Computational Physics Assignment, Autumn 2019

S. S. Mall

I. Floating Point Variables

The aim of this question is to determine the machine accuracy of the system used and compare to the theoretical value.
The "Machine Epsilon" ϵ is defined as the difference between the value 1.0 and the smallest representable value above it. Hence this value can be used as the relative accuracy of the system as any number smaller than it cannot be operated upon without losing accuracy.

In IEEE 754 standard, the theoretical value of $1+\epsilon$ is given by setting the exponent to zero (before adding the bias of 127) and having the only non-zero term of the mantissa be the least significant bit. Because there are 8 bits delegated to the exponent and one to the sign, this leaves 23 bits for a 32-bit system and 55 bits for a 64-bit system. For a bit in position n of the mantissa, the denary value is 2^{-n}.

Subtracting 1 from this value gives ϵ, which is calculated to be 1.192×10^{-7} for a 32-bit system and 2.78×10^{-17} for a 64-bit system. Numpy also allows for extended precision which allocates more bits to the number, but this is variable and depends on the system being used.

The empirical calculation of ϵ was found by first initialising a value of $a_0 = 1$ as either a numpy.float32 or numpy.float64 datatype, and a multiplier variable of 0.5. A value of eps is added to the value of 1.0. If the sum of a_0 and eps is not equal to a_0 then eps is multiplied by 0.5 to be made smaller.

This is repeated until the two values are equal so that the operation cannot be properly carried out. Because this is the first value above 1 that cannot be completely represented, ϵ is given by the previous value of eps, which is eps divided by the multiplier.

The empirical value of ϵ for 32-bit representation agrees with the theoretical value of 1.192×10^{-7}. However, the result for 64-bit representation was 8 times larger at $2.22 \times 10^{-16} \approx 2^{-52}$. This value was consistent with the output of the numpy finfo function which suggests that 3 bits have been removed from the mantissa for use elsewhere.

The result for extended precision representation was equal to the result for 64-bit representation for the system used.

II. Matrix Methods

This question was to develop a matrix solver by decomposing the input matrix A upper and lower matrices U and L respectively such that $L.U = A$.

A. Decomposition

By writing out the explicit terms of general L and U matrices, the terms of the $L.U$ matrix can be found in terms of L_{ij} and U_{ij} and compared to matrix A to solve for the terms $(LU)_{ij}$.
Because U has unitary diagonal, the matrix LU can be written (for a 3×3 matrix) as

$$\begin{bmatrix} l_{00} & u_{01} & u_{02} \\ l_{10} & l_{11} & u_{12} \\ l_{20} & l_{21} & l_{22} \end{bmatrix}$$

and $L.U$ as

$$\begin{bmatrix} l_{00} & u_{01}u_{01} & l_{00}u_{02} \\ l_{10} & l_{10}u_{01} + l_{11} & l_{10}u_{02} + l_{11}u_{12} \\ l_{20} & l_{20}u_{01} + l_{21} & l_{20}u_{02} + l_{21}u_{12} + l_{22} \end{bmatrix}.$$

The first thing to note is that the terms $(L.U)_{i0} = l_{i0}$ can be found directly as they are equal to A_{i0}.
Secondly considering any term $(L.U)_{ij}$, the term is only composed of terms from the LU matrix with lower or equal i and j (due to the definition of matrix multiplication). So the LU matrix can be initialised with all terms zero and by solving each term successively in i and j, the only unknown in $(L.U)_{ij}$ is $(LU)_{ij}$.
Finally, the sought term in $(L.U)_{ij}$ is multiplied by l_{ii} if $i < j$.

Using the definition of matrix multiplication and $(L.U)_{ij} = A_{ij}$ along with these observations, it can be shown that

$$(LU)_{ij} = A_{ij} - \sum_{k=0}^{j-1} (LU)_{ik}(LU)_{kj} \quad ; \; i \geq j \tag{1}$$

and

$$(LU)_{ij} = \frac{1}{(LU)_{ii}} . \left(A_{ij} - \sum_{k=0}^{i-1} (LU)_{ik}(LU)_{kj} \right) \quad ; \; i < j \tag{2}$$

The formula for each element of LU depends only on elements from A and other elements from LU, rather than pulling variables from separate L and U matrices which would be less efficient. The triangular matrices L and U are easily extracted from this matrix and the numpy.dot function has been used to check that $L.U = A$.

