Computational Physics

Eitan Lees

2025-02-24

Table of contents

Preface

I first learned to program in 2011 in Dr. Richard O. Gray's computational physics course.

It was a revelatory experience that shaped my life. I enjoyed the class so much that I served as the TA the following year and later earned a PhD in Computational Science.

I've kept the lecture notes on my various computers ever since. I want to convert the notes into a Quarto book as a conservation project. In addition to replicating the source material I want to add some examples from Python and Scipy.

Colophon

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

To learn more about Quarto books, visit https://quarto.org/docs/books

1 Interpolation

1.1 Linear Interpolation

A common computational problem in physics involves determining the value of a particular function at one or more points of interest from a tabulation of that function. For instance, we may wish to calculate the index of refraction of a type of glass at a particular wavelength, but be faced with the problem that that particular wavelength is not explicitly in the tabulation. In such cases, we need to be able to interpolate in the table to find the value of the function at the point of interest. Let us take a particular example.

BK-7 is a type of common optical crown glass. Its index of refraction n varies as a function of wavelength; for shorter wavelengths n is larger than for longer wavelengths, and thus violet light is refracted more strongly than red light, leading to the phenomenon of dispersion. The index of refraction is tabulated in Table 1.1.

Let us suppose that we wish to find the index of refraction at a wavelength of 5000. Unfortunately, that wavelength is not found in the table, and so we must estimate it from the values in the table. We must make some assumption about how n varies between the tabular values. Presumably it varies in a smooth sort of way and does not take wild excursions between the tabulated values. The simplest and quite often an entirely adequate assumption to make is that the actual function varies linearly between the tabulated values. This is the basis of **linear interpolation**.

> **Exercise 4.1**
>
> Determine, by hand, the value of the index of refraction of BK7 at 5000 using linear interpolation.

How do we carry out linear interpolation on the computer? Let us suppose that the function is tabulated at N points and takes on the values $y_1, y_2, y_3 ... y_N$ at the points $x_1, x_2, x_3 ... x_N$, and that we want to find the value of the function y at a point x that lies someplace in the interval between x_1 and x_N.

The first thing that we must do is to bracket x, that is we must find a j such that $x_j < x \leq x_{j+1}$. This can be accomplished by the following code fragment:

Table 1.1: Refractive Index for BK7 Glass

$\lambda()$	n
3511	1.53894
3638	1.53648
4047	1.53024
4358	1.52669
4416	1.52611
4579	1.52462
4658	1.52395
4727	1.52339
4765	1.5231
4800	1.52283
4861	1.52238
4880	1.52224

$\lambda()$	n
4965	1.52165
5017	1.5213
5145	1.52049
5320	1.51947
5461	1.51872
5876	1.5168
5893	1.51673
6328	1.51509
6438	1.51472
6563	1.51432
6943	1.51322
7860	1.51106

$\lambda()$	n
8210	1.51037
8300	1.51021
8521	1.50981
9040	1.50894
10140	1.50731
10600	1.50669
13000	1.50371
15000	1.5013
15500	1.50068
19701	1.495
23254	1.48929

```
for(i=1;i<N;i++) {
  if(xn[i] < x && xn[i+1] >= x) {
    j = i;
    break;
  }
}
```

where the `xn`'s are the tabulated points. When the `if` statement is satisfied, `j` is assigned the value of `i` and the procedure drops out of the loop. Please note that this is not the most efficient way to accomplish this task, especially if N is very large. We will look at a more efficient way later on.

Once we have bracketed x, we can find the equation of the line between the points (x_j, y_j) and (x_{j+1}, y_{j+1}). This equation will be of the form $y = mx + b4where\text{m\$}$ is the slope and b is the y-intercept. As we all know, the slope is given by

$$m = \frac{y_{j+1} - y_j}{x_{j+1} - x_j}$$

and the intercept can be found by substituting one point, say, (x_j, y_j) into the resulting equation. Thus,

$$b = y - mx = y_j - \frac{y_{j+1} - y_j}{x_{j+1} - x_j} x_j$$

yielding for the equation of the line, after some rearrangement,

$$y == y_j + \left(\frac{y_{j+1} - y_j}{x_{j+1} - x_j}\right)(x - x_j)$$

It is left to the student to show (for future reference) that this equation may be rewritten

$$y = Ay_j + By_{j+1}$$

where

$$A = \frac{x_{j+1} - x}{x_{j+1} - x_j}$$

and

$$B = \frac{x - x_j}{x_{j+1} - x_j}$$

i Exercise 4.2

Write a C-function that will linearly interpolate the tabular data for the index of refraction of BK-7 and return a value for n for wavelengths between 3511 and 23254. Write a driver program that will use this function to prompt the user for a wavelength and then print to screen the corresponding value of n.

i Exercise 4.3

The file `data/boiling.dat` contains data in two columns for the boiling point of water at different atmospheric pressures. The first column is the pressure in millibars, the second is the corresponding boiling point temperature in degrees Celsius. Write a C-function that initializes two vectors, `P` and `T` with the data in that data file (don't read in the datafile – hardwire the data into your program), accepts the pressure as a double floating-point parameter, and returns the value of the temperature of the boiling point at that pressure. You should also write a driver program that will prompt the user for an atmospheric pressure, check whether it is within the limits of the data ($50 \leq P \leq 2150$), calls your C-function, and prints to the screen the boiling point of water at that pressure.

1.2 Polynomial Interpolation

Linear interpolation is good enough for government work, and it is even better than that. Because it is simple and makes the simplest possible assumption about the data, it should be employed in all cases except where it is manifestly inadequate. There are such cases. Sometimes the function being interpolated is very non-linear or has been tabulated at such wide intervals

that linear interpolation would lead to large errors. Some applications demand more than simply the functional values at the interpolated points; sometimes the derivative of the function is required as well. With linear interpolation, the derivative is a constant between the tabulated points, and may actually be undefined at the tabulated points!

For such applications it may be best to interpolate using a polynomial interpolating function or functions. It can be shown that the following poly- nomial $P(x)$ of degree $N-1$ will exactly pass through the N tabulated points of the function $y = f(x)$:

$$\begin{aligned} P(x) = & \frac{(x-x_2)(x-x_3)\cdots(x-x_N)}{(x_1-x_2)(x_1-x_3)\cdots(x_1-x_N)}\, y_1 \\ & + \frac{(x-x_1)(x-x_3)\cdots(x-x_N)}{(x_2-x_1)(x_2-x_3)\cdots(x_2-x_N)}\, y_2 \\ & + \cdots \\ & + \frac{(x-x_1)(x-x_2)\cdots(x-x_{N-1})}{(x_N-x_1)(x_N-x_2)\cdots(x_N-x_{N-1})}\, y_N \end{aligned}$$

The problem with the direct application of the polynomial $P(x)$ is that for tabulations with many points, it can lead to very high degree polynomials. For instance, if a function is tabulated at 100 points, the above equation would yield a polynomial of degree 99! Such a polynomial could potentially fluctuate wildly between the tabulated points and thus not be a good representation of the actual function. $P(x)$ is more usually applied to subsets of the tabulated points. For instance, if the polynomial is applied to subsets of 3 points, it yields parabolic interpolation, which can be much superior to linear interpolation if the function has a number of minima and maxima.

One problem with parabolic 3-point interpolation is that when it comes to bracketing x, there is an ambiguity – does one bracket between x_1 and x_2 or between x_2 and x_3? This is one reason why cubic 4-point interpolation is more commonly practiced – the bracketing is then between x_2 and x_3 with no ambiguity. Such interpolation is also called Lagrangian 4-point interpolation. The Lagrangian 4-point interpolation equation can be written:

$$L(x) = \frac{(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_2)(x_1-x_3)(x_1-x_4)}\,y_1$$
$$+\frac{(x-x_1)(x-x_3)(x-x_4)}{(x_2-x_1)(x_2-x_3)(x_2-x_4)}\,y_2$$
$$+\frac{(x-x_1)(x-x_2)(x-x_4)}{(x_3-x_1)(x_3-x_2)(x_3-x_4)}\,y_3$$
$$+\frac{(x-x_1)(x-x_2)(x-x_3)}{(x_4-x_1)(x_4-x_2)(x_4-x_3)}\,y_4$$

which the student can easily verify by reference to the equation for $P(x)$.

To apply this interpolation equation, the user should first bracket x between x_j and x_{j+1} as before, but now identify x_j with x_2 in the above equation and x_{j+1} with x_3. It then follows that x_1 will be x_{j-1} and x_4 will be x_{j+2}. The perceptive student will see that this will lead to a problem at the endpoints. For instance, if x is situated between the first two tabulated points, x_{j-1} will be undefined. Likewise, if x is situated between the last two tabulated points, x_{j+2} will be undefined. Thus in those intervals, the user must either interpolate linearly, or, in the first case, use the polynomial that would be used for an x bracketed between the 2nd and 3rd tabulated points, and similarly for the last case.

Exercise 4.4

Write a C-function that will implement the 4-point Lagrangian interpolation formula above. For the endpoints, use the polynomial that would have been defined for the adjacent interval as described above. Modify the driver program in Exercise 4.2 (interpolation in a table of the wavelength and the index of refraction for BK-7 glass) to use this new C-function. Compare the results between the two programs.

Exercise 4.5

Write a C-function that will implement the 4-point Lagrangian interpolation formula above. For the endpoints, use the polynomial that would have been defined for the adjacent interval as described above. Modify the driver program in Exercise 4.3 (interpolation in a table of atmospheric pressure and the boiling point of water) to use this new C-function. Compare the results between the two programs.

We have only scratched the surface of the subject of interpolation. The subject of Extrapolation – finding a value for a function outside the range of the defined points – is much more dangerous. Interpolation schemes can be used for extrapolation, but only with great care!

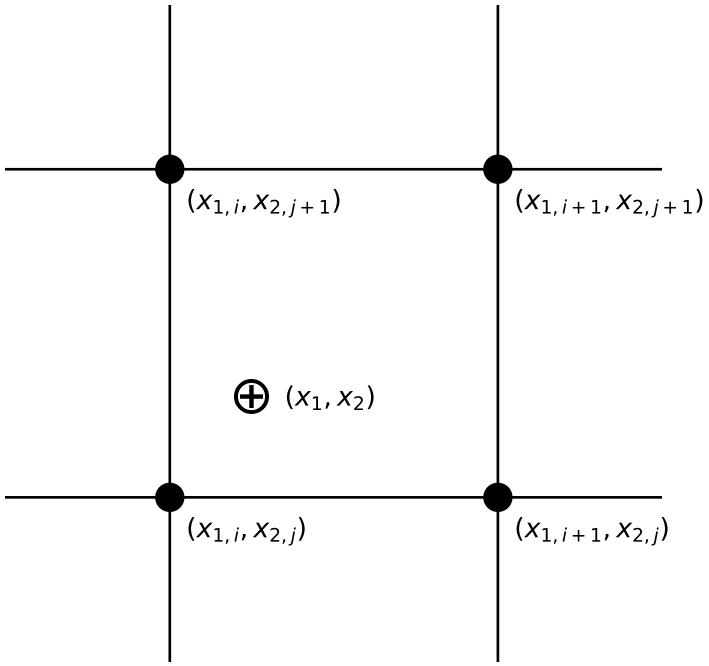
Before we leave the subject of interpolation, let us examine one further subject, that of efficiently bracketing x. If we have a set of x's (say in a vector `x[i]`) in numerical order that we must find interpolated values for, there is a simple time saving step that we can use, implemented in the following fragment of code (assume `x` is a vector of dimension `n`):

```
j = 1;
for(k=1;k<=n;k++) {
  for(i=j;i<N;i++) {
    if(xn[i] < x[k] && xn[i+1] >= x[k]) {
      j = i;
      break;
    }
  }
  here is your interpolation function code
}
```

Notice that the second `for` loop begins at `i=j` and not `i=1`; since the x_k's are in numerical order, if x_k is bracketed between `xn[j]` and `xn[j+1]`, there is no need to search beginning at `i=1` for x_{k+1} because it will either be bracketed between the same pair or later pairs (note that we are obviously also assuming that the `xn[j]`'s are in numerical order). This can save an enormous amount of time in a code that needs to interpolate in a large (say $N > 100$) table. Another useful trick is that of bisection. Further notes on bisection and other techniques for efficiently bracketing x can be found in *Numerical Recipes* (Press et al. 1992).

