Computational Physics Assignment

Samuel Lashwood, 01190892

November 7, 2018

1 Floating Point Variables

The machine accuracy gives the smallest number that can be meaningfully subtracted from 1. Due to numbers being stored by Python in binary, the size of a number can be effectively reduced by dividing by 2, as this will shift the mantissa of the floating point to make it smaller. A function was written to divide a float by 2 whilst the result of that float subtracted from 1 was not 1, and append it to a list of numbers that can be meaningfully subtracted from 1. The final value on this list is the machine accuracy. Due to the functional nature of this process, it was easily repeated with floats of different precision: single, double and extended using the *numpy.float(num)* (where *(num)* is a power of 2) data types. For Python 3 on a standard college *HP EliteDesk* Windows machine, the standard machine accuracy (found just using standard floats) was 1.11×10^{-16}, showing that double precision was used as it agreed with the result from *numpy.float64*. The precision for single and extended floats were 5.96×10^{-8} and 1.93×10^{-34} respectively. Theoretically, I expect the machine accuracy to be 2^{-m} where m is the length of the mantissa in that precision scheme as this is the smallest number that can be stored in as a float of that precision that can be meaningfully manipulated. The results I have obtained exactly follow this prediction.

Working with the extended floats presented some issues as the standard college machines run windows 10, which doesn't support *numpy.float128*, so the *mpmath.mpf* data type had to be used with a specified mantissa length of 112. This precision can be increased as much as the machine processor will allow, so this module was a good substitute-further levels of precision could be investigated (if time would allow!).

Issues also arose initially in operating with regular floats in the function, as this automatically converted all floats used into doubles. This was overcome by making the other numbers required in the function for operations arguments and passing the correct floats to the function in different calls.

I used the function *np.finfo(np.longdouble)* to confirm my findings with the machine specs.

2 Matrix Methods

In order to decompose matrices into upper and lower triangular matrices, Crout's method was implemented using iterative processes to build up the decomposition matrices. Iterative processes were used to best implement the sums in the standard equations and allow the matrices to be constructed in the order required by Crout's method (in solving dependant simultaneous equations with increasing length). Pivoting was implemented (inefficiently) to sort the rows of the matrices such that there were no zeros on the leading diagonal of the input matrix. This was tested using a few basic examples. This process was not optimized as it was not assessed. Due to the inclusion of pivoting, the function to decompose a matrix had an optional argument for the matrix on the RHS of a given matrix equation problem. This function returned both the upper and lower triangular matrices for use in later functions, the combination matrix required and the pivoted RHS matrix if input.

Using the decomposition function, a simple function was written to find the determinant of the input matrix (the trace of the upper triangular matrix) and a solving function was written to solve matrix equations as required. This solving function called the decomposition function in order to include pivoting and used back-forward substitution to obtain the solution of the matrix equation. The arguments of the function were changed to be the input matrix such that pivoting could be carried out in the general case and so that only one call was needed in the interpolation task (see section 3). This function was written to take matrix equations of any dimension, so it could be used to find the inverse of a matrix (A) by solving the equation

$$LUA^{-1} = \mathbb{1}. \tag{1}$$

The decomposed combination matrix and solution to the given equation were found to be

$$\begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 1 & 8 & 4 & 0 & 0 \\ 0 & 1.13 & 15.5 & 10 & 0 \\ 0 & 0 & -1.4194 & 45.1935 & 25 \\ 0 & 0 & 0 & -1.2170 & 29.5753 \end{bmatrix} \text{ and } \begin{bmatrix} 0.4562 \\ 0.6315 \\ -0.5129 \\ 0.0576 \\ 0.2028 \end{bmatrix}$$

respectively. The determinant was found to be 497220. These results were validated by multiplying the given matrix by the found solution to recover the given RHS matrix and using *numpy*'s inbuilt methods to find the determinant.The inverse of the given matrix was found to be

$$\begin{bmatrix} 0.3794 & -0.0461 & 0.0039 & -0.0048 & -0.0020 \\ -0.1382 & 0.1382 & -0.0117 & 0.0145 & 0.0060 \\ 0.0263 & -0.0263 & 0.0234 & -0.0290 & -0.0121 \\ 0.0717 & -0.0717 & 0.0637 & 0.0449 & 0.0187 \\ 0.0657 & -0.0657 & 0.0584 & 0.0411 & 0.0338 \end{bmatrix}$$

where all floating point values have been rounded to four decimal places. This was validated by multiplying this with the given matrix to obtain the identity matrix, which it did. There were a couple of small elements where zeros were expected in the identity matrix, but this is attributed to floating point errors.