B. Substitution

Once the matrix has been decomposed into triangular matrices, solving by forward and backward substitution is easy to implement:

The forward substitution function is passed the decomposed matrix LU and extracts the lower matrix L. This solves for the equation $\underline{\underline{L}}.\underline{y} = \underline{b}$ where $\underline{y} = \underline{\underline{U}}.\underline{x}$. The equation $\underline{\underline{U}}.\underline{x} = \underline{y}$ is then solved for $\underline{x}$.

C. Testing and Results

The matrix solver was tested at each step using an online example of Crout's Method at
https://numericalmethodsece101.weebly.com/croutrsquos-method.html

which carried each step of the method out explicitly for a 3×3 set of simultaneous equations. After passing this test, the solver was tested against several online examples of higher-dimension matrix questions aimed at A-Level mathematics students.

Inputting the given matrix A and vector b into the matrix solver yields a result for x of

$$\underline{x} = \begin{bmatrix} 0.456571 \\ 0.630288 \\ -0.510575 \\ 0.0538916 \\ 0.196132 \end{bmatrix} \tag{3}$$

This final result was tested using the numpy.dot function to ensure that $\underline{\underline{A}}.\underline{x} = \underline{b}$ was satisfied.

D. Determinant

Because Det$(M.N)$ = Det(M).Det(N), the determinant of A is equal to the product of the determinants of the upper and lower matrix, which for triangular matrices is just the product of the diagonal terms. Furthermore, the upper matrix has a unitary diagonal so the determinant of A is the product of diagonal elements of L. For the given matrix A, this is computed to be 514032.0.

E. Inverse

The inverse A^{-1} of a matrix A must satisfy $A \,.\, A^{-1} = I$, where I is the identity matrix. A^{-1} can be found by solving the matrix equation

$$\underline{\underline{A}} \cdot \underline{(A)^{-1}_{:i}} = \underline{(I)^{-1}_{:i}} \tag{4}$$

where the subscript $(: i)$ represents the i^{th} column of the matrix. This is solved for each column of A^{-1} individually which are then collected to form the matrix. A note to make here is that when choosing the column from a matrix, the numpy.transpose function must be used to change it from a row to column vector. Acting this on the give matrix A returns the matrix (to two significant figures)

$$\begin{bmatrix} 0.3795 & -0.04618 & 0.0040 & -0.0047 & -0.0019 \\ -0.14 & 0.14 & -0.012 & 0.014 & 0.0058 \\ 0.027 & -0.027 & 0.024 & -0.028 & -0.011 \\ 0.070 & -0.070 & 0.063 & 0.044 & 0.018 \\ 0.064 & -0.064 & 0.056 & 0.040 & 0.033 \end{bmatrix} \tag{5}$$

This was tested by acting the numpy dot function on A and A^{-1} to check if their product is the identity matrix. The diagonal terms were all 1.0 but all non-diagonal terms after the first column were on the order 10^{-16}, though this is on the order of the machine accuracy of a 64-bit system and so is within the uncertainty of the operations.

III. Interpolation

A. Linear Interpolation

This form of interpolation builds a function which passes through all the discreet data points (x_i, f_i) and is linear for all x between them. This leads to discontinuity in the first derivative and so is unlikely to represent the physical processes that produce the data.

In order to satisfy the above condition, the function $f(x)$ must take the form [1]

$$f(x) = \frac{x_{j+1} - x}{x_{j+1} - x_j} \,.f_j + \frac{x - x_j}{x_{j+1} - x_j} \,.f_{j+1} \tag{6}$$

where x_j is the closest given data point before x, and x_{j+1} the closest data point after it. The value of $(j + 1)$ is found by iterating through all x_i and noting the first instance where the data point is greater then or equal to x. When plotting, an array of equally spaced x of length 100 times the length of the data array over the same range is initialised and each x is passed through the linear interpolator function to create a new array of f values.

