Computational Physics Assignment

Blackett Laboratory, Imperial College London

November 01, 2019

This assignment was completed using Python with the Numpy, Scipy and Matplotlib packages.

I. FLOATING POINT VARIABLES

The floating-point machine accuracy of a system is quantified by the machine epsilon of the system. Machine epsilon [1] is the smallest number, ε, such that a computer is able to differentiate between 1.0 and $1.0 + \varepsilon$. Theoretical values for ε are shown in Table 1.

Name	Bits	Precision p	Theoretical ε $b^{-(p-1)}$
Single	32	24	$1.192092895 \times 10^{-7}$
Double	64	53	$2.220446049 \times 10^{-16}$
Extended	128	113	$1.925929944 \times 10^{-34}$

TABLE 1: Floating point format specification as defined in the IEEE 754 standard [2] with base $b = 2$. The precision includes one implicit bit, so the theoretical ε is a function of $p - 1$.

We use a linear search algorithm to approximate machine epsilon. Initially, we make a guess $\varepsilon_0 = 1.0$ and check if there exists a smaller number $\varepsilon_1 = 0.5\varepsilon_0$ such that $1.0 + \varepsilon_1 > 1.0$. If ε_1 exists, we update the initial guess $\varepsilon_{n+1} = 0.5\varepsilon_n$ and repeat the check in a while loop. If no such ε_{n+1} exists, we return the current value of ε_n.

The implementation we use can accept a float-generating function as an optional argument. This allows us to quickly check the machine epsilon of different floating point precision numbers. The empirical estimates for ε are found in Table 2.

We find a strong correspondence between the empirical estimates for ε and the theoretical values calculated in Table 1. The real machine epsilon for the default `float` function can be found by passing this as an argument to `np.finfo()` and retrieving the `eps` attribute of the returned value. We find machine epsilon to be $\varepsilon = 2.220446049 \times 10^{-16}$, which corresponds to the theoretical value for double precision floats as expected. Note, the extended precision float in Numpy is machine-dependent with 64 bits on a x64 machine and 80 bits on a x86 machine.

Name	Bits	Empirical ε
`np.float32`	32	1.1920929×10^{-7}
`np.float64`	64	$2.220446049 \times 10^{-16}$
`float`	64	$2.220446049 \times 10^{-16}$
`np.longdouble`	64	$2.220446049 \times 10^{-16}$

TABLE 2: Empirical values of machine accuracy computed using different floating point formats. The default `float` function in Python generates a double precision number.

II. MATRIX METHODS

Crout's method can be used to write a square matrix $\bar{\bar{A}}$ as a product of two triangular matrices $\bar{\bar{A}} = \bar{\bar{L}}\bar{\bar{U}}$ where $\bar{\bar{L}}$ is a lower triangular matrix with ones on its diagonals and $\bar{\bar{U}}$ is an upper triangle matrix. Suppose matrix $\bar{\bar{A}}$ has elements $a_{(i,j)}$ with $i, j = 1, \dots, N$. Then, the lower matrix and upper matrix have elements $l_{(i,j)}$ and $u_{(i,j)}$, respectively, with $l_{(i,i)} = 1$. The expressions for the elements of the lower and upper matrices are [1]:

$$u_{(i,j)} = a_{(i,j)} - \sum_{k=1}^{i-1} l_{(i,k)} u_{(k,j)}$$

$$l_{(i,j)} = \frac{1}{u_{(j,j)}} \left(a_{(i,j)} - \sum_{k=1}^{j-1} l_{(i,k)} u_{(k,j)} \right)$$

where we are required to update the values for each column in order as shown in Fig. 2.1.

An advantage of Crout's algorithm is that each element of $\bar{\bar{L}}$ and $\bar{\bar{U}}$ is only dependent on the elements to the left and above it, as shown in Fig. 2.2. Therefore, matrix $\bar{\bar{A}}$ can be transformed in place without needing to create a new $N \times N$ matrix to store the decomposed values.

