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. Rather than posting the original notes unchanged, I'm converting them into a Quarto book, possibly updating examples from C to Python and adding some of my own material.

Note: Most of the content and examples are taken from Dr. Gray's original course materials.

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!

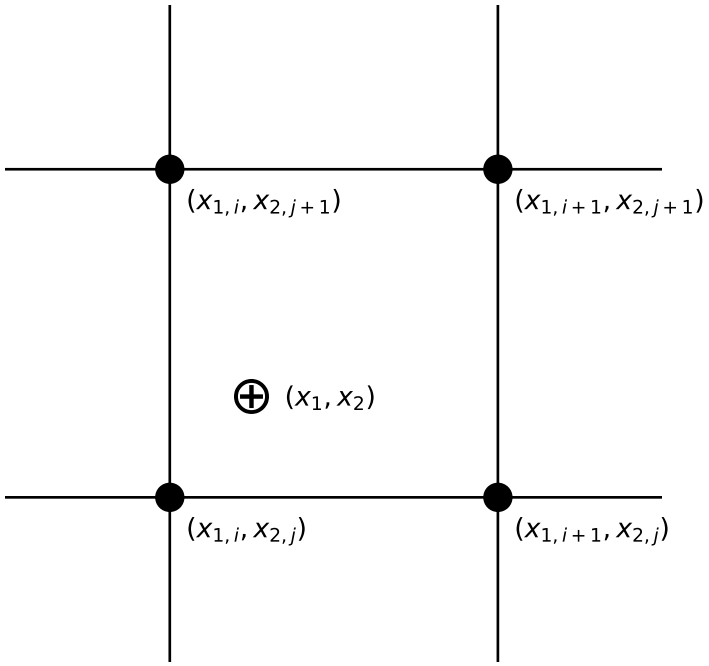
Before we leave the subject of interpolation, let us examine one further subject, that of efficiently bracketing x. If we have a set of x's (say in a vector `x[i]`) in numerical order that we must find interpolated values for, there is a simple time saving step that we can use, implemented in the following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

