Project A Report - Numerical Methods

Richard M. G. Boyne
CID: 01057503
Word Count: aprox 1700 (Captions, bibliography and Appendices discounted)

October 13, 2017

Overview

This report is for computational project A[1] for which there were five tasks requiring coding solutions were set. It is accompanied by "Project_A_code.zip" containing all the code in python used for this report.

Machine Accuracy

Every numerical data type has a round-off error quantified by its machine accuracy[2] ϵ, defined as the smallest number that can be added to 1 without the result rounding back to 1. The aim of this section was to find the machine accuracy for different sized floating point variables.

The smallest value that can be stored is found by taking the sum of 1 and increasingly negative powers of 2 until the result is rounded down to 1. The code for this section was written as a function that operates generally, accepting any data type and number value. To ensure the number was never cased as a different data type after each relevant operation the result was recast as the original type. This section supports a validate function that checks the code with several data types including all integer and all possible sizes of floating points.

An issue found by writing this section is that many machines do not support pythons 128-bit floating point variables and as such could not be tested. The other tested data types are shown in table 1 agree with the expected values from floating point theory (see Appendix A) and from literature for IEEE 754 standard floating points[2] (to at least the 3 decimal places given there).

To conclude a simple algorithm has been used to find the machine accuracy values in agreement with literature to at least 3 decimal places.

Variable Type	**Machine Accuracy (ϵ)**
int-32	1
int-64	1
float-16	9.7656×10^{-4}
float-32 (single)	1.19209×10^{-7}
float-64 (double)	$2.22044604925 \times 10^{-16}$

Table 1: Machine accuracy's for various numpy library variable types (value of one used). Agrees with theory given in Appendix A

Matrix methods

A standard method for solving large scale matrix equations (and by extrapolation systems of linear equations) is LU decomposition[2]. It has the advantage of being able to solve for multiple initial conditions quickly and find the inverse or determinant of a matrix easily. The aim of this section was to take a square matrix and decompose it into the upper and lower triangular matrices, then use them in forward backward substitution to solve matrix equations of the form $A * x = b$ for x.

The equations used here are seen in Numerical Recipes[2]. As several repeatedly used values are computed in this section that all stem from the original matrix it makes sense to store them in a class instance so that they may be kept together and repeatedly used at a later time without recalculation. The upper, lower matrices, inverse and determinant found are all suitably verified via the numpy library to ensure they are correct to within rounding errors. An issue here is that for large matrices this value can sometimes overflow a float-64 often causing a numpy infinity answer so caution must be used (this issue is also present for the reference numpy determinant). A validation function also exists that uses a random matrix and times the code. It demonstrates that a (200, 200) matrix can be decomposed with inverse and determinate found correctly in approximately 2 seconds.

The solutions to the given problem are shown below and have been verified.

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 3 & 8 & 4 & 0 & 0 \\ 0 & 9 & 20 & 10 & 0 \\ 0 & 0 & 22 & 51 & -25 \\ 0 & 0 & 0 & -55 & 60 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1.5 & 6.5 & 4 & 0 & 0 \\ 0 & 1.385 & 14.462 & 10 & 0 \\ 0 & 0 & 1.521 & 35.787 & -25 \\ 0 & 0 & 0 & -1.537 & 21.578 \end{bmatrix} \tag{1}$$

Matrix given (left) decomposes into upper and lower matrices which are combined (right) as $L + U - I$ where I is the identity matrix.

$$\mathbf{A}^{-1} = \begin{bmatrix} 0.712 & -0.141 & 0.046 & -0.017 & -0.007 \\ -0.424 & 0.282 & -0.093 & 0.033 & 0.014 \\ 0.313 & -0.209 & 0.151 & -0.054 & -0.022 \\ -0.245 & 0.164 & -0.118 & 0.078 & 0.032 \\ -0.225 & 0.150 & -0108 & 0.071 & 0.046 \end{bmatrix} \qquad \mathbf{A} * \begin{bmatrix} 0.338 \\ 1.325 \\ -1.653 \\ 1.723 \\ 1.720 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ -4 \\ 8 \\ 9 \end{bmatrix} \tag{2}$$

Inverse of matrix given (left), and the solution to the matrix equation supplied (right). The determinant of matrix given is 145180.0.

