

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 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 (that is, 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 allow 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.

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 provide 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 0.15.1. See <http://www.scipy.org> for more information.

Importing Modules

The SciPy package `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 (with 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 is 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

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2, \\ &\dots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m. \end{aligned}$$

This is a linear system of m equations in n unknown variables $\{x_1, x_2, \dots, x_n\}$, and 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:

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

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 of 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 less 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 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, 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 this: 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

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

Thus, a bound for the relative error in the solution x , given a relative error in the b vector, is 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:

$$\begin{aligned} 2x_1 + 3x_2 &= 4 \\ 5x_1 + 4x_2 &= 23 \end{aligned}$$

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 $(-1, 2)$.

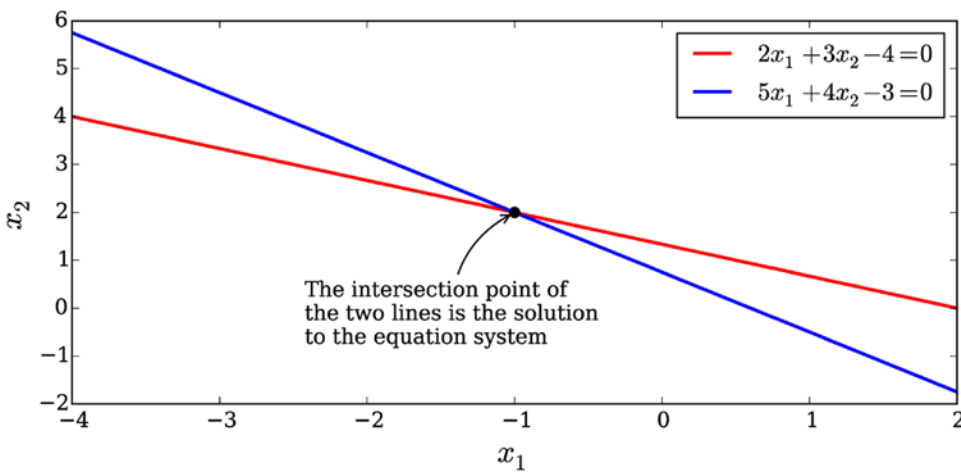


Figure 5-1. Graphical representation of a system of two linear equations

We can define this problem in SymPy by creating matrix objects for A and b , and compute 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]:  $\frac{\sqrt{27+2\sqrt{170}}}{\sqrt{27-2\sqrt{170}}}$ 
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 multiplying 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 solve $Ux = y$ with backwards substitution. Owing 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 solving 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 from 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]: P, L, U = la.lu(A)
In [27]: L
Out[27]: array([[ 1. ,  0. ],
               [ 0.4,  1. ]])
In [28]: U
Out[28]: array([[ 5. ,  4. ],
               [ 0. ,  1.4]])
In [29]: L*U
Out[29]: array([[ 5. ,  0. ],
               [ 0. ,  1.4]])
In [30]: la.solve(A, b)
Out[30]: 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 code below (In [36]) 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 numbers. In this example we solve the equation system

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

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 [31]: p = sympy.symbols("p", positive=True)
In [32]: A = sympy.Matrix([[1, sympy.sqrt(p)], [1, 1/sympy.sqrt(p)]])
In [33]: b = sympy.Matrix([1, 2])
In [34]: x = A.solve(b)
In [35]: x
Out[35]: 
$$\begin{pmatrix} \frac{2p-1}{p-1} \\ -\frac{\sqrt{p}}{p-1} \end{pmatrix}$$


```

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 [36]: # 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(symbolic - numerical)")
...: axes[0].set_xlabel(r'$x$', 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'$x$', fontsize=18)

