CHAPTER 5

Equation Solving

In the previous chapters, we have discussed general methodologies and techniques, namely, array-based numerical computing, symbolic computing, and visualization. These methods are the cornerstones of scientific computing that make up a fundamental toolset we have at our disposal when attacking computational problems.

Starting from this chapter, we begin to explore how to solve problems from different domains of applied mathematics and computational sciences, using the basic techniques introduced in the previous chapters. The topic of this chapter is algebraic equation solving. This is a broad topic that requires the application of theory and approaches from multiple fields of mathematics. In particular, when discussing equation solving, we have to distinguish between univariate and multivariate equations (i.e., equations that contain one unknown variable or many unknown variables). In addition, we need to distinguish between linear and nonlinear equations. This classification is useful because solving equations of these different types requires applying different mathematical methods and approaches.

We begin with linear equation systems, which are tremendously useful and have important applications in every field of science. The reason for this universality is that linear algebra theory allows us to straightforwardly solve linear equations, while nonlinear equations are difficult to solve in general and typically require more complicated and computationally demanding methods. Because linear systems are readily solvable, they are also an important tool for local approximations of nonlinear systems. For example, by considering small variations from an expansion point, a nonlinear system can often be approximated by a linear system in the local vicinity of the expansion point. However, a linearization can only describe local properties, and for global analysis of nonlinear problems, other techniques are required. Such methods typically employ iterative approaches for gradually constructing an increasingly accurate estimate of the solution.

R. Johansson, *Numerical Python*, https://doi.org/10.1007/978-1-4842-4246-9_5

In this chapter, we use SymPy for solving equations symbolically, when possible, and use the linear algebra module from the SciPy library for numerically solving linear equation systems. For tackling nonlinear problems, we will use the root-finding functions in the `optimize` module of SciPy.

SciPy SciPy is a Python library, the collective name of the scientific computing environment for Python, and the umbrella organization for many of the core libraries for scientific computing with Python. The library, `scipy`, is in fact rather a collection of libraries for high-level scientific computing, which are more or less independent of each other. The SciPy library is built on top of NumPy, which provides the basic array data structures and fundamental operations on such arrays. The modules in SciPy provide domain-specific high-level computation methods, such as routines for linear algebra, optimization, interpolation, integration, and much more. At the time of writing, the most recent version of SciPy is 1.1.0. See `www.scipy.org` for more information.

Importing Modules

The SciPy module `scipy` should be considered a collection of modules that are selectively imported when required. In this chapter we will use the `scipy.linalg` module, for solving linear systems of equations, and the `scipy.optimize` module, for solving nonlinear equations. In this chapter we assume that these modules are imported as:

```
In [1]: from scipy import linalg as la
In [2]: from scipy import optimize
```

In this chapter we also use the NumPy, SymPy, and Matplotlib libraries introduced in the previous chapters, and we assume that those libraries are imported following the previously introduced convention:

```
In [3]: import sympy
In [4]: sympy.init_printing()
In [5]: import numpy as np
In [6]: import matplotlib.pyplot as plt
```

To get the same behavior in both Python 2 and Python 3 with respect to integer division, we also include the following statement (which is only necessary in Python 2):

```
In [7]: from __future__ import division
```

Linear Equation Systems

An important application of linear algebra is solving systems of linear equations. We have already encountered linear algebra functionality in the SymPy library, in Chapter 3. There are also linear algebra modules in the NumPy and SciPy libraries, `numpy.linalg` and `scipy.linalg`, which together provide linear algebra routines for numerical problems, that is, for problems that are completely specified in terms of numerical factors and parameters.

In general, a linear equation system can be written on the form

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1,$$

$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2,$$

$$\ldots$$

$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m.$$

This is a linear system of m equations in n unknown variables $\{x_1, x_2, \ldots, x_n\}$, where a_{mn} and b_m are known parameters or constant values. When working with linear equation systems, it is convenient to write the equations in matrix form:

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix},$$

or simply $Ax = b$, where A is a $m \times n$ matrix, b is a $m \times 1$ matrix (or m-vector), and x is the unknown $n \times 1$ solution matrix (or n-vector). Depending on the properties of the matrix A, the solution vector x may or may not exist, and if a solution does exist, it is not necessarily unique. However, if a solution exists, then it can be interpreted as an expression for the vector b as a linear combination of the columns of the matrix A, where the coefficients are given by the elements in the solution vector x.

A system for which $n < m$ is said to be underdetermined, because it has fewer equations than unknown and therefore cannot completely determine a unique solution. If, on the other hand, $m > n$, then the equations are said to be overdetermined. This will in general lead to conflicting constraints, resulting in that a solution does not exist.

Square Systems

Square systems with $m = n$ is an important special case. It corresponds to the situation where the number of equations equals the number of unknown variables, and it can therefore potentially have a unique solution. In order for a unique solution to exist, the matrix A must be *nonsingular*, in which case the inverse of A exists, and the solution can be written as $x = A^{-1}b$. If the matrix A is singular, that is, the rank of the matrix is less than n, $\text{rank}(A) < n$, or, equivalently, if its determinant is zero, $\det A = 0$, then the equation $Ax = b$ can either have no solution or infinitely many solutions, depending on the right-hand side vector b. For a matrix with rank deficiency, $\text{rank}(A) < n$, there are columns or rows that can be expressed as linear combinations of other columns or vectors, and they therefore correspond to equations that do not contain any new constraints, and the system is really underdetermined. Computing the rank of the matrix A that defines a linear equation system is therefore a useful method that can tell us whether the matrix is singular or not and therefore whether there exists a solution or not.

When A has full rank, the solution is guaranteed to exist. However, it may or may not be possible to accurately compute the solution. The *condition number* of the matrix, $\text{cond}(A)$, gives a measure of how well or poorly conditioned a linear equation system is. If the conditioning number is close to 1, if the system is said to be *well conditioned* (a condition number 1 is ideal), and if the condition number is large, the system is said to be *ill-conditioned*. The solution to an equation system that is ill-conditioned can have large errors. An intuitive interpretation of the condition number can be obtained from a simple error analysis. Assume that we have a linear equation system on the form $Ax = b$, where x is the solution vector. Now consider a small variation of b, say δb, which gives a corresponding change in the solution, δx, given by $A(x+\delta x) = b+\delta b$. Because of linearity of the equation, we have $A\delta x = \delta b$. An important question to consider now is: how large is the relative change in x compared to the relative change in b? Mathematically we can formulate this question in terms of the ratios of the norms of these vectors. Specifically, we are interested in comparing $\|\delta x\|/\|x\|$ and $\|\delta b\|/\|b\|$, where $\|x\|$ denotes the norm of x. Using the matrix norm relation $\|Ax\| \leq \|A\| \cdot \|x\|$, we can write

$$\frac{\|\delta x\|}{\|x\|} = \frac{\|A^{-1}\delta b\|}{\|x\|} \leq \frac{\|A^{-1}\| \cdot \|\delta b\|}{\|x\|} = \frac{\|A^{-1}\| \cdot \|b\|}{\|x\|} \cdot \frac{\|\delta b\|}{\|b\|} \leq \|A^{-1}\| \cdot \|A\| \cdot \frac{\|\delta b\|}{\|b\|}$$

A bound of the relative error in the solution x, given a relative error in the b vector, is therefore given by $\text{cond}(A) \equiv \|A^{-1}\| \cdot \|A\|$, which by definition is the condition number of the matrix A. This means that for linear equation systems characterized by a matrix A that is ill-conditioned, even a small perturbation in the b vector can give large errors in the solution vector x. This is particularly relevant in numerical solution using floating-point numbers, which are only approximations to real numbers. When solving a system of linear equations, it is therefore important to look at the condition number to estimate the accuracy of the solution.

The rank, condition number, and norm of a symbolic matrix can be computed in SymPy using the `Matrix` methods `rank`, `condition_number`, and `norm`, and for numerical problems, we can use the NumPy functions `np.linalg.matrix_rank`, `np.linalg.cond`, and `np.linalg.norm`. For example, consider the following system of two linear equations:

$$2x_1 + 3x_2 = 4$$
$$5x_1 + 4x_2 = 3$$

These two equations correspond to lines in the (x_1,x_2) plane, and their intersection is the solution to the equation system. As can be seen in Figure 5-1, which graphs the lines corresponding to the two equations, the lines intersect at $(-1, 2)$.

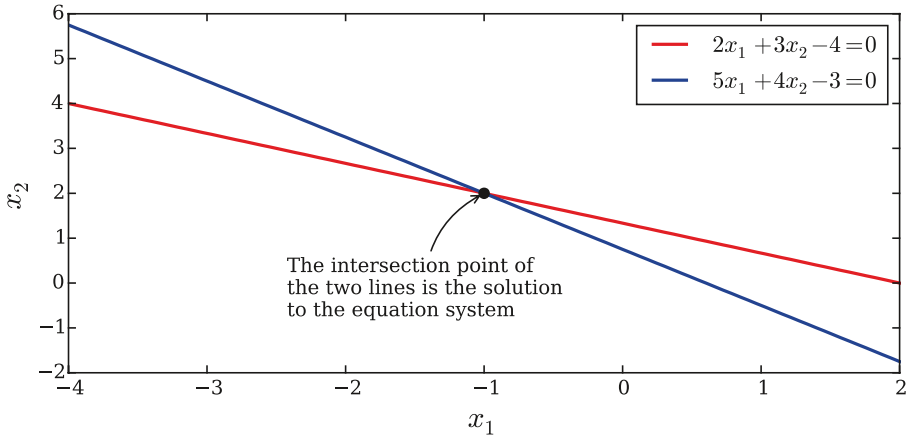

Figure 5-1. *Graphical representation of a system of two linear equations*

We can define this problem in SymPy by creating matrix objects for A and b and computing the rank, condition number, and norm of the matrix A using

```
In [8]: A = sympy.Matrix([[2, 3], [5, 4]])
In [9]: b = sympy.Matrix([4, 3])
In [10]: A.rank()
Out[10]: 2
In [11]: A.condition_number()
```

Out[11]: $\dfrac{\sqrt{27+2\sqrt{170}}}{\sqrt{27-2\sqrt{170}}}$

```
In [12]: sympy.N(_)
Out[12]: 7.58240137440151
In [13]: A.norm()
```

Out[13]: $3\sqrt{6}$

We can do the same thing in NumPy/SciPy using NumPy arrays for A and b and functions from the `np.linalg` and `scipy.linalg` modules:

```
In [14]: A = np.array([[2, 3], [5, 4]])
In [15]: b = np.array([4, 3])
In [16]: np.linalg.matrix_rank(A)
Out[16]: 2
In [17]: np.linalg.cond(A)
Out[17]: 7.5824013744
In [18]: np.linalg.norm(A)
Out[18]: 7.34846922835
```

A direct approach to solving the linear problem is to compute the inverse of the matrix A and multiply it with the vector b, as used, for example, in the previous analytical discussions. However, this is not the most efficient computational method to find the solution vector x. A better method is LU factorization of the matrix A, such that $A = LU$ and where L is a lower triangular matrix and U is an upper triangular matrix. Given L and U, the solution vector x can be efficiently constructed by first solving $Ly = b$ with forward substitution and then solving $Ux = y$ with backward substitution. Owning to the fact that L and U are triangular matrices, these two procedures are computationally efficient.

In SymPy we can perform a symbolic LU factorization by using the `LUdecomposition` method of the `sympy.Matrix` class. This method returns new `Matrix` objects for the L and U matrices, as well as a row swap matrix. When we are interested in solving an equation system $Ax = b$, we do not explicitly need to calculate the L and U matrices, but rather we can use the `LUsolve` method, which performs the LU factorization internally and solves the equation system using those factors. Returning to the previous example, we can compute the L and U factors and solve the equation system using

```
In [19]: A = sympy.Matrix([[2, 3], [5, 4]])
In [20]: b = sympy.Matrix([4, 3])
In [21]: L, U, _ = A.LUdecomposition()
In [22]: L
```

Out[22]:
$$\begin{bmatrix} 1 & 0 \\ 5/2 & 1 \end{bmatrix}$$

```
In [23]: U
```

Out[23]:
$$\begin{bmatrix} 2 & 3 \\ 0 & -7/2 \end{bmatrix}$$

```
In [24]: L * U
```

Out[24]:
$$\begin{bmatrix} 2 & 3 \\ 5 & 4 \end{bmatrix}$$

```
In [25]: x = A.solve(b); x  # equivalent to A.LUsolve(b)
```

Out[25]:
$$\begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

For numerical problems we can use the `la.lu` function form SciPy's linear algebra module. It returns a permutation matrix P and the L and U matrices, such that $A = PLU$. Like with SymPy, we can solve the linear system $Ax = b$ without explicitly calculating the L and U matrices by using the `la.solve` function, which takes the A matrix and the b vector as arguments. This is in general the preferred method for solving numerical linear equation systems using SciPy.

```
In [26]: A = np.array([[2, 3], [5, 4]])
In [27]: b = np.array([4, 3])
In [28]: P, L, U = la.lu(A)
In [29]: L
Out[29]: array([[ 1. ,  0. ],
                [ 0.4,  1. ]])
```

```
In [30]: U
Out[30]: array([[ 5. ,  4. ],
                [ 0. ,  1.4]])
In [31]: P.dot(L.dot(U))
Out[31]: array([[ 2., 3.],
                [ 5., 4.]])
In [32]: la.solve(A, b)
Out[32]: array([-1.,  2.])
```

The advantage of using SymPy is of course that we may obtain exact results and we can also include symbolic variables in the matrices. However, not all problems are solvable symbolically, or it may give exceedingly lengthy results. The advantage of using a numerical approach with NumPy/SciPy, on the other hand, is that we are guaranteed to obtain a result, although it will be an approximate solution due to floating-point errors. See the following code (`In [38]`) for an example that illustrates the differences between the symbolic and numerical approaches and for an example that show numerical approaches can be sensitive for equation systems with large condition number. In this example we solve the equation system

$$\begin{pmatrix} 1 & \sqrt{p} \\ 1 & \dfrac{1}{\sqrt{p}} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

which for $p = 1$ is singular and for p in the vicinity of one is ill-conditioned. Using SymPy, the solution is easily found to be

```
In [33]: p = sympy.symbols("p", positive=True)
In [34]: A = sympy.Matrix([[1, sympy.sqrt(p)], [1, 1/sympy.sqrt(p)]])
In [35]: b = sympy.Matrix([1, 2])
In [36]: x = A.solve(b)
In [37]: x
```

Out[37]:

$$\begin{pmatrix} \dfrac{2p-1}{p-1} \\ -\dfrac{\sqrt{p}}{p-1} \end{pmatrix}$$

A comparison between this symbolic solution and the numerical solution is shown in Figure 5-2. Here the errors in the numerical solution are due to numerical floating-point errors, and the numerical errors are significantly larger in the vicinity of $p = 1$, where the system has a large condition number. Also, if there are other sources of errors in either A or b, the corresponding errors in x can be even more severe.

```
In [38]: # Symbolic problem specification
    ...: p = sympy.symbols("p", positive=True)
    ...: A = sympy.Matrix([[1, sympy.sqrt(p)], [1, 1/sympy.sqrt(p)]])
    ...: b = sympy.Matrix([1, 2])
    ...:
    ...: # Solve symbolically
    ...: x_sym_sol = A.solve(b)
    ...: Acond = A.condition_number().simplify()
    ...:
    ...: # Numerical problem specification
    ...: AA = lambda p: np.array([[1, np.sqrt(p)], [1, 1/np.sqrt(p)]])
    ...: bb = np.array([1, 2])
    ...: x_num_sol = lambda p: np.linalg.solve(AA(p), bb)
    ...:
    ...: # Graph the difference between the symbolic (exact) and numerical
           results.
    ...: fig, axes = plt.subplots(1, 2, figsize=(12, 4))
    ...:
    ...: p_vec = np.linspace(0.9, 1.1, 200)
    ...: for n in range(2):
    ...:     x_sym = np.array([x_sym_sol[n].subs(p, pp).evalf() for pp in
             p_vec])
    ...:     x_num = np.array([x_num_sol(pp)[n] for pp in p_vec])
    ...:     axes[0].plot(p_vec, (x_num - x_sym)/x_sym, 'k')
    ...: axes[0].set_title("Error in solution\n(numerical - symbolic)/
         symbolic")
    ...: axes[0].set_xlabel(r'$p$', fontsize=18)
    ...:
    ...: axes[1].plot(p_vec, [Acond.subs(p, pp).evalf() for pp in p_vec])
    ...: axes[1].set_title("Condition number")
    ...: axes[1].set_xlabel(r'$p$', fontsize=18)
```