This code could be improved to accept matrices requiring pivoting and raise errors on being passed in a linearly dependent (i.e. unsolvable) matrix. It could also be optimized by using array slicing more over for loops to improve run time, for example solving all columns in b simultaneously as an array operation rather than looping over each. However this would only be significant for large matrices with m,n of order 10^4 and above. Note this code is limited to solve up to matrices of size 10^3 in reasonable time.

To conclude LU decomposition has been implemented for square matrices with reasonable run time up to size 10^3 matrices. From the decomposition the determinant, inverse an solutions to matrix equations have been correctly found with a limitation being a too large determinant causing overflows.

Interpolation

Interpolation is used to generate a continuous function that follows a set of points to a predetermined number of continuous derivatives. The aim of this section was to find a linear interpolation and cubic interpolation for a given set of points.

The equations used in this algorithm are shown in Appendix B. For the cubic spline interpolation there are N 2^{nd} derivative unknowns with N-2 equations, hence the 2^{nd} derivatives for the first and last points are set to zero so the equation is solvable. The matrix equation this generates can be solved using the previous sections LU decomposition. In setting up the matrix equation the boundary conditions are set by using rows filled with zeros and a single 1 in the unsolved matrix making the implementation simpler than alternative methods (see Appendix B.

The code for this section is written as several functions to give a more logical flow of the process and a validation function exist that simply interpolates for a set of random y values over uniform x values.

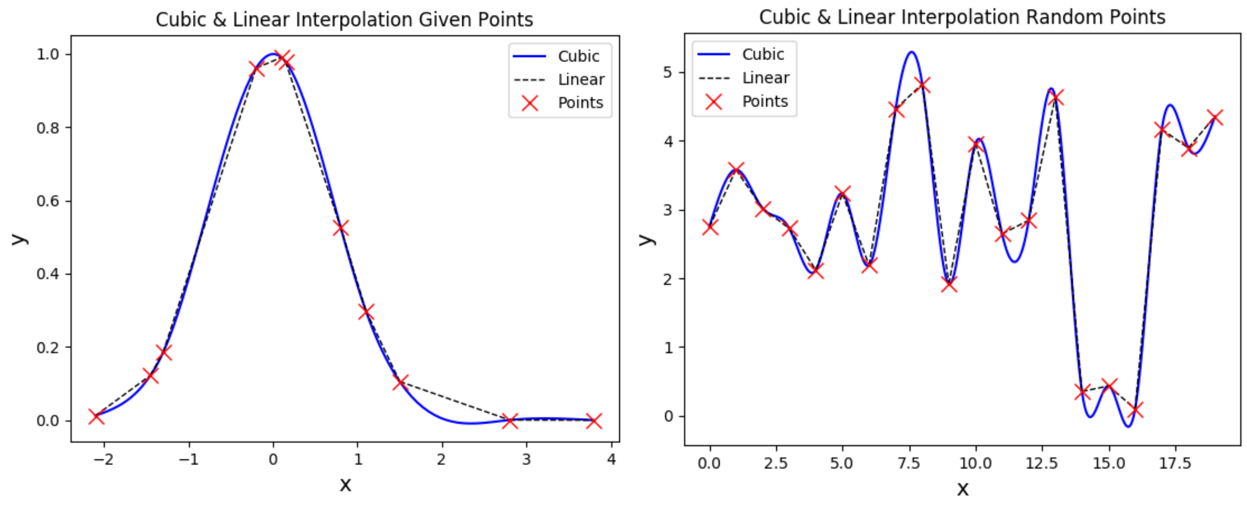

Figure 1: Cubic and Linear interpolation of the given set of data points (left) and a random set of y values. The interpolations here are reasonable, see Appendix B figure 4 for unreasonable plots.

The linear interpolation works well for any distribution of points (however is discontinuous to the 0^{th} order). The cubic is also sensible provided a smooth curve is initially sampled or the x-spacing of points is reasonably large, fitting to random points in both x and y gives far poorer results (see Appendix B) due to some points being close on the x-axis with large y value differences. This is a limitation of the cubic interpolation method, not this particular implementation.

To conclude both linear and cubic interpolation have been implemented for random and sampled functions to give reasonable fits that are discontinuous in the 0^{th} and 2^{nd} derivatives respectively with a limitation that closely spaced x values can cause large abnormalities for this plot in certain cases.