```

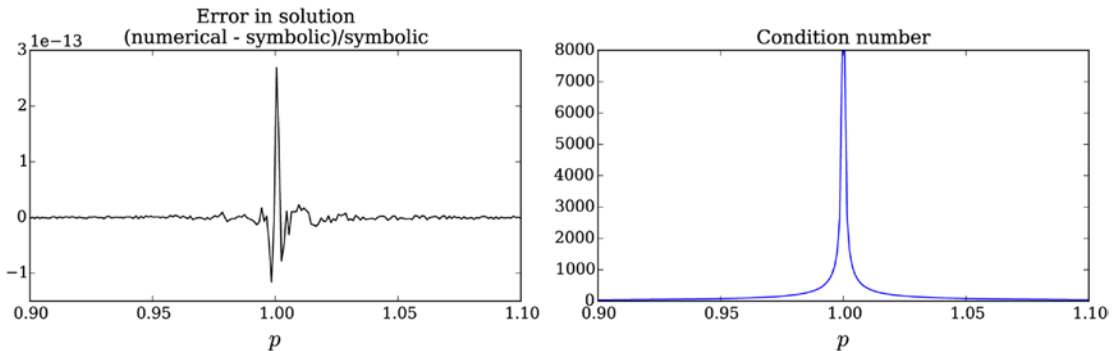


Figure 5-2. Graph of the relative numerical errors (left) and condition number (right) as a function of the parameter p

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 that 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 [37]: x_vars = sympy.symbols("x_1, x_2, x_3")
In [38]: A = sympy.Matrix([[1, 2, 3], [4, 5, 6]])
In [39]: x = sympy.Matrix(x_vars)
In [40]: b = sympy.Matrix([7, 8])
In [41]: sympy.solve(A*x - b, x_vars)
Out[41]: {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 equations system.

On the other hand, if the system 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 $\{(x_i, y_i)\}_{i=1}^m$ 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 give a *best fit* for the overdetermined system. A natural definition of best fit for the overdetermined system $Ax \approx b$, is to minimize the sum of square error, $\min_x \sum_{i=1}^m (r_i)^2$, 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 above, 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 [42]: # 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, s = 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)

```

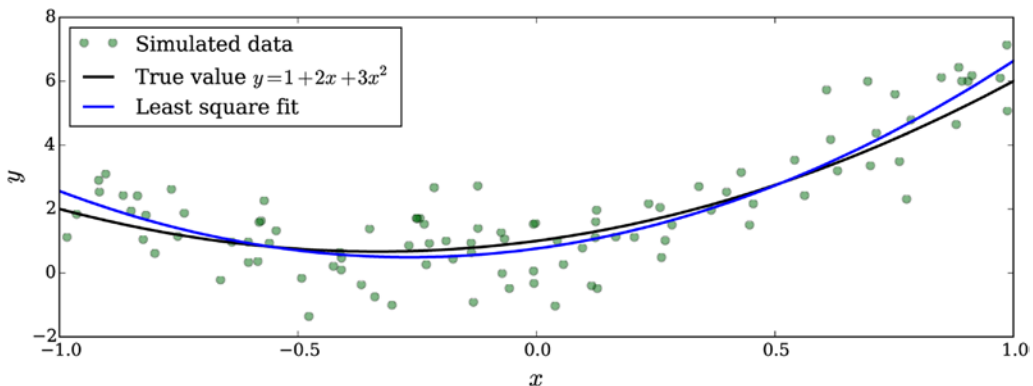


Figure 5-3. Linear least square fit

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 [43]), 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). In 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 [43]: # 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)