3 Interpolation

Two different methods of interpolation, linear and cubic spline, were applied to the same set of data. The linear interpolater was implemented as a function that bubble sorted the input data into order by the first data element, identified the data points either side of the desired value and then used the standard formula to interpolate the data to that value. The cubic spline did the same thing, but created and solved a matrix equation for the second derivatives at every data point using the standard formulae and the natural spline boundary conditions to interpolate to a desired value. This matrix equation was comprised of the N linear simultaneous equations encoding the relations between the second derivatives of the functions- as described by the formulae given in lectures. The matrix equation was solved using the solving function written in the previous question (see section 2). Both of these were then carried out over a large range of values to cover the whole data range. This was possible (and makes the functions more generally applicable) as the interpolaters ignore values outside of the data range by returning 'None' for the interpolated value. Both results were plotted along with the original data points for visual checking, as shown in figure 1. Each point was interpolated with a step of 0.1 to

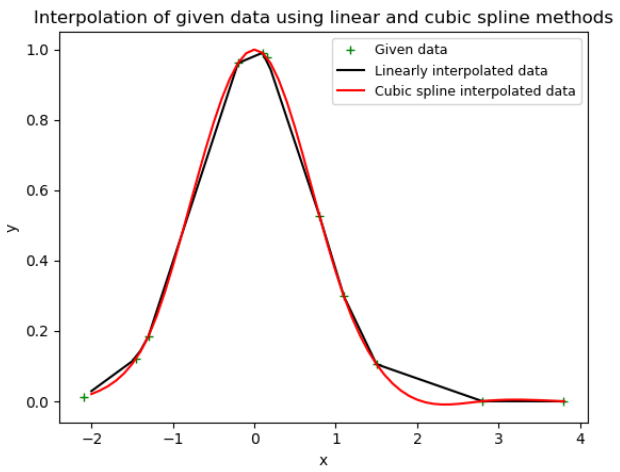


Figure 1: The plot of the interpolated data and original data. The Linearly interpolated values form straight line connecting the data whilst the spline values form a smooth curve through all points, as expected.

give sufficient smoothness in the cubic spline result whilst still executing quickly. The fact that *matplotlib.pyplot.plot* simply ignores 'None' values was relied upon.

The data points themselves were added to the linear interpolater output set of points so that the points either side weren't joined ignoring the original data (see figure 2).

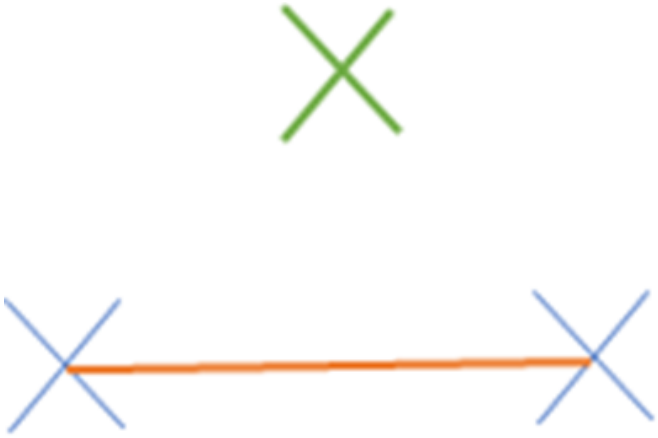


Figure 2: The error that can occur around the data points if not included in the interpolated data. Here, the blue interpolated points are connected as the green data point is ignored.