1.3 Interpolation in 2 dimensions

Some problems require interpolation in a two-dimensional grid of data. Let us suppose, for instance, that $y = y(x_1, x_2)$ where x_1 and x_2 are the two independent variables. The functional value y is tabulated on a Cartesian grid, and so the first thing the programmer must do is to bracket the desired point (x_1, x_2) in this grid (see figure below):



This bracketing can be done by bracketing in the two dimensions one at a time, using the technique discussed earlier.

The simplest interpolation technique in two dimensions is *bilinear interpolation*. If we define

$$\begin{aligned} y_1 &= y(x_{1,i}, x_{2,j}) \\ y_2 &= y(x_{1,i+1}, x_{2,j}) \\ y_3 &= y(x_{1,i+1}, x_{2,j+1}) \\ y_4 &= y(x_{1,i}, x_{2,j+1}) \end{aligned}$$

i.e., working our way counterclockwise around the above figure, then the interpolation formulae are:

$$\begin{aligned} t &= (x_1 - x_{1,i})/(x_{1,i+1} - x_{1,i}) \\ u &= (x_2 - x_{2,j})/(x_{2,j+1} - x_{2,j}) \end{aligned}$$

which make both t and u lie between 0 and 1. Then,

$$y(x_1, x_2) = (1-t)(1-u)y_1 + t(1-u)y_2 + tuy_3 + (1-t)uy_4$$

where (x_1, x_2) are the coordinates of the desired point.

Exercise 4.6

Exercise 4.6: A good example of the need to carry out interpolation in two dimensions is found in the calculation of partition functions. In certain plasma-physics contexts, it is necessary to calculate the partition functions of atoms and ions.
A partition function, U, is essentially an overall "statistical weight" for the atom, calculated by carrying out a weighted sum — weighted according to the populations of the levels — of the statistical weights for all of the energy levels in the atom.
At low temperatures, the partition function is simply the statistical weight of the ground level of the atom, but at higher temperatures, the partition function becomes larger, as the populations in the excited levels become significant.
The partition function is also a function of density, as at high densities in the plasma, the outermost energy levels are effectively "stripped off," resulting in a lowering of the ionization energy, usually denoted as ΔE. Thus, $U = U(T, \Delta E)$.
Because the calculation for a partition function can be very complex, they are usually calculated and tabulated so that users need not carry out the full calculation. The following table is a tabulation for the partition function for the hydrogen atom:

Table 1.5: The Hydrogen Partition Function

T (K)	ΔE = 0.10	ΔE = 0.50	ΔE = 1.00	ΔE = 2.00
3250	2.000	2.000	2.000	2.000
10083	2.000	2.000	2.000	2.000
14188	2.025	2.006	2.005	2.004
15643	2.068	2.016	2.012	2.009
17246	2.168	2.037	2.027	2.020
19014	2.384	2.080	2.058	2.040
20963	2.814	2.162	2.114	2.078
23111	3.610	2.308	2.213	2.142
25480	4.991	2.551	2.377	2.246

Write a function that will perform a 2-D interpolation in this table, and use it to determine the partition function for Hydrogen at the following points $(T, \Delta E)$: (16000, 0.25), (18500, 1.50), (19000, 0.15), (25023, 1.99).

Exercise 4.7

The study of plasmas is an important field of physics, and is essential in the understanding of the interiors of stars and the functioning of fusion reactors.
Hydrogen becomes increasingly ionized (hydrogen loses its electron when ionized) with increasing temperature, but the density of electrons, N_e, also plays an important role.

The following table gives the ratio of ionized hydrogen atoms to all forms of hydrogen (neutral + ionized) as a function of both T (in kelvins) and N_e (number of electrons per cubic centimeter).
Write a C-function and driver that will interpolate in this table. The driver should prompt the user for T (in kelvins) and N_e. Note that the table is tabulated in terms of $\log N_e$.

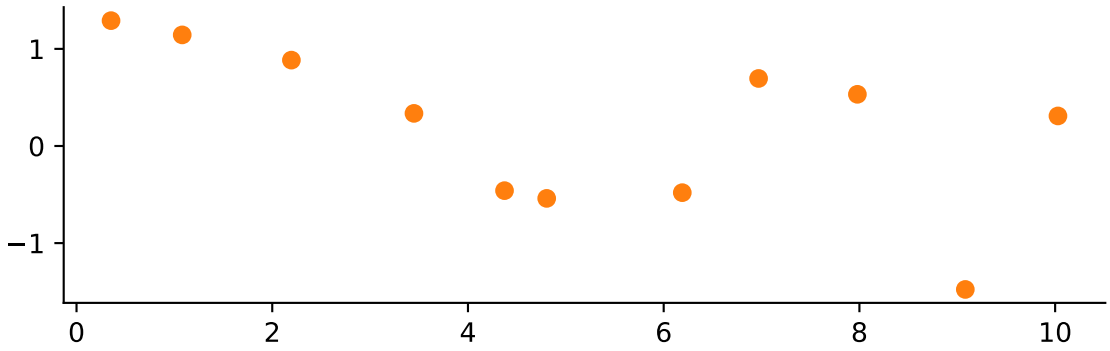
Table 1.6: Hydrogen Ionization: Ratio of Ionized to Total

T(K)	10.0	11.0	12.0	13.0	14.0	15.0	16.0	17.0
1000.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
2000.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
3000.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
4000.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
5000.0	0.0017	0.0002	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
6000.0	0.2980	0.0407	0.0042	0.0004	0.0000	0.0000	0.000	0.0000
7000.0	0.9582	0.6961	0.1864	0.0224	0.0023	0.0002	0.000	0.0000
8000.0	0.9979	0.9791	0.8241	0.3191	0.0448	0.0047	0.000	0.0000
9000.0	0.9998	0.9980	0.9804	0.8335	0.3335	0.0477	0.005	0.0005
10000.0	1.0000	0.9997	0.9971	0.9713	0.7719	0.2529	0.032	0.0034
11000.0	1.0000	0.9999	0.9994	0.9939	0.9425	0.6211	0.140	0.0161
12000.0	1.0000	1.0000	0.9998	0.9984	0.9841	0.8606	0.381	0.0581
13000.0	1.0000	1.0000	0.9999	0.9995	0.9948	0.9503	0.656	0.1607
14000.0	1.0000	1.0000	1.0000	0.9998	0.9980	0.9807	0.835	0.3373
15000.0	1.0000	1.0000	1.0000	0.9999	0.9992	0.9917	0.922	0.5448

1.4 Interpolation in Python

Let's say you have run an experiment and have some data.

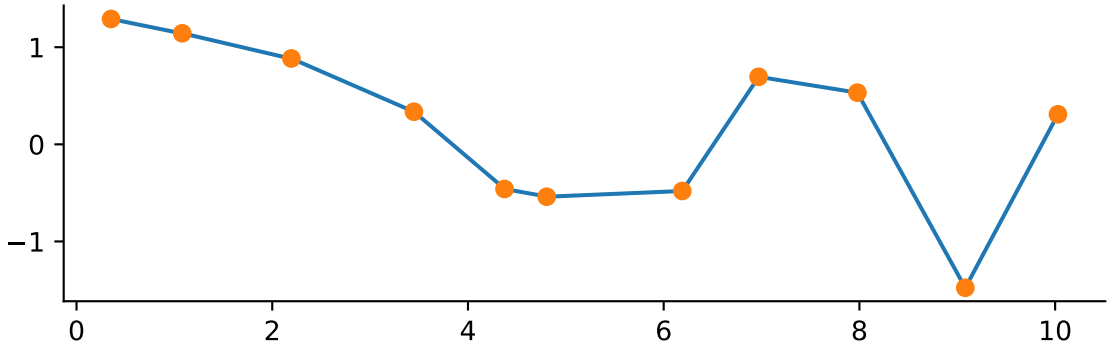
```
# Pretend you don't know the functional form of the data.
num_samples = 11
x_data = np.linspace(0, 10, num=num_samples) + np.random.normal(scale=.2, size=num_samples)
y_data = np.cos(-x_data**2 / 9.0)            + np.random.normal(scale=.2, size=num_samples)

fig, ax = plt.subplots()
ax.plot(x_data, y_data, 'o', color='C1')
```



Now you want to connect the dots … I mean interpolate your data.

Numpy has a method (`numpy.interp`) for simple linear interpolation. The method takes in an array of new `x`'s where you want to calculate the new `y`'s. The original `x` and `y` points are also needed.

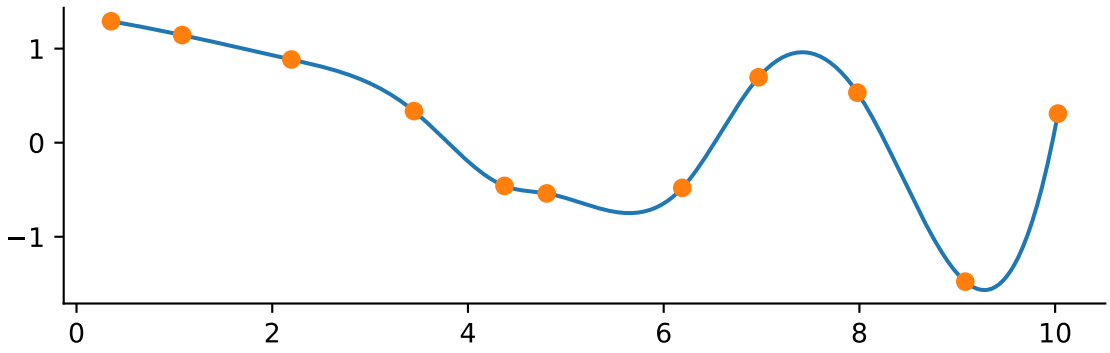
```
x_interp = np.linspace(x_data.min(), x_data.max(), num=1001)
y_interp = np.interp(x_interp, x_data, y_data)

fix, ax = plt.subplots()
ax.plot(x_interp, y_interp)
ax.plot(x_data, y_data, 'o')
```



If you want a fancier smooth line SciPy has got you covered! The `interpolate` subpackage contains many methods to suit your needs. These methods follow a different pattern than the Numpy approach. When using the SciPy `interpolate` methods you provide the original data points and are returned a function you can call to interpolate new values.