B. Cubic spline

As mentioned, the linear interpolator is likely to be nonphysical for most systems. In this case, a cubic spline may be more suitable as in addition to the condition of $f(x = x_i) = f_i$, it also imposes the condition that the first derivative of f must be continuous at points x_i. This leads to the functional form of f of:

$$f(x) = A(x)f_j + B(x)f_{j+1} + C(x)f''_j + D(x)f''_{j+1} \tag{7}$$

where $A(x)$ and $B(x)$ are the pre-factors to f_j and f_{j+1} given in Eqn. 6, and $C(x)$ and $D(x)$ are easily found as linear functions of A and B. The f_j are given but the second derivatives are set by imposing continuity of the first derivative at all x_i.

This leads to the following expression for each i [1]:

$$\frac{x_i - x_{i-1}}{6} f''_{i-1} + \frac{x_{i+1} - x_{i-1}}{3} f''_i + \frac{x_{i+1} - x_i}{6} f''_{i+1} = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} - \frac{f_i - f_{i-1}}{x_i - x_{i-1}} \tag{8}$$

Note that f''_0 will depend on f''_{-1} and f''_{n-1} will depend on f''_n, both of which are undefined for a vector of n second derivatives. Because of this, natural boundary conditions are imposed so that the second derivatives at the first and last point are zero.

This set of n coupled equations is most easily solved as a matrix where $\underline{der}$ is a vector of all the second derivatives and $\underline{rhs}$ is the right side of Eqn. 8 for each i. The matrix $\underline{\underline{\text{mat}}}$ is derived from the factors before the second derivative terms in Eqn. 8; the i^{th} row of the matrix only contains non-zero terms in the i^{th}, $(i-1)^{th}$, and $(i+1)^{th}$ positions so that the matrix is tri-diagonal. Furthermore, it can be observed from Eqn. 8 that the matrix should have some reflective symmetry about the diagonal, i.e. $mat_{i,i+1} = mat_{i+1,i}$; this provides a useful check to ensure that the matrix has been constructed properly.

However, there are several subtleties due to the natural boundary conditions. Firstly, we are not solving an equation containing a $n \times n$ matrix but instead a $m \times m = (n-2) \times (n-2)$ matrix where the first and last rows and columns have been removed because the first and last derivatives are no longer unknown.

Secondly, the first and last rows of the $m \times m$ matrix depend on f''_0 and f''_{n-1} (which are equal to zero) so these rows of the matrix only contain two terms. Hence they are computed manually after iterating through the other rows. For larger matrices, this is more efficient than incorporating them in to the loop and checking each position if it is the first or last row.

Finally, because the first row of $\underline{\underline{mat}}$ and the first element of $\underline{der}$ have been removed, the i^{th} row of $\underline{\underline{mat}}$ and i^{th} element of $\underline{der}$ actually solve for the $(i+1)^{th}$ second derivative. Hence a value of 1 must be added to the subscript referring to a position in the given data vectors x_i and f_i.

Once $\underline{\underline{mat}}$ and $\underline{rhs}$ have been defined, they are passed to the matrix solver written for question 2 to solve for the vector $\underline{der}$ and zero terms are appended to the first and last positions. Once the second derivative has been found at every point x_i, $f(x)$ can be computed for any x in the range of the given data after finding the value of j by the same method as for the linear interpolator.

The same array of x values is passed to the linear interpolator and cubic spline, with the results shown in Fig. 1