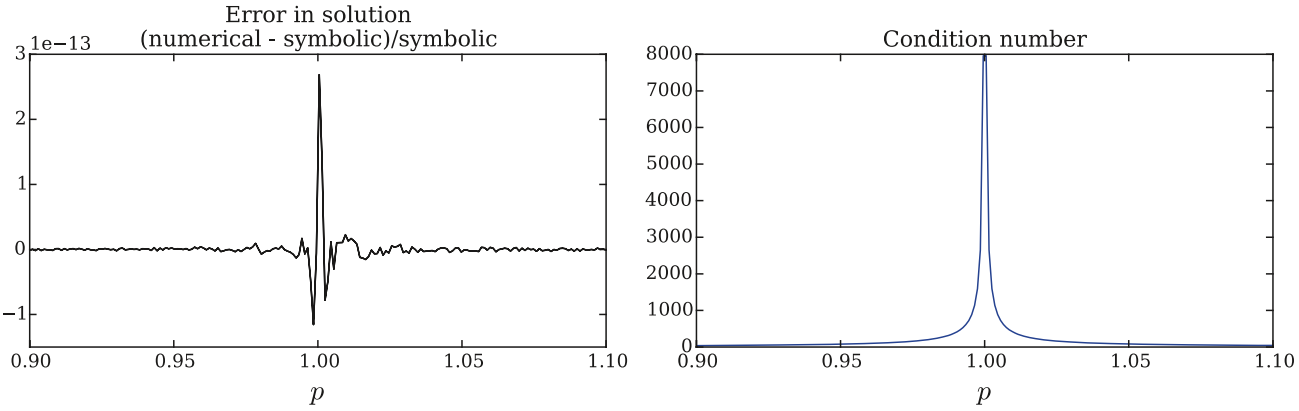

Figure 5-2. *Graph of the relative numerical errors (left) and condition number (right) as a function of the parameter p*

Rectangular Systems

Rectangular systems, with $m \neq n$, can be either underdetermined or overdetermined. Underdetermined systems have more variables than equations, so the solution cannot be fully determined. Therefore, for such a system, the solution must be given in terms of the remaining free variables. This makes it difficult to treat this type of problem numerically, but a symbolic approach can often be used instead.

For example, consider the underdetermined linear equation system

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 8 \end{pmatrix}.$$

Here we have three unknown variables, but only two equations impose constraints on the relations between these variables. By writing this equation as $Ax - b = 0$, we can use the SymPy `sympy.solve` function to obtain a solution for x_1 and x_2 parameterized by the remaining free variable x_3:

```
In [39]: x_vars = sympy.symbols("x_1, x_2, x_3")
In [40]: A = sympy.Matrix([[1, 2, 3], [4, 5, 6]])
In [41]: x = sympy.Matrix(x_vars)
In [42]: b = sympy.Matrix([7, 8])
In [43]: sympy.solve(A*x - b, x_vars)
Out[43]: {x_1 = x_3 - 19/3, x_2 = -2x_3 + 20/3}
```

Here we obtained the symbolic solution $x_1 = x_3 - 19/3$ and $x_2 = -2x_3+20/3$, which defines a line in the three-dimensional space spanned by $\{x_1, x_2, x_3\}$. Any point on this line therefore satisfies this underdetermined equation system.

On the other hand, if the system is overdetermined and has more equations than unknown variables, $m > n$, then we have more constraints than degrees of freedom, and in general there is no exact solution to such a system. However, it is often interesting to find an approximate solution to an overdetermined system. An example of when this situation arises is data fitting: say we have a model where a variable y is a quadratic polynomial in the variable x, so that $y = A+Bx+Cx^2$, and that we would like to fit this model to experimental data. Here y is nonlinear in x, but y is linear in the three unknown coefficients A, B, and C, and this fact can be used to write the model as a linear equation system. If we collect data for m pairs $\left\{\left(x_i, y_i\right)\right\}_{i=1}^{m}$ of the variables x and y, we can write the model as an $m \times 3$ equation system:

$$\begin{pmatrix} 1 & x_1 & x_1^2 \\ \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}.$$

If $m = 3$, we can solve for the unknown model parameters A, B, and C, assuming the system matrix is nonsingular. However, it is intuitively clear that if the data is noisy and if we were to use more than three data points, we should be able to get a more accurate estimate of the model parameters.

However, for $m > 3$, there is in general no exact solution, and we need to introduce an approximate solution that gives the *best fit* for the overdetermined system $Ax \approx b$. A natural definition of the best fit for this system is to minimize the sum of square errors, $\min_x \sum_{i=1}^{m} (r_i)^2$, where $r = b - Ax$ is the residual vector. This leads to the *least square* solution of the problem $Ax \approx b$, which minimizes the distances between the data points and the linear solution. In SymPy we can solve for the least square solution of an overdetermined system using the `solve_least_squares` method, and for numerical problems, we can use the SciPy function `la.lstsq`.

The following code demonstrates how the SciPy `la.lstsq` method can be used to fit the example model considered in the preceding section, and the result is shown in Figure 5-3. We first define the true parameters of the model, and then we simulate

measured data by adding random noise to the true model relation. The least square problem is then solved using the `la.lstsq` function, which in addition to the solution vector x also returns the total sum of square errors (the residual `r`), the rank `rank` and the singular values `sv` of the matrix A. However, in the following example, we only use the solution vector `x`.

```
In [44]: # define true model parameters
    ...: x = np.linspace(-1, 1, 100)
    ...: a, b, c = 1, 2, 3
    ...: y_exact = a + b * x + c * x**2
    ...:
    ...: # simulate noisy data
    ...: m = 100
    ...: X = 1 - 2 * np.random.rand(m)
    ...: Y = a + b * X + c * X**2 + np.random.randn(m)
    ...:
    ...: # fit the data to the model using linear least square
    ...: A = np.vstack([X**0, X**1, X**2])  # see np.vander for alternative
    ...: sol, r, rank, sv = la.lstsq(A.T, Y)
    ...:
    ...: y_fit = sol[0] + sol[1] * x + sol[2] * x**2
    ...: fig, ax = plt.subplots(figsize=(12, 4))
    ...:
    ...: ax.plot(X, Y, 'go', alpha=0.5, label='Simulated data')
    ...: ax.plot(x, y_exact, 'k', lw=2, label='True value $y = 1 + 2x +
         3x^2$')
    ...: ax.plot(x, y_fit, 'b', lw=2, label='Least square fit')
    ...: ax.set_xlabel(r"$x$", fontsize=18)
    ...: ax.set_ylabel(r"$y$", fontsize=18)
    ...: ax.legend(loc=2)
```

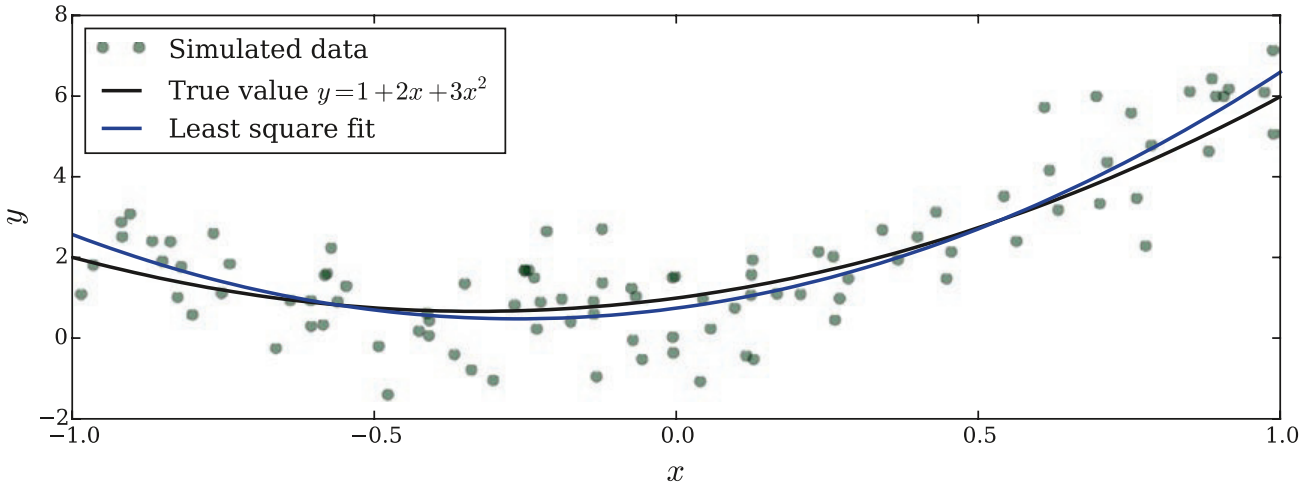

Figure 5-3. *Linear least square fit*

A good fit of data to a model obviously requires that the model used to describe the data correspond well to the underlying process that produced the data. In the following example (`In [45]`), and in Figure 5-4, we fit the same data used in the previous example to linear model and to a higher-order polynomial model (up to order 15). The former case corresponds to underfitting, where we have used a too simple model for the data, and the latter case corresponds to overfitting, where we have used a too complex model for the data, and thus fit the model not only to the underlying trend but also to the measurement noise. Using an appropriate model is an important and delicate aspect of data fitting.

```
In [45]: # fit the data to the model using linear least square:
    ...: # 1st order polynomial
    ...: A = np.vstack([X**n for n in range(2)])
    ...: sol, r, rank, sv = la.lstsq(A.T, Y)
    ...: y_fit1 = sum([s * x**n for n, s in enumerate(sol)])
    ...:
    ...: # 15th order polynomial
    ...: A = np.vstack([X**n for n in range(16)])
    ...: sol, r, rank, sv = la.lstsq(A.T, Y)
    ...: y_fit15 = sum([s * x**n for n, s in enumerate(sol)])
    ...:
    ...: fig, ax = plt.subplots(figsize=(12, 4))
    ...: ax.plot(X, Y, 'go', alpha=0.5, label='Simulated data')
```

```
...: ax.plot(x, y_exact, 'k', lw=2, label='True value $y = 1 + 2x +
     3x^2$')
...: ax.plot(x, y_fit1, 'b', lw=2, label='Least square fit [1st
     order]')
...: ax.plot(x, y_fit15, 'm', lw=2, label='Least square fit [15th
     order]')
...: ax.set_xlabel(r"$x$", fontsize=18)
...: ax.set_ylabel(r"$y$", fontsize=18)
...: ax.legend(loc=2)
```

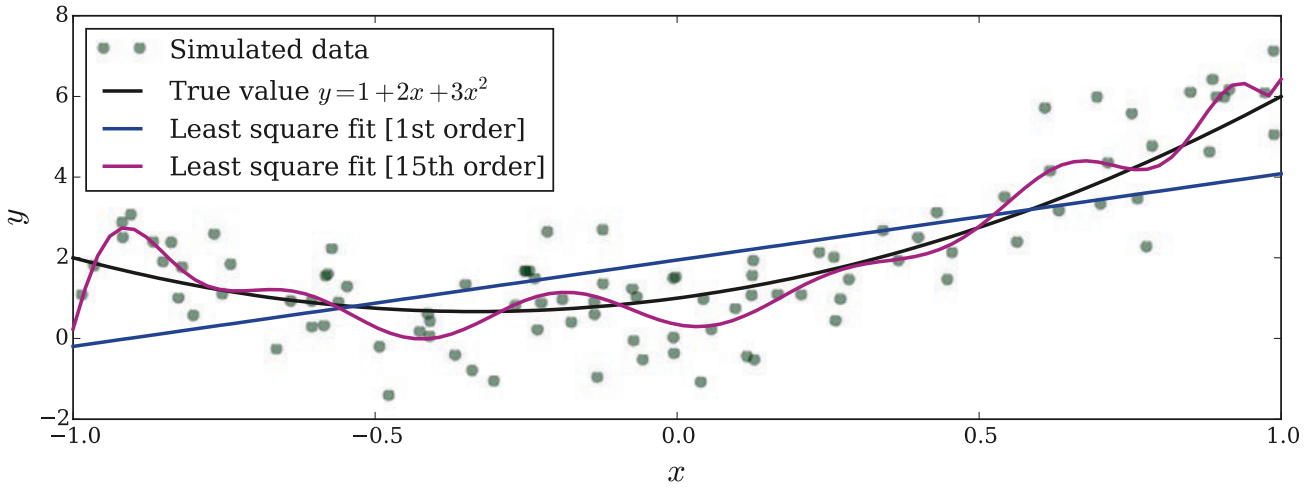

Figure 5-4. *Graph demonstrating underfitting and overfitting of data using the linear least square method*

Eigenvalue Problems

A special system of equations of great theoretical and practical importance is the eigenvalue equation $Ax = \lambda x$, where A is a $N \times N$ square matrix, x is an unknown vector, and λ is an unknown scalar. Here x is an eigenvector and λ an eigenvalue of the matrix A. The eigenvalue equation $Ax = \lambda x$ closely resembles the linear equation system $Ax = b$, but note that here both x and λ are unknown, so we cannot directly apply the same techniques to solve this equation. A standard approach to solve this eigenvalue problem is to rewrite the equation as $(A - I\lambda)x = 0$ and note that for there to exist a nontrivial solution, $x \neq 0$, the matrix $A - I\lambda$ must be singular, and its determinant must be zero, $\det(A - I\lambda) = 0$. This gives a polynomial equation (the characteristic polynomial) of Nth order whose N roots give the N eigenvalues $\{\lambda_n\}_{n=1}^{N}$. Once the eigenvalues are known, the equation $(A - I\lambda_n)x_n = 0$ can be solved for the nth eigenvector x_n using standard forward substitution.