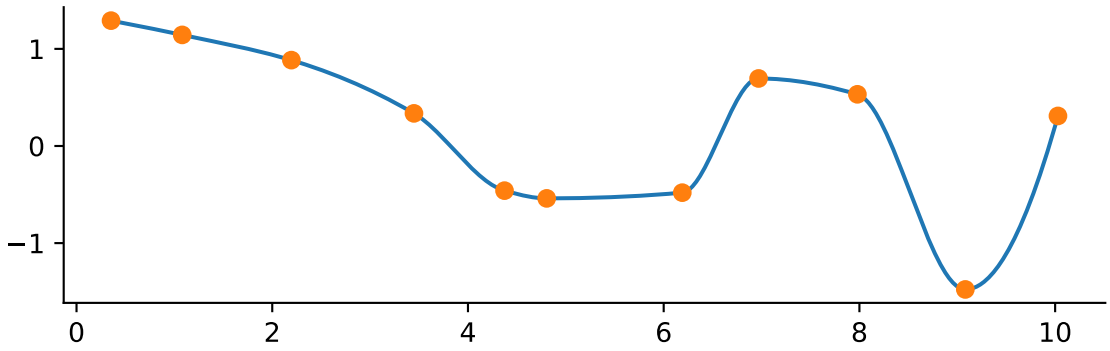
As an example consider the `CubicSpline` method

```
from scipy.interpolate import CubicSpline

f = CubicSpline(x_data, y_data)

y_interp = f(x_interp)

fix, ax = plt.subplots()
ax.plot(x_interp, y_interp)
ax.plot(x_data, y_data, 'o')
```



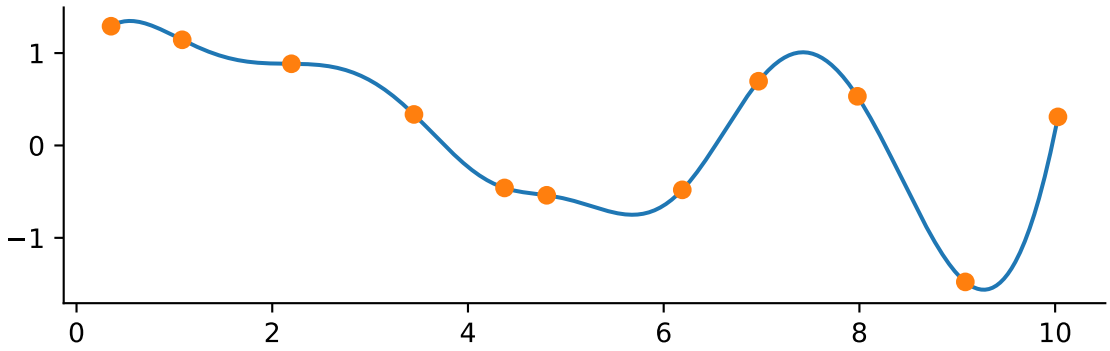
Quick and easy! There are other interpolation methods. For instance, maybe you are concerned with the line overshooting the data. An alternative is to use a so-called *monotone* cubic interpolant which attempts to preserve the local shape implied by the data. An example is the `PchipInterpolator`.

```
from scipy.interpolate import PchipInterpolator

f = PchipInterpolator(x_data, y_data)

y_interp = f(x_interp)

fix, ax = plt.subplots()
ax.plot(x_interp, y_interp)
ax.plot(x_data, y_data, 'o')
```



These cubic methods are typically fine for most application. There is also a generalized method for making interpolating splines of any degree but use with caution. You may introduce unexpected wiggles!

```
from scipy.interpolate import make_interp_spline

f = make_interp_spline(x_data, y_data, k=5)

y_interp = f(x_interp)

fix, ax = plt.subplots()
ax.plot(x_interp, y_interp)
ax.plot(x_data, y_data, 'o')
```



There are other methods for 2-D interpolation and smoothing as well. Have a look at the SciPy Interpolation User Guide for more details.

2 Finding the Root of a Function

The need to determine the roots of a function – i.e. where that function crosses the x-axis – is a recurring problem in practical and computational physics. For example, the density of matter in the interior of a star may be approximately represented by a type of function called a polytropic function. The edge or surface of the star occurs where the appropriate polytropic function goes to zero. There are many other examples. All of you know how to find analytically the roots of a parabola. For more complex functions, it may not be possible or easy to find the roots analytically, and so it is necessary to use numerical methods. The numerical method that is used depends upon whether or not it is possible to determine the derivative of the function. The function, for instance, may be given to us as a "black box" – it may be in tabular form, or given to us as a "C" function, or it may be known only through a recursion equation. In such a case, the appropriate method would be to more and more closely bracket the root until we know it to the precision that we desire. Alternately, we may be able to differentiate the function. In that case, we can use the Newton-Raphson method (or similar methods) to zero in on the root.

2.1 The Method of Bracketing and Bisection

The first thing that we need to know in root-finding is an approximate location of the root we are interested in. How can we determine this? The easiest way is to use a plotting program, such as `gnuplot`, `EXCEL`, etc., to plot the function. Another way is to calculate the function at a number of points. Then, unless the function is pathological in some way, we can say that a root exists in the interval (a, b) if $f(a)$ and $f(b)$ have opposite signs. An exception to this is a function with a singularity in this interval. Suppose c is in the interval (a, b), and the function is given by

$$f(x) = \frac{1}{x - c}$$

Then, $f(x)$ has a singularity at c and not a root. There are other pathological functions such as $\sin(1/x)$, which has infinitely many roots in the vicinity of $x = 0$. So, the user must be aware of such problems, as such pathologies will give problems to even the most clever programs.

Let us assume that our function is fairly free of pathologies and that we know that there is a root in the interval (a, b) and that we want to find it more exactly.

The simplest way to proceed is using the method of *bisection*. This method is straightforward. If we know that a root is in (a, b) because $f(a)$ has a different sign from $f(b)$, then evaluate the

function f at the midpoint of the interval, $(a+b)/2$, and examine its sign. Replace whichever limit (e.g. a or b) that has the same sign. Repeat the process until the interval is so small that we know the location of the root to the desired accuracy.

The "desired" accuracy must be within limits. As we have discussed before in class and lab, since computers use a fixed number of binary digits to represent floating-point numbers, there is a limit to the accuracy with which computers can represent numbers. To be precise, the smallest floating-point number which, when added to the floating-point number 1.0, produces a floating-point number different from 1.0, is termed the *machine accuracy*, ϵ_m. Using single precision, most machines have $\epsilon_m \approx 1 \times 10^{-8}$. Using double precision, $\epsilon_m \approx 1 \times 10^{-16}$. It is always a good idea to back off from these numbers, and thus setting ϵ_m to 10^{-6} and 10^{-14} respectively are practical limits. Sometimes even those limits are too optimistic. Let us suppose that our $(n-1)^{\text{th}}$ and n^{th} evaluations of the midpoint are x_{n-1} and x_n, and that the root was initially in the interval (a, b). We would then be justified in stopping our search if $|x_n - x_{n-1}| < \epsilon_m |b - a|$.

> **Exercise 5.1**
>
> The polynomial $f(x) = 5x^2 + 9x - 80$ has a root in the interval $(3, 4)$. Find the root to a precision of 10^{-6} using the *bisection* method. Verify your root analytically.

Plotting Functions in `gnuplot`

When finding roots, it is a good idea to plot the function to get an idea of the position of the roots. This can be done in `gnuplot`. To plot the function of Exercise 5.1, enter the following commands:

```
gnuplot> f(x) = 5*x**2 + 9*x - 80
gnuplot> plot f(x)
gnuplot> g(x) = 0
gnuplot> replot g(x)
```

Note that exponentiation in `gnuplot` is accomplished with "`**`". Notice in the plot that $f(x)$ has a root at about -5 and another between 3 and 4.

> **Exercise 5.2**
>
> Bessel functions $J_0(x)$, $J_1(x)$, etc. often turn up in mathematical physics, especially in the context of optics and diffraction. The file `comphys.c` has a canned version of the Bessel function $J_0(x)$. The function declaration (contained in the file `comphys.h`) for this function is

```
float bessj0(float x)
```

Modify your program from Exercise 5.1 to find the first two positive roots of $J_0(x)$.

Exercise 5.3

The viscosity of air (in micropascal seconds) is given to good accuracy by the following polynomial (valid between $T = 100\,\mathrm{K}$ and $600\,\mathrm{K}$):

$$\eta = 3.61111 \times 10^{-8} T^3 - 6.95238 \times 10^{-5} T^2 + 0.0805437\,T - 0.3$$

Use this polynomial to find the temperature T at which $\eta = 20.1\,\mu\mathrm{Pa}\cdot\mathrm{s}$.

Exercise 5.4

An atomic state decays with two time constants τ_1 and τ_2 and can be described by the equation:

$$N = \frac{N_0}{2}\left(e^{-t/\tau_1} + e^{-t/\tau_2}\right)$$

Find, by the method of bisection, the time at which $N = N_0/2$, i.e., the half-life of the state. Let $\tau_1 = 5.697$ and $\tau_2 = 9.446$. The program should first display (with `gnuplot`) the function (use $N_0 = 100.0$) and then prompt the user for an initial bracket. Hints: `gnuplot` does not recognize the variable `t`, so use `x` instead. You may wish to add the command `set xrange [0:15]` just before the `plot f(x)` command to give the plot a nice scaling. Use `exp(x)` for e^x in `gnuplot`.

2.2 The Newton-Raphson Method

If we can calculate the first derivative of our function, then we can use a speedier (although somewhat more dangerous) method called the Newton-Raphson method.

Let us suppose that we have a function as illustrated below, which has a root at $x = x_r$, and that we have a rough estimate for that root of $x = x_0$. Let us evaluate the derivative of this function at x_0, $f'(x_0)$. This derivative will be the slope of the tangent line to the function at the point $(x_0, f(x_0))$, and you can see that where it crosses the x-axis, x_1, is a much better estimate to the root of the function than x_0.

The process can be repeated to give an even better estimate, and so on. The equation of the first tangent line is (the student should verify):

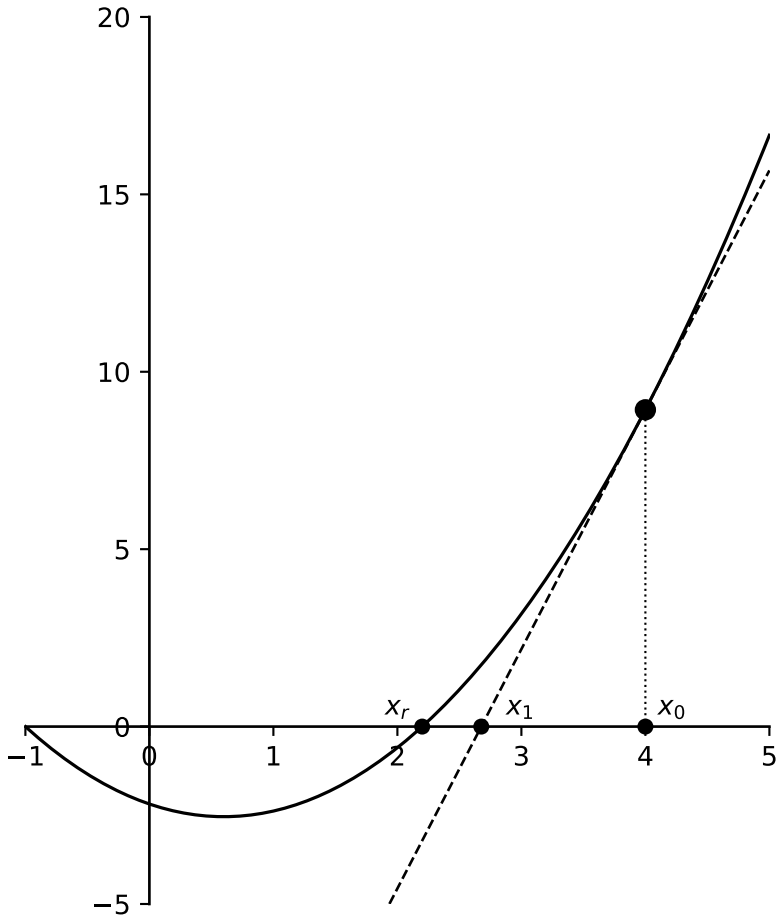
$$y = f'(x_0)(x - x_0) + f(x_0)$$

If we set $y = 0$ to find the root of this line, we get

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

From this, we can derive the Newton–Raphson formula for the n^{th} estimate of the root:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$



Exercise 5.5

Write a program to find the root for the polynomial of Exercise 5.1 using the Newton–Raphson method. Begin with an estimate $x_0 = 4$. See what happens if you use a different starting point. Explore both positive and negative values for the starting point.