One pitfall in Crout's method is an instability in the values of $l_{(i,j)}$ which require a division by $u_{(j,j)}$. A full solution would implement pivoting so that division by zero never occurs, but we will simply raise a `ValueError` for the provided matrix if $u_{(j,j)}$ is sufficiently close to zero.

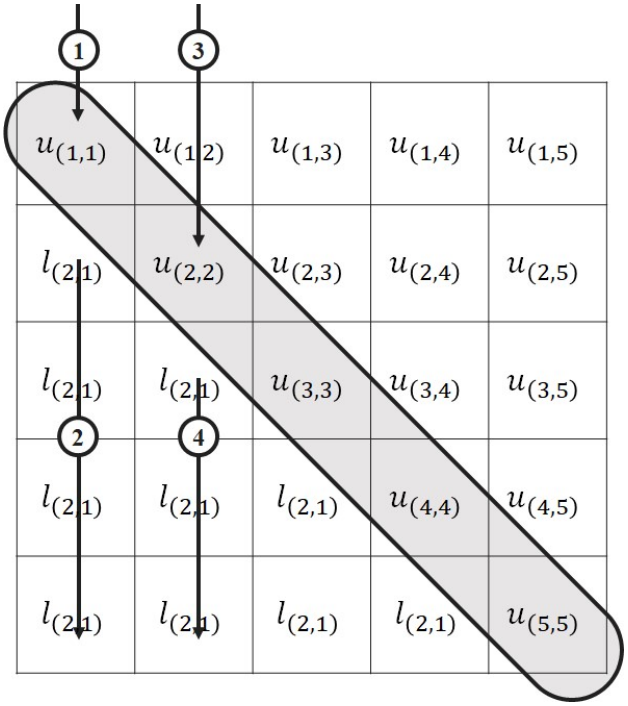

FIG 2.1: The values of the lower and upper matrices are calculated in the order indicated by the arrows.

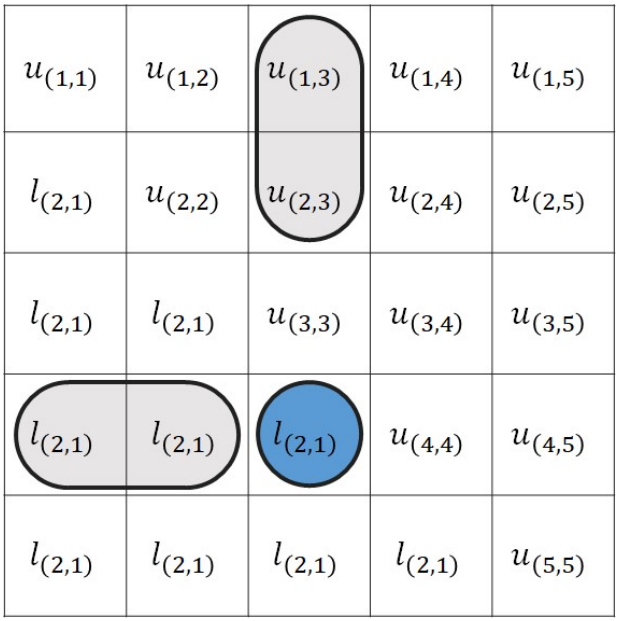

FIG 2.2: The value of each element (highlighted in blue) is calculated from the values highlighted in grey. Therefore, if the matrix is updated in the order shown in Fig. 2.1, the changes can be made in place.

As requested in the brief, the `crout_LU` function returns a transformed matrix $\bar{\bar{A}}^* = \bar{\bar{L}} + \bar{\bar{U}} - \bar{\bar{I}}$ where $\bar{\bar{I}}$ is the $N \times N$ identity matrix. To separate this into its lower and upper triangular components, we use the function `get_lu_matrix`. To calculate the determinant, note $\det(\bar{\bar{A}}) = \det(\bar{\bar{U}}) = \prod_i u_{(i,i)}$ since $\det(\bar{\bar{L}}) = 1$ and the determinant of a triangular matrix is the product of its diagonal elements.