```

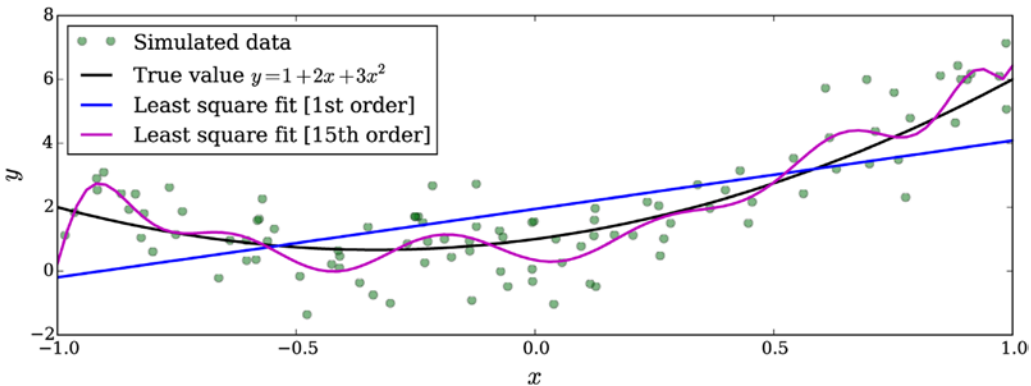


Figure 5-4. Graph demonstrating underfitting and overfitting of data using the linear least square method

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 noting 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 N th 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 n th 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` method of the `Matrix` class, which is 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 [44]: eps, delta = sympy.symbols("epsilon, Delta")
In [45]: H = sympy.Matrix([[eps, delta], [delta, -eps]])

```

In [46]: H

Out[46]: $\begin{pmatrix} \varepsilon & \Delta \\ \Delta & -\varepsilon \end{pmatrix}$

In [47]: H.eigenvals()

Out[47]: $\{-\sqrt{\varepsilon^2 + \Delta^2} : 1, \sqrt{\varepsilon^2 + \Delta^2} : 1\}$

In [48]: H.eigenvecs()

Out[48]: $\left(\left(-\sqrt{\varepsilon^2 + \Delta^2}, 1, \left[\begin{array}{c} -\frac{\Delta}{\varepsilon + \sqrt{\varepsilon^2 + \Delta^2}} \\ 1 \end{array} \right] \right), \left(\sqrt{\varepsilon^2 + \Delta^2}, 1, \left[\begin{array}{c} -\frac{\Delta}{\varepsilon - \sqrt{\varepsilon^2 + \Delta^2}} \\ 1 \end{array} \right] \right) \right)$

The return value of the `eigenvals` method is 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 `eigenvecs` 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 `eigenvecs`, and verify that the two eigenvectors are orthogonal using for example:

In [49]: (eval1, _, vec1), (eval2, _, vec2) = H.eigenvecs()

In [50]: sympy.simplify(vec1[0].T * vec2[0])

Out[50]: [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 cumbersome to work with even using a computer algebra system such as SymPy. Thus, for larger systems we must resort to a fully numerical approach. For this we can use the `la.eigenvals` 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 is stored in a NumPy array with real values. For example, to solve a numerical eigenvalue problem with `la.eig` we can use:

In [51]: A = np.array([[1, 3, 5], [3, 5, 3], [5, 3, 9]])

In [52]: evals, evcs = la.eig(A)

In [53]: evals

Out[53]: array([13.35310908+0.j, -1.75902942+0.j, 3.40592034+0.j])

In [54]: evcs

Out[54]: array([[0.42663918, 0.90353276, -0.04009445],
[0.43751227, -0.24498225, -0.8651975],
[0.79155671, -0.35158534, 0.49982569]])

In [55]: la.eigvalsh(A)

Out[55]: array([-1.75902942, 3.40592034, 13.35310908])

Since the matrix in this example is symmetric, we could use `la.eigh` and `la.eigvalsh`, giving a real-valued eigenvalue arrays, as shown in the last cell of the example.

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 of 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 variable with a power greater than 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 $f(x) = 0$, 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 [56]: x, a, b, c = sympy.symbols("x, a, b, c")
In [57]: sympy.solve(a + b*x + c*x**2, x)
Out[57]: [(-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 [58]: sympy.solve(a * sympy.cos(x) - b * sympy.sin(x), x)
Out[58]: [-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 contains both polynomial expressions and elementary functions, such as $\sin x = x$, are often transcendental, and does 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 [59]: 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, considering the following example (In [60]), 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 [60]: 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)
```