Exercise 5.6

Write a program to solve the problem of Exercise 5.3 using the Newton–Raphson method. Begin with an estimate of 200 K.

Exercise 5.7

Write a program to solve the problem of Exercise 5.4 using the Newton–Raphson method.

An obvious danger to the Newton-Raphson method is that if there is a minimum or a maximum between the root and your starting point, the method will either not converge to the desired root at all, but to another one, or wander out to $\pm\infty$ at an ever accelerating pace. In addition, if you happen by bad luck to choose your beginning point at a maximum or minimum point, your program will never converge, but will blow up! Hence, as always, you should have a pretty good idea of the nature of your function before you attempt to find and "polish" roots.

Exercise 5.8

Use the fact that

$$\frac{dJ_0(x)}{dx} = -J_1(x)$$

to calculate the first two roots of $J_0(x)$.
The code for $J_1(x)$ is in the file `comphys.c`, and the function definition is

```
float bessj1(float x);
```

which is contained in `comphys.h`.

Exercise 5.9 An Iterative Problem

In ballistics, it is common to solve the equations of motion of a particle shot at an angle, θ, above a horizontal surface with an initial velocity v_0. The effects of air resistance play an important role that is often neglected in introductory or intermediate coursework. If the force of air resistance can be modeled as:

$$F_x = -km\dot{x} \tag{1}$$

$$F_y = -km\dot{y} \tag{2}$$

then the $x(t)$ and $y(t)$ solutions are:

$$x(t) = \frac{v_0 \cos\theta}{k}\left(1 - e^{-kt}\right) \tag{3}$$

$$y(t) = -\frac{gt}{k} + \frac{kv_0 \sin\theta + g}{k^2}\left(1 - e^{-kt}\right) \tag{4}$$

The time of flight of the projectile (the time elapsed before the projectile lands) is given as:

$$T = \frac{v_0 \sin\theta + g}{gk}\left(1 - e^{-kT}\right) \tag{5}$$

Notice that T appears on both sides of equation (5). This is known as a "transcendental equation" which can be solved numerically using iteration. Note that this equation collapses to the simple expression for T when resistance is neglected ($k = 0$):

$$T(k = 0) = \frac{2v_0 \sin\theta}{g} \tag{6}$$

The range of motion (the distance traveled by the projectile before landing) is:

$$R = x(t = T) \tag{7}$$

In the exercises below, use $\theta = 60^\circ$, $v_0 = 600\,\mathrm{m/s}$, and $|g| = 9.8\,\mathrm{m/s^2}$.

a. Solve for T (equation 5) numerically using iteration. In the first iteration, use the non-resistive expression for T (equation 6). Iterate until the solution converges to within a user-defined tolerance (error) – e.g., ΔT over successive iterations is less than the tolerance. Use a value of $k = 0.005$ for this part only.

b. Find the value of k that allows the projectile to travel 1500 meters ($R = 1500\,\mathrm{m}$).

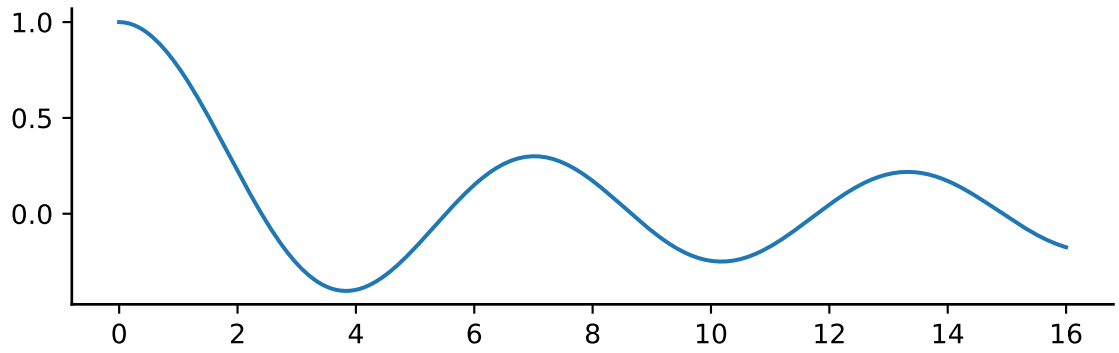
c. [**GRAD STUDENTS ONLY**] Plot y vs. x using gnuplot for k values of 0.0, 0.005, 0.01, 0.02, 0.04, and 0.08. Also, show analytically how to get equation (6) from equation (5) (*Hint: Use perturbation methods – i.e., expand* $1 - e^{-kT}$ *in a power series expansion, and keep only the first few terms*).

2.3 Root Finding in Python

In SciPy all of the root finding methods are in the `optimize` subpackage.

To show off how it is done consider this suspiciously familiar function

```
import numpy as np
from scipy.special import j0
import matplotlib.pyplot as plt
plt.rcParams.update({
  'figure.figsize': (7, 2),
  'axes.spines.top': False,
  'axes.spines.right': False
  })

x = np.linspace(0, 16, 400)
fig, ax = plt.subplots()
ax.plot(x, j0(x))
```

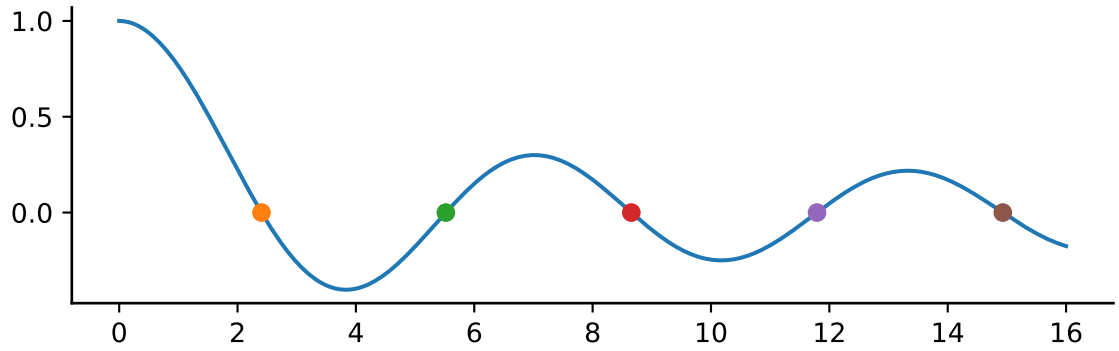


All of the root finding algorithems for scalar functions can be accessed via a single method `root_scalar`.

Let's use a different method for each root! We need to supply a bracket for where to limit our search and a method name.

```
from scipy.optimize import root_scalar

method = ['bisect' ,'brentq' ,'brenth' ,'ridder' ,'toms748']
bracket = [[2,4], [5,7], [8, 10], [11,13], [14, 16]]
results = []

for m, b in zip(method, bracket):
```

```
    results.append(root_scalar(j0, bracket=b, method=m))

fig, ax = plt.subplots()
ax.plot(x, j0(x))

for result in results:
    ax.plot(result.root, 0, 'o')
```



A call to `root_scaler` retuns a `RootResults` object which contains more information than only the root value.

```
results[0]
```

```
      converged: True
           flag: converged
 function_calls: 42
     iterations: 40
           root: 2.404825557694494
         method: bisect
```

For a full overview of the supported methods see the `root_scalar` documentation.

Optimization is a vast subject of which we will cover more in the next chapter.

3 Minimization and Maximization of Functions

The determination of the minima or the maxima of functions, including 1-D and multi-dimensional functions, is an important problem not only in physics but throughout quantitative science. For instance, in physics and astronomy, many quantities come to a maximum at an intermediate value of a certain parameter. In the spectra of stars, the hydrogen lines come to their maximum strength at a temperature of about 10,000 K. Stars with temperatures both lower and higher than this have relatively weak hydrogen lines. The investigation of this phenomenon can lead to insights into the physics of stars. The theory of orbits in classical mechanics leads to the concept of an *effective potential*. Depending upon the nature of the central force and the angular momentum of the particle, the effective potential can have a minimum at a certain radius. A stable circular orbit is then possible at that radius. Around black holes, this effective potential has not only a minimum but also a maximum. At the minimum, stable circular orbits are possible. Inside of the maximum, the particle is irretrievably drawn into the black hole. Knowledge of the location of this maximum is of obvious importance if you are orbiting around a black hole!

The determination of a minimum of a multi-dimensional function is related to the question of optimization. For instance, a common question to ask in physics is: "What combination of parameters yields the best fit of a model to the experimental or observational results?" "

That this is a problem related to minimization becomes clear when one considers a function $f(x_1, x_2, x_3, ...)$ which is equal to the sum of the squared differences between the theoretical model and the experimental results. The variables $x_1, x_2, x_3, ...$ are the model parameters, and we can consider them to define an n-dimensional space. The function f then takes on a value at each point in this n-dimensional space, and the problem reduces to finding the global minimum of f.

Finding the maximum of a function is equivalent to finding a minimum, because if $f(x)$ has a maximum at x_m, then $-f(x)$ has a minimum at the same point x_m. Thus, in what follows, we will consider algorithms to determine the *minimum* of a function only.

Let us first begin with 1-D functions, and then move on to multi-dimensional functions.

3.1 The Minimization of 1-D Functions

Analogous to Chapter 2, in which we considered how to find the root of a 1-D function, we can divide the problem into functions for which we can determine the first derivative and those for

which we cannot. Let us first consider the case for which we cannot determine the derivative, or are too lazy to do so.

3.1.1 Bracketing a Minimum in 1-D

In Section 2.1 we used the method of bisection to zero in on the root of a 1-D function. We can use a similar method here, but we must first define what we mean by "bracketing" a minimum. A minimum can be bracketed only by a triplet of points $a < b < c$ such that $f(b)$ is less than both $f(a)$ and $f(c)$. Only then do we know that the function has a minimum in the interval (a, c) (see Figure 3.1). To zero in on the minimum, we choose a new point x, which is either between a and b, or between b and c. Let us suppose x is between b and c. We evaluate $f(x)$. If $f(b) < f(x)$, then we replace c with x. If, however, $f(b) > f(x)$, then the new bracketing triplet is (b, x, c).
We continue until we know the position of the minimum to a precision consistent with both our needs and the machine precision.

How do we choose x? There are a number of ways to choose x, but a straightforward way is to take x halfway through the larger of the two intervals (a, b) and (b, c).

We use the following algorithm:

- **If** the interval (a, b) is larger than (b, c):
  Take $x = (a + b)/2$ and find $f(x)$.
  - If $f(b) < f(x)$, the new interval is (x, b, c).
  - If $f(b) > f(x)$, the new interval is (a, x, b).
- **If** the interval (b, c) is larger than (a, b):
  Take $x = (b + c)/2$ and find $f(x)$.
  - If $f(b) < f(x)$, the new interval is (a, b, x).
  - If $f(b) > f(x)$, the new interval is (b, x, c).

Exercise 6.1

Using the above algorithm, find the minimum of the polynomial

$$f(x) = x^3 - 24x^2 + 6x + 15$$

using the beginning bracket $(11, 15, 20)$. Print out intermediate results so that you know how many iterations the algorithm went through.