Both SymPy and the linear algebra package in SciPy contain solvers for eigenvalue problems. In SymPy, we can use the `eigenvals` and `eigenvects` methods of the `Matrix` class, which are able to compute the eigenvalues and eigenvectors of some matrices with elements that are symbolic expressions. For example, to compute the eigenvalues and eigenvectors of symmetric 2×2 matrix with symbolic elements, we can use

```
In [46]: eps, delta = sympy.symbols("epsilon, Delta")
In [47]: H = sympy.Matrix([[eps, delta], [delta, -eps]])
In [48]: H
```

Out[48]: $\begin{pmatrix} \varepsilon & \Delta \\ \Delta & -\varepsilon \end{pmatrix}$

```
In [49]: H.eigenvals()
```

Out[49]: $\left\{-\sqrt{\varepsilon^2+\Delta^2}:1, \sqrt{\varepsilon^2+\Delta^2}:1\right\}$

```
In [50]: H.eigenvects()
```

Out[50]: $\left[\left(-\sqrt{\varepsilon^2+\Delta^2}, 1, \left[\begin{bmatrix} -\frac{\Delta}{\varepsilon+\sqrt{\varepsilon^2+\Delta^2}} \\ 1 \end{bmatrix}\right]\right), \left(\sqrt{\varepsilon^2+\Delta^2}, 1, \left[\begin{bmatrix} -\frac{\Delta}{\varepsilon-\sqrt{\varepsilon^2+\Delta^2}} \\ 1 \end{bmatrix}\right]\right)\right]$

The return value of the `eigenvals` method is a dictionary where each eigenvalue is a key, and the corresponding value is the multiplicity of that particular eigenvalue. Here the eigenvalues are $-\sqrt{\varepsilon^2+\Delta^2}$ and $\sqrt{\varepsilon^2+\Delta^2}$, each with multiplicity one. The return value of `eigenvects` is a bit more involved: a list is returned where each element is a tuple containing an eigenvalue, the multiplicity of the eigenvalue, and a list of eigenvectors. The number of eigenvectors for each eigenvalue equals the multiplicity. For the current example, we can unpack the value returned by eigenvects and verify that the two eigenvectors are orthogonal using, for example,

```
In [51]: (eval1, _, evec1), (eval2, _, evec2) = H.eigenvects()
In [52]: sympy.simplify(evec1[0].T * evec2[0])
Out[52]: [0]
```

Obtaining analytical expressions for eigenvalues and eigenvectors using these methods is often very desirable indeed, but unfortunately it only works for small matrices. For anything larger than a 3×3, the analytical expression typically becomes extremely lengthy and cumbersome to work with even using a computer algebra system such as SymPy. Therefore, for larger systems we must resort to a fully numerical

approach. For this we can use the `la.eigvals` and `la.eig` functions in the SciPy linear algebra package. Matrices that are either Hermitian or real symmetric have real-valued eigenvalues, and for such matrices, it is advantageous to instead use the functions `la.eigvalsh` and `la.eigh`, which guarantees that the eigenvalues returned by the function are stored in a NumPy array with real values. For example, to solve a numerical eigenvalue problem with `la.eig`, we can use

```
In [53]: A = np.array([[1, 3, 5], [3, 5, 3], [5, 3, 9]])
In [54]: evals, evecs = la.eig(A)
In [55]: evals
Out[55]: array([ 13.35310908+0.j,  -1.75902942+0.j,   3.40592034+0.j])
In [56]: evecs
Out[56]: array([[ 0.42663918,  0.90353276, -0.04009445],
                [ 0.43751227, -0.24498225, -0.8651975 ],
                [ 0.79155671, -0.35158534,  0.49982569]])
In [57]: la.eigvalsh(A)
Out[57]: array([ -1.75902942,   3.40592034,  13.35310908])
```

Since the matrix in this example is symmetric, we could use `la.eigh` and `la.eigvalsh`, giving real-valued eigenvalue arrays, as shown in the cell `Out[57]` in the preceding code listing.

Nonlinear Equations

In this section we consider *nonlinear* equations. Systems of linear equations, as considered in the previous sections of this chapter, are of fundamental importance in scientific computing because they are easily solved and can be used as important building blocks in many computational methods and techniques. However, in natural sciences and in engineering disciplines, many, if not most, systems are intrinsically nonlinear.

A linear function $f(x)$ by definition satisfies additivity $f(x+y) = f(x)+f(y)$ and homogeneity $f(\alpha x) = \alpha f(x)$, which can be written together as the superposition principle $f(\alpha x+\beta y) = \alpha f(x)+\beta f(y)$. This gives a precise definition of linearity. A *nonlinear* function, in contrast, is a function that does not satisfy these conditions. Nonlinearity is therefore a much broader concept, and a function can be nonlinear in many different ways. However, in general, an expression that contains a variable with a power greater that one is nonlinear. For example, x^2+x+1 is nonlinear because of the x^2 term.

A nonlinear equation can always be written on the form $f(x) = 0$, where $f(x)$ is a nonlinear function and we seek the value of x (which can be a scalar or a vector) such that $f(x)$ is zero. This x is called the root of the function $f(x)$, and equation solving is therefore often referred to as *root finding*. In contrast to the previous section of this chapter, in this section we need to distinguish between univariate equation solving and multivariate equations, in addition to single equations and system of equations.

Univariate Equations

A univariate function is a function that depends only on a single variable $f(x)$, where x is a scalar, and the corresponding univariate equation is on the form $f(x) = 0$. Typical examples of this type of equation are polynomials, such as $x^2 - x + 1 = 0$, and expressions containing elementary functions, such as $x^3 - 3\sin(x) = 0$ and $\exp(x) - 2 = 0$. Unlike for linear systems, there are no general methods for determining if a nonlinear equation has a solution, or multiple solutions, or if a given solution is unique. This can be understood intuitively from the fact that graphs of nonlinear functions correspond to curves that can intersect $x = 0$ in an arbitrary number of ways.

Because of the vast number of possible situations, it is difficult to develop a completely automatic approach to solving nonlinear equations. Analytically, only equations on special forms can be solved exactly. For example, polynomials of up to 4th order, and in some special cases also higher orders, can be solved analytically, and some equations containing trigonometric and other elementary functions may be solvable analytically. In SymPy we can solve many analytically solvable univariate and nonlinear equations using the `sympy.solve` function. For example, to solve the standard quadratic equation $a + bx + cx^2 = 0$, we define an expression for the equation and pass it to the `sympy.solve` function:

```
In [58]: x, a, b, c = sympy.symbols("x, a, b, c")
In [59]: sympy.solve(a + b*x + c*x**2, x)
Out[59]: [(-b + sqrt(-4*a*c + b**2))/(2*c), -(b + sqrt(-4*a*c + b**2))/(2*c)]
```

The solution is indeed the well-known formula for the solution of this equation. The same method can be used to solve some trigonometric equations:

```
In [60]: sympy.solve(a * sympy.cos(x) - b * sympy.sin(x), x)
Out[60]: [-2*atan((b - sqrt(a**2 + b**2))/a), -2*atan((b +
         sqrt(a**2 + b**2))/a)]
```

However, in general nonlinear equations are typically not solvable analytically. For example, equations that contain both polynomial expressions and elementary functions, such as $\sin x = x$, are often transcendental and do not have an algebraic solution. If we attempt to solve such an equation using SymPy, we obtain an error in the form of an exception:

```
In [61]: sympy.solve(sympy.sin(x)-x, x)
...
NotImplementedError: multiple generators [x, sin(x)]
No algorithms are implemented to solve equation -x + sin(x)
```

In this type of situation, we need to resort to various numerical techniques. As a first step, it is often very useful to graph the function. This can give important clues about the number of solutions to the equation and their approximate locations. This information is often necessary when applying numerical techniques to find good approximations to the roots of the equations. For example, consider the following example (`In [62]`), which plots four examples of nonlinear functions, as shown in Figure 5-5. From these graphs, we can immediately conclude that the plotted functions, from left to right, have two, three, one, and a large number of roots (at least within the interval that is being graphed).

```
In [62]: x = np.linspace(-2, 2, 1000)
    ...: # four examples of nonlinear functions
    ...: f1 = x**2 - x - 1
    ...: f2 = x**3 - 3 * np.sin(x)
    ...: f3 = np.exp(x) - 2
    ...: f4 = 1 - x**2 + np.sin(50 / (1 + x**2))
    ...:
    ...: # plot each function
    ...: fig, axes = plt.subplots(1, 4, figsize=(12, 3), sharey=True)
    ...:
    ...: for n, f in enumerate([f1, f2, f3, f4]):
    ...:     axes[n].plot(x, f, lw=1.5)
    ...:     axes[n].axhline(0, ls=':', color='k')
    ...:     axes[n].set_ylim(-5, 5)
    ...:     axes[n].set_xticks([-2, -1, 0, 1, 2])
    ...:     axes[n].set_xlabel(r'$x$', fontsize=18)
    ...:
```

```
...: axes[0].set_ylabel(r'$f(x)$', fontsize=18)
...:
...: titles = [r'$f(x)=x^2-x-1$', r'$f(x)=x^3-3\sin(x)$',
...:           r'$f(x)=\exp(x)-2$', r'$f(x)=\sin\left(50/(1+x^2)\
               right)+1-x^2$']
...: for n, title in enumerate(titles):
...:     axes[n].set_title(title)
```

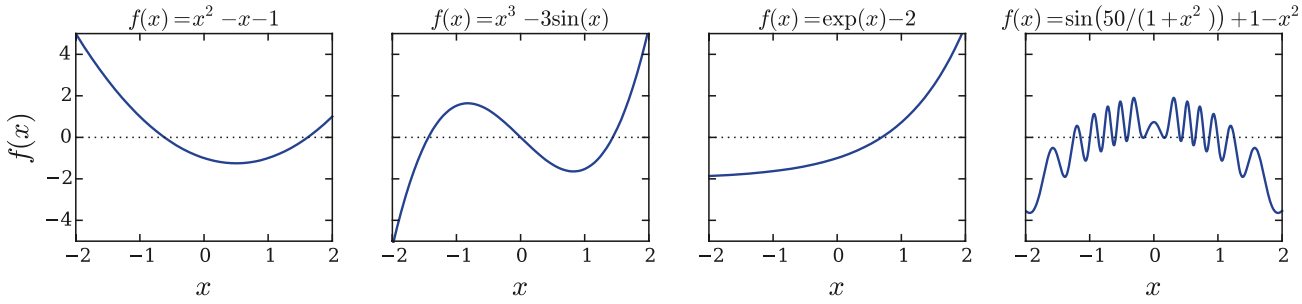

Figure 5-5. *Graphs of four examples of nonlinear functions*

To find the approximate location of a root to an equation, we can apply one of the many techniques for numerical root finding, which typically applies an iterative scheme where the function is evaluated at successive points until the algorithm has narrowed in on the solution, to the desired accuracy. Two standard methods that illustrate the basic idea of how many numerical root-finding methods work are the bisection method and the Newton method.

The bisection method requires a starting interval $[a, b]$ such that $f(a)$ and $f(b)$ have different signs. This guarantees that there is at least one root within this interval. In each iteration, the function is evaluated in the middle point m between a and b, and sign of the function is different at a and m, and then the new interval $[a, b = m]$ is chosen for the next iteration. Otherwise the interval $[a = m, b]$ is chosen for the next iteration. This guarantees that in each iteration, the function has a different sign at the two endpoints of the interval, and in each iteration the interval is halved and therefore converges toward a root of the equation. The following code example demonstrates a simple implementation of the bisection method with a graphical visualization of each step, as shown in Figure 5-6.

```
In [63]: # define a function, desired tolerance and starting interval [a, b]
    ...: f = lambda x: np.exp(x) - 2
    ...: tol = 0.1
    ...: a, b = -2, 2
    ...: x = np.linspace(-2.1, 2.1, 1000)
    ...:
    ...: # graph the function f
    ...: fig, ax = plt.subplots(1, 1, figsize=(12, 4))
    ...:
    ...: ax.plot(x, f(x), lw=1.5)
    ...: ax.axhline(0, ls=':', color='k')
    ...: ax.set_xticks([-2, -1, 0, 1, 2])
    ...: ax.set_xlabel(r'$x$', fontsize=18)
    ...: ax.set_ylabel(r'$f(x)$', fontsize=18)
    ...:
    ...: # find the root using the bisection method and visualize
    ...: # the steps in the method in the graph
    ...: fa, fb = f(a), f(b)
    ...:
    ...: ax.plot(a, fa, 'ko')
    ...: ax.plot(b, fb, 'ko')
    ...: ax.text(a, fa + 0.5, r"$a$", ha='center', fontsize=18)
    ...: ax.text(b, fb + 0.5, r"$b$", ha='center', fontsize=18)
    ...:
    ...: n = 1
    ...: while b - a > tol:
    ...:     m = a + (b - a)/2
    ...:     fm = f(m)
    ...:
    ...:     ax.plot(m, fm, 'ko')
    ...:     ax.text(m, fm - 0.5, r"$m_%d$" % n, ha='center')
    ...:     n += 1
    ...:
    ...:     if np.sign(fa) == np.sign(fm):
```

```
...:         a, fa = m, fm
...:     else:
...:         b, fb = m, fm
...:
...: ax.plot(m, fm, 'r*', markersize=10)
...: ax.annotate("Root approximately at %.3f" % m,
...:             fontsize=14, family="serif",
...:             xy=(a, fm), xycoords='data',
...:             xytext=(-150, +50), textcoords='offset points',
...:             arrowprops=dict(arrowstyle="->",
                 connectionstyle="arc3, rad=-.5"))
...:
...: ax.set_title("Bisection method")
```

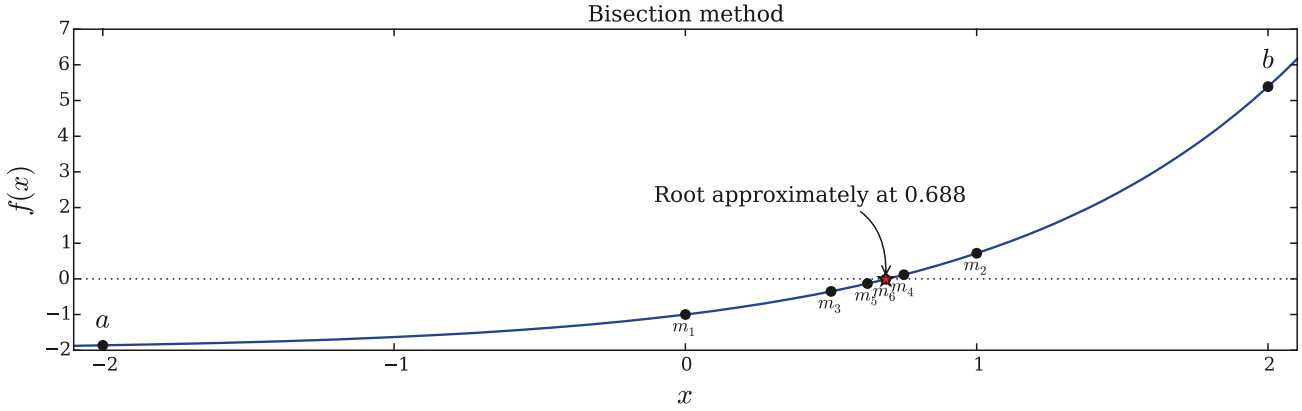

Figure 5-6. *Graphical visualization of how the bisection method works*

Another standard method for root finding is Newton's method, which converges faster than the bisection method discussed in the previous paragraph. While the bisection method only uses the sign of the function at each point, Newton's method uses the actual function values to obtain a more accurate approximation of the nonlinear function. In particular, it approximates the function $f(x)$ with its first-order Taylor expansion $f(x+dx) = f(x)+dx\,f'(x)$, which is a linear function whose root is easily found to be $x - f(x)/f'(x)$. Of course, this does not need to be a root of the function f(x), but in many cases it is a good approximation for getting closer to a root of $f(x)$. By iterating this scheme, $x_{k+1} = x_k - f(x_k)/f'(x_k)$, we may approach the root of the function. A potential problem with this method is that it fails if $f'(x_k)$ is zero at some point x_k. This special case would have to be dealt with in a real implementation of this method. The following

example (`In [64]`) demonstrates how this method can be used to solve for the root of the equation $\exp(x) - 2 = 0$, using SymPy to evaluate the derivative of the function $f(x)$, and Figure 5-7 visualizes the steps in this root-finding process.

```
In [64]: # define a function, desired tolerance and starting point xk
    ...: tol = 0.01
    ...: xk = 2
    ...:
    ...: s_x = sympy.symbols("x")
    ...: s_f = sympy.exp(s_x) - 2
    ...:
    ...: f = lambda x: sympy.lambdify(s_x, s_f, 'numpy')(x)
    ...: fp = lambda x: sympy.lambdify(s_x, sympy.diff(s_f, s_x), 'numpy')(x)
    ...:
    ...: x = np.linspace(-1, 2.1, 1000)
    ...:
    ...: # setup a graph for visualizing the root finding steps
    ...: fig, ax = plt.subplots(1, 1, figsize=(12, 4))
    ...: ax.plot(x, f(x))
    ...: ax.axhline(0, ls=':', color='k')
    ...:
    ...: # iterate Newton's method until convergence to the desired
           tolerance has been reached
    ...: n = 0
    ...: while f(xk) > tol:
    ...:     xk_new = xk - f(xk) / fp(xk)
    ...:
    ...:     ax.plot([xk, xk], [0, f(xk)], color='k', ls=':')
    ...:     ax.plot(xk, f(xk), 'ko')
    ...:     ax.text(xk, -.5, r'$x_%d$' % n, ha='center')
    ...:     ax.plot([xk, xk_new], [f(xk), 0], 'k-')
    ...:
    ...:     xk = xk_new
    ...:     n += 1
    ...:
    ...: ax.plot(xk, f(xk), 'r*', markersize=15)
```

```
...: ax.annotate("Root approximately at %.3f" % xk,
...:             fontsize=14, family="serif",
...:             xy=(xk, f(xk)), xycoords='data',
...:             xytext=(-150, +50), textcoords='offset points',
...:             arrowprops=dict(arrowstyle="->",
                 connectionstyle="arc3, rad=-.5"))
...:
...: ax.set_title("Newtown's method")
...: ax.set_xticks([-1, 0, 1, 2])
```