We apply Crout's method to the tridiagonal matrix given and find $\det(\bar{\bar{A}}) = 514032$ with

$$\bar{\bar{L}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1.125 & 1 & 0 & 0 \\ 0 & 0 & -1.4193584 & 1 & 0 \\ 0 & 0 & 0 & -1.21698787 & 1 \end{bmatrix}$$

$$\bar{\bar{U}} = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 0 & 8 & 4 & 0 & 0 \\ 0 & 0 & 15.5 & 10 & 0 \\ 0 & 0 & 0 & 45.19354839 & -25 \\ 0 & 0 & 0 & 0 & 30.57530335 \end{bmatrix}$$

To ensure that the decomposition was performed correctly, we verified that $\bar{\bar{L}}\bar{\bar{U}} = \bar{\bar{A}}$ using `np.matmul(l, u)`. The value of the determinant was also checked using Mathematica.

To find the solution to $\bar{\bar{A}}\vec{x} = \vec{b}$, we implement a forward substitution and backward subtitution routine to first find $\vec{y}$ such that $\bar{\bar{L}}\vec{y} = \vec{b}$ and then solve for $\vec{x}$ such that $\bar{\bar{U}}\vec{x} = \vec{y}$. We can transform $\vec{b} \to \vec{y}$ in place during forward substitution and then $\vec{y} \to \vec{x}$ during backward substitution. The solution to the required equation is given by the vector

$$\vec{x} = \begin{bmatrix} 0.04565708 \\ 0.63028761 \\ -0.51057522 \\ 0.05389159 \\ 0.19613176 \end{bmatrix}$$

To find the inverse of a matrix, we simply solve the equations $\bar{\bar{A}}\vec{x}_i = \vec{b}_i$ where $\vec{x}_i$ is the i-th column of $\bar{\bar{A}}^{-1}$ and $\vec{b}_i$ is the i-th column of the identity matrix. The `invert` function calculates the columns of the inverse matrix and stacks these into a square matrix.

$$\bar{\bar{A}}^{-1} = \begin{bmatrix} 0.380 & -0.046 & 0.004 & -0.005 & -0.002 \\ -0.139 & 0.139 & -0.012 & 0.014 & 0.006 \\ 0.027 & -0.027 & 0.024 & -0.028 & -0.012 \\ 0.070 & -0.070 & 0.062 & 0.044 & 0.018 \\ 0.064 & -0.064 & 0.056 & 0.040 & 0.033 \end{bmatrix}$$

We use `np.matmul(A, invA)` to verify that the product of the two matrices is the identity matrix to within machine accuracy.

III. INTERPOLATION

For interpolation, we sort the tabulated data by its independent variable, which allows us to make use of the efficient `np.searchsorted` function. The interpolation routines return a function that can be

used to find interpolated values without needing to sort the data or solve the derivative matrix equation each time.

For linear interpolation [1], we use the equation $y = A(x)y_{i-1} + B(x)y_i$ with

$$A(x) = \frac{x_i - x}{x_i - x_{i-1}}$$
$$B(x) = \frac{x - x_{i-1}}{x_i - x_{i-1}}$$

For cubic splines [1], we write $y = A(x)y_{i-1} + B(x)y_i + C(x)y''_{i-1} + D(x)y''_i$. To find the second derivatives, y''_i, we impose continuity on the first derivative at each data point, yielding a system of coupled equations for y''_i with $i = 2, \dots, N-1$.

$$\alpha y''_{i-1} + \beta y''_i + \gamma y''_{i+1} = \delta$$

$$\alpha = \frac{x_i - x_{i-1}}{6}$$
$$\beta = \frac{x_{i+1} - x_{i-1}}{3}$$
$$\gamma = \frac{x_{i+1} - x_i}{6}$$
$$\delta = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

