82, 'Erzurum': 13.7, 'İzmir': 8.23}
```

Üç istasyonda da en sıcak ay temmuz, en soğuk ay ocaktır. Yarı açıklıklar 11,82, 13,70 ve 8,23'tür; model genlikleri 11,5, 14 ve 8,5'tir. Aylık ortalama dalganın tepesini biraz düzleştirir: tepe çevresindeki

 CC BY-NC-SA 4.0

31 günlük bir pencerede kosinüs dalgasının ortalaması, tepe değerinin yaklaşık $0{,}988$ katıdır. Kalan farklar ay ortalamalarına giren gürültüden gelir; Önerme 15.1 gereği bir ay ortalamasındaki gürültünün standart sapması yaklaşık $2{,}5/\sqrt{30} \approx 0{,}46$'dır. ■

Alıştırma 15.10 (Mevsim Dalgasını En Küçük Karelerle Kestirmek). $\omega = 2\pi/365$ olmak üzere Ankara'nın eksik olmayan ölçümlerine

$$T(t) \approx c_0 + c_1 \cos \omega t + c_2 \sin \omega t$$

modelini en küçük kareler yöntemiyle uydurun ve buradan yıllık ortalama m'yi, genlik a'yı ve en soğuk günü veren t_0'ı kestirin.

Çözüm.
Model katsayılarda doğrusaldır; dolayısıyla bu bir doğrusal en küçük kareler problemidir. Sütunları 1, $\cos \omega t$ ve $\sin \omega t$ olan A matrisiyle $Ac \approx y$ sistemini NumPy ile Lineer Cebir bölümündeki `np.linalg.lstsq` ile çözeriz. Gün numarası t, tarihten 1 Ocak çıkarılarak bulunur. Katsayılardan parametrelere geçmek için

$$m - a\cos\omega(t - t_0) = m - a\cos\omega t_0 \cos \omega t - a \sin \omega t_0 \sin \omega t$$

özdeşliğini kullanırız: $c_0 = m$, $c_1 = -a\cos\omega t_0$ ve $c_2 = -a \sin \omega t_0$. Buradan $a = \sqrt{c_1^2 + c_2^2}$ ve $\omega t_0 = \operatorname{atan2}\,(-c_2, -c_1)$ bulunur.

```
import numpy as np
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
ank = df[df["istasyon"] == "Ankara"].dropna(subset=["sicaklik"])
t = (ank["tarih"] - pd.Timestamp("2025-01-01")).dt.days.to_numpy()
y = ank["sicaklik"].to_numpy()
w = 2 * np.pi / 365
A = np.column_stack([np.ones(t.size), np.cos(w * t), np.sin(w * t)])
coef, *_ = np.linalg.lstsq(A, y, rcond=None)
c0, c1, c2 = coef
amp = np.hypot(c1, c2)
t0 = np.arctan2(-c2, -c1) / w
resid = y - A @ coef
print(len(t), np.round(coef, 3))
print(round(c0, 2), round(amp, 2), round(t0, 1))
print(round(resid.std(ddof=3), 3))
```

Çıktı:

```
357 [ 12.005 -10.924  -2.94 ]
12.01 11.31 15.3
2.65
```

357 ölçümden $m \approx 12{,}01$, $a \approx 11{,}31$ ve $t_0 \approx 15{,}3$ gün kestirilir; gerçek değerler 12, 11,5 ve 15'tir. Artıkların standart sapması $2{,}65$'tir ve gürültünün $2{,}5$'lik standart sapmasına yakındır; `ddof=3`, üç parametre kestirdiğimiz için paydayı $n - 3$ yapar. ■

Alıştırma 15.11 (Mevsim Etkisi Çıkarılınca Korelasyon). İstasyonların sıcaklıkları arasındaki korelasyonu, her istasyonun 31 günlük merkezî hareketli ortalamasını çıkardıktan sonra yeniden hesaplayın.

Çözüm.
`pivot` ile geniş tabloyu kurarız. DataFrame'in `interpolate` ve `rolling` yöntemleri her sütuna ayrı ayrı uygulanır; böylece üç istasyonun hareketli ortalaması tek satırda bulunur. Ölçülen değerden hareketli

 CC BY-NC-SA 4.0

ortalamayı çıkarınca mevsim dalgası gider, gürültü kalır. Eksik günler ile baştaki ve sondaki 15'er gün `NaN` olur ve `corr` bunları atar; `count()` her sütunda kullanılabilen değer sayısını gösterir.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
wide = df.pivot(index="tarih", columns="istasyon", values="sicaklik")
trend = wide.interpolate().rolling(31, center=True).mean()
anom = wide - trend
print(wide.corr().round(3))
print(anom.corr().round(3))
print(anom.count().to_dict())
```