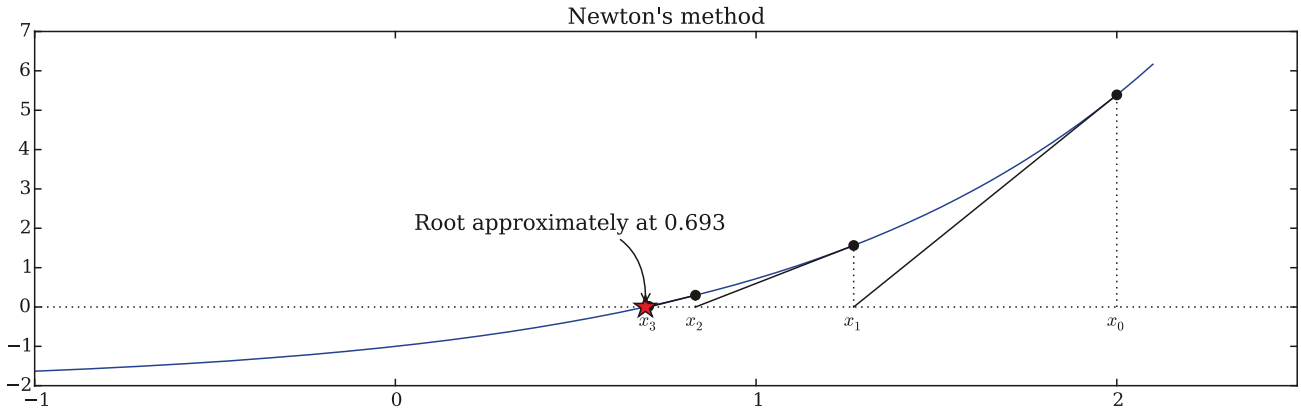

Figure 5-7. *Visualization of the root-finding steps in Newton's method for the equation exp(x) − 2 = 0*

A potential issue with Newton's method is that it requires both the function values and the values of the derivative of the function in each iteration. In the previous example, we used SymPy to symbolically compute the derivatives. In an all-numerical implementation, this is of course not possible, and a numerical approximation of the derivative would be necessary, which would in turn require further function evaluations. A variant of Newton's method that bypasses the requirement to evaluate function derivatives is the secant method, which uses two previous function evaluations to obtain a linear approximation of the function, which can be used to compute a new estimate of the root. The iteration formula for the secant method is $x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$. This is only one example of the many variants and possible refinements on the basic idea of Newton's method. State-of-the-art implementations of numerical root-finding functions typically use the basic idea of either the bisection method of Newton's method or a combination of both but additionally use various refinement strategies, such as higher-order interpolations of the function to achieve faster convergence.

The SciPy optimize module provides multiple functions for numerical root finding. The `optimize.bisect` and `optimize.newton` functions implement variants of bisection and Newton methods. The `optimize.bisect` takes three arguments: first a Python function (e.g., a lambda function) that represents the mathematical function for the equation for which a root is to be calculated and the second and third arguments are the lower and upper values of the interval for which to perform the bisection method. Note that the sign of the function has to be different at the points a and b for the bisection method to work, as discussed earlier. Using the `optimize.bisect` function, we can calculate the root of the equation $\exp(x) - 2 = 0$ that we used in the previous examples, using

```
In [65]: optimize.bisect(lambda x: np.exp(x) - 2, -2, 2)
Out[65]: 0.6931471805592082
```

As long as $f(a)$ and $f(b)$ indeed have different signs, this is guaranteed to give a root within the interval $[a, b]$. In contrast, the function `optimize.newton` for Newton's method takes a function as the first argument and an initial guess for the root of the function as the second argument. Optionally, it also takes an argument for specifying the derivative of the function, using the `fprime` keyword argument. If `fprime` is given, Newton's method is used; otherwise the secant method is used instead. To find the root of the equation $\exp x - 2 = 0$, with and without specifying its derivative, we can use

```
In [66]: x_root_guess = 2
In [67]: f = lambda x: np.exp(x) - 2
In [68]: fprime = lambda x: np.exp(x)
In [69]: optimize.newton(f, x_root_guess)
Out[69]: 0.69314718056
In [70]: optimize.newton(f, x_root_guess, fprime=fprime)
Out[70]: 0.69314718056
```

Note that with this method, we have less control over which root is being computed, if the function has multiple roots. For instance, there is no guarantee that the root the function returns is the closest one to the initial guess; we cannot know in advance if the root is larger or smaller than the initial guess.

The SciPy `optimize` module provides additional functions for root finding. In particular, the `optimize.brentq` and `optimize.brenth` functions, which are variants of the bisection method, also work on an interval where the function changes sign. The

`optimize.brentq` function is generally considered the preferred all-around root-finding function in SciPy. To find a root of the same equation that we considered previously, using `optimize.brentq` and `optimize.brenth` functions, we can use

```
In [71]: optimize.brentq(lambda x: np.exp(x) - 2, -2, 2)
Out[71]: 0.6931471805599453
In [72]: optimize.brenth(lambda x: np.exp(x) - 2, -2, 2)
Out[72]: 0.6931471805599381
```

Note that these two functions take a Python function for the equation as the first argument and the lower and upper values of the sign-changing interval as the second and third arguments.

Systems of Nonlinear Equations

In contrast to a linear system of equations, we cannot in general write a system of nonlinear equations as a matrix-vector multiplication. Instead we represent a system of multivariate nonlinear equations as a vector-valued function, for example, $f: \mathbb{R}^N \rightarrow \mathbb{R}^N$, that takes an N-dimensional vector and maps it to another N-dimensional vector. Multivariate systems of equations are much more complicated to solve than univariate equations, partly because there are so many more possible behaviors. As a consequence, there is no method that strictly guarantees convergence to a solution, such as the bisection method for a univariate nonlinear equation, and the methods that do exist are much more computationally demanding than the univariate case, especially as the number of variables increases.

Not all methods that we previously discussed for univariate equation solving can be generalized to the multivariate case. For example, the bisection method cannot be directly generalized to a multivariate equation system. On the other hand, Newton's method can be used for multivariate problems, and in this case its iteration formula is $x_{k+1} = x_k - J_f(x_k)^{-1} f(x_k)$, where $J_f(x_k)$ is the Jacobian matrix of the function $f(x)$, with elements $[J_f(x_k)]_{ij} = \partial f_i(x_k)/\partial x_j$. Instead of inverting the Jacobian matrix, it is sufficient to solve the linear equation system $J_f(x_k)\delta x_k = -f(x_k)$ and update x_k using $x_{k+1} = x_k + \delta x_k$. Like the secant variants of the Newton method for univariate equation systems, there are also variants of the multivariate method that avoid computing the Jacobian by estimating it from previous function evaluations. Broyden's method is a popular example of this type of secant updating method for multivariate equation systems. In the SciPy `optimize`

module, `broyden1` and `broyden2` provide two implementations of Broyden's method using different approximations of the Jacobian, and the function `optimize.fsolve` provides an implementation of a Newton-like method, where optionally the Jacobian can be specified, if available. The functions all have a similar function signature: The first argument is a Python function that represents the equation to be solved, and it should take a NumPy array as the first argument and return an array of the same shape. The second argument is an initial guess for the solution, as a NumPy array. The `optimize.fsolve` function also takes an optional keyword argument `fprime`, which can be used to provide a function that returns the Jacobian of the function $f(x)$. In addition, all these functions take numerous optional keyword arguments for tuning their behavior (see the docstrings for details).

For example, consider the following system of two multivariate and nonlinear equations

$$\begin{cases} y - x^3 - 2x^2 + 1 = 0 \\ y + x^2 - 1 = 0 \end{cases},$$

which can be represented by the vector-valued function $f([x_1, x_2]) = [x_2 - {x_1}^3 - 2{x_1}^2 + 1, x_2 + {x_1}^2 - 1]$. To solve this equation system using SciPy, we need to define a Python function for $f([x_1, x_2])$ and call, for example, the `optimize.fsolve` using the function and an initial guess for the solution vector:

```
In [73]: def f(x):
    ...:     return [x[1] - x[0]**3 - 2 * x[0]**2 + 1, x[1] + x[0]**2 - 1]
In [74]: optimize.fsolve(f, [1, 1])
Out[74]: array([ 0.73205081,  0.46410162])
```

The `optimize.broyden1` and `optimize.broyden2` can be used in a similar manner. To specify a Jacobian for `optimize.fsolve` to use, we need to define a function that evaluates the Jacobian for a given input vector. This requires that we first derive the Jacobian by hand or, for example, using SymPy

```
In [75]: x, y = sympy.symbols("x, y")
In [76]: f_mat = sympy.Matrix([y - x**3 -2*x**2 + 1, y + x**2 - 1])
In [77]: f_mat.jacobian(sympy.Matrix([x, y]))
```

Out[77]:
$$\begin{pmatrix} -3x^2 - 4x & 1 \\ 2x & 1 \end{pmatrix}$$

which we can then easily implement as a Python function that can be passed to the `optimize.fsolve` function:

```
In [78]: def f_jacobian(x):
    ...:     return [[-3*x[0]**2-4*x[0], 1], [2*x[0], 1]]
In [79]: optimize.fsolve(f, [1, 1], fprime=f_jacobian)
Out[79]: array([ 0.73205081,  0.46410162])
```

As with the Newton's method for a univariate nonlinear equation system, the initial guess for the solution is important, and different initial guesses may result in different solutions that are found for the equations. There is no guarantee that any particular solution is found, although the proximity of the initial guess to the true solution often is correlated with convergence to that particular solution. When possible, it is often a good approach to graph the equations that are being solved, to give a visual indication of the number of solutions and their locations. For example, the following code demonstrates how three different solutions can be found to the equation systems we are considering here, by using different initial guesses with the `optimize.fsolve` function. The result is shown in Figure 5-8.

```
In [80]: def f(x):
    ...:     return [x[1] - x[0]**3 - 2 * x[0]**2 + 1,
    ...:             x[1] + x[0]**2 - 1]
    ...:
    ...: x = np.linspace(-3, 2, 5000)
    ...: y1 = x**3 + 2 * x**2 -1
    ...: y2 = -x**2 + 1
    ...:
    ...: fig, ax = plt.subplots(figsize=(8, 4))
    ...:
    ...: ax.plot(x, y1, 'b', lw=1.5, label=r'$y = x^3 + 2x^2 - 1$')
    ...: ax.plot(x, y2, 'g', lw=1.5, label=r'$y = -x^2 + 1$')
    ...:
    ...: x_guesses = [[-2, 2], [1, -1], [-2, -5]]
    ...: for x_guess in x_guesses:
    ...:     sol = optimize.fsolve(f, x_guess)
    ...:     ax.plot(sol[0], sol[1], 'r*', markersize=15)
    ...:
```

```
...:     ax.plot(x_guess[0], x_guess[1], 'ko')
...:     ax.annotate("", xy=(sol[0], sol[1]), xytext=(x_guess[0],
                    x_guess[1]),
...:                 arrowprops=dict(arrowstyle="->", linewidth=2.5))
...:
...: ax.legend(loc=0)
...: ax.set_xlabel(r'$x$', fontsize=18)
```

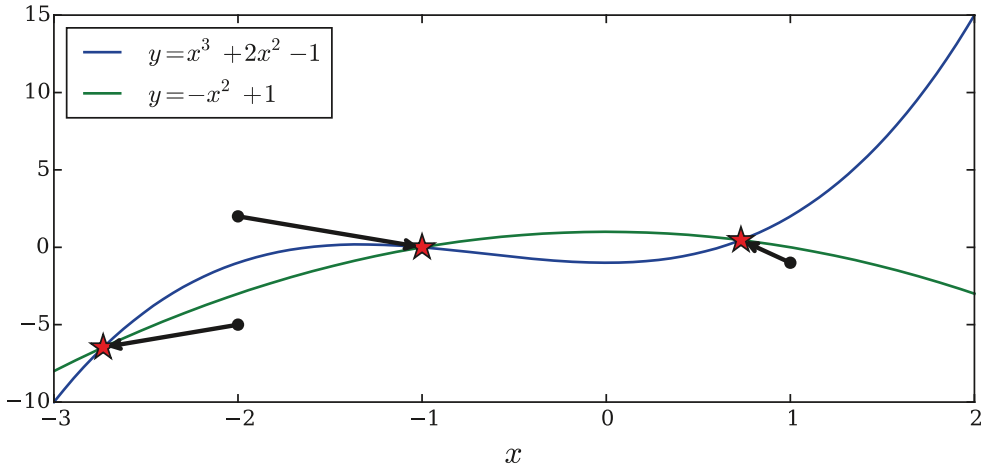

Figure 5-8. *Graph of a system of two nonlinear equations. The solutions are indicated with red stars and the initial guess with a black dot and an arrow to the solution each initial guess eventually converged to.*

By systematically solving the equation systems with different initial guesses, we can build a visualization of how different initial guesses converge to different solutions. This is done in the following code example, and the result is shown in Figure 5-9. This example demonstrates that even for this relatively simple example, the regions of initial guesses that converge to different solutions are highly nontrivial, and there are also missing dots which corresponds to initial guesses for which the algorithm fails to converge to any solution. Nonlinear equation solving is a complex task, and visualizations of different types can often be a valuable tool when building an understanding of the characteristics of a particular problem.

```
In [81]: fig, ax = plt.subplots(figsize=(8, 4))
    ...:
    ...: ax.plot(x, y1, 'k', lw=1.5)
    ...: ax.plot(x, y2, 'k', lw=1.5)
```

```
...:
...: sol1 = optimize.fsolve(f, [-2,  2])
...: sol2 = optimize.fsolve(f, [ 1, -1])
...: sol3 = optimize.fsolve(f, [-2, -5])
...: sols = [sol1, sol2, sol3]
...: for idx, s in enumerate(sols):
...:     ax.plot(s[0], s[1], colors[idx]+'*', markersize=15)
...:
...: colors = ['r', 'b', 'g']
...: for m in np.linspace(-4, 3, 80):
...:     for n in np.linspace(-15, 15, 40):
...:         x_guess = [m, n]
...:         sol = optimize.fsolve(f, x_guess)
...:         idx = (abs(sols - sol)**2).sum(axis=1).argmin()
...:         ax.plot(x_guess[0], x_guess[1], colors[idx]+'.')
...:
...: ax.set_xlabel(r'$x$', fontsize=18)
```

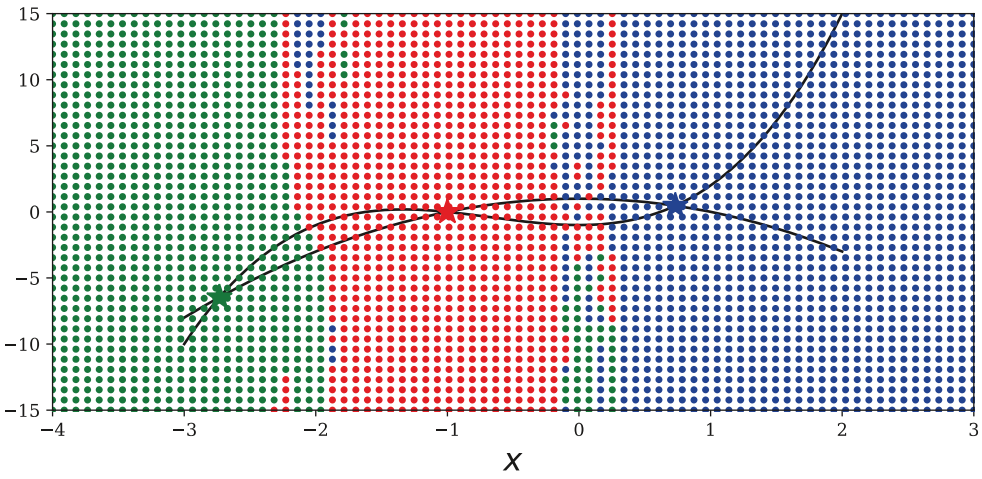

Figure 5-9. *Visualization of the convergence of different initial guesses to different solutions. Each dot represents an initial guess, and its color encodes which solution it eventually converges to. The solutions are marked with correspondingly color-coded stars.*

Summary

In this chapter we have explored methods for solving algebraic equations using the SymPy and SciPy libraries. Equation solving is one of the most elementary mathematical tools for computational sciences, and it is both an important component in many algorithms and methods and has direct applications in many problem-solving situations. In some cases, analytical algebraic solutions exist, especially for equations that are polynomials or contain certain combinations of elementary functions, and such equations can often be handled symbolically with SymPy. For equations with no algebraic solution, and for larger systems of equations, numerical methods are usually the only feasible approach. Linear equation systems can always be systematically solved, and for this reason there is an abundance of important applications for linear equation systems, be it for originally linear systems or as approximations to originally nonlinear systems. Nonlinear equation solving requires a different set of methods, and it is in general much more complex and computationally demanding compared to linear equation systems. In fact, solving linear equation systems is an important step in the iterative methods employed in many of the methods that exist to solve nonlinear equation systems. For numerical equation solving, we can use the linear algebra and optimization modules in SciPy, which provide efficient and well-tested methods for numerical root finding and equation solving of both linear and nonlinear systems.