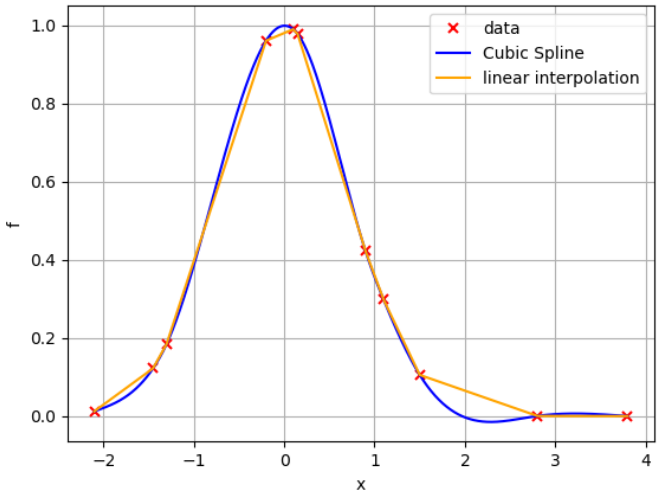

Fig. 1. A data set has been passed to a linear and cubic spline interpolator. It can be seen that the linear interpolator has discontinuities in the first derivative but the spline does not.

IV. Fourier Transforms

The Convolution Theorem states that for two functions $g(t)$ and $h(t)$, the convolution $(g * h)$ is given by

$$(f * g) = \mathscr{F}^{-1}\Big(\mathscr{F}(g)\,\mathscr{F}(h)\,.\Delta t\,.e^{i\phi}\Big) \tag{9}$$

where the Fourier transforms of g and h are functions of frequency and ϕ and Δt are a phase shift and scaling factor.

The function used to Fourier transform was scipy.fftpack.fft and the inverse transform scipy.ffftpack.ifft.

However there are two key factors that need to be considered: padding and aliasing.

The fftpack.fft function assumes the function transformed is periodic with period T so by padding both functions with 0 values will increase the period compared to the relevant values of the function. the condition for circular and linear convolution to be equivalent is for the length of the padded array to be at least twice the unpadded array [2].

Although increasing T would increase the resolution $\Delta\omega$ in the frequency domain, the resolution in the time domain would be decreased by the same factor (as the unpadded function is only over $\frac{N}{2}$) so this does not increase the quality of the convolution.

Padding is also used to ensure that the length of the array is a power of 2, i.e. $N = 2^m$ where m is an integer, but the written code takes the argument m so N is defined to be a power of 2.

Secondly, aliasing occurs when the transformed function (in frequency domain) has non-zero values outside the range $(-\omega_{max}, \omega_{max})$ where $\omega_{max} = \frac{N\pi}{T}$. The range of the transformed functions can be estimated and suitable values of T and ω can be chosen to ensure ω_{max} is large enough to accommodate this.

The Fourier Transform of a box function is a sinc function which falls off on the order of the width of the box function. The transform of a Gaussian is another Gaussian with a width inverse to that of the original. Using these. it can be seen that

$N = 2^{10}$ and $T = 40$ is sufficient for aliasing to be mitigated. The functions are acted over the range -10 to 10 with padding up to position 30.

Finally, a shift and scaling is required. Firstly, the shift is because fftpack.fft (and the inverse function) take the g and h arrays but do not take any information regarding the axes. Hence the function defines the start of the array to be at position $t = 0$. To rectify this, a phase of

$$\phi = -2\pi i \omega t_0 \tag{10}$$

is added, where ω is found using the numpy fftpack.fftfreq function and t_0 is the distance between the first position in the array and the defined zero point.

Secondly, the Δt factor is because the fftpack function assumes each timestep is equal to one and so must be scaled.

the convolution is shown in Fig. 2.

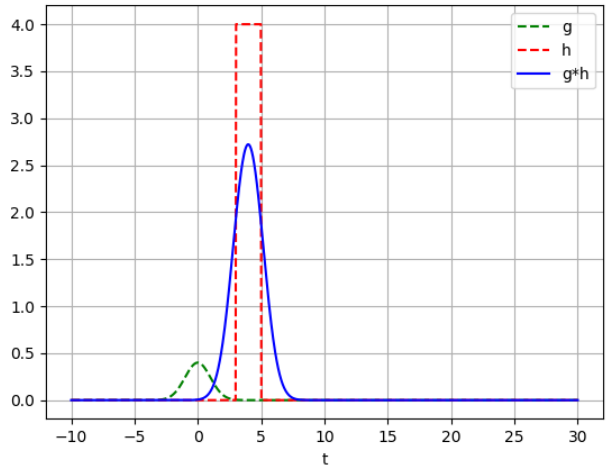

Fig. 2. The convolution of a Gaussian and rectangular function, carried out using the convolution theorem and the scipy fftpack module.