For the natural cubic spline, we specify $y''_1 = y''_N = 0$. We write the above system as a matrix equation and use the `crout_LU` and `solve_matrix_eqn` routines to find solutions for y_i''. The values for $C(x)$ and $D(x)$ are

$$C(x) = \frac{1}{6}A(x)[A(x)^2 - 1](x_{i+1} - x_i)^2$$
$$D(x) = \frac{1}{6}B(x)[B(x)^2 - 1](x_{i+1} - x_i)^2$$

The matrix equation is solved and the second derivatives are stored when `cubic_spline` is called. Subsequent calls to the interpolation function only need to compute the expressions for $A(x)$, $B(x)$, $C(x)$ and $D(x)$ for a given x.

The data provided is interpolated with `linear_interp` and `cubic_spline` to produce the graphs in Fig. 3.1.

IV. FOURIER TRANSFORMS

The convolution theorem states that the convolution of a signal $h(t)$ with a response $g(t)$ satisfies the equation $\mathcal{F}(h * g) = \mathcal{F}(h)\mathcal{F}(g)$. We use this principle with the fast Fourier transform algorithm in Numpy to compute the convolution of two discretely-sampled functions.

The Fourier transform of the top-hat signal is a sinc function which is not bandwidth-limited so aliasing will always cause some distortion. We compute the exact Fourier transform of the top-hat signal as

$$\mathcal{F}(h) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} h(t)e^{-i\omega t}\,\mathrm{d}t$$
$$= \frac{8e^{-4i\omega}}{\sqrt{2\pi}}\left(\frac{\sin\omega}{\omega}\right)$$

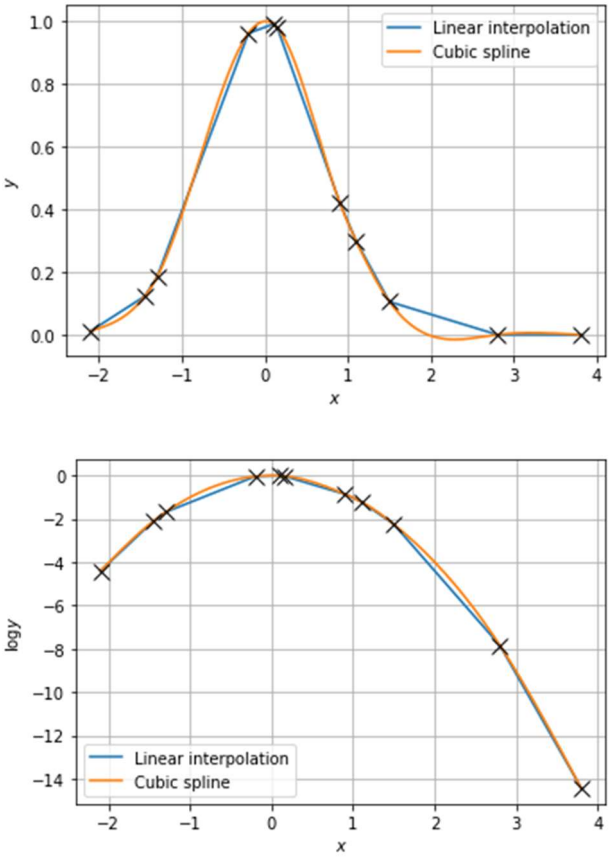

FIG 3.1: A plot of the data provided with linear interpolation (blue) and natural cubic spline interpolation (orange) applied. A plot of $\log y$ against x produces a cubic spline with less oscillatory behaviour due to the gradual change in $\log y$.