Further Reading

Equation solving is a basic numerical technique whose methods are covered in most introductory numerical analysis texts. A good example of a book that covers these topics is Heath (2001), and W.H. Press (2007) gives a practical introduction with implementation details.

References

Heath, M. (2001). *Scientific Computing*. Boston: McGraw-Hill.

W.H. Press, S. T. (2007). *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge: Cambridge University Press.

CHAPTER 6

Optimization

In this chapter, we will build on Chapter 5 about equation solving and explore the related topic of solving optimization problems. In general, optimization is the process of finding and selecting the optimal element from a set of feasible candidates. In mathematical optimization, this problem is usually formulated as determining the extreme value of a function on a given domain. An extreme value, or an optimal value, can refer to either the minimum or maximum of the function, depending on the application and the specific problem. In this chapter we are concerned with the optimization of real-valued functions of one or several variables, which optionally can be subject to a set of constraints that restricts the domain of the function.

The applications of mathematical optimization are many and varied, and so are the methods and algorithms that must be employed to solve optimization problems. Since optimization is a universally important mathematical tool, it has been developed and adapted for use in many fields of science and engineering, and the terminology used to describe optimization problems varies between fields. For example, the mathematical function that is optimized may be called a cost function, loss function, energy function, or objective function, to mention a few. Here we use the generic term objective function.

Optimization is closely related to equation solving because at an optimal value of a function, its derivative, or gradient in the multivariate case, is zero. The converse, however, is not necessarily true, but a method to solve optimization problems is to solve for the zeros of the derivative or the gradient and test the resulting candidates for optimality. This approach is not always feasible though, and often it is required to take other numerical approaches, many of which are closely related to the numerical methods for root finding that was covered in Chapter 5.

In this chapter we discuss using SciPy's optimization module `optimize` for nonlinear optimization problems, and we will briefly explore using the convex optimization library `cvxopt` for linear optimization problems with linear constraints. This library also has powerful solvers for quadratic programming problems.

R. Johansson, *Numerical Python*, https://doi.org/10.1007/978-1-4842-4246-9_6

cvxopt The convex optimization library cvxopt provides solvers for linear and quadratic optimization problems. At the time of writing, the latest version is 1.1.9. For more information, see the project's web site http://cvxopt.org. Here we use this library for constrained linear optimization.

Importing Modules

Like in the previous chapter, here we use the `optimize` module from the SciPy library. Here we assume that this module is imported in the following manner:

```
In [1]: from scipy import optimize
```

In the later part of this chapter, we also look at linear programming using the `cvxopt` library, which we assume to be imported in its entirety without any alias:

```
In [2]: import cvxopt
```

For basic numerics, symbolics, and plotting, here we also use the NumPy, SymPy, and Matplotlib libraries, which are imported and initialized using the conventions introduced in earlier chapters:

```
In [3]: import matplotlib.pyplot as plt
In [4]: import numpy as np
In [5]: import sympy
In [6]: sympy.init_printing()
```

Classification of Optimization Problems

Here we restrict our attention to mathematical optimization of real-valued functions, with one or more dependent variables. Many mathematical optimization problems can be formulated in this way, but a notable exception is optimization of functions over discrete variables, for example, integers, which is beyond the scope of this book.

A general optimization problem of the type considered here can be formulated as a minimization problem, $\min_x f(x)$, subject to sets of m equality constraints $g(x) = 0$ and p inequality constraints $h(x) \leq 0$. Here $f(x)$ is a real-valued function of x, which can be a

scalar or a vector $x = (x_0, x_1, \ldots, x_n)^T$, while $g(x)$ and $h(x)$ can be vector-valued functions: $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, $g : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ and $h : \mathbb{R}^n \longrightarrow \mathbb{R}^p$. Note that maximizing $f(x)$ is equivalent to minimizing $-f(x)$, so without loss of generality, it is sufficient to consider only minimization problems.

Depending on the properties of the objective function $f(x)$ and the equality and inequality constraints $g(x)$ and $h(x)$, this formulation includes a rich variety of problems. A general mathematical optimization on this form is difficult to solve, and there are no efficient methods for solving completely generic optimization problems. However, there are efficient methods for many important special cases, and in optimization it is therefore important to know as much as possible about the objective functions and the constraints in order to be able to solve a problem.

Optimization problems are classified depending on the properties of the functions $f(x)$, $g(x)$, and $h(x)$. First and foremost, the problem is *univariate* or *one dimensional* if x is a scalar, $x \in \mathbb{R}$, and *multivariate* or *multidimensional* if x is a vector, $x \in \mathbb{R}^n$. For high-dimensional objective functions, with larger n, the optimization problem is harder and more computationally demanding to solve. If the objective function and the constraints all are linear, the problem is a linear optimization problem, or *linear programming* problem.[1] If either the objective function or the constraints are nonlinear, it is a nonlinear optimization problem, or a *nonlinear programming* problem. With respect to constraints, important subclasses of optimization are unconstrained problems, and those with linear and nonlinear constraints. Finally, handling equality and inequality constraints requires different approaches.

As usual, nonlinear problems are much harder to solve than linear problems, because they have a wider variety of possible behaviors. A general nonlinear problem can have both local and global minima, which turns out to make it very difficult to find the global minima: iterative solvers may often converge to local minima rather than the global minima or may even fail to converge altogether if there are both local and global minima. However, an important subclass of nonlinear problems that can be solved efficiently is *convex problems*, which is directly related to the absence of strictly local minima and the existence of a unique global minimum. By definition, a function is convex on an interval $[a, b]$ if the values of the function on this interval lie below the line through the endpoints $(a, f(a))$ and $(b, f(b))$. This condition, which can be readily generalized to the multivariate case, implies a number of important properties, such as

[1] For historical reasons optimization problems are often referred to as *programming* problems, which are not related to computer programming.

the existence of a unique minimum on the interval. Because of strong properties like this one, convex problems can be solved efficiently even though they are nonlinear. The concepts of local and global minima, and convex and nonconvex functions, are illustrated in Figure 6-1.

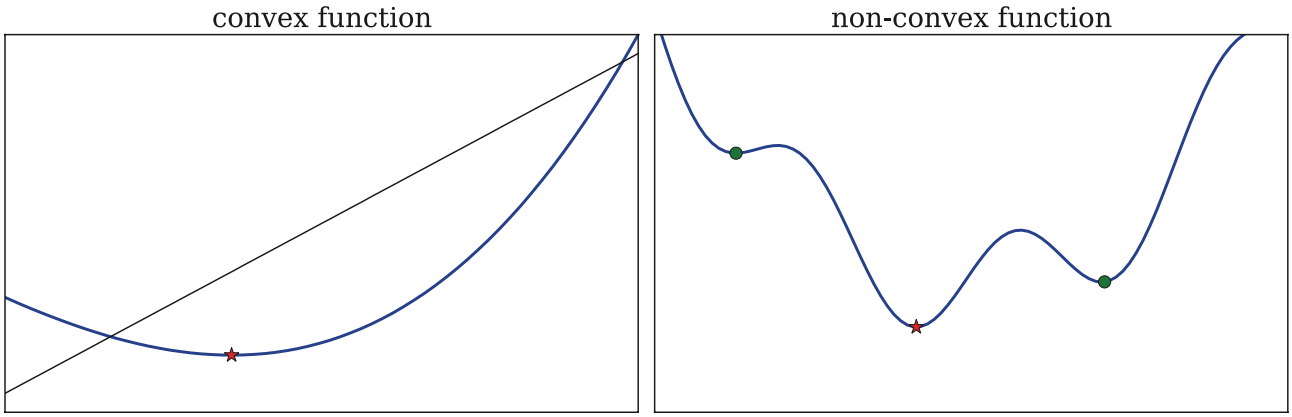

Figure 6-1. *Illustration of a convex function (left) and a nonconvex function (right) with a global minimum and two local minima*

Whether the objective function $f(x)$ and the constraints $g(x)$ and $h(x)$ are continuous and smooth are properties that have very important implications for the methods and techniques that can be used to solve an optimization problem. Discontinuities in these functions, or their derivatives or gradients, cause difficulties for many of the available methods of solving optimization problems, and in the following, we assume that these functions are indeed continuous and smooth. On a related note, if the function itself is not known exactly, but contains noise due to measurements or for other reasons, many of the methods discussed in the following may not be suitable.

Optimization of continuous and smooth functions are closely related to nonlinear equation solving, because extremal values of a function $f(x)$ correspond to points where its derivative, or gradient, is zero. Finding candidates for the optimal value of $f(x)$ is therefore equivalent to solving the (in general nonlinear) equation system $\nabla f(x) = 0$. However, a solution to $\nabla f(x) = 0$, which is known as a stationary point, does not necessarily correspond to a minimum of $f(x)$; it can also be maximum or a saddle point; see Figure 6-2. Candidates obtained by solving $\nabla f(x) = 0$ should therefore be tested for optimality. For unconstrained objective functions, the higher-order derivatives, or Hessian matrix

$$\left\{H_f(x)\right\}_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j},$$

for the multivariate case, can be used to determine if a stationary point is a local minimum or not. In particular if the second-order derivative is positive, or the Hessian positive definite, when evaluated at stationary point x^*, then x^* is a local minimum. Negative second-order derivative, or negative definite Hessian, corresponds to a local maximum, and a zero second-order derivative, or an indefinite Hessian, corresponds to saddle point.

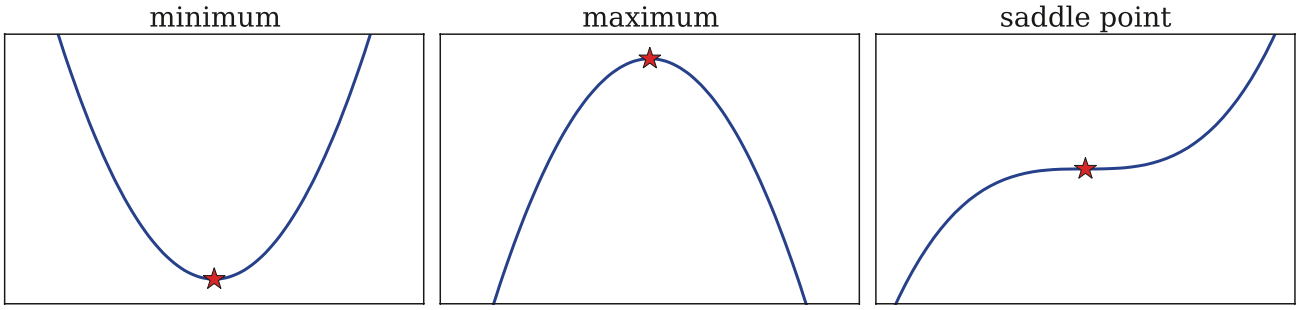

Figure 6-2. *Illustration of different stationary points of a one-dimensional function*

Algebraically solving the equation system $\nabla f(x) = 0$ and testing the candidate solutions for optimality is therefore one possible strategy for solving an optimization problem. However, it is not always a feasible method. In particular, we may not have an analytical expression for $f(x)$ from which we can compute the derivatives, and the resulting nonlinear equation system may not be easy to solve, especially not to find all of its roots. For such cases, there are alternative numerical optimization approaches, some of which have analogs among the root-finding methods discussed in Chapter 5. In the remaining part of this chapter, we explore the various classes of optimization problems and how such problems can be solved in practice using available optimization libraries for Python.

Univariate Optimization

Optimization of a function that only depends on a single variable is relatively easy. In addition to the analytical approach of seeking the roots of the derivative of the function, we can employ techniques that are similar to the root-finding methods for univariate functions, namely, bracketing methods and Newton's method. Like the bisection method for univariate root finding, it is possible to use bracketing and iteratively refine an interval using function evaluations alone. Refining an interval $[a, b]$ that contains a minimum can be achieved by evaluating the function at two interior points x_1 and

x_2, $x_1 < x_2$, and selecting $[x_1, b]$ as new interval if $f(x_1) > f(x_2)$, and $[a, x_2]$ otherwise. This idea is used in the *golden section search* method, which additionally uses the trick of choosing x_1 and x_2 such that their relative positions in the $[a, b]$ interval satisfy the golden ratio. This has the advantage of allowing to reuse one function evaluation from the previous iteration and thus only requiring one new function evaluation in each iteration but still reducing the interval with a constant factor in each iteration. For functions with a unique minimum on the given interval, this approach is guaranteed to converge to an optimal point, but this is unfortunately not guaranteed for more complicated functions. It is therefore important to carefully select the initial interval, ideally relatively close to an optimal point. In the SciPy `optimize` module, the function `golden` implements the golden search method.

As the bisection method for root finding, the golden search method is a (relatively) safe but a slowly converging method. Methods with better convergence can be constructed if the values of the function evaluations are used, rather than only comparing the values to each other (which is similar to using only the sign of the functions, as in the bisection method). The function values can be used to fit a polynomial, for example, a quadratic polynomial, which can be interpolated to find a new approximation for the minimum, giving a candidate for a new function evaluation, after which the process can be iterated. This approach can converge faster but is riskier than bracketing and may not converge at all or may converge to local minima outside the given bracket interval.

Newton's method for root finding is an example of a quadratic approximation method that can be applied to find a function minimum, by applying the method to the derivative rather than the function itself. This yields the iteration formula $x_{k+1} = x_k - f'(x_k)/f''(x_k)$, which can converge quickly if started close to an optimal point but may not converge at all if started too far from the optimal value. This formula also requires evaluating both the derivative and the second-order derivative in each iteration. If analytical expressions for these derivatives are available, this can be a good method. If only function evaluations are available, the derivatives may be approximated using an analog of the secant method for root finding.

A combination of the two previous methods is typically used in practical implementations of univariate optimization routines, giving both stability and fast convergence. In SciPy's `optimize` module, the `brent` function is such a hybrid method, and it is generally the preferred method for optimization of univariate functions with SciPy. This method is a variant of the golden section search method that uses inverse parabolic interpolation to obtain faster convergence.

Instead of calling the `optimize.golden` and `optimize.brent` functions directly, it is convenient to use the unified interface function `optimize.minimize_scalar`, which dispatches to the `optimize.golden` and `optimize.brent` functions depending on the value of the `method` keyword argument, where the currently allowed options are `'Golden'`, `'Brent'`, or `'Bounded'`. The last option dispatches to `optimize.fminbound`, which performs optimization on a bounded interval, which corresponds to an optimization problem with inequality constraints that limit the domain of objective function $f(x)$. Note that the `optimize.golden` and `optimize.brent` functions may converge to a local minimum outside the initial bracket interval, but `optimize.fminbound` would in such circumstances return the value at the end of the allowed range.