4 Fourier Transforms

In order to convolve the given functions (figure 3(a)), said functions were implemented as simple functions to call and then sampled at regular intervals between a start and end point. The exponential function was padded by setting all values with an absolute value of 3 or more to return 0. This was done to reduce the 'leaking' of the function from the aliased frequency spectra into the examined result spectra (eliminate circular convolution). The number of points in this sampling interval was chosen to be a power of 2 and even about 0, so that the fast Fourier transform method worked well and was centred for the convolution. The Fast Fourier Transform of these sets of data were then found using the *numpy.fft.fft* function. By using the convolution theorem:

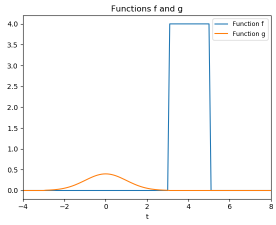
$$\mathfrak{F}(f(t) \circledast g(t)) = \frac{1}{2\pi}\mathfrak{F}(f(t))\mathfrak{F}(g(t)), \tag{2}$$

the given functions were spectrally convolved by computing

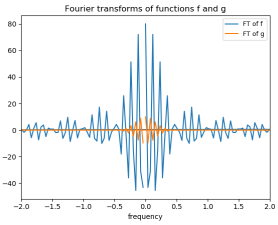
$$h(t) = f(t) \circledast g(t) = \mathfrak{F}^{-1}(\frac{1}{2\pi}\mathfrak{F}(f(t))\mathfrak{F}(g(t))). \tag{3}$$

The Fourier transforms of the given functions were plotted (using the *numpy.fft.fftfreq* function to obtain the frequency space intervals in the correct order). These were used to validate the sampling density by obtaining a smooth curve, see figure 3(b).

The absolute values of the convolved values were then plotted on the same real space using the *numpy.fft.fftshift*



(a) The two given functions to convolve plotted in real space



(b) The Fourier transforms of the given functions plotted in frequency space. These transforms were used with the convolution theorem to produce the convolution.

function to automatically alias the data. This function centred the frequency spectrum of the Fourier transform to 0 for the even sampling region, so that the convolution was in the correct place in the spectrum. The complex conjugate of one of the Fourier transforms in the convolution theorem was used to partially cancel the imaginary parts of the convolution, so that the absolute value at each sampled point was of the correct amplitude.

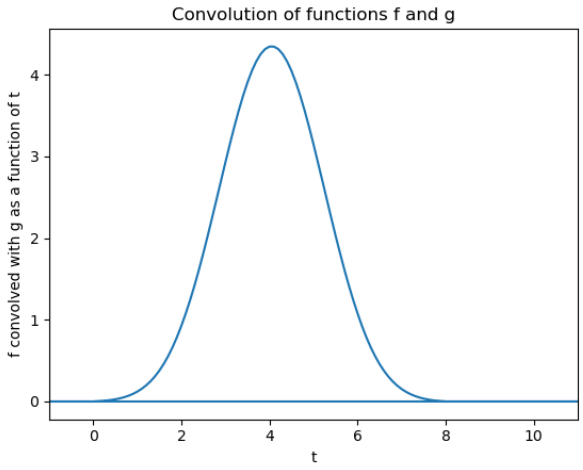


Figure 3: The convolution of the two given functions plotted in real space. The first and last points in the convolution are joined.

5 Random Numbers

A uniform random number distribution of 10^5 samples was generated using the *numpy.random.uniform* function with an arbitrary seed. This common function was used as it has a large periodicity (good pseudo-randomness) and can be easily seeded. This distribution was then plotted in a histogram of 50 bins (see figure 4). This bin number was chosen as it both eliminates any small fluctuations in the uniformity and still shows a large number of intervals.

The transformation method was the used to produce

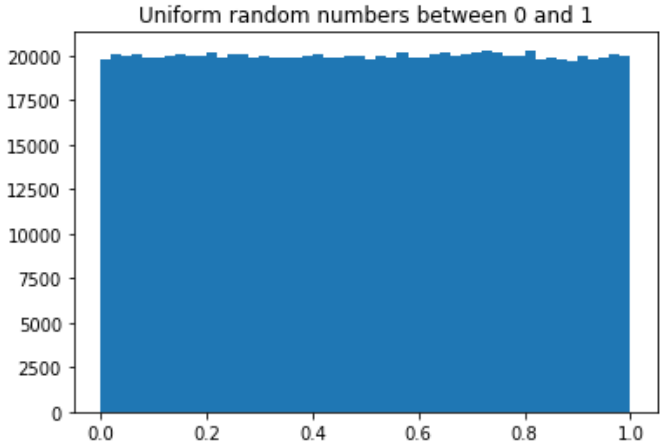


Figure 4: The histogram of the 10^5 normalised uniform random numbers showing uniformity with some random fluctuations.