Note that the contributions to the frequency decay as ω^{-1} so choosing a high enough sampling rate such that the Nyquist frequency $\omega_{\text{max}} = \pi/\Delta t > 100$ ensures that the aliased frequencies will contribute less than 1% of the contribution by frequencies close to $\omega = 0$. This requires that our sampling rate satisfies $\Delta t < \pi/100 = 0.03$.

The sinc function has a peak contribution at $\omega = 0$ which means we require $\omega_{\text{min}} = 2\pi/T$ to be as small as possible to ensure these contributions are not neglected. Therefore, we also wish to maximise the interval, T, over which we sample the signal function.

We use zero padding to prevent distortion due to end effects. The signal and the response function need

to be sufficiently padded to ensure the periodicity properties of the Fourier transform do not distort the required convolution.

For the response function, it is necessary to convert the function to wrap-around format and apply zero padding to the middle values. We assume that once the height of the Gaussian is a hundredth of its peak height, the contributions can be neglected and $g(t) = 0$ for all values where $g(t) < g(0)/100$. This gives $t > \sqrt{2 \ln 100} = 3.03485$. We choose $t = 3.5$ so that $g(t) = 0$ for $|t| > 3.5$ and this means we require a sampling interval of width $T > 2t = 7$. The zero padded wrap-around response function is shown in Fig. 4.1.

To make use of the efficient FFT algorithm, we require the number of samples to be a power of two. Let us choose a sampling interval of width $T = 22$ centred at $t = 4$, which is the centre of the top-hat function. Our sampling rate must satisfy $\Delta t < 0.03$, so the minimum number of samples required is $N > 22/0.03 = 733$ samples. The smallest power of two greater than 733 is $2^{10} = 1024$ samples. The resulting signal, response and convolution functions are shown in Fig. 4.2.

The convolution requested is straightforward and has an analytic solution given by (see Appendix for derivation)

$$p(t) = 2\left[\text{erf}\left(\frac{t-3}{\sqrt{2}}\right) - \text{erf}\left(\frac{t-5}{\sqrt{2}}\right)\right]$$

A plot on Python is used to verify that the FFT convolution and the analytic solution are indeed the same, as shown in Fig. 4.2.

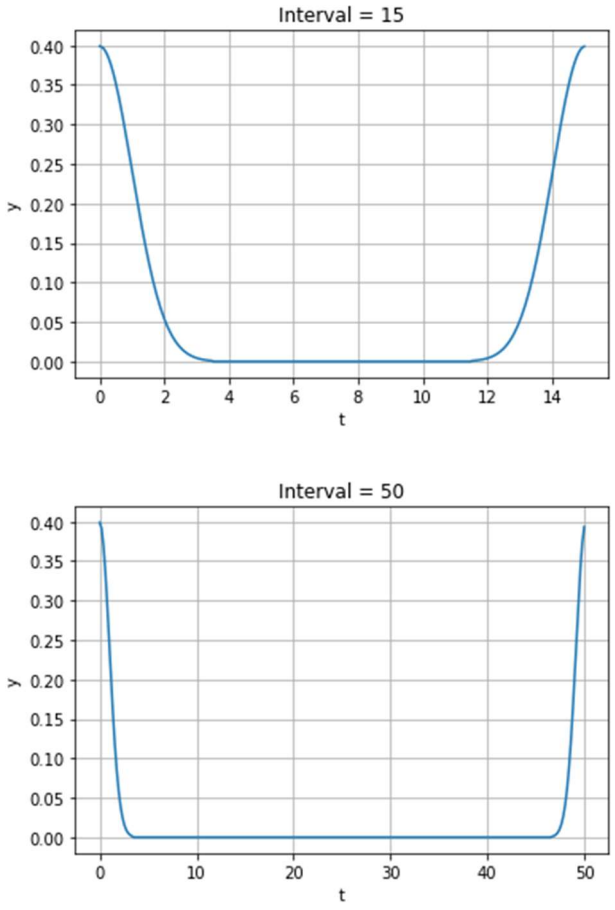

FIG 4.1: Wrap-around format of the Gaussian response function $g(t)$ for sampling intervals $T = 15$ and $T = 50$. The half-width of the Gaussian is taken to be $\Delta t = 3.5$ so $g(t) = 0$ for $|t| > 3.5$. Note that values for $t < 0$ are wrapped around to the opposite side of the sample space and the values in the middle are padded with zeros.