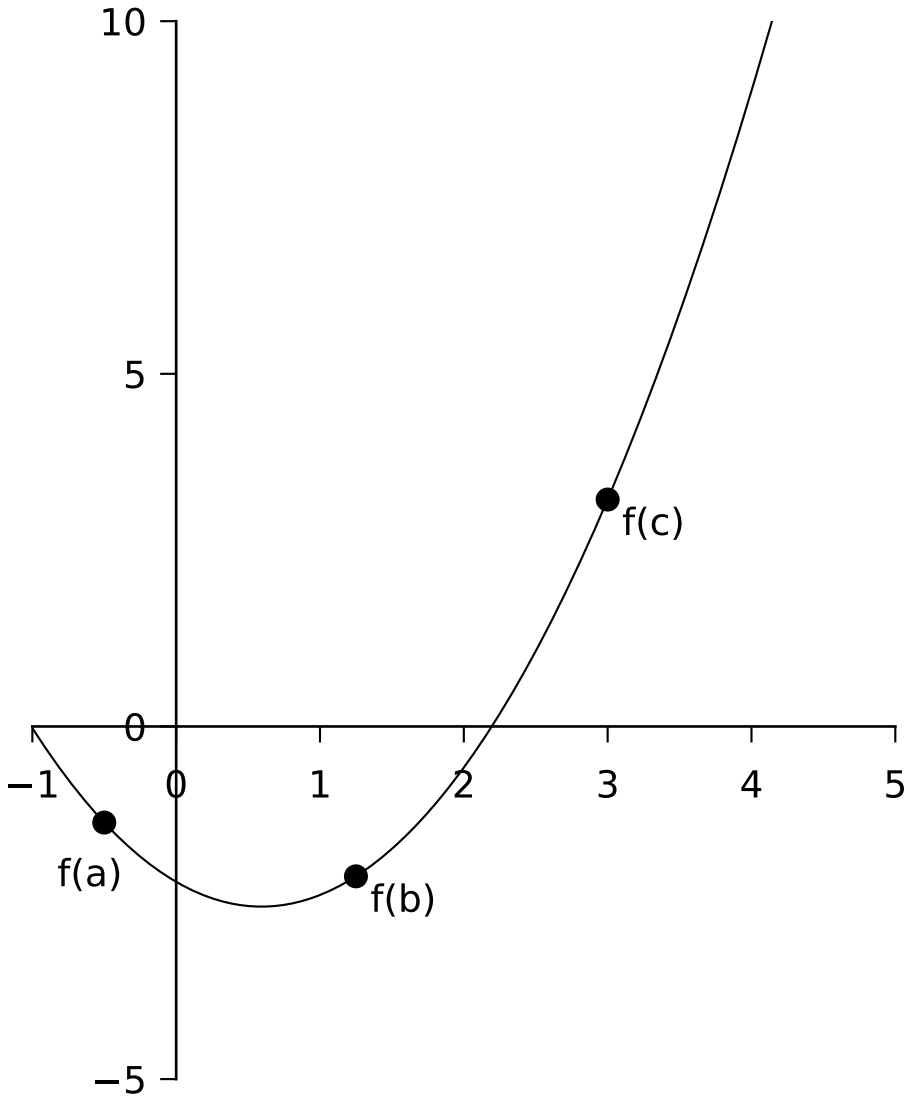


Figure 3.1: A parabola we want to minimize

Exercise 6.2

Legendre polynomials appear in the solutions to Laplace's equation in electrostatics, particularly for systems with spherical symmetries. For example, the general solution for the potential in 3-space, arrived at via separation of variables, takes the form:

$$V(r,\theta) = \sum_{n=0}^{\infty} \left(A_n r^n + \frac{B_n}{r^{n+1}} \right) P_n(\cos\theta)$$

where the coefficients A_n and B_n are determined by boundary conditions, and r and θ are the radial distance and polar angle, respectively, in spherical coordinates.
P_n is the Legendre polynomial for the nth harmonic, defined via the Rodrigues Formula as:

$$P_n(x) = \frac{1}{2^n n!} \left(\frac{d}{dx} \right)^n (x^2 - 1)$$

Write a C program to find the minimum of the $n = 3$ polynomial

$$P_3(x) = \frac{1}{2}(5x^3 - 3x)$$

using the initial bracket $(0.0, 0.5, 1.0)$. Print out intermediate results so that you know how many iterations the algorithm went through.

Exercise 6.3

Recall that the Bessel function $J_0(x)$ is included in `comphys.c` as:

```
float bessj0(float x)
```

Write a program that first invokes gnuplot to plot $J_0(x)$ between $x = 0$ and $x = 15$, and then prompts the user to enter a triplet (`a, b, c`) to bracket one of the displayed minima of $J_0(x)$. The program should then use bracketing and bisection to find the x value of that minimum to a tolerance of 1×10^{-5}. Hint: You can get gnuplot to plot $J_0(x)$ with the command:

```
gnuplot> plot besj0(x)
```

3.1.2 Bracketing a Minimum in 1-D with Derivatives

In the previous section, we cornered the minimum of our function by successively reducing the size of the bracketing interval until we knew the location of the minimum to the desired precision. We did this by arbitrarily selecting a new evaluation point in the center of the *largest* interval, (a, b) or (b, c). Now, this is inefficient, as the minimum may not be in the largest interval, and so our algorithm actually wasted nearly 50% of its steps. If we can calculate the derivative of the function, we can use this information to determine which interval the minimum is actually in and thus avoid wasting steps. Furthermore, if we retain the derivatives of the function at all bracketing points a, b, and c, then we can utilize linear *extrapolation* to predict where the derivative of the function goes to zero — i.e., the location of the minimum. This scheme should zero in on the minimum with far fewer steps than the previous algorithm.

This is how we proceed:

1. Evaluate the function and its derivative at each of the bracketing points a, b, and c.
2. Examine $f'(b)$:
   - If $f'(b) > 0$, the minimum is in the interval (a, b).
   - If $f'(b) < 0$, the minimum is in (b, c). See Figure 3.2.
3. If $f'(b) < 0$, use $f'(a)$ and $f'(b)$ to extrapolate the derivative to 0.
   The straight line that passes through the points $(a, f'(a))$ and $(b, f'(b))$ is given by:

   $$y' = f'(a) + \frac{f'(b) - f'(a)}{b - a}(x - a)$$

   (The student should verify this). This can be used to extrapolate to $y' = 0$ by setting $y' = 0$ and solving for x, yielding:

   $$x = a - \frac{f'(a)(b - a)}{f'(b) - f'(a)}$$

   We then calculate $f(x)$ and $f'(x)$, and the new interval is (b, x, c).
4. If $f'(b) > 0$, use $f'(b)$ and $f'(c)$ to extrapolate the derivative to 0.
   The straight line that passes through the points $(b, f'(b))$ and $(c, f'(c))$ is:

   $$y' = f'(b) + \frac{f'(c) - f'(b)}{c - b}(x - b)$$

   (The student should verify this.)

   This can be used to extrapolate to $y' = 0$ by setting $y' = 0$ and solving for x, yielding:

$$x = b - \frac{f'(b)(c-b)}{f'(c) - f'(b)}$$

We then calculate $f(x)$ and $f'(x)$, and the new interval is (a, x, b).

5. Go back to step (2) and continue until the minimum is isolated to the desired accuracy.

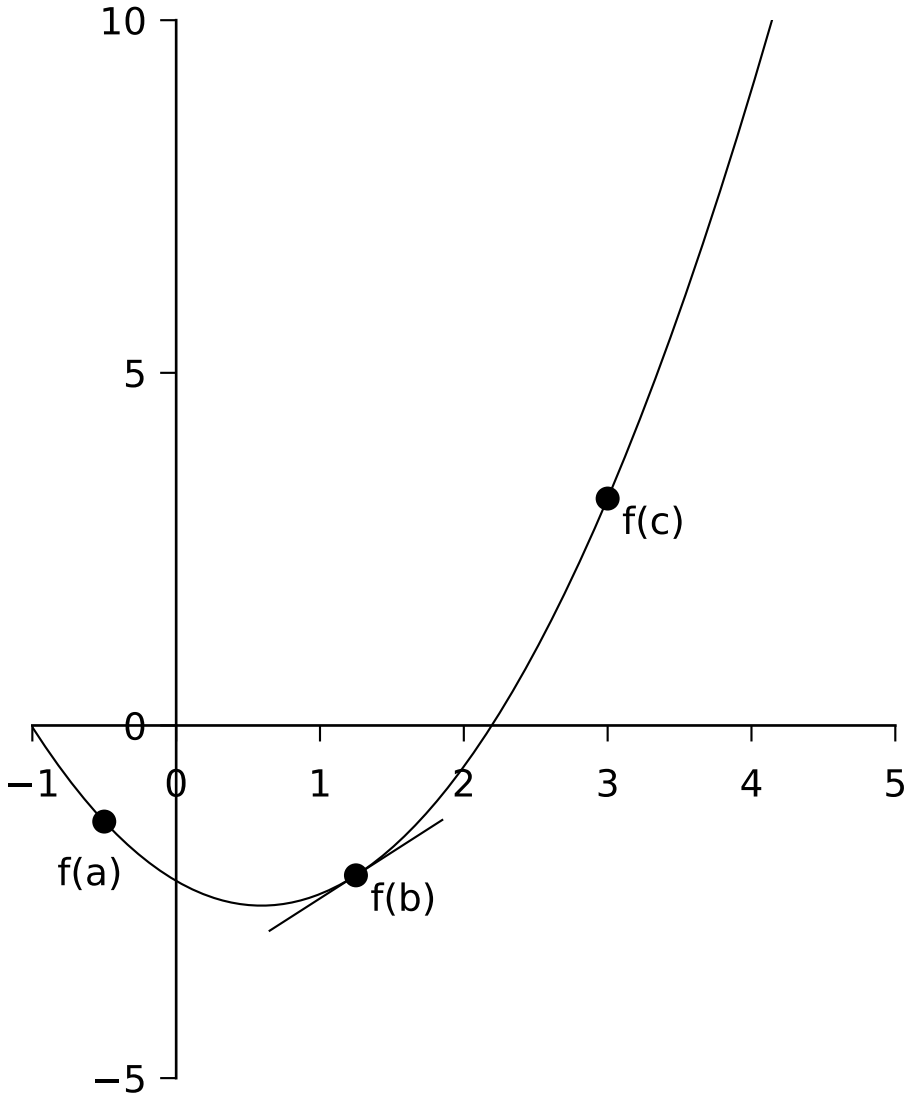


Figure 3.2: A parabola with derivative information

Exercise 6.4

Modify your program for **Exercise 6.1** to include derivative information.
Does the modified program take fewer steps to converge on the minimum?
Does it do a better job at finding the minimum? How do you know?

Exercise 6.5

Re-do **Exercise 6.2** using derivative information.

Exercise 6.6

Re-do **Exercise 6.3** using derivative information. Recall that:

$$\frac{dJ_0(x)}{dx} = -J_1(x)$$

Exercise 6.7 — The Derivation of Wien's Law

Wien's law is a property of the blackbody spectrum and can be written as:

$$\lambda_{\max}(\text{Å}) = \frac{2.9 \times 10^7 \ \text{ÅK}}{T(\text{K})}$$

meaning that the wavelength of the maximum of the blackbody (Planck) spectrum is inversely proportional to the temperature.
The flux emitted by a blackbody (in units of erg s ^{1} cm ^{2} Å ^{1}) is given by:

$$B(\lambda) = \frac{1.19106 \times 10^{27}}{\lambda^5 \left(e^{1.43879 \times 10^8/(\lambda T)} - 1\right)}$$

This function is coded in the C function `planck` found in **comphys.c**. The function definition is:

```
float planck(float wave, float T);
```

where wave is the wavelength in Ångströms and T is the temperature in Kelvin.
Use the method of this section to determine the wavelength of the maximum of $B(\lambda)$ for the following temperatures: $T = 2000\text{K}, 5000\text{K}, 8000\text{K}, 11000\text{K}, 14000\text{K}$ and use gnuplot to plot these wavelengths against $1/T$ and show that the slope of the line is $2.9 \times 10^7, \text{ÅK}$. To learn how to get gnuplot to do this for you, see the "aside" below. *Hint*: keep in mind that $B(\lambda)$ has a *maximum*, so determine the minimum of $-B(\lambda)$. Another question to consider — how will you find the initial values of (a, b, c) which bracket the minimum of $-B(\lambda)$? This is an important question — you will need to devise a C function that will come up with these three numbers.