Fourier transforms

Fourier transforms are frequently used for analysis of sound and experimental data as well as a computational trick to speed up certain processes. This section was aimed at utilizing numpy's Fast Fourier Transform[2] (FFT) to rapidly convolve a signal and response function.

This method exploits the property that:

$$\mathcal{F}\{h(t) * r(t)\} = \mathcal{F}\{h(t)\}\mathcal{F}\{r(t)\} \tag{3}$$

where $\mathcal{F}$ is the Fourier transform and $*$ as the convolution with $h(t)$ as a signal function and $r(t)$ as a response function.

The numpy FFT expects response functions shifted using the periodic assumption of the Fourier transform from figure 2 upper to lower so that the resulting FFT product is not shifted in frequency space. The numpy convolution function was used as a reference (see figure 2). Both inverse FFT product and the numpy convolution are not normalized, hence their integrals depend on the number of points used for the Fourier transform. To normalize the functions are multiplied by the time step so that the integral of the resultant convolution is not dependent on the sampling of the signal and response functions. To keep FFT as fast as possible 2^n points were always sampled between a desired range, hence padding zeros is replaced with increasing resolution over the predetermined range which is more useful.

To try and understand why these measures were needed a separate FFT function was written and the results of which are shown to be the same of the numpy FFT function (see Appendix C). This gave insight to the normalization requirement as if more points are used in the transform the area under the original curve is found to be larger hence so is the transformed curve, however no further insight as to the origin of the required response signal shift was gained.

The results from the inverse FFT product and numpy convolution, figure 2 are the same and match what is expected from the original functions by inspection, showing that this transform method is correct.

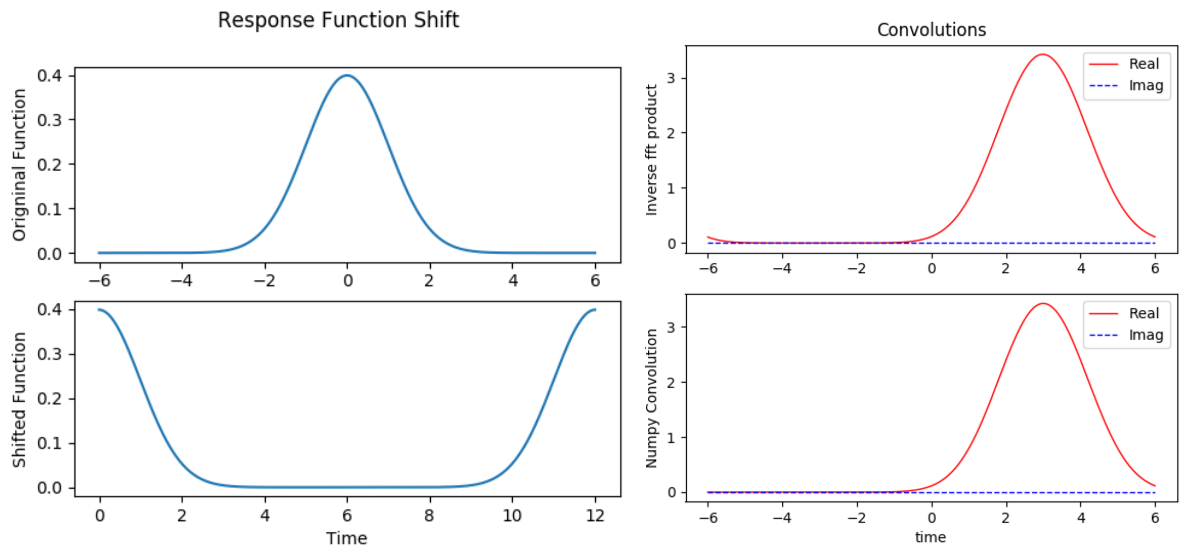

Figure 2: Left - Original response function (upper) and shifted response function under the periodic assumption used by the Fast Fourier Transform. The original function gives a shifted convolution under both numpy convolution and inverse of the FFT product where as the shifted function does not.
right - convolution of the a box signal function and a shifted gaussian response function both via inverse FFT product (upper) and numpy convolution (lower). Both are normalised and seen to be the same.

To conclude convolution using the inverse FFT product has been implemented and shown to be the same as pre-existing convolution libraries provided the response function is given over the correct range. Normalization has been used to keep the convolution constant under change of sampling distance.