V. Random Numbers

A. Uniform Distribution

The numpy.random.unform function uses a Mersenne Twister to generate random numbers with a period of $2^{19937} - 1$ based on a single seed that can be set using the numpy.random.seed function. The binned distribution of 10^5 random numbers over 100 bins generated between 0 and 1 by this function is given in Fig. 3.

B. Transformation Method

These uniform deviates were used to generate 10^5 random numbers according to a Probability Density Function (PDF):

$$P(x) = \frac{1}{2}\cos\left(\frac{x}{2}\right) \tag{11}$$

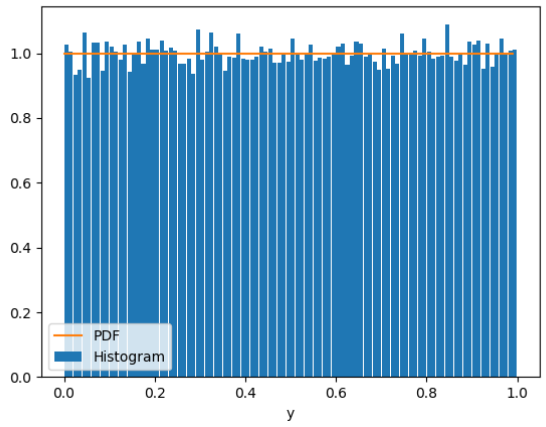

Fig. 3. A histogram of the distribution of 10^5 random deviates over the range 0 to 1. The yellow line represents the constant probability function.

This was achieved using the transformation method, whereby the initial uniform PDF $U(x)$ is transformed to the new PDF $P(x)$. Using the fact that transforming variables cannot change probability, i.e. $U(x).dx = P(y).dy$, then the cumulative distribution functions must also be equal. Because $U(x) = 1$ in the range 0 to 1, it can be shown that

$$y = F^{-1}(x) \tag{12}$$

where x is the uniform deviate, y is the random number distributed according to P, and F^{-1} is the inverse of the Cumulative Distribution Function (CDF).

Seeking F^{-1} for the PDF given in Eqn. 11 first requires finding the corresponding CDF. This is found to be:

$$F(x) = \int_{-\infty}^{x} P(y')\,dy' = \sin\left(\frac{x}{2}\right) \tag{13}$$

Inverting this yields

$$y = F^{-1}(x) = 2\arcsin(x) \tag{14}$$

Passing the uniform deviates to the inverse CDF produces 10^5 random numbers distributed according to $P(x)$ within the range of 0 to π, i.e. the range of arcsin for the domain 0 to 1. This is confirmed by the binned histogram in Fig. 4.

C. Rejection Method

A second method for generating random number according to a given PDF is by rejection. Here random numbers are generated according to the PDF

$$P_2(x) = \frac{2}{\pi}\cos^2\left(\frac{x}{2}\right) \tag{15}$$

over the same domain 0 to 1.

This is done by choosing random points (y_i, p_i), where y_i is distributed according to some comparison function $f(y)$ (which is chosen to be greater than the PDF at all points) over the range y_{min} to y_{max} , and p_i is uniformly distributed from

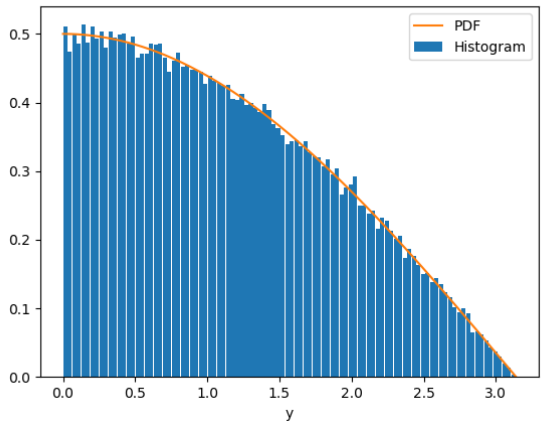

Fig. 4. A binned histogram (100 bins) of 10^5 numbers randomly generated according to $P(x) = 0.5\cos(0.5x)$ by the transformation. The intial uniform deviates were generated using the Mersenne Twister as part of the numpy uniform function.