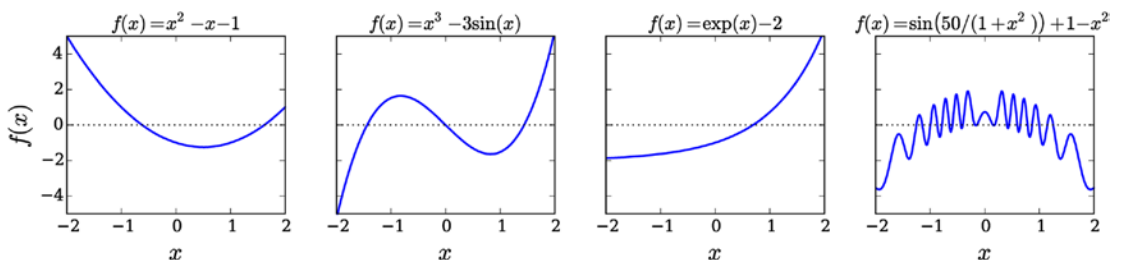


Figure 5-5. Graphs of four examples of nonlinear functions

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 Newton method.

The bisection method requires a starting interval $[a, b]$ such that $f(a)$ and $f(b)$ have different sign. 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 , 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 different signs at the two endpoints of the interval, and in each iteration the interval is halved, and therefore converges towards 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 [61]: # 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")

```

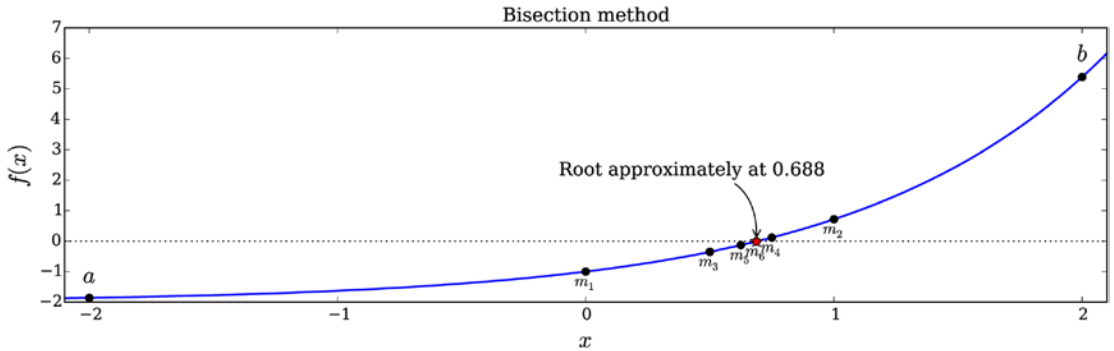


Figure 5-6. Graphical visualization of how the bisection method works

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 if $f'(x_k)$ is zero at some point x_k . This special case would have to be dealt in a real implementation of this method. The following example (In [62]) 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 [62]: # 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_new], [0, f(xk)], color='k', ls=':')
...:     ax.plot(xk, f(xk), 'ko')
...:     ax.text(xk, -0.5, 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])

```

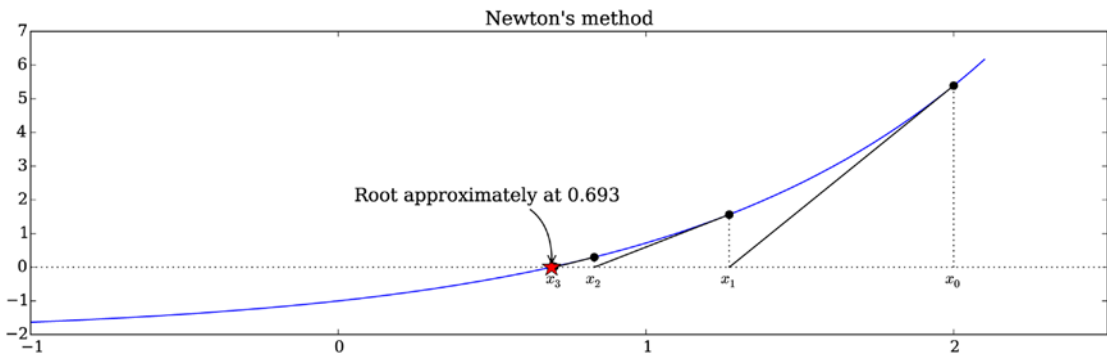


Figure 5-7. Visualization of the root-finding steps in Newton's method for the equation $\exp(x) - 2 = 0$