As an example for illustrating these techniques, consider the following classic optimization problem: Minimize the area of a cylinder with unit volume. Here, suitable variables are the radius r and height h of the cylinder, and the objective function is $f([r, h]) = 2\pi r^2 + 2\pi rh$, subject to the equality constraint $g([r, h]) = \pi r^2 h - 1 = 0$. As this problem is formulated here, it is a two-dimensional optimization problem with an equality constraint. However, we can algebraically solve the constraint equation for one of the dependent variables, for example, $h = 1/\pi r^2$, and substitute this into the objective function to obtain an unconstrained one-dimensional optimization problem: $f(r) = 2\pi r^2 + 2/r$. To begin with, we can solve this problem symbolically using SymPy, using the method of equating the derivative of $f(r)$ to zero:

```
In [7]: r, h = sympy.symbols("r, h")
In [8]: Area = 2 * sympy.pi * r**2 + 2 * sympy.pi * r * h
In [9]: Volume = sympy.pi * r**2 * h
In [10]: h_r = sympy.solve(Volume - 1)[0]
In [11]: Area_r = Area.subs(h_r)
In [12]: rsol = sympy.solve(Area_r.diff(r))[0]
In [13]: rsol
Out[13]: 2^(2/3)/(2*pi^(1/3))
In [14]: _.evalf()
Out[14]: 0.541926070139289
```

Now verify that the second derivative is positive and that `rsol` corresponds to a minimum:

```
In [15]: Area_r.diff(r, 2).subs(r, rsol)
Out[15]: 12π
In [16]: Area_r.subs(r, rsol)
Out[16]: 3∛(2π)

In [17]: _.evalf()
Out[17]: 5.53581044593209
```

For simple problems this approach is often feasible, but for more realistic problems, we typically need to resort to numerical techniques. To solve this problem using SciPy's numerical optimization functions, we first define a Python function `f` that implements the objective function. To solve the optimization problem, we then pass this function to, for example, `optimize.brent`. Optionally we can use the `brack` keyword argument to specify a starting interval for the algorithm:

```
In [18]: def f(r):
    ...:     return 2 * np.pi * r**2 + 2 / r
In [19]: r_min = optimize.brent(f, brack=(0.1, 4))
In [20]: r_min
Out[20]: 0.541926077256
In [21]: f(r_min)
Out[21]: 5.53581044593
```

Instead of calling `optimize.brent` directly, we could use the generic interface for scalar minimization problems `optimize.minimize_scalar`. Note that to specify a starting interval in this case, we must use the `bracket` keyword argument:

```
In [22]: optimize.minimize_scalar(f, bracket=(0.1, 4))
Out[22]:  nit: 13
          fun: 5.5358104459320856
            x: 0.54192606489766715
         nfev: 14
```

All these methods give that the radius that minimizes the area of the cylinder is approximately 0.54 (the exact result from the symbolic calculation is $2^{2/3}/2\sqrt[3]{\pi}$) and a minimum area of approximately 5.54 (the exact result is $3\sqrt[3]{2\pi}$). The objective function

that we minimized in this example is plotted in Figure 6-3, where the minimum is marked with a red star. When possible, it is a good idea to visualize the objective function before attempting a numerical optimization, because it can help in identifying a suitable initial interval or a starting point for the numerical optimization routine.

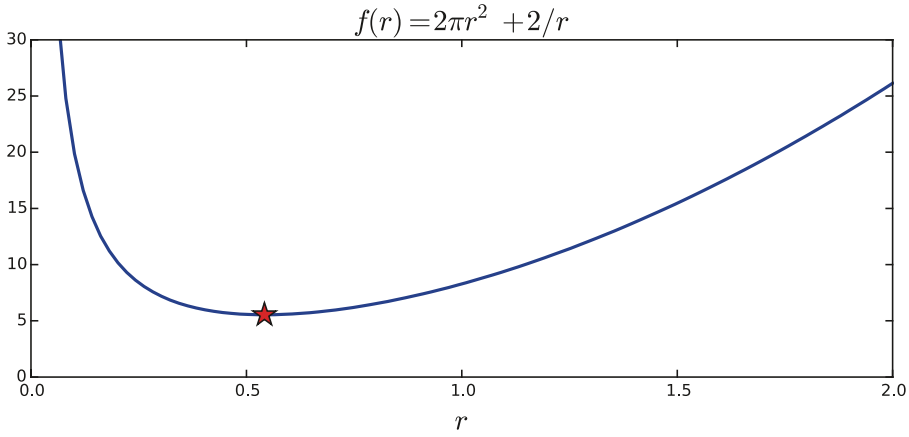

Figure 6-3. *The surface area of a cylinder with unit volume as a function of the radius r*

Unconstrained Multivariate Optimization

Multivariate optimization is significantly harder than the univariate optimization discussed in the previous section. In particular, the analytical approach of solving the nonlinear equations for roots of the gradient is rarely feasible in the multivariate case, and the bracketing scheme used in the golden search method is also not directly applicable. Instead we must resort to techniques that start at some point in the coordinate space and use different strategies to move toward a better approximation of the minimum point. The most basic approach of this type is to consider the gradient $\nabla f(x)$ of the objective function $f(x)$ at a given point x. In general, the negative of the gradient, $-\nabla f(x)$, always points in the direction in which the function $f(x)$ decreases the most. As minimization strategy, it is therefore sensible to move along this direction for some distance α_k and then iterate this scheme at the new point. This method is known as the *steepest descent method*, and it gives the iteration formula $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$, where α_k is a free parameter known as the *line search parameter* that describes how far along the given direction to move in each iteration. An appropriate α_k can, for example, be selected by solving the one-dimensional optimization problem $\min_{ak} f\left(x_k - \alpha_k \nabla f\left(x_k\right)\right)$. This method is guaranteed to make progress and eventually converge to a minimum

of the function, but the convergence can be quite slow because this method tends to overshoot along the direction of the gradient, giving a zigzag approach to the minimum. Nonetheless, the steepest descent method is the conceptual basis for many multivariate optimization algorithms, and with suitable modifications, the convergence can be speed up.

Newton's method for multivariate optimization is a modification of the steepest descent method that can improve convergence. As in the univariate case, Newton's method can be viewed as a local quadratic approximation of the function, which when minimized gives an iteration scheme. In the multivariate case, the iteration formula is $x_{k+1} = x_k - H_f^{-1}(x_k)\nabla f(x_k)$, where compared to the steepest descent method, the gradient has been replaced with the gradient multiplied from the left with the inverse of Hessian matrix for the function.[2] In general this alters both the direction and the length of the step, so this method is not strictly a steepest descent method and may not converge if started too far from a minimum. However, when close to a minimum, it converges quickly. As usual there is a trade-off between convergence rate and stability. As it is formulated here, Newton's method requires both the gradient and the Hessian of the function.

In SciPy, Newton's method is implemented in the function `optimize.fmin_ncg`. This function takes the following arguments: a Python function for the objective function, a starting point, a Python function for evaluating the gradient, and (optionally) a Python function for evaluating the Hessian. To see how this method can be used to solve an optimization problem, we consider the following problem: $\min_x f(x)$ where the objective function is $f(x) = (x_1 - 1)^4 + 5(x_2 - 1)^2 - 2x_1x_2$. To apply Newton's method, we need to calculate the gradient and the Hessian. For this particular case, this can easily be done by hand. However, for the sake of generality, in the following we use SymPy to compute symbolic expressions for the gradient and the Hessian. To this end, we begin by defining symbols and a symbolic expression for the objective function, and then use the `sympy.diff` function for each variable to obtain the gradient and Hessian in symbolic form:

```
In [23]: x1, x2 = sympy.symbols("x_1, x_2")
In [24]: f_sym = (x1-1)**4 + 5 * (x2-1)**2 - 2*x1*x2
In [25]: fprime_sym = [f_sym.diff(x_) for x_ in (x1, x2)]
```

[2] In practice, the inverse of the Hessian does not need to be computed, and instead we can solve the linear equation system $H_f(x_k)y_k = -\nabla f(x_k)$ and use the integration formula $x_{k+1} = x_k + y_k$.

```
In [26]: # Gradient
    ...: sympy.Matrix(fprime_sym)
```

Out[26]: $\begin{bmatrix} -2x_2 + 4(x_1 - 1)^3 \\ -2x_1 + 10x_2 - 10 \end{bmatrix}$

```
In [27]: fhess_sym = [[f_sym.diff(x1_, x2_) for x1_ in (x1, x2)] for x2_ in
         (x1, x2)]
In [28]: # Hessian
    ...: sympy.Matrix(fhess_sym)
```

Out[28]: $\begin{bmatrix} 12(x_1 - 1)^2 & -2 \\ -2 & 10 \end{bmatrix}$

Now that we have a symbolic expression for the gradient and the Hessian, we can create vectorized functions for these expressions using `sympy.lambdify`.

```
In [29]: f_lmbda = sympy.lambdify((x1, x2), f_sym, 'numpy')
In [30]: fprime_lmbda = sympy.lambdify((x1, x2), fprime_sym, 'numpy')
In [31]: fhess_lmbda = sympy.lambdify((x1, x2), fhess_sym, 'numpy')
```

However, the functions produced by `sympy.lambdify` take one argument for each variable in the corresponding expression, and the SciPy optimization functions expect a vectorized function where all coordinates are packed into one array. To obtain functions that are compatible with the SciPy optimization routines, we wrap each of the functions generated by `sympy.lambdify` with a Python function that rearranges the arguments:

```
In [32]: def func_XY_to_X_Y(f):
    ...:     """
    ...:     Wrapper for f(X) -> f(X[0], X[1])
    ...:     """
    ...:     return lambda X: np.array(f(X[0], X[1]))
In [33]: f = func_XY_to_X_Y(f_lmbda)
In [34]: fprime = func_XY_to_X_Y(fprime_lmbda)
In [35]: fhess = func_XY_to_X_Y(fhess_lmbda)
```

Now the functions `f`, `fprime`, and `fhess` are vectorized Python functions on the form that, for example, `optimize.fmin_ncg` expects, and we can proceed with a numerical optimization of the problem at hand by calling this function. In addition to the functions that we have prepared from SymPy expressions, we also need to give a starting point for the Newton method. Here we use (0, 0) as the starting point.

```
In [36]: x_opt = optimize.fmin_ncg(f, (0, 0), fprime=fprime, fhess=fhess)
         Optimization terminated successfully.
              Current function value: -3.867223
              Iterations: 8
              Function evaluations: 10
              Gradient evaluations: 17
              Hessian evaluations: 8
In [37]: x_opt
Out[37]: array([ 1.88292613,  1.37658523])
```

The routine found a minimum point at $(x_1, x_2) = (1.88292613, 1.37658523)$, and diagnostic information about the solution was also printed to standard output, including the number of iterations and the number of function, gradient, and Hessian evaluations that were required to arrive at the solution. As usual it is illustrative to visualize the objective function and the solution (see Figure 6-4):

```
In [38]: fig, ax = plt.subplots(figsize=(6, 4))
    ...: x_ = y_ = np.linspace(-1, 4, 100)
    ...: X, Y = np.meshgrid(x_, y_)
    ...: c = ax.contour(X, Y, f_lmbda(X, Y), 50)
    ...: ax.plot(x_opt[0], x_opt[1], 'r*', markersize=15)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...: plt.colorbar(c, ax=ax)
```

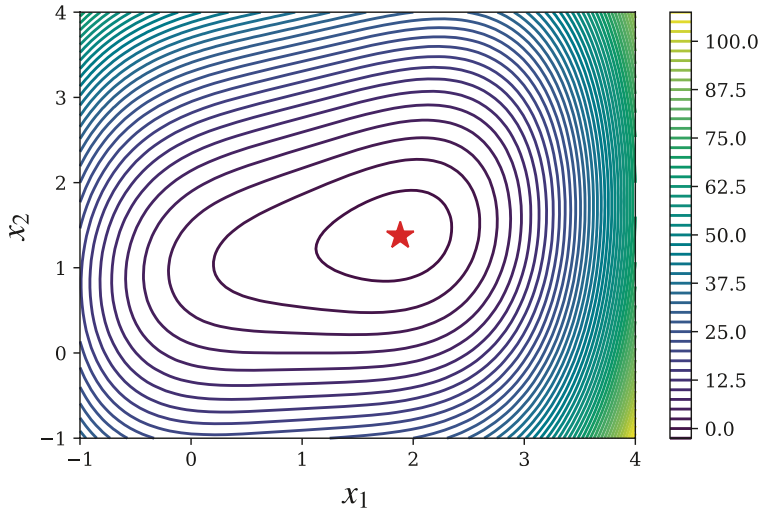

Figure 6-4. *Contour plot of the objective function* $f(x) = (x_1 - 1)^4 + 5(x_2 - 1)^2 - 2x_1x_2$. *The minimum point is marked by a red star.*

In practice, it may not always be possible to provide functions for evaluating both the gradient and the Hessian of the objective function, and often it is convenient with a solver that only requires function evaluations. For such cases, several methods exist to numerically estimate the gradient or the Hessian or both. Methods that approximate the Hessian are known as quasi-Newton methods, and there are also alternative iterative methods that completely avoid using the Hessian. Two popular methods are the Broyden-Fletcher-Goldfarb-Shanno (BFGS) and the conjugate gradient methods, which are implemented in SciPy as the functions `optimize.fmin_bfgs` and `optimize.fmin_cg`. The BFGS method is a quasi-Newton method that can gradually build up numerical estimates of the Hessian, and also the gradient, if necessary. The conjugate gradient method is a variant of the steepest descent method and does not use the Hessian, and it can be used with numerical estimates of the gradient obtained from only function evaluations. With these methods, the number of function evaluations that are required to solve a problem is much larger than for Newton's method, which on the other hand also evaluates the gradient and the Hessian. Both `optimize.fmin_bfgs` and `optimize.fmin_cg` can optionally accept a function for evaluating the gradient, but if not provided, the gradient is estimated from function evaluations.

The preceding problem given, which was solved with the Newton method, can also be solved using the `optimize.fmin_bfgs` and `optimize.fmin_cg`, without providing a function for the Hessian:

```
In [39]: x_opt = optimize.fmin_bfgs(f, (0, 0), fprime=fprime)
         Optimization terminated successfully.
             Current function value: -3.867223
             Iterations: 10
             Function evaluations: 14
             Gradient evaluations: 14
In [40]: x_opt
Out[40]: array([ 1.88292605,  1.37658523])

In [41]: x_opt = optimize.fmin_cg(f, (0, 0), fprime=fprime)
         Optimization terminated successfully.
             Current function value: -3.867223
             Iterations: 7
             Function evaluations: 17
             Gradient evaluations: 17
```

```
In [42]: x_top
Out[42]: array([ 1.88292613,  1.37658522])
```

Note that here, as shown in the diagnostic output from the optimization solvers in the preceding section, the number of function and gradient evaluations is larger than for Newton's method. As already mentioned, both of these methods can also be used without providing a function for the gradient as well, as shown in the following example using the `optimize.fmin_bfgs` solver:

```
In [43]: x_opt = optimize.fmin_bfgs(f, (0, 0))
         Optimization terminated successfully.
             Current function value: -3.867223
             Iterations: 10
             Function evaluations: 56
             Gradient evaluations: 14
In [44]: x_opt
Out[44]: array([ 1.88292604,  1.37658522])
```

In this case the number of function evaluations is even larger, but it is clearly convenient to not have to implement functions for the gradient and the Hessian.