0 to $f(y_i)$. This is essentially choosing a random number lying below the curve of the comparison function . If the point lies above the PDF – i.e. $p_i > P_2(y_i)$ – then the point is rejected and a new random point is chosen and tested - this is repeated until 10^5 trials are successful.

For the PDF given in Eqn. 15, a suitable comparison function is

$$f_2(x) = \frac{2}{\pi}\cos\left(\frac{x}{2}\right) \tag{16}$$

which lies close above $P_2(x)$ as seen in Fig. 5

In order to carry out the transformation, the function must be normalised before integrating and inverting as per Section V-B. Normalising this yields the same function as in Eqn. 11 and so the inverse CDF is also given by Eqn. 14.

It is easy to see from Fig. 5 that the random numbers are generated according to $P_2(x)$.

Increasing the efficiency of the generator means fewer trials are necessary so the theoretical efficiency of the generator for a large sample can be given as the ratio of the areas of the PDF and comparison function:

$$\epsilon = \frac{\int_{-\infty}^{\infty} P_2(x)}{\int_{-\infty}^{\infty} f_2(x)} \tag{17}$$

Calculating this using the functional forms given in Eqn. 15 and Eqn. 16 over the domain 0 to π yields an efficiency of $\frac{\pi}{4} = 0.785$. This is the probability that a trial is successful so the expected number of trials is the inverse of this multiplied by 10^5 (the number of required successes). This gives an expected number of trials of $127,324$, which is easily compared to the number of trials by implementing a counter in the program. However the exact number of trials depends on the seed of the RTG used. In this case, the number of trials was $127,371$ which gives only a 4% difference to the expected value.

Assuming the time taken to trial and reject a chosen random point is negligible, the time taken for the code to run is given by the time taken to generate a single random number multiplied by the number of numbers generated. Because the comparison function of question 5.c is the same as the PDF of 5.b, the time taken per trial is approximately equal so the ratio of time taken is equal to the ratio of numbers generated, which for 5.b is 10^5 and for $5.c$ is $10^5 \times \frac{1}{\epsilon}$. The time taken for question 5.c is expected to be 1.27 times that of 5.b. However, the timeit function was used (averaged over 10 runs) and measured the time taken for 5.b to be $1.19s$ and the time taken for 5.c as $11.22s$.

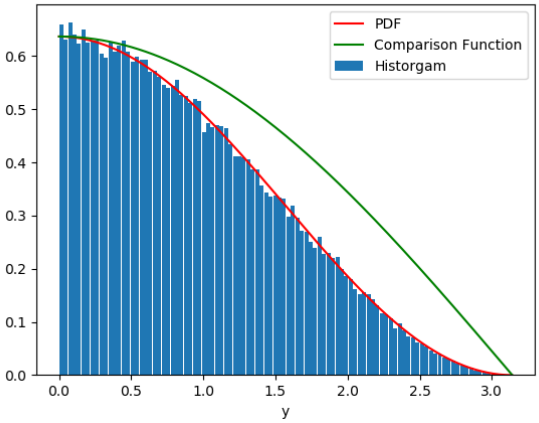

Fig. 5. 10^5 random numbers are generated via the rejection method according to a given PDF. They are chosen according to a comparison function and rejected if above the PDF.

VI. References

1) Imperial College London Physics Department - Computational Physics Notes, Autumn 2019 (Yoshi Uchida nad Mark Scott): https://bb.imperial.ac.uk/bbcswebdav/pid-1691421-dt-content-rid-5391766_1/courses/12074.201910/CP2019-Chapter06-RandomNumbersAndMonteCarloMethods.pdf
2) Mathworks Support - Linear and Circular Convolution: https://uk.mathworks.com/help/signal/ug/linear-and-circular-convolution.html