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. A state-of-the-art implementations numerical root-finding functions typically use the basic idea of either the bisection method or Newton's method, or a combination of both, but additionally uses 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 (for example 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 is the lower and upper value 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 [63]: optimize.bisect(lambda x: np.exp(x) - 2, -2, 2)
Out[63]: 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 first argument, and an initial guess for the root of the function as 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 [64]: x_root_guess = 2
In [65]: f = lambda x: np.exp(x) - 2
In [66]: fprime = lambda x: np.exp(x)
In [67]: optimize.newton(f, x_root_guess)
Out[67]: 0.69314718056
In [68]: optimize.newton(f, x_root_guess, fprime=fprime)
Out[68]: 0.69314718056
```

Note that with this method we have less control over which root is being computed, if the function have multiple roots. For instance, there is no guarantee that the root the function returns is the closest one to the initial guess, we can we cannot know in advance if the root that 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, and 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 [69]: optimize.brentq(lambda x: np.exp(x) - 2, -2, 2)
Out[69]: 0.6931471805599453
In [70]: optimize.brenth(lambda x: np.exp(x) - 2, -2, 2)
Out[70]: 0.6931471805599381
```

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

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 a 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 increase.

Not all methods discussed in for univariate equation solving can be generalized to the multivariate case. In particular, the bisection method cannot be directly generalized to a multivariate equation system. Newton's method, however, can be generalized to the multivariate equation systems, in which case the 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 for 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` provides 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 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 $f([x_1, x_2]) = [x_2 - x_1^3 - 2x_1^2 + 1, x_2 + x_1^2 - 1]$. To solve this equation system using SciPy, we need 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 [71]: def f(x):
...:     return [x[1] - x[0]**3 - 2 * x[0]**2 + 1, x[1] + x[0]**2 - 1]
In [72]: optimize.fsolve(f, [1, 1])
Out[72]: 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 [73]: x, y = sympy.symbols("x, y")
In [74]: f_mat = sympy.Matrix([y - x**3 - 2*x**2 + 1, y + x**2 - 1])
```

```
In [75]: f_mat.jacobian(sympy.Matrix([x, y]))
```

```
Out[75]:  $\begin{pmatrix} -3x^2 - 4x & 1 \\ 2x & 1 \end{pmatrix}$ 
```

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

```
In [76]: def f_jacobian(x):
...:     return [-3*x[0]**2-4*x[0], 1], [2*x[0], 1]
In [77]: optimize.fsolve(f, [1, 1], fprime=f_jacobian)
Out[77]: array([ 0.73205081,  0.46410162])
```

As with new 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 are found for to the equations. There is no guarantee that any particular solution is found, although 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, code below 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 [78]: 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)
```