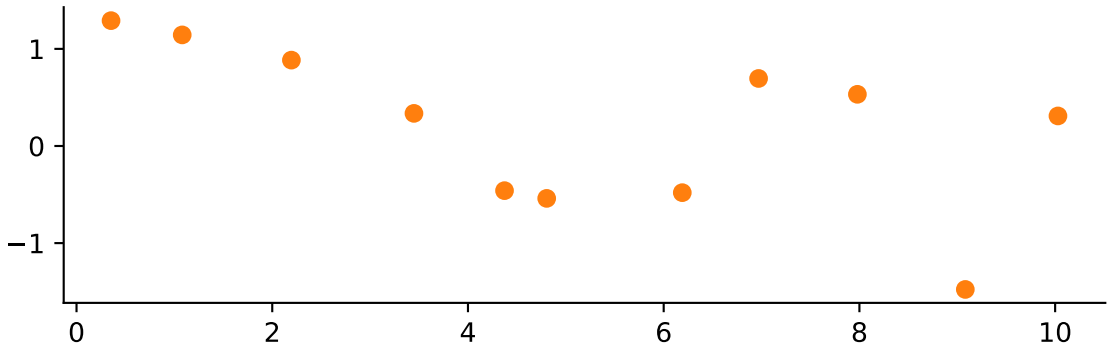
T(K)	10.0	11.0	12.0	13.0	14.0	15.0	16.0	17.0
1000.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
2000.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
3000.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
4000.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.000	0.0000
5000.0	0.0017	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')
```

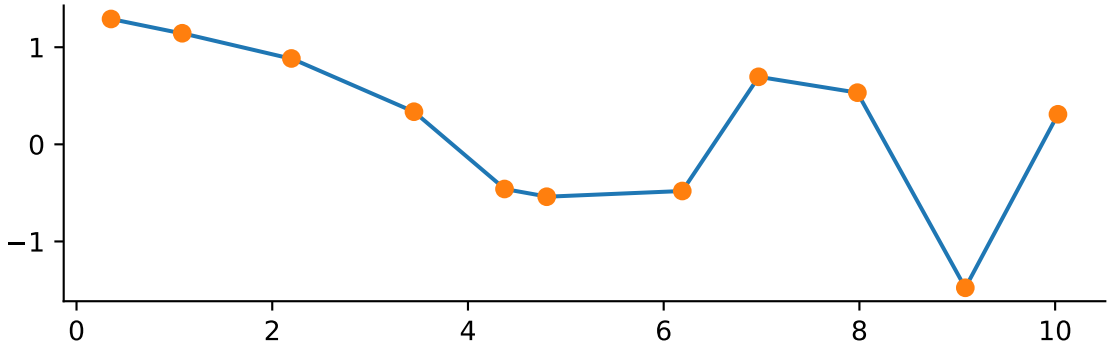


Now you want to connect the dots … I mean interpolate your data.

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.

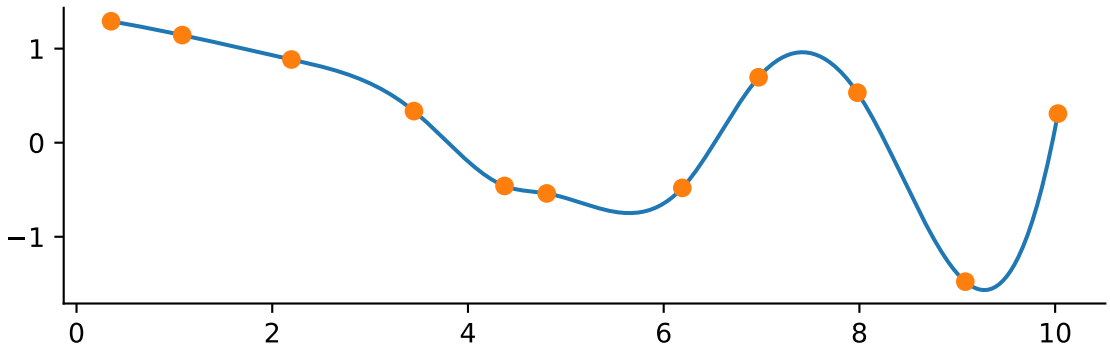
As an example consider the `CubicSpline` method

```
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')
```



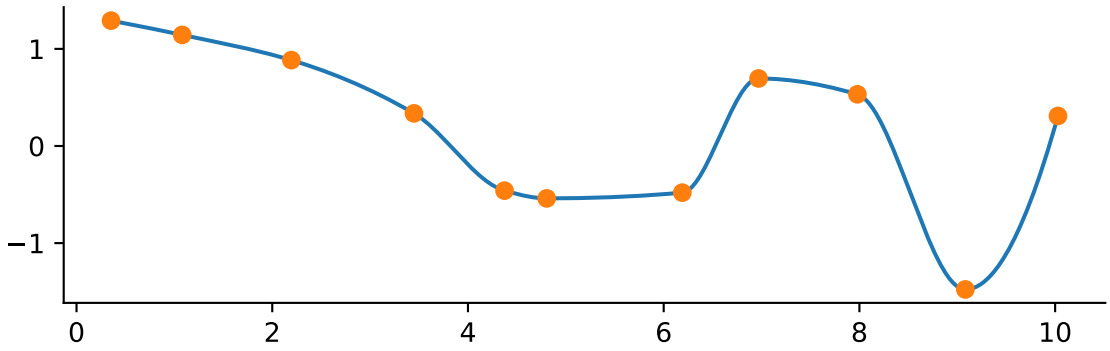
Quick and easy! There are other interpolation methods. For instance, maybe you are concerned with the line overshooting the data. An alternative is to use a so-called *monotone* cubic interpolant which attempts to preserve the local shape implied by the data. An example is the `PchipInterpolator`.