3.1.3 An Aside - How to get Gnuplot to fit a line to data

Let us suppose that we have some data in the file `data.dat` and we want to get gnuplot to fit a regression line to those data. This is how to do it manually – the same commands can be used in "response" mode. Carry out the following steps:

```
gnuplot > plot "data.dat" using 1:2
gnuplot > f(x) = a + b*x
gnuplot > fit f(x) "data.dat" using 1:2 via a,b
gnuplot > replot f(x)
```

Gnuplot will calculate the regression line, and then display the values of the coefficients a and b along with associated errors.

How does one plot, say, the reciprocal of the first column of `data.dat` against the square of the second column? Use the following gnuplot command:

```
gnuplot > plot "data.dat" using (1/$1):($2*$2)
```

3.2 The Minimization of Multi-dimensional Functions

The subject of the minimization of multidimensional functions, i.e. functions of the form $y = f(x_1, x_2, x_3, ...)$ is a huge subject, and we will only be able to scratch the surface.
We begin with the so-called *downhill simplex method*.

3.2.1 The Multidimensional Downhill Simplex Method

A *simplex* is a geometrical figure consisting, in N dimensions, of $N+1$ vertices and all their interconnecting line segments, faces, etc. For $N = 2$, the simplex is a triangle; for $N = 3$, it is a tetrahedron, etc. In 1-D minimization, it is possible to bracket the minimum, but in multidimensional space, this is impossible. What we can do, however, is to *surround* the minimum with a *simplex* and then shrink that *simplex* until we know the position of the minimum to the desired precision. The way the *simplex* method works is as follows. Let us suppose we begin with a rough estimate of the position of the minimum in N-dimensional space. This point $(x_1, x_2, \ldots, x_N)$ makes up one of the vertices of the simplex. We then supply the remaining vertices of the simplex by adding a small value to each of the coordinates, thus:

$$
\begin{aligned}
&(x_1 + \Delta x_1,\ x_2,\ \ldots,\ x_N),\\
&(x_1,\ x_2 + \Delta x_2,\ \ldots,\ x_N),\\
&\ldots,\\
&(x_1,\ x_2,\ \ldots,\ x_N + \Delta x_N)
\end{aligned}
$$

The Δx_j's can all be the same, or they can be different, according to the needs of the problem. These $N+1$ points define the initial simplex.

The downhill simplex method now takes a number of steps in the "downhill" direction in an attempt to find the minimum of the function. Most of these steps are in the form of reflections, in which the "highest" vertex of the simplex (i.e. the vertex at which the function evaluates to the highest value) is reflected through the opposite face of the simplex to a lower point. Other steps (see Figure 3.3) involve reflection and expansion, or contraction. The goal is to surround the minimum and then contract on it until we know the minimum to the required precision. If you could watch the simplex in action, it would appear somewhat like an amoeba, first pursuing its prey and then engulfing it.

The C function `amoeba` — from *Numerical Recipes* (Press et al. 1992) — implements the downhill simplex method. Since this algorithm may be a bit difficult for you to code yourself, we have given you this routine in `comphys.c`, and you will explore a number of applications of this routine.

The function definition for amoeba (contained in `comphys.h`) is:

```
void amoeba(float **p, float y[], int ndim, float ftol,
            float (*funk)(float []), int *nfunk)
```

Here are the specific instructions for the use of `amoeba`, taken from *Numerical Recipes* (Press et al. 1992):

> Multidimensional minimization of the function `funk(x)` where `x[1...ndim]` is a vector in `ndim` dimensions, by the downhill simplex method of Nelder and Mead. The

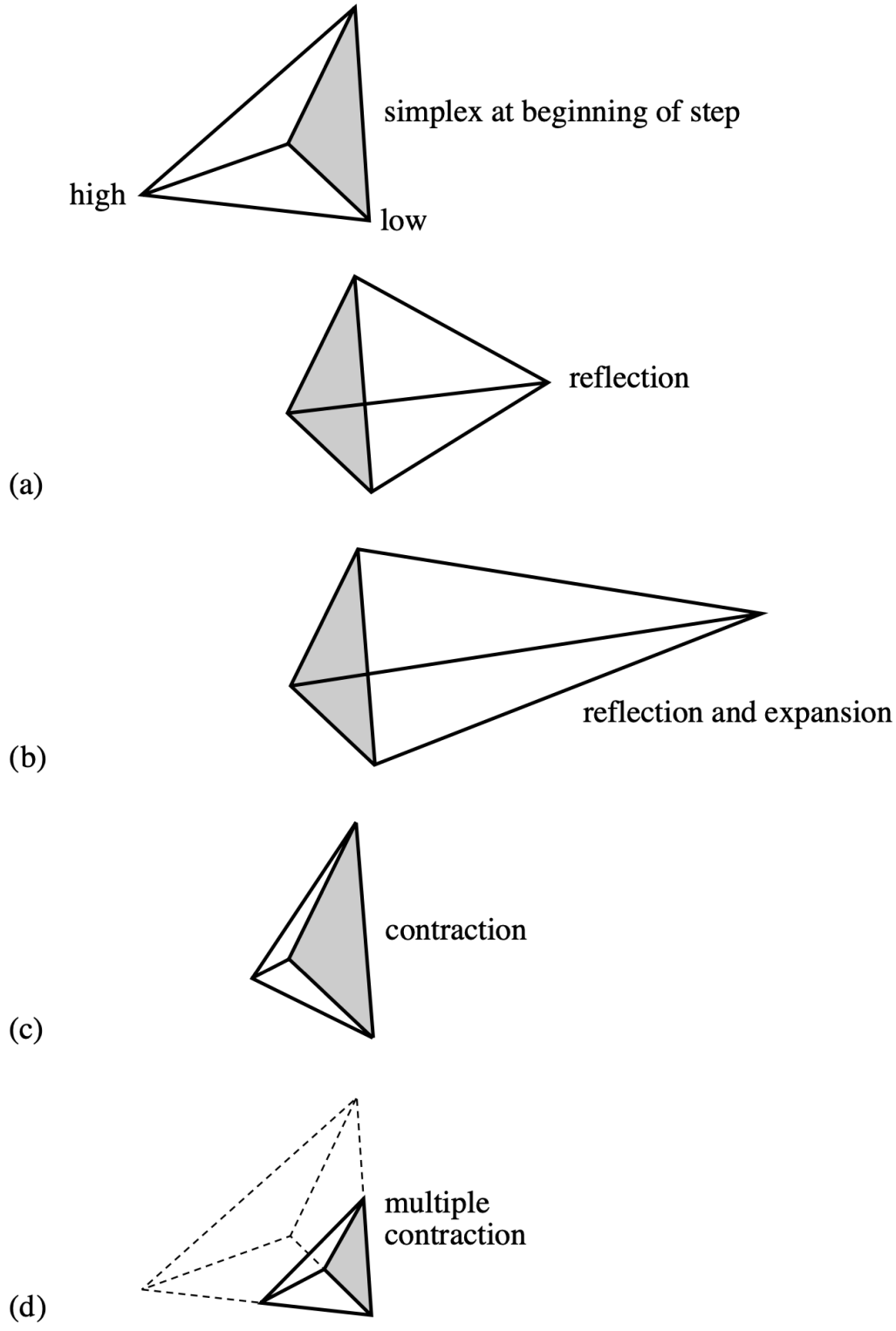


Figure 3.3: Possible outcomes for a step in the downhill simplex method

matrix `p[1..ndim+1][1..ndim]` is input. Its `ndim + 1` rows are `ndim`-dimensional vectors which are the vertices of the starting simplex. Also input is the vector `y[1..ndim+1]`, whose components must be preinitialized to the values of `funk` evaluated at the `ndim + 1` vertices (rows) of `p`; and `ftol`, the fractional convergence tolerance to be achieved in the function value. On output, `p` and `y` will have been reset to `ndim + 1` new points all within `ftol` of a minimum function value, and `nfunk` gives the number of function evaluations taken.

This certainly sounds complicated, but it is fairly easy to use in practice.
For an example, let us look at a C-program that will implement `amoeba` to solve a simple minimization problem where we find the minimum of the function

$$f(x, y) = x^2 + (y - 1)^2 + 1.0$$

```
#include <stdio.h>
#include <math.h>
#include "comphys.h"
float f(float x[]);

int main()
{
  float **p,*y;
  int nfunk,i;
  float x0,y0;

  /* Allocate memory for the p matrix and the y vector */

  p = matrix(1,3,1,2);
  y = vector(1,3);

  /* initial point */

  x0 = 5.0;
  y0 = 5.0;

  /* define the vertices of the simplex - in this case a triangle */

  p[1][1] = x0;
  p[1][2] = y0;
  p[2][1] = x0 + 1.0;
  p[2][2] = y0;
  p[3][1] = x0;
```

```
  p[3][2] = y0 + 1.0;

  /* initiate y[i] */

  for(i=1;i<=3;i++) y[i] = f(p[i]);

  /* invoke amoeba with ftol = 0.001 */

  amoeba(p,y,2,0.001,f,&nfunk);

  /* the best value for the minimum is obtained by averaging the final
  p's. */

  x0 = (p[1][1]+p[2][1]+p[3][1])/3.0;
  y0 = (p[1][2]+p[2][2]+p[3][2])/3.0;

  printf("\nThe minimum is found at x0 = %f, y0 = %f\n",x0,y0);
  printf("The number of function evaluations is %d\n",nfunk);
  return(0);
}

/* here is our function that we want to miminize */

float f(float x[])
{
  float z;
  z = x[1]*x[1] + (x[2]-1.0)*(x[2]-1.0) + 1.0;
  /* print out intermediate values so we can see what is happening */
  printf("\nx[1] = %f x[2] = %f z = %f",x[1],x[2],z);
  return(z);
}
```

The initial simplex looks like this:

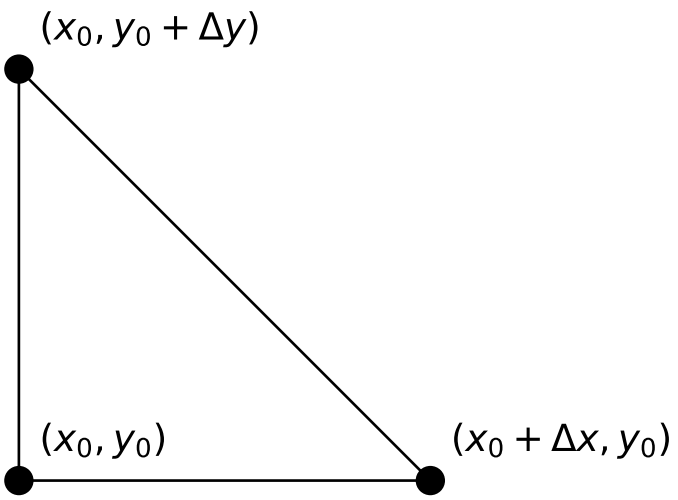
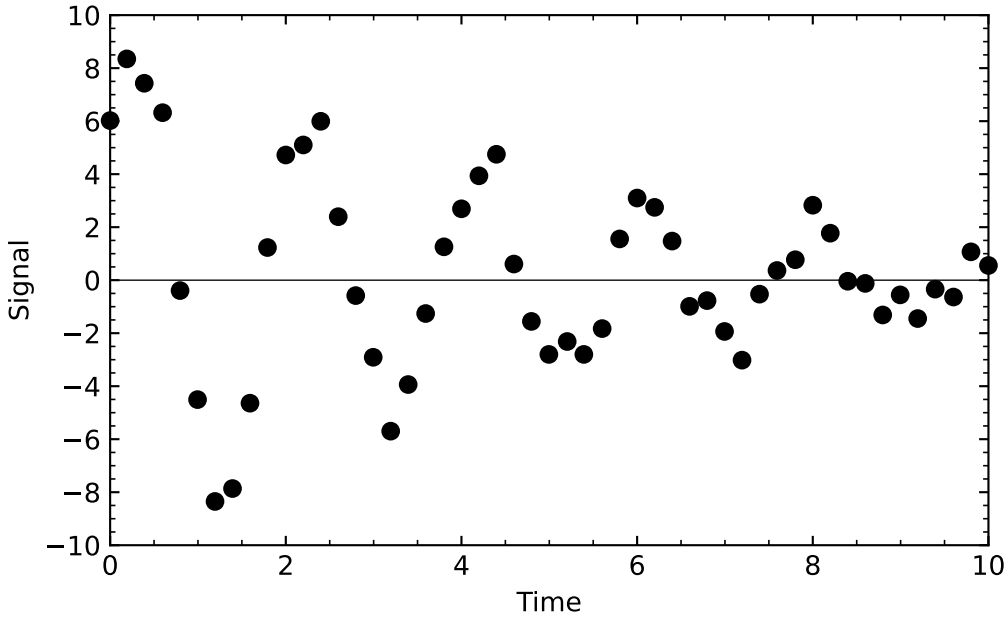


Figure 3.4: A 3D simplex creates a triangle to encapulate the minima

In this case, both Δx and Δy are 1.