Çıktı:

```
istasyon  Ankara  Erzurum  İzmir
istasyon
Ankara     1.000    0.920  0.876
Erzurum    0.920    1.000  0.901
İzmir      0.876    0.901  1.000
istasyon  Ankara  Erzurum  İzmir
istasyon
Ankara     1.000    0.057 -0.062
Erzurum    0.057    1.000 -0.016
İzmir     -0.062   -0.016  1.000
{'Ankara': 330, 'Erzurum': 322, 'İzmir': 330}
```

Ham sıcaklıklar arasındaki 0,88–0,92 korelasyonlar, mevsim etkisi çıkarılınca $-0{,}06$ ile 0,06 arasına iner. Modelde istasyonların günlük gürültüleri bağımsızdır; yüksek korelasyonun tek kaynağı ortak mevsim dalgasıydı. Gerçek verilerde komşu istasyonların sapmaları da birlikte hareket eder, çünkü aynı hava sistemlerinden etkilenirler; ama o ilişkiyi görmek için de önce ortak eğilimi çıkarmak gerekir. Korelasyonun 0 olmasının bağımsızlık anlamına gelmediğini Olasılık Teorisi notlarındaki örnek gösterir. ■

Alıştırma 15.12 (Yıllık Yağışın Yarısı). İzmir'de 2025 yılı boyunca düşen toplam yağışın yarısı hangi gün tamamlanmıştır?

Çözüm.
`cumsum()` her gün için yılbaşından o güne kadar düşen toplam yağışı verir. `cum >= total / 2` boolean serisi o günden itibaren hep `True`'dur. Boolean bir serinin `idxmax()` değeri ilk `True` değerin etiketi olduğundan aradığımız günü verir. Son satır, aylık toplamların en büyüğünün hangi ayda olduğunu da yazdırır.

```
import pandas as pd

df = pd.read_csv("olcumler.csv", parse_dates=["tarih"])
izm = df[df["istasyon"] == "İzmir"].set_index("tarih")["yagis"]
cum = izm.cumsum()
total = cum.iloc[-1]
reached = cum >= total / 2
day = reached.idxmax()                # ilk True değerin etiketi
print(round(total, 1), day.date(), round(cum.loc[day], 1))
print(izm.resample("ME").sum().idxmax().month)
```

Çıktı:

```
403.9 2025-06-06 203.3
6
```

Yıllık toplam 403,9 mm'dir ve yarısı (201,95 mm) 6 Haziran'da aşılmıştır; o günün sonunda birikimli toplam 203,3 mm'dir. Yılın ortası 2 Temmuz olduğu hâlde yarıya daha erken ulaşıldı: yılın en yağışlı günü (33,7 mm) 6 Haziran'dır ve en yağışlı ay da hazirandır. Modelde yağış olasılığı mevsime bağlı olmadığından bu erkenlik yalnız bir rastlantıdır. ■

Bu bölümde pandas'ın iki temel yapısını, Series ve DataFrame'i tanıdık. Etiketlerin işlemlerde nasıl hizalandığını, CSV dosyalarını okuyup yazmayı, `loc` ve `iloc` ile seçimi, eksik değerleri, betimsel istatistikleri, `groupby` ile ayır, uygula, birleştir düzenini, `merge` ile tablo birleştirmeyi ve zaman serilerini gördük. Bu, dersin son bölümüdür: Python'ın temel yapılarından başlayıp NumPy, Matplotlib, SymPy ve SciPy'dan geçerek bir veri tablosunun analizine geldik. Burada hesapladığımız ortalama, varyans, korelasyon ve yüzdeliklerin arkasındaki kuram için Matematiksel İstatistik notlarına geçebilir, dersin genel planı için Müfredat ve Giriş sayfasına dönebilirsiniz.

 CC BY-NC-SA 4.0