Random numbers

Generating random numbers is at the heart of many computational algorithms and are often required to be random over a given distribution. This section was aimed and taking a uniform distribution from the numpy library to generate a non-uniform distribution of random numbers over probability density function (pdf).

The numpy library generates numbers over the [0,1) range, rather than the required [0,1]. However the probability of getting a 1 in the interval [0,1] with a uniform distribution is 1/number of float-64 values in [0,1) which is negligible considering we are taking order 10^5 samples so here these intervals can be considered the same.

A verify function is written here that takes the mean square difference of the normalized histograms from the expected distributions.

For an analytically invertible the transformation method can be used where a normalised cumulative distribution function (cdf) is used to directly map a uniform distribution from [0,1] to a desired pdf, figure 3 shows this for pdf $\frac{1}{2}Sin(x)$.

For a function with unknown inverse the rejection method is used where a comparison function $C(y)$ that we can distribute random numbers x over and is always greater than the desired pdf is needed. By selecting a random number n for each x and rejecting it if $n > C(x)$ we get a distribution over any pdf. To reduce the number of rejections a comparison function close to the desired pdf is needed, an example with pdf $\frac{2}{\pi}Sin^2(x)$ is shown in figure 3.

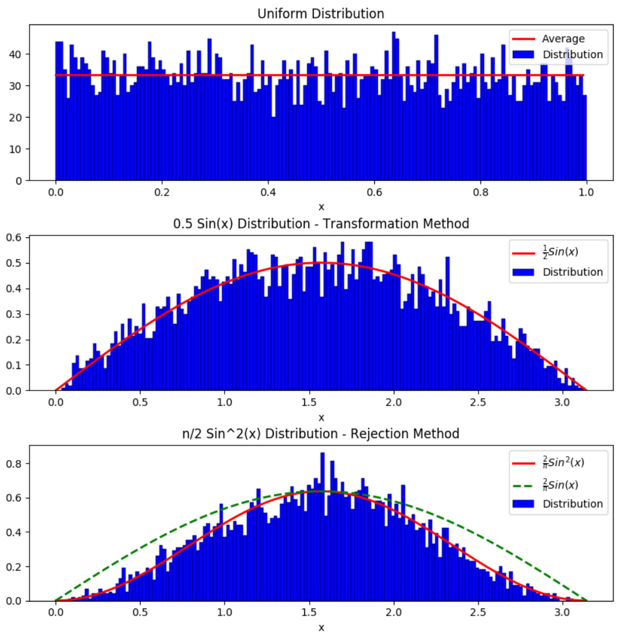

Figure 3: 10^5 *random numbers generated over:*
top - uniform distribution over the range [0,1)
middle - pdf $\frac{1}{2}Sin(x)$ *using the transformation method with cdf* $Cos^{-1}(1-2x)$ *over the range [0,π)*
bottom - pdf $\frac{2}{\pi}Sin^2(x)$ *using the rejection method with* $\frac{2}{\pi}Sin(x)$ *comparison function using b) as the initial number generator over the range [0,π)*

Although the comparison function and transformation function are different, after normalization are identical hence the x's for the rejection method can be found from the previously used transformation method. A verification function exists which finds the mean square difference from the expected pdfs for each normalised distribution and found they all tend to zero in the large number limit. These are all relatively low which would be expected for large number of generated numbers showing the algorithms here are valid. It also times the code to take under 1s for 5000 numbers.

To conclude, random numbers from the [0,1) range have been shifted to different range and pdfs via transformation and rejection methods with the latter being optimized by a pre-found distribution on a closely fitting comparison function. All run relatively quickly (under 1s for 5000 numbers) and in high number limit tend to the desired pdfs.

References

[1] Y. Uchida and P. Scot, "Project a," 2017. Given Task sheet.

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes.* Cambridge University Press, third ed., 2007. Pages: Floating Points 9–10, LU Decomposition 48–50, Interpolation 120–122, Fourier Transforms 608 –617, Random Numbers 340–341.