> **Exercise 6.8**
>
> In this exercise, we explore a very useful application of function minimization — optimizing a fit to a model. Consider the following data, which come from an exponentially decaying oscillating system:



The data are slightly noisy, but we can model these data with the equation

$$f(t) = Ae^{-\gamma t}\sin(Bt + C)$$

We can consider the parameters A, γ, B, and C as variables, and ask our simplex algorithm `amoeba` to roam around in this four-dimensional space seeking the best fit (i.e., the fit with the smallest residual or χ^2) between this model and the data in the above figure. These data are contained in the file `data/decay.out`. The first thing you must do is to read these data into your program. Consider the following code fragment, which can serve as the beginning of your program:

```
#include <stdio.h>
#include <math.h>
#include "comphys.h"
float *t,*yi;
int k;
float f(float x[]);

int main()
{
  float **p,*y;
  int nfunk,i;
  float A,gam,B,C;
  FILE *in;

  p = matrix(1,5,1,4);
  y = vector(1,5);
  t = vector(0,200);
  yi = vector(0,200);

  if((in = fopen("decay.out","r")) == NULL) {
    printf("\nCannot find input file\n");
    exit(1);
  }

  k = 0;
  while(fscanf(in,"%f %f",&t[k],&yi[k]) != EOF) k++;

  fclose(in);

  // ...

}
```

What this code fragment does is to define two external vectors, `*t` and `*yi`, and then allocate space for them in the lines:

```
t = vector(0,200);
yi = vector(0,200);
```

The next lines in the code open the file `decay.out` and then read the data from `decay.out` into the vectors `t` and `yi`. The file handle `in` is then closed.

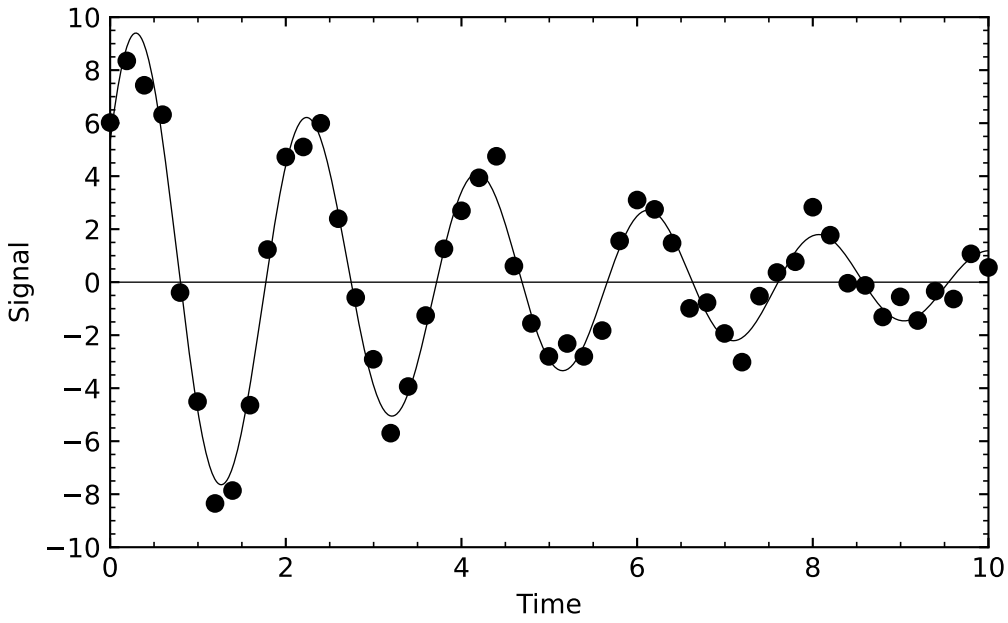
The reason why we used external vectors was so that our function `f`, which is used by `amoeba`, would have access to these vectors. The function `f` calculates the residual or χ^2 between the model and the data in `decay.out` using:

$$\chi^2 = \sum_{i=0}^{k} (y_i - f(t_i))^2$$

This value for χ^2 is then returned. This is accomplished in the code:

```
float f(float x[])
{
  float y1,chi2;
  int i;

  chi2 = 0.0;
  for(i=0;i<k;i++) {
    y1 = x[1]*exp(-x[2]*t[i])*sin(x[3]*t[i] + x[4]);
    chi2 += (y1 - yi[i])*(y1 - yi[i]);
  }
  printf("\nchi2 = %f",chi2);
  return(chi2);
}
```

What remains for you to do is to figure out reasonable starting values for A, γ, B, and C, use them to initialize the components of p, and then to invoke amoeba. The next figure shows a plot of what I got.



Exercise 6.9

The restricted 3-body problem of classical mechanics leads to a non-dimensionalized potential energy surface given by the equation:

$$V(x,y) = \left\{ \frac{\xi_2}{[(x-\xi_1)^2+y^2]^{1/2}} - \frac{\xi_1}{[(x-\xi_2)^2+y^2]^{1/2}} - \frac{1}{2}(x^2+y^2) \right\}$$

where $0 < \xi_1 < 1$ and $\xi_2 = \xi_1 - 1$. It turns out that this potential surface has 5 extrema, three minima (L_1, L_2, and L_3), which all lie along the x-axis, and two maxima (L_4 and L_5). For small values of ξ_1, L_4 will be in the second quadrant and approximately located at a point 1 unit away from the origin along a line inclined 60° to the $-x$ axis. As ξ_1 increases, L_4 moves closer to the y-axis and lies on the y-axis for $\xi_1 = 0.5$. L_5 is located similarly in the third quadrant. Write a program that will prompt the user for ξ_1, check that $\xi_1 \leq 0.50$, and then find the coordinates of L_4 using the simplex method.

3.2.2 Powell's Method

The downhill simplex method has the virtue that it is easy to implement, and often will do the job as well as other much more sophisticated methods. A problem with the simplex method is that it uses more function evaluations than other multidimensional minimization techniques, and if evaluating your function is computationally expensive, then the simplex method can be

very slow. So, if you are dealing with a complicated, computationally expensive function, it is worthwhile to investigate other methods.

One straightforward way to minimize a multidimensional function is to use the methods of Section 3.1 to minimize first along one direction, then along the next and so on until you spiral into the minimum — i.e., if $y = f(x_1, x_2, x_3, ...)$, hold $x_2, x_3, ...$ constant and minimize f along x_1, then hold $x_1, x_3, ...$ constant and minimize along x_2, and so on, cycling through the directions until you know the minimum to the desired precision. This is actually a very good algorithm to follow in most situations, especially if the minimum of your function is nice and symmetric, as is quite often the case. However, the method becomes very inefficient if the minimum is found at the bottom of a long, narrow valley. Then the method will have to cycle through the directions many, many times, bouncing against the sides of this valley to get to the minimum.

The key is to abandon the coordinate directions $x_1, x_2, x_3, ...$ and instead use a new basis set of "conjugate directions", with the idea that the first of these directions will be along the direction of maximum descent — i.e., along the direction of that long narrow valley. The algorithm moves as far as it can along this direction; it then abandons that direction and finds the next direction of maximum descent, and so on, until the actual function minimum is found. This is the basis of Powell's method, which is implemented in the C-function `powell` found in `comphys.c`. The function definition is

```
void powell(float p[], float **xi, int n, float ftol,
int *iter, float *fret, float (*func)(float []))
```

The *Numerical Recipes* instructions for this function are as follows:

> Minimization of a function `func` of `n` variables. Input consists of an initial starting point `p[1...n]`; an initial matrix `xi[1..n][1..n]`, whose columns contain the initial set of directions (usually the n unit vectors); and `ftol`, the fractional tolerance in the function value such that failure to decrease by more than this amount on one iteration signals doneness. On output, `p` is set to the best point found, `xi` is the then-current direction set, `fret` is the returned function value at `p`, and `iter` is the number of iterations taken.

Let us use Powell's method to minimize the function $f(x, y) = x^2 + (y-1)^2 + 1.0$ of the previous section. The following program does the job:

```
#include <stdio.h>
#include <math.h>
#include "comphys.h"
float f(float x[]);

int main()
```

```
{
  float *p,**xi,fret;
  int iter=0;
  int i,j;
  float x0,y0;

  /* Allocate memory for the p vector and the xi matrix */
  p = vector(1,2);
  xi = matrix(1,2,1,2);

  /* initial point */
  x0 = 5.0;
  y0 = 5.0;

  p[1] = x0;
  p[2] = y0;

  /* Initial directions */
  for(i=1;i<=2;i++) {
    for(j=1;j<=2;j++) {
        if(i == j) xi[i][j] = 1.0;
        else xi[i][j] = 0.0;
    }
  }

  /* invoke powell with ftol = 0.001 */

  powell(p,xi,2,0.001,&iter,&fret,f);

  printf("\nThe minimum is found at x = %f, y = %f",p[1],p[2]);
  printf("\nThe value of f at the minimum is %f\n",fret);
  return(0);
}

/* here is our function that we want to minimize */

float f(float x[])
{
  char tmp[10];
  float z;
  z = x[1]*x[1] + (x[2]-1.0)*(x[2]-1.0) + 1.0;
  /* print out intermediate values so we can see what is happening */
```

```
  printf("\nx[1] = %f x[2] = %f z = %f",x[1],x[2],z);
  return(z);
}
```

> **Exercise 6.10**
>
> Repeat Exercise 6.8 using Powell's method.

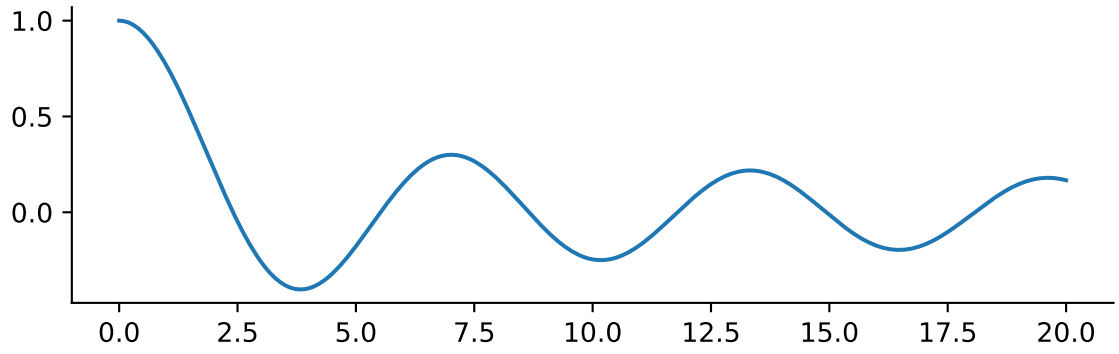
> **Exercise 6.11**
>
> Repeat Exercise 6.9 using Powell's method.