```
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')
```



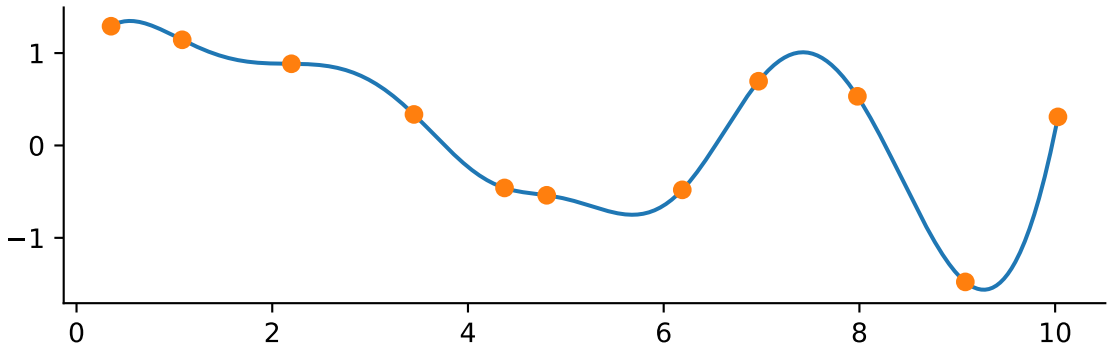
These cubic methods are typically fine for most application. There is also a generalized method for making interpolating splines of any degree but use with caution. You may introduce unexpected wiggles!

```
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):

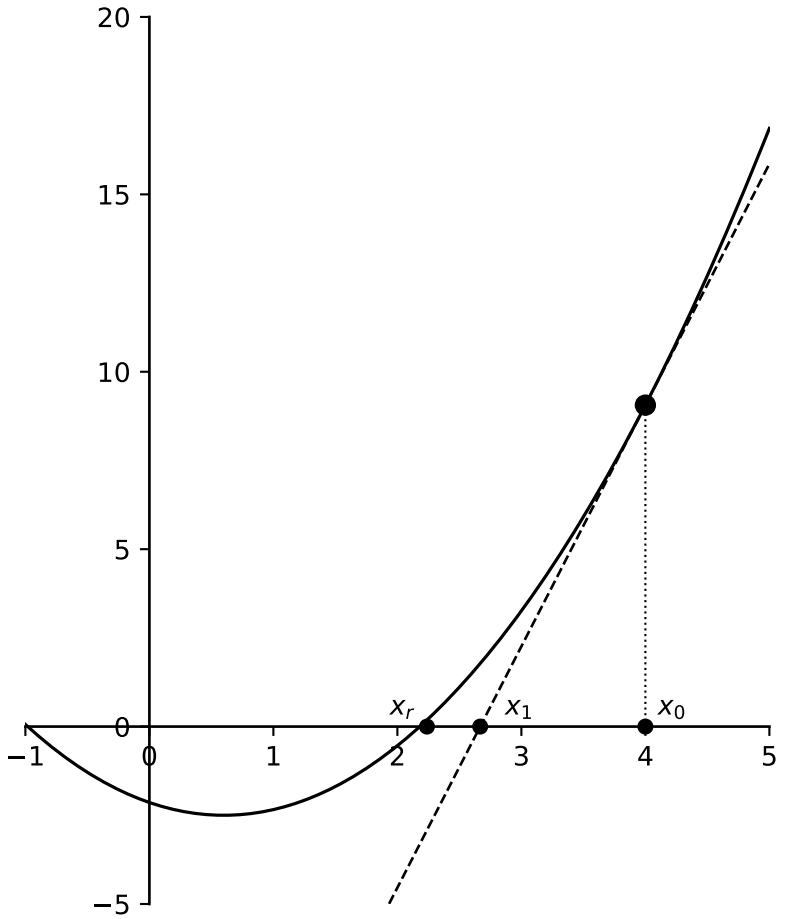
$$y = f'(x_0)(x - x_0) + f(x_0)$$

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*).

3 Minimization and Maximization of Functions

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.