In general, the BFGS method is often a good first approach to try, in particular if neither the gradient nor the Hessian is known. If only the gradient is known, then the BFGS method is still the generally recommended method to use, although the conjugate gradient method is often a competitive alternative to the BFGS method. If both the gradient and the Hessian are known, then Newton's method is the method with the fastest convergence in general. However, it should be noted that although the BFGS and the conjugate gradient methods theoretically have slower convergence than Newton's method, they can sometimes offer improved stability and can therefore be preferable. Each iteration can also be more computationally demanding with Newton's method compared to quasi-Newton methods and the conjugate gradient method, and especially for large problems, these methods can be faster in spite of requiring more iterations.

The methods for multivariate optimization that we have discussed so far all converge to a local minimum in general. For problems with many local minima, this can easily lead to a situation when the solver easily gets stuck in a local minimum, even if a global minimum exists. Although there is no complete and general solution to this problem, a practical approach that can partially alleviate this problem is to use a brute force search over a coordinate grid to find a suitable starting point for an

iterative solver. At least this gives a systematic approach to find a global minimum within given coordinate ranges. In SciPy, the function `optimize.brute` can carry out such a systematic search. To illustrate this method, consider the problem of minimizing the function $4 \sin x\pi + 6 \sin y\pi + (x-1)^2 + (y-1)^2$, which has a large number of local minima. This can make it tricky to pick a suitable initial point for an iterative solver. To solve this optimization problem with SciPy, we first define a Python function for the objective function:

```
In [45]: def f(X):
   ...:      x, y = X
   ...:      return (4 * np.sin(np.pi * x) + 6 * np.sin(np.pi * y)) +
             (x - 1)**2 + (y - 1)**2
```

To systematically search for the minimum over a coordinate grid, we call `optimize.brute` with the objective function `f` as the first parameter and a tuple of `slice` objects as the second argument, one for each coordinate. The `slice` objects specify the coordinate grid over which to search for a minimum value. Here we also set the keyword argument `finish=None`, which prevents the `optimize.brute` from automatically refining the best candidate.

```
In [46]: x_start = optimize.brute(f, (slice(-3, 5, 0.5),
         slice(-3, 5, 0.5)), finish=None)
In [47]: x_start
Out[47]: array([ 1.5,  1.5])
In [48]: f(x_start)
Out[48]: −9.5
```

On the coordinate grid specified by the given tuple of `slice` objects, the optimal point is $(x_1, x_2) = (1.5, 1.5)$, with corresponding objective function minimum -9.5. This is now a good starting point for a more sophisticated iterative solver, such as `optimize.fmin_bfgs`:

```
In [49]: x_opt = optimize.fmin_bfgs(f, x_start)
         Optimization terminated successfully.
              Current function value: -9.520229
              Iterations: 4
              Function evaluations: 28
              Gradient evaluations: 7
```

```
In [50]: x_opt
Out[50]: array([ 1.47586906,  1.48365788])
In [51]: f(x_opt)
Out[51]: -9.52022927306
```

Here the BFGS method gave the final minimum point $(x_1, x_2) = (1.47586906, 1.48365788)$, with the minimum value of the objective function −9.52022927306. For this type of problem, guessing the initial starting point easily results in that the iterative solver converges to a local minimum, and the systematic approach that `optimize.brute` provides is frequently useful.

As always, it is important to visualize the objective function and the solution when possible. The following two code cells plot a contour graph of the current objective function and mark the obtained solution with a red star (see Figure 6-5). As in the previous example, we need a wrapper function for reshuffling the parameters of the objective function because of the different conventions of how the coordinated vectors are passed to the function (separate arrays and packed into one array, respectively).

```
In [52]: def func_X_Y_to_XY(f, X, Y):
    ...:     """
    ...:     Wrapper for f(X, Y) -> f([X, Y])
    ...:     """
    ...:     s = np.shape(X)
    ...:     return f(np.vstack([X.ravel(), Y.ravel()])).reshape(*s)
In [53]: fig, ax = plt.subplots(figsize=(6, 4))
    ...: x_ = y_ = np.linspace(-3, 5, 100)
    ...: X, Y = np.meshgrid(x_, y_)
    ...: c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 25)
    ...: ax.plot(x_opt[0], x_opt[1], 'r*', markersize=15)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...: plt.colorbar(c, ax=ax)
```

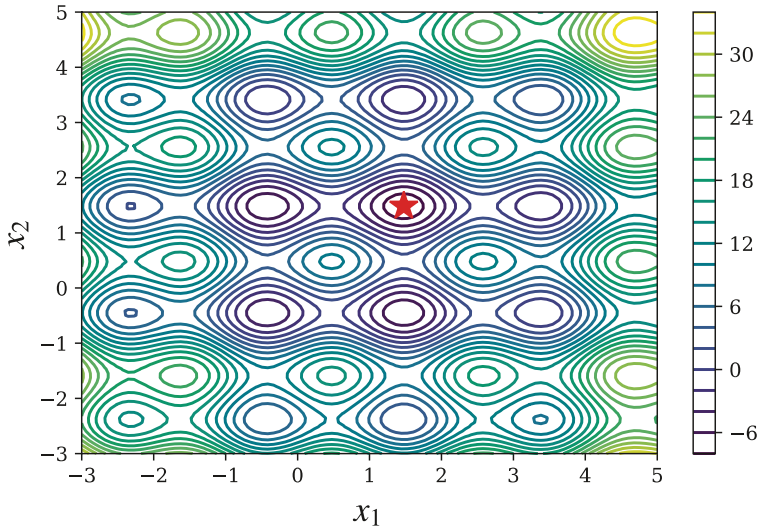

Figure 6-5. *Contour plot of the objective function $f(x) = 4 \sin x\pi + 6 \sin y\pi + (x-1)^2 + (y-1)^2$. The minimum is marked with a red star.*

In this section, we have explicitly called functions for specific solvers, for example, `optimize.fmin_bfgs`. However, like for scalar optimization, SciPy also provides a unified interface for all multivariate optimization solver with the function `optimize.minimize`, which dispatches out to the solver-specific functions depending on the value of the `method` keyword argument (remember, the univariate minimization function that provides a unified interface is `optimize.scalar_minimize`). For clarity, here we have favored explicitly calling functions for specific solvers, but in general it is a good idea to use `optimize.minimize`, as this makes it easier to switch between different solvers. For example, in the previous example, where we used `optimize.fmin_bfgs` in the following way,

```
In [54]: x_opt = optimize.fmin_bfgs(f, x_start)
```

we could just as well have used

```
In [55]: result = optimize.minimize(f, x_start, method= 'BFGS')
In [56]: x_opt = result.x
```

The `optimize.minimize` function returns an instance of `optimize.OptimizeResult` that represents the result of the optimization. In particular, the solution is available via the `x` attribute of this class.

Nonlinear Least Square Problems

In Chapter 5 we encountered linear least square problems and explored how they can be solved with linear algebra methods. In general, a least square problem can be viewed as an optimization problem with the objective function $g(\beta) = \sum_{i=0}^{m} r_i(\beta)^2 = \left\| r(\beta) \right\|^2$, where $r(\beta)$ is a vector with the residuals $r_i(\beta) = y_i - f(x_i, \beta)$ for a set of m observations (x_i, y_i). Here β is a vector with unknown parameters that specifies the function $f(x, \beta)$. If this problem is nonlinear in the parameters β, it is known as a nonlinear least square problem, and since it is nonlinear, it cannot be solved with the linear algebra techniques discussed in Chapter 5. Instead, we can use the multivariate optimization techniques described in the previous section, such as Newton's method or a quasi-Newton method. However, this nonlinear least square optimization problem has a specific structure, and several methods that are tailored to solve this particular optimization problem have been developed. One example is the Levenberg-Marquardt method, which is based on the idea of successive linearizations of the problem in each iteration.

In SciPy, the function `optimize.leastsq` provides a nonlinear least square solver that uses the Levenberg-Marquardt method. To illustrate how this function can be used, consider a nonlinear model on the form $f(x, \beta) = \beta_0 + \beta_1 \exp(-\beta_2 x^2)$ and a set of observations (x_i, y_i). In the following example, we simulate the observations with random noise added to the true values, and we solve the minimization problem that gives the best least square estimates of the parameters β. To begin with, we define a tuple with the true values of the parameter vector β and a Python function for the model function. This function, which should return the y value corresponding to a given x value, takes as first argument the variable x, and the following arguments are the unknown function parameters:

```
In [57]: beta = (0.25, 0.75, 0.5)
In [58]: def f(x, b0, b1, b2):
    ...:     return b0 + b1 * np.exp(-b2 * x**2)
```

Once the model function is defined, we generate randomized data points that simulate the observations.

```
In [59]: xdata = np.linspace(0, 5, 50)
In [60]: y = f(xdata, *beta)
In [61]: ydata = y + 0.05 * np.random.randn(len(xdata))
```

With the model function and observation data prepared, we are ready to start solving the nonlinear least square problem. The first step is to define a function for the residuals given the data and the model function, which is specified in terms of the yet-to-be determined model parameters β.

```
In [62]: def g(beta):
    ...:     return ydata - f(xdata, *beta)
```

Next we define an initial guess for the parameter vector and let the `optimize.leastsq` function solve for the best least square fit for the parameter vector:

```
In [63]: beta_start = (1, 1, 1)
In [64]: beta_opt, beta_cov = optimize.leastsq(g, beta_start)
In [65]: beta_opt
Out[65]: array([ 0.25733353,  0.76867338,  0.54478761])
```

Here the best fit is quite close to the true parameter values $(0.25, 0.75, 0.5)$, as defined earlier. By plotting the observation data and the model function for the true and fitted function parameters, we can visually confirm that the fitted model seems to describe the data well (see Figure 6-6).

```
In [66]: fig, ax = plt.subplots()
    ...: ax.scatter(xdata, ydata, label='samples')
    ...: ax.plot(xdata, y, 'r', lw=2, label='true model')
    ...: ax.plot(xdata, f(xdata, *beta_opt), 'b', lw=2, label='fitted model')
    ...: ax.set_xlim(0, 5)
    ...: ax.set_xlabel(r"$x$", fontsize=18)
    ...: ax.set_ylabel(r"$f(x, \beta)$", fontsize=18)
    ...: ax.legend()
```

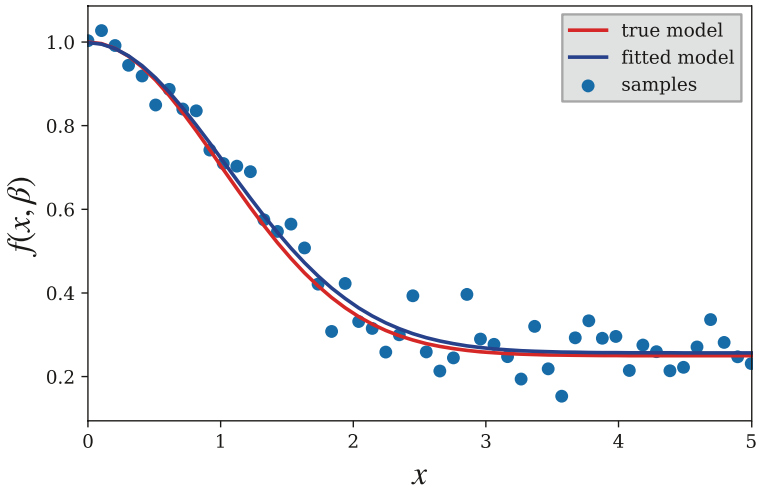

Figure 6-6. *Nonlinear least square fit to the function* $f(x, \beta) = \beta_0 + \beta_1 \exp(-\beta_2 x^2)$ *with* $\beta = (0.25, 0.75, 0.5)$

The SciPy `optimize` module also provides an alternative interface to nonlinear least square fitting, through the function `optimize.curve_fit`. This is a convenience wrapper around `optimize.leastsq`, which eliminates the need to explicitly define the residual function for the least square problem. The previous problem could therefore be solved more concisely using the following:

```
In [67]: beta_opt, beta_cov = optimize.curve_fit(f, xdata, ydata)
In [68]: beta_opt
Out[68]: array([ 0.25733353,  0.76867338,  0.54478761])
```

Constrained Optimization

Constraints add another level of complexity to optimization problems, and they require a classification of their own. A simple form of constrained optimization is the optimization where the coordinate variables are subject to some bounds. For example: $\min_x f(x)$ subject to $0 \leq x \leq 1$. The constraint $0 \leq x \leq 1$ is simple because it only restricts the range of the coordinate without dependencies on the other variables. This type of problems can be solved using the L-BFGS-B method in SciPy, which is a variant of the BFGS method we used earlier. This solver is available through the function `optimize.fmin_l_bgfs_b` or via `optimize.minimize` with the `method` argument set to `'L-BFGS-B'`. To define the coordinate boundaries, the `bound` keyword argument must be used, and its value should be a list of tuples that contain the minimum and maximum value of each constrained variable. If the minimum or maximum value is set to `None`, it is interpreted as an unbounded.

As an example of solving a bounded optimization problem with the L-BFGS-B solver, consider minimizing the objective function $f(x) = (x_1 - 1)^2 - (x_2 - 1)^2$ subject to the constraints $2 \leq x_1 \leq 3$ and $0 \leq x_2 \leq 2$. To solve this problem, we first define a Python function for the objective functions and tuples with the boundaries for each of the two variables in this problem, according to the given constraints. For comparison, in the following code, we also solve the unconstrained optimization problem with the same objective function, and we plot a contour graph of the objective function where the unconstrained and constrained minimum values are marked with blue and red stars, respectively (see Figure 6-7).

```
In [69]: def f(X):
    ...:     x, y = X
    ...:     return (x - 1)**2 + (y - 1)**2
In [70]: x_opt = optimize.minimize(f, [1, 1], method='BFGS').x
In [71]: bnd_x1, bnd_x2 = (2, 3), (0, 2)
In [72]: x_cons_opt = optimize.minimize(f, [1, 1], method='L-BFGS-B',
    ...:                                 bounds=[bnd_x1, bnd_x2]).x
In [73]: fig, ax = plt.subplots(figsize=(6, 4))
    ...: x_ = y_ = np.linspace(-1, 3, 100)
    ...: X, Y = np.meshgrid(x_, y_)
    ...: c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 50)
    ...: ax.plot(x_opt[0], x_opt[1], 'b*', markersize=15)
    ...: ax.plot(x_cons_opt[0], x_cons_opt[1], 'r*', markersize=15)
    ...: bound_rect = plt.Rectangle((bnd_x1[0], bnd_x2[0]),
    ...:                            bnd_x1[1] - bnd_x1[0], bnd_x2[1] -
                                    bnd_x2[0], facecolor="grey")
    ...: ax.add_patch(bound_rect)
    ...: ax.set_xlabel(r"$x_1$", fontsize=18)
    ...: ax.set_ylabel(r"$x_2$", fontsize=18)
    ...: plt.colorbar(c, ax=ax)
```

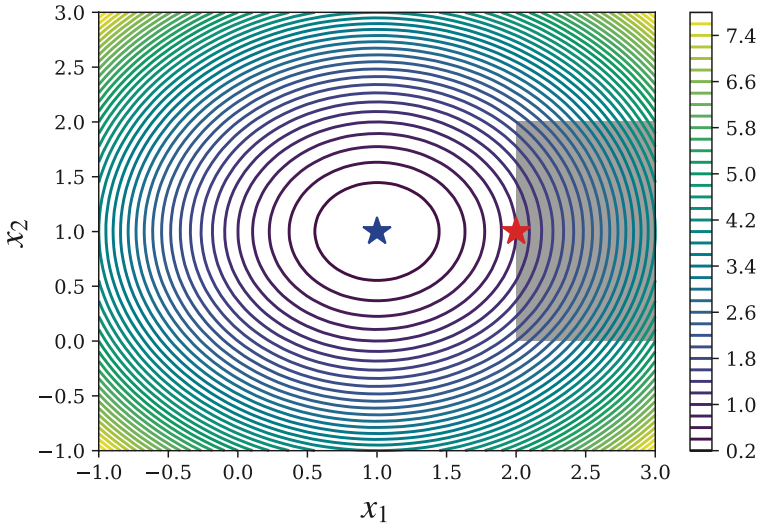

Figure 6-7. *Contours of the objective function f(x), with the unconstrained (blue star) and constrained minima (red star). The feasible region of the constrained problem is shaded in gray.*

Constraints that are defined by equalities or inequalities that include more than one variable are somewhat more complicated to deal with. However, there are general techniques also for this type of problems. For example, using the Lagrange multipliers, it is possible to convert a constrained optimization problem to an unconstrained problem by introducing additional variables. For example, consider the optimization problem $\min_x f(x)$ subject to the equality constraint $g(x) = 0$. In an unconstrained optimization problem, the gradient of $f(x)$ vanishes at the optimal points, $\nabla f(x) = 0$. It can be shown that the corresponding condition for constrained problems is that the negative gradient lies in the space supported by the constraint normal, i.e., $-\nabla f(x) = \lambda J_g^T(x)$. Here $J_g(x)$ is the Jacobian matrix of the constraint function $g(x)$ and λ is the vector of Lagrange multipliers (new variables). This condition arises from equating to zero the gradient of the function $\Lambda(x, \lambda) = f(x) + \lambda^T g(x)$, which is known as the Lagrangian function. Therefore, if both $f(x)$ and $g(x)$ are continuous and smooth, a stationary point (x_0, λ_0) of the function $\Lambda(x, \lambda)$ corresponds to an x_0 that is an optimum of the original constrained optimization problem. Note that if $g(x)$ is a scalar function (i.e., there is only one constraint), then the Jacobian $J_g(x)$ reduces to the gradient $\nabla g(x)$.