3.3 Optimization in Python

Optimization is a vast subject. SciPy has got you covered with the optimize subpackages, which we previously met in Chapter 2 for all of our root finding needs.

For scalar functions the `minimize_scalar` method is used. As an example we will return to our favorite wiggle

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import j0
from scipy.optimize import minimize_scalar
plt.rcParams.update({
  'figure.figsize': (7, 2),
  'axes.spines.top': False,
  'axes.spines.right': False
  })

x = np.linspace(0, 20, 400)
plt.plot(x, j0(x))
```



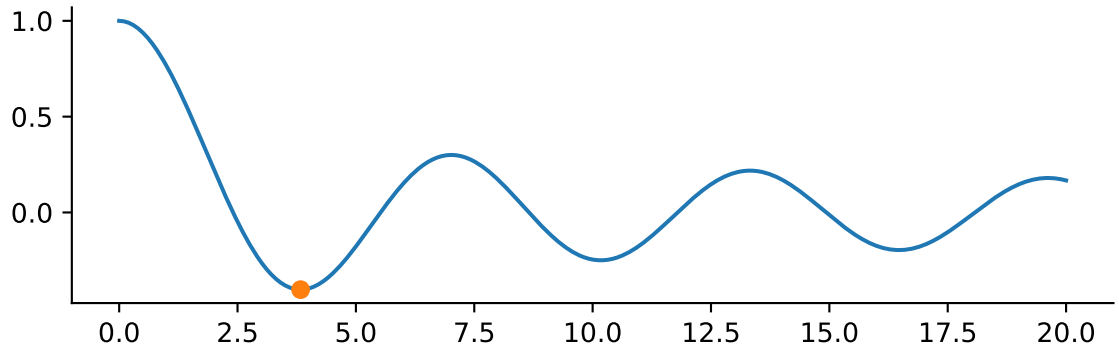
The call to `minimize_scalar` returns an object with other information about the result.

```
result = minimize_scalar(j0)
result
```

```
 message:
          Optimization terminated successfully;
          The returned value satisfies the termination criteria
          (using xtol = 1.48e-08 )
 success: True
     fun: -0.4027593957025531
       x: 3.8317059554863437
     nit: 9
    nfev: 13
```

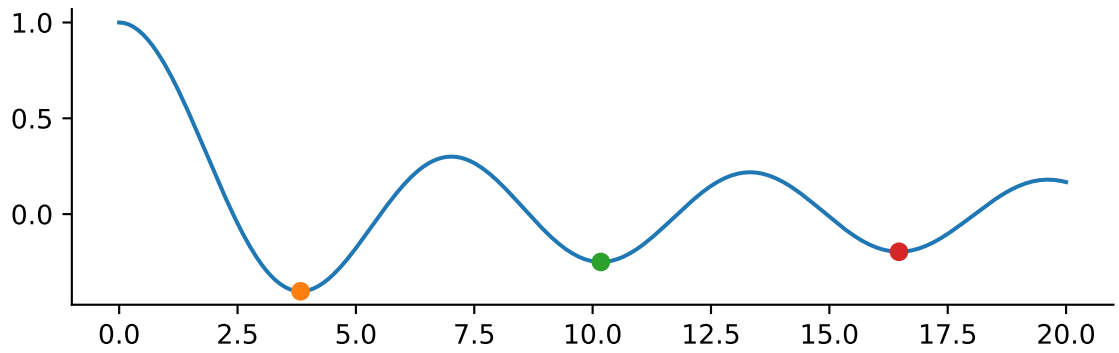
Notice there are many function evaluations. If your function is expensive to evaluate, minimization can be expensive.

```
plt.plot(x, j0(x))
plt.plot(result.x, result.fun, 'o')
```



Note this is a local minium. Passing a bracket we can find multiple local minima

```
results = []
bracket = [[2,4], [8, 10], [14, 16]]

for b in bracket:
    results.append(minimize_scalar(j0, bracket=b))

fig, ax = plt.subplots()
ax.plot(x, j0(x))

for mini in results:
    ax.plot(mini.x, mini.fun, 'o')
```



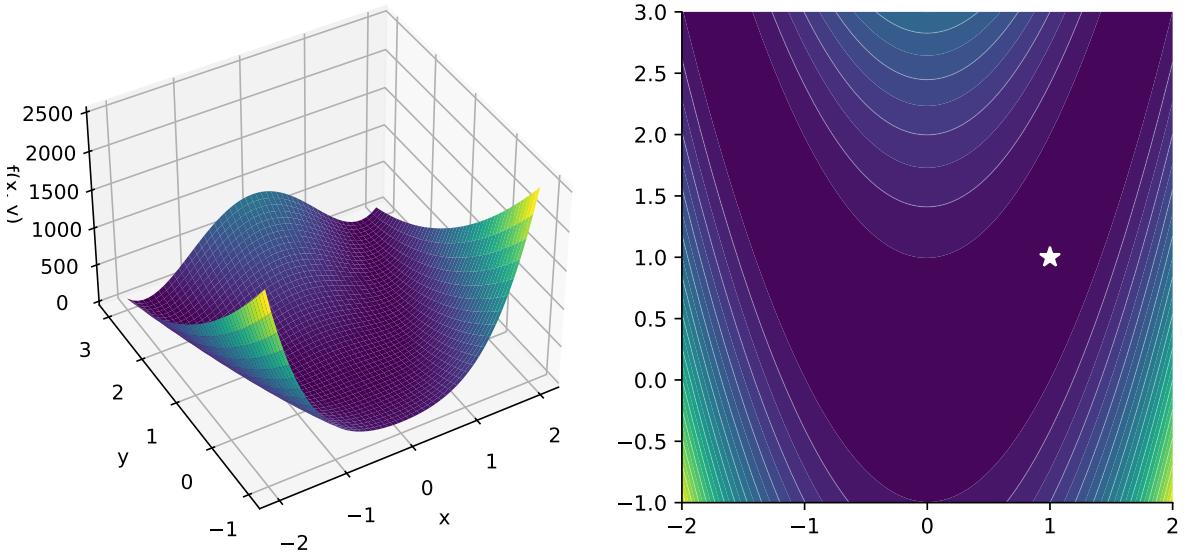
For a multi-dimensional example consider the example of the Rosenbrock function

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} 100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2$$

Don't worry too much about the exact form of the equation. The Rosenbrock function is a horseshoe type valley shown in Figure 3.5a. The Rosenbrock function is used as a benchmark for optimization techniques as it can be difficult for methods to navigate the shallow curved valley.

The Rosenbrock function as a minimum at $f(x, y) = f(1, 1) = 0$ shown in Figure 3.5b.

The Rosenbrock function is so commonly used in optimization that SciPy provides an method `rosen`.



(a) The funtion forms a shallow horseshoe valley (b) The minimum of the fucntion is at $(x, y) = (1, 1)$

Figure 3.5: The Rosenbrock function

```
from scipy.optimize import rosen

print(rosen([1, 2]))
print(rosen([-15, 25]))
print(rosen([1,1]))
```

```
100.0
4000256.0
0.0
```

Let's say we start at a point $(-1.5, 2.5)$ and we want to minimize the function. For multivariet minimization, SciPy provides the `minimize` method.

```
from scipy.optimize import minimize
x0 = np.array([-1.5, 2.5])
result = minimize(rosen, x0)
result
```

```
  message: Optimization terminated successfully.
  success: True
```

```
   status: 0
      fun: 2.008444162310784e-11
        x: [ 1.000e+00  1.000e+00]
      nit: 36
      jac: [-3.771e-07  1.852e-07]
 hess_inv: [[ 5.000e-01  1.000e+00]
            [ 1.000e+00  2.005e+00]]
     nfev: 144
     njev: 48
```

Again as with the root finding methods, the results contain more than just the minimum. In this case we also get the number of function evaluates and an estimate of the jacobian and hessian. The defauly method is `'BFGS'` but you can pass which method you want to use such as `Nelder-Mead`

```
minimize(rosen, x0, method='Nelder-Mead')
```

```
       message: Optimization terminated successfully.
       success: True
        status: 0
           fun: 6.446298475670153e-10
             x: [ 1.000e+00  1.000e+00]
           nit: 97
          nfev: 182
 final_simplex: (array([[ 1.000e+00,  1.000e+00],
                        [ 1.000e+00,  9.999e-01],
                        [ 1.000e+00,  1.000e+00]]), array([ 6.446e-10,  1.465e-
09,  1.635e-09]))
```

in which case we are returned the final simplex. Notice there were a few more function evaluations but not so bad compared to the default.

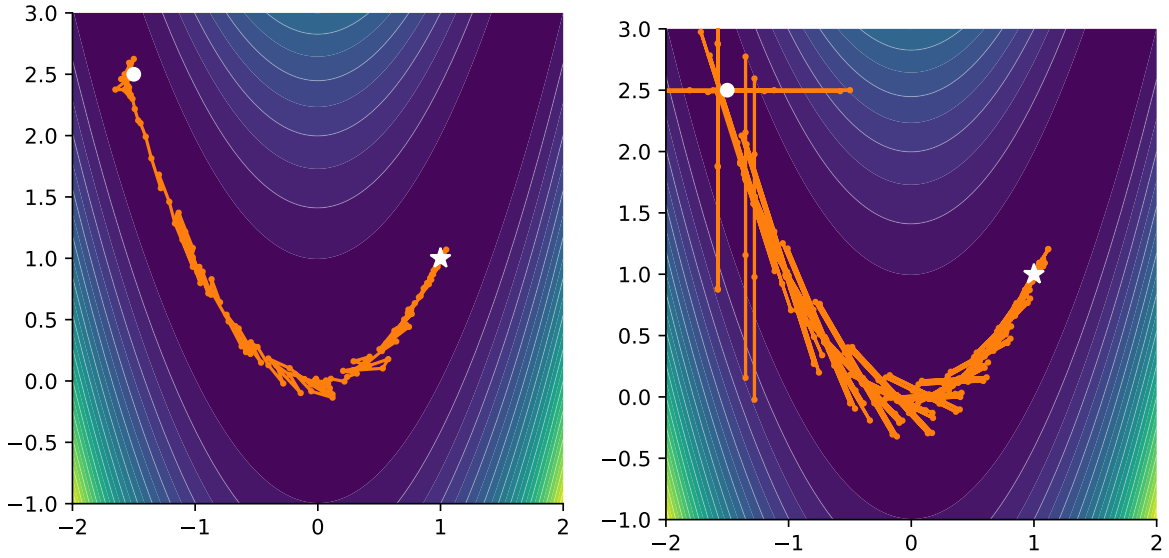
Consider our good old friend `Powell`

```
minimize(rosen, x0, method='Powell')
```

```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 9.860761315262648e-30
       x: [ 1.000e+00  1.000e+00]
```

```
  nit: 26
direc: [[ 4.521e-02  9.181e-02]
        [ 1.075e-07  1.814e-07]]
 nfev: 686
```

and notice how poorly it performs on this function. Navigating around the valley resulted in many more function evaluations. If can be instructive to view both methods side by side and see what path they take.



(a) Nelder-Mead: Takes a rough path but gets there in due time

(b) Powell: Really struggles with the diagonal route.

Figure 3.6: Comparing optimization traces

4 Numerical Integration

5 Numerical Integration of Ordinary Differential Equations

6 The Modeling of Data

References

Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. *Numerical Recipes in c (2nd Ed.): The Art of Scientific Computing*. USA: Cambridge University Press.