Appendices

A Floating Point Theory

A float variable stores numbers as a combination of exponent and mantissa. In theory the smallest value that can be stored in a float of a fixed exponent (caused by the fixed value of 1) is given by the length of the mantissa. Therefore for an exponent of zero a 32-bit float can store a 1×2^{-23} and a 64-bit float can store a 1×2^{-53}. These values are only possible as the phantom is ignored by python's implementation of floating points to provided one extra usable bit in the mantissa[2]. Due to he dependence on the exponent the smallest value that can be added and kept will depend on the value of the number to be added to, hence the value 1 is used in the definition of machine accuracy.

B Interpolation

The equations that need to be solved for interpolation are

$$y(x) = A(x)y_i + B(x)y_{i+1} + C(x)y_i'' + D(x)y_{i+1}'' \tag{4}$$

$$\alpha_i y_{i-1}'' + \beta_i y_i'' + \gamma_i y_{i+1}'' = \delta_i \tag{5}$$

where A, B, C, D all depend on the x_i and x_{i+1} points and $\alpha_i, \beta_i, \gamma_i, \delta_i$ depend on x_i, x_{i-1} and x_{i+1} and $y(x)$ is the interpolation function between the x_i and x_{i+1} points (the equations for these are found in Numerical Recipes[2]).

Equation (5) is turned into the following n by n matrix equation with bounty conditions of $y_0, y_{n-1} = 0$ set by the first and last rows of the n by n matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ \alpha_1 & \beta_1 & \gamma_1 & 0 & \cdots & 0 \\ 0 & \alpha_2 & \beta_2 & \gamma_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \alpha_{n-2} & \beta_{n-2} & \gamma_{n-2} \\ 0 & \cdots & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 0 \\ \delta_1 \\ \delta_2 \\ \vdots \\ \delta_{n-2} \\ 0 \end{bmatrix} \tag{6}$$

An interpolation of points that are too close together on the x axis with a large difference in the y axis give large abnormalities in the fitted cubic spline (figure 4).

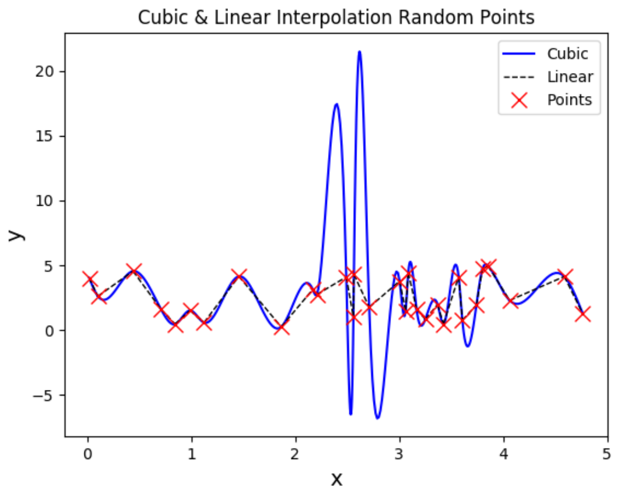

Figure 4: Cubic spline with random x and y values so some x values are very close with large y value differences. This plot was generated with the validate function using the parameters n= 30, s=10, xrand = True

C Fast Fourier Transforms

In attempting to understand Fast Fourier Transforms the code ”fft_attempt” was written that gives the same results as numpy fft. It operates using the FFT equations in Numerical Recipes[2].

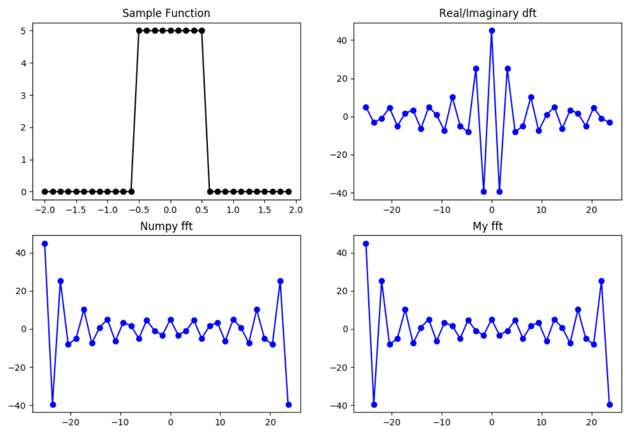

Figure 5: FFt of a box function (top left) compared with a discrete Fourier transform (top right), numpy FFT (bottom left) and FFT found by fft_attempt (bottom right), as can be seen the FFTs are the same and simple a shift of the discrete Fourier transform