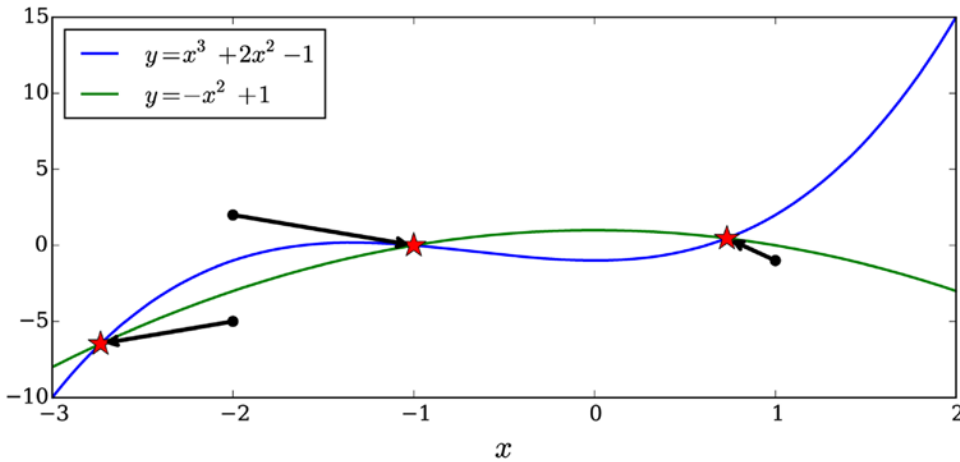


Figure 5-8. Graph of a system of two nonlinear equations. The solutions are indicated with red stars, and the initial guess with a black dot and an arrow to the solution each initial guess eventually converged to

By systematically solving the equation systems with different initial guesses, we can build visualization of how different initial guesses converges to different solutions. This is done in the code example below, and the result is shown in Figure 5-9. This examples demonstrates that even for this relatively simple example, the regions of initial guesses that converges to different solutions is highly nontrivial, and there are also missing dots that correspond 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 for the characteristics of a particular problem.

```
In [79]: fig, ax = plt.subplots(figsize=(8, 4))
...:
...: ax.plot(x, y1, 'k', lw=1.5, label=r'$y = x^3 + 2x^2 - 1$')
...: ax.plot(x, y2, 'k', lw=1.5, label=r'$y = -x^2 + 1$')
...:
...: sol1 = optimize.fsolve(f, [-2, 2])
...: sol2 = optimize.fsolve(f, [ 1, -1])
...: sol3 = optimize.fsolve(f, [-2, -5])
...:
...: 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)
...:
...:         for idx, s in enumerate([sol1, sol2, sol3]):
...:             if abs(s-sol).max() < 1e-8:
...:                 ax.plot(sol[0], sol[1], colors[idx]+'*', markersize=15)
...:                 ax.plot(x_guess[0], x_guess[1], colors[idx]+'.')
...:
...: ax.set_xlabel(r'$x$', fontsize=18)
```

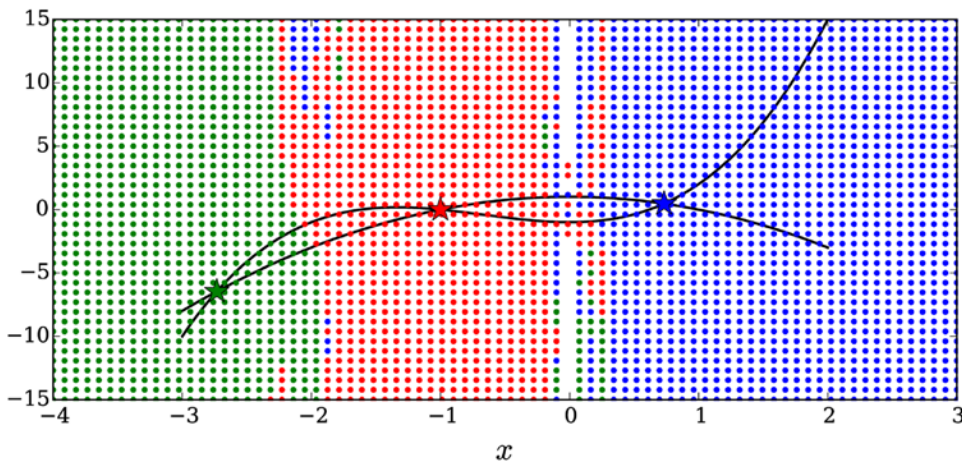


Figure 5-9. Visualization of the convergence of different initial guesses to different solutions. Each dot represents an initial guess, and its color encodes which solution it eventually converges to. The solutions are marked with correspondingly color-coded stars

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 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 books that cover these topics is (Heath, 2001) and (W.H. Press, 2007), which give 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.