10^5 random numbers distributed according to the PDF

$$P(x) = \frac{1}{2} sin(x). \tag{4}$$

According to the transformation method, for a PDF, P(x), the uniform random numbers, x, must be transformed by y(x) where

$$y(x) = F^{-1}(x) \; and \; F(y) = \int_0^y P(\phi) d\phi. \tag{5}$$

For the given PDF, it was found that

$$F(y) = \int_0^y \frac{1}{2} sin(\phi) d\phi = -\frac{1}{2}(cos(y) - 1) \tag{6}$$

and therefore

$$y(x) = F^{-1}(x) = cos^{-1}(1 - 2x). \tag{7}$$

This was used to transform the set of uniform random numbers to be distributed correctly. This distribution was then plotted (see figure 5).

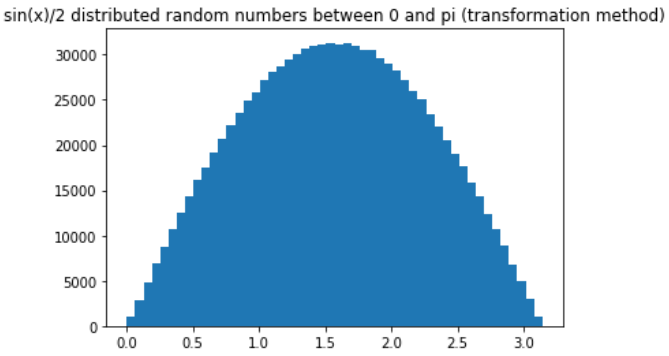


Figure 5: The histogram of the 10^5 random numbers distributed with PDF in equation 4 showing the expected sinusoidal distribution form.

The rejection method was then used to produce 10^5 random numbers distributed according to the PDF,

$$P(x) = \frac{2}{\pi} sin^2(x), \tag{8}$$

first using a constant comparison function and then using

$$C(x) = \frac{2}{\pi} sin(x) \tag{9}$$

as a comparison function. The rejection method with the sinusoidal comparison function produced a well distributed histogram with clear sin^2 shape, as can be see in figure 6. New random numbers had to be sampled for these comparisons to remove any correlation between the sample set and comparison criteria. For the second comparison, the uniform random numbers from the previous question were scaled by $\frac{4}{\pi}$ so as to cover the whole domain for the comparison function (a set of random numbers weighted by the comparison function was produced using the transformation method as in the previous case. Comparative list comprehensions were used for these implementations of the rejection method and written as functions so that the *timeit.timeit* method could be used to time the duration of the functions running. The size of the list of initially produced random numbers was then increased so that 10^5 samples could be produced using the rejection method, as some are rejected.

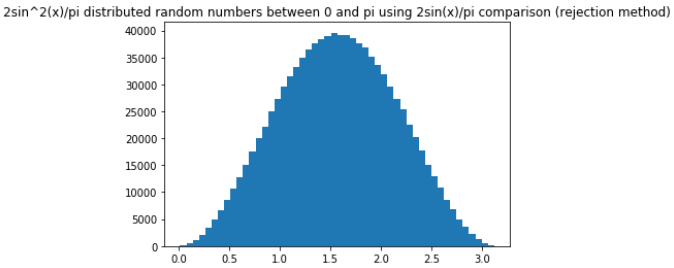


Figure 6: The histogram of the 10^5 random numbers distributed with PDF in equation 8 showing the expected sinusoidal squared distribution form.

The ratio of time to create the weighted random number distributions was found to be 2.0678, which was obtained by timing 10 iterations of the rejection method and dividing by the time for 10 iterations of the transformation method. Multiple iterations were used to eliminate any random fluctuations in machine processing time. It is expected that the rejection method would take more time as more operations must be carried out for each comparison. Also, I produce a list of slightly more than 10^5 numbers (for ease of solving the problem), so time is taken up by this.