To illustrate this technique, consider the problem of maximizing the volume of a rectangle with sides of length x_1, x_2, and x_3, subject to the constraint that the total surface area should be unity: $g(x) = 2x_1x_2 + 2x_0x_2 + 2x_1x_0 - 1 = 0$. To solve this optimization problem

using Lagrange multipliers, we form the Lagrangian $\Lambda(\mathrm{x}) = f(x)+\lambda g(x)$ and seek the stationary points for $\nabla\Lambda(\mathrm{x}) = 0$. With SymPy, we can carry out this task by first defining the symbols for the variables in the problem, then constructing expressions for $f(x)$, $g(x)$, and $\Lambda(\mathrm{x})$,

```
In [74]: x = x0, x1, x2, l = sympy.symbols("x_0, x_1, x_2, lambda")
In [75]: f = x0 * x1 * x2
In [76]: g = 2 * (x0 * x1 + x1 * x2 + x2 * x0) - 1
In [77]: L = f + l * g
```

and finally computing $\nabla\Lambda(\mathrm{x})$ using `sympy.diff` and solving the equation $\nabla\Lambda(\mathrm{x}) = 0$ using `sympy.solve`:

```
In [78]: grad_L = [sympy.diff(L, x_) for x_ in x]
In [79]: sols = sympy.solve(grad_L)
In [80]: sols
```

Out[80]: $$\left[\left\{\lambda : -\frac{\sqrt{6}}{24}, x_0 : \frac{\sqrt{6}}{6}, x_1 : \frac{\sqrt{6}}{6}, x_2 : \frac{\sqrt{6}}{6}\right\}, \left\{\lambda : \frac{\sqrt{6}}{24}, x_0 : -\frac{\sqrt{6}}{6}, x_1 : -\frac{\sqrt{6}}{6}, x_2 : -\frac{\sqrt{6}}{6}\right\}\right]$$

This procedure gives two stationary points. We could determine which one corresponds to the optimal solution by evaluating the objective function for each case. However, here only one of the stationary points corresponds to a physically acceptable solution: since x_i is the length of a rectangle side in this problem, it must be positive. We can therefore immediately identify the interesting solution, which corresponds to the intuitive result $x_0 = x_1 = x_2 = \frac{\sqrt{6}}{6}$ (a cube). As a final verification, we evaluate the constraint function and the objective function using the obtained solution:

```
In [81]: g.subs(sols[0])
Out[81]: 0
In [82]: f.subs(sols[0])
```

Out[82]: $$\frac{\sqrt{6}}{36}$$

This method can be extended to handle inequality constraints as well, and there exist various numerical methods of applying this approach. One example is the method known as sequential least square programming, abbreviated as SLSQP, which is available in the SciPy as the `optimize.slsqp` function and via `optimize.minimize`

with `method='SLSQP'`. The `optimize.minimize` function takes the keyword argument `constraints`, which should be a list of dictionaries that each specifies a constraint. The allowed keys (values) in this dictionary are `type` (`'eq'` or `'ineq'`), `fun` (constraint function), `jac` (Jacobian of the constraint function), and `args` (additional arguments to constraint function and the function for evaluating its Jacobian). For example, the constraint dictionary describing the constraint in the previous problem would be `dict(type='eq', fun=g)`.

To solve the full problem numerically using SciPy's SLSQP solver, we need to define Python functions for the objective function and the constraint function:

```
In [83]: def f(X):
    ...:     return -X[0] * X[1] * X[2]
In [84]: def g(X):
    ...:     return 2 * (X[0]*X[1] + X[1] * X[2] + X[2] * X[0]) - 1
```

Note that since the SciPy optimization functions solve minimization problems, and here we are interested in maximization, the function `f` is here the negative of the original objective function. Next we define the constraint dictionary for $g(x) = 0$ and finally call the `optimize.minimize` function

```
In [85]: constraint = dict(type='eq', fun=g)
In [86]: result = optimize.minimize(f, [0.5, 1, 1.5], method='SLSQP',
         constraints=[constraint])
In [87]: result
Out[87]:  status: 0
         success: True
            njev: 18
            nfev: 95
             fun: -0.068041368623352985
               x: array([ 0.40824187,  0.40825127,  0.40825165])
         message: 'Optimization terminated successfully.'
             jac: array([-0.16666925, -0.16666542, -0.16666527,  0.])
             nit: 18
In [88]: result.x
Out[88]: array([ 0.40824187,  0.40825127,  0.40825165])
```

As expected, the solution agrees well with the analytical result obtained from the symbolic calculation using Lagrange multipliers.

To solve problems with inequality constraints, all we need to do is to set `type='ineq'` in the constraint dictionary and provide the corresponding inequality function. To demonstrate minimization of a nonlinear objective function with a nonlinear inequality constraint, we return to the quadratic problem considered previously but in this case with inequality constraint $g(x) = x_1 - 1.75 - (x_0 - 0.75)^4 \geq 0$. As usual, we begin by defining the objective function and the constraint function, as well as the constraint dictionary:

```
In [89]: def f(X):
    ...:     return (X[0] - 1)**2 + (X[1] - 1)**2
In [90]: def g(X):
    ...:     return X[1] - 1.75 - (X[0] - 0.75)**4
In [91]: constraints = [dict(type='ineq', fun=g)]
```

Next, we are ready to solve the optimization problem by calling the `optimize.minimize` function. For comparison, here we also solve the corresponding unconstrained problem.

```
In [92]: x_opt = optimize.minimize(f, (0, 0), method='BFGS').x
In [93]: x_cons_opt = optimize.minimize(f, (0, 0), method='SLSQP',
         constraints=constraints).x
```

To verify the soundness of the obtained solution, we plot the contours of the objective function together with a shaded area representing the feasible region (where the inequality constraint is satisfied). The constrained and unconstrained solutions are marked with a red and a blue star, respectively (see Figure 6-8).

```
In [94]: fig, ax = plt.subplots(figsize=(6, 4))
In [95]: x_ = y_ = np.linspace(-1, 3, 100)
    ...: X, Y = np.meshgrid(x_, y_)
    ...: c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 50)
    ...: ax.plot(x_opt[0], x_opt[1], 'b*', markersize=15)
    ...: ax.plot(x_, 1.75 + (x_-0.75)**4, 'k-', markersize=15)
    ...: ax.fill_between(x_, 1.75 + (x_-0.75)**4, 3, color='grey')
    ...: ax.plot(x_cons_opt[0], x_cons_opt[1], 'r*', markersize=15)
    ...:
```

```
...: ax.set_ylim(-1, 3)
...: ax.set_xlabel(r"$x_0$", fontsize=18)
...: ax.set_ylabel(r"$x_1$", fontsize=18)
...: plt.colorbar(c, ax=ax)
```

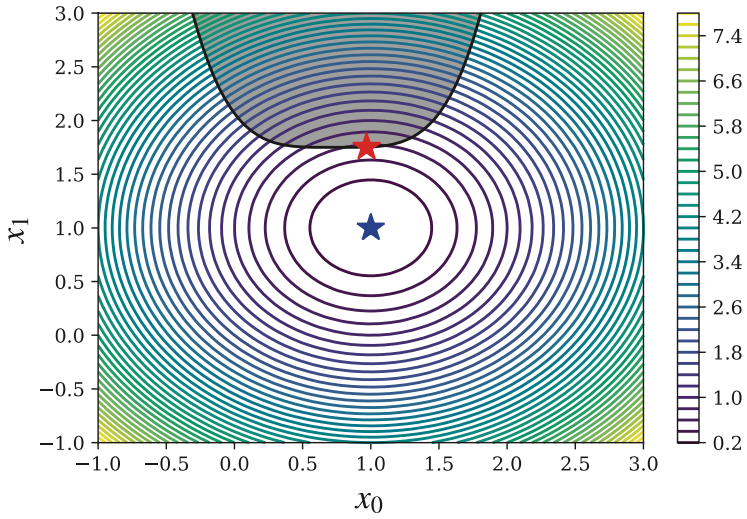

Figure 6-8. *Contour plot of the objective function with the feasible region of the constrained problem shaded gray. The red and blue stars are the optimal points in the constrained and unconstrained problems, respectively.*

For optimization problems with *only* inequality constraints, SciPy provides an alternative solver using the constrained optimization by linear approximation (COBYLA) method. This solver is accessible either through `optimize.fmin_cobyla` or `optimize.minimize` with `method='COBYLA'`. The previous example could just as well have been solved with this solver, by replacing `method='SLSQP'` with `method='COBYLA'`.

Linear Programming

In the previous section, we considered methods for very general optimization problems, where the objective function and constraint functions all can be nonlinear. However, at this point it is worth taking a step back and considering a much more restricted type of optimization problem, namely, *linear programming*, where the objective function is linear and all constraints are linear equality or inequality constraints. The class of problems is clearly much less general, but it turns out that linear programming has many important applications, and they can be solved vastly more efficiently than general nonlinear problems. The reason for this is that linear problems have properties that enable completely different methods to be used. In particular, the solution to a linear

optimization problem must necessarily lie on a constraint boundary, so it is sufficient to search the vertices of the intersections of the linear constraint functions. This can be done efficiently in practice. A popular algorithm for this type of problems is known as *simplex*, which systematically moves from one vertex to another until the optimal vertex has been reached. There are also more recent interior point methods that efficiently solve linear programming problems. With these methods, linear programming problems with thousands of variables and constraints are readily solvable.

Linear programming problems are typically written in the so-called standard form: $\min_x c^T x$ where $Ax \leq b$ and $x \geq 0$. Here c and x are vectors of length n, and A is a $m \times n$ matrix and b a m-vector. For example, consider the problem of minimizing the function $f(x) = -x_0+2x_1 - 3x_2$, subject to the three inequality constraints $x_0+x_1 \leq 1$, $-x_0+3x_1 \leq 2$, and $-x_1+x_2 \leq 3$. On the standard form, we have $c = (-1, 2, -3)$, $b = (1, 2, 3)$, and

$$A = \begin{pmatrix} 1 & 1 & 0 \\ -1 & 3 & 0 \\ 0 & -1 & 1 \end{pmatrix}.$$

To solve this problem, here we use the `cvxopt` library, which provides the linear programming solver with the `cvxopt.solvers.lp` function. This solver expects as arguments the c, A, and b vectors and matrix used in the standard form introduced in the preceding text, in the given order. The `cvxopt` library uses its own classes for representing matrices and vectors, but fortunately they are interoperable with NumPy arrays via the array interface[3] and can therefore be cast from one form to another using the `cvxopt.matrix` and `np.array` functions. Since NumPy array is the de facto standard array format in the scientific Python environment, it is sensible to use NumPy array as far as possible and only convert to cvxopt matrices when necessary, i.e., before calling one of the solvers in `cvxopt.solvers`.

To solve the stated example problem using the `cvxopt` library, we therefore first create NumPy arrays for the A matrix and the c and b vectors and convert them to `cvxopt` matrices using the `cvxpot.matrix` function:

```
In [96]: c = np.array([-1.0, 2.0, -3.0])
In [97]: A = np.array([[ 1.0, 1.0, 0.0],
                       [-1.0, 3.0, 0.0],
                       [ 0.0, -1.0, 1.0]])
```

[3]For details, see http://docs.scipy.org/doc/numpy/reference/arrays.interface.html.

```
In [98]: b = np.array([1.0, 2.0, 3.0])
In [99]: A_ = cvxopt.matrix(A)
In [100]: b_ = cvxopt.matrix(b)
In [101]: c_ = cvxopt.matrix(c)
```

The cvxopt compatible matrices and vectors c_, A_, and b_ can now be passed to the linear programming solver cvxopt.solvers.lp:

```
In [102]: sol = cvxopt.solvers.lp(c_, A_, b_)
         Optimal solution found.
In [103]: sol
Out[103]: {'dual infeasibility': 1.4835979218054372e-16,
           'dual objective': -10.0,
           'dual slack': 0.0,
           'gap': 0.0,
           'iterations': 0,
           'primal infeasibility': 0.0,
           'primal objective': -10.0,
           'primal slack': -0.0,
           'relative gap': 0.0,
           'residual as dual infeasibility certificate': None,
           'residual as primal infeasibility certificate': None,
           's': <3x1 matrix, tc='d'>,
           'status': 'optimal',
           'x': <3x1 matrix, tc='d'>,
           'y': <0x1 matrix, tc='d'>,
           'z': <3x1 matrix, tc='d'>}
In [104]: x = np.array(sol['x'])
In [105]: x
Out[105]: array([[ 0.25],
                 [ 0.75],
                 [ 3.75]])
In [106]: sol['primal objective']
Out[106]: -10.0
```

The solution to the optimization problem is given in terms of the vector x, which in this particular example is $x = (0.25, 0.75, 3.75)$, which corresponds to the $f(x)$ value -10. With this method and the `cvxopt.solvers.lp` solver, linear programming problems with hundreds or thousands of variables can readily be solved. All that is needed is to write the optimization problem on the standard form and create the c, A, and b arrays.

Summary

Optimization – to select the best option from a set of alternatives – is fundamental in many applications in science and engineering. Mathematical optimization provides a rigorous framework for systematically treating optimization problems, if they can be formulated as a mathematical problem. Computational methods for optimization are the tools with which such optimization problems are solved in practice. In a scientific computing environment, optimization therefore plays a very important role. For scientific computing with Python, the SciPy library provides efficient routines for solving many standard optimization problems, which can be used to solve a vast variety of computational optimization problems. However, optimization is a large field in mathematics, requiring a different array of methods for solving different types of problems, and there are several optimization libraries for Python that provide specialized solvers for specific types of optimization problems. In general, the SciPy `optimize` module provides good and flexible general-purpose solvers for a wide variety of optimization problems, but for specific types of optimization problems, there are also many specialized libraries that provide better performance or more features. An example of such a library is `cvxopt`, which complements the general-purpose optimization routines in SciPy with efficient solvers for linear and quadratic problems.

Further Reading

For an accessible introduction to optimization, with more detailed discussions of the numerical properties of several of the methods introduced in this chapter, see, for example, Heath (2002). For a more rigorous and in-depth introduction to optimization, see, for example, E.K.P. Chong (2013). A thorough treatment of convex optimization is given by the creators of the `cvxopt` library in the excellent book (S. Boyd, 2004), which is also available online at `http://stanford.edu/~boyd/cvxbook`.

References

E.K.P. Chong, S. Z. (2013). *An Introduction to Optimization* (4th ed.). New York: Wiley.

Heath, M. (2002). *Scientific Computing: An introductory Survey* (2nd ed.). Boston: McGraw-Hill.

S. Boyd, L. V. (2004). *Convex Optimization.* Cambridge: Cambridge University Press.