V. RANDOM NUMBERS

For random numbers in Python, the common options are the built-in `random.random` or `np.random.rand` in Numpy. Both implementations use the Mersenne Twister, a deterministic psuedorandom number generator. The speed of the algorithm is comparable to other PRNGs, but its periodicity of order 2^{19937} is significantly greater [3]. We opt to use the Numpy implementation, which offers support for parallelisation when generating a list of random numbers.

We use a random, but fixed, seed and generate 10^5 uniformly distributed random numbers, as shown in Fig. 5.1.

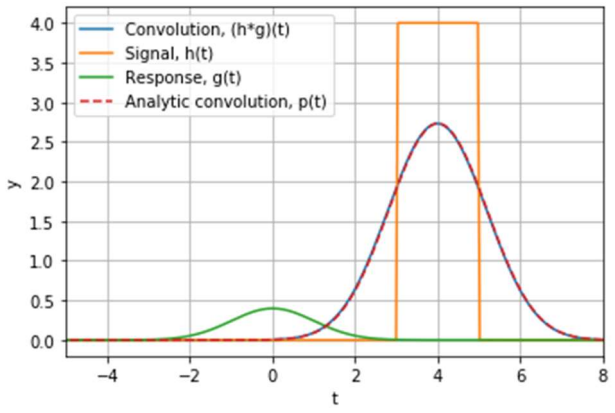

FIG 4.2: A plot of the signal (solid, orange), response (solid, green), FFT convolution (solid, blue) and analytic convolution (dashed, red) for sampling interval $T = 22$ with 1024 samples.

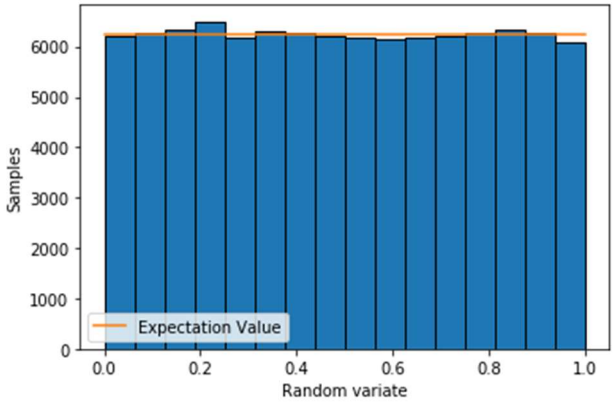

FIG 5.1: A histogram of 10^5 numbers generated from a uniform distribution. The expectation value is shown in orange.

For the transformation method, a PDF, $f(x)$, has cumulative distribution function $F(x)$ given by

$$F(x) = \int_{-\infty}^{x} f(x)\,\mathrm{d}x$$

Suppose $f(x) = (1/2)\cos(x/2)$ with $x \in [0, \pi]$ then we have $F^{-1}(y) = 2\sin^{-1}(y)$. For a list of 10^5 random numbers $x_i \in [0,1]$, $X_i = F^{-1}(x_i)$ are distributed according to $f(x)$. A histogram of X_i and the corresponding CDF comparison is shown in Fig. 5.2.

For the rejection method, we sample 10^5 numbers from a uniform distribution, $u_i \in [0,1]$ and 10^5 numbers, y_i, from a known probability distribution f. To sample from a distribution g, we choose the number M such that $g(y_i)/f(y_i) \leq M$ for all possible y_i. Then, if $u_i < g(y_i)/Mf(y_i)$, we accept u_i as a random sample from g. Otherwise, we generate new values (u_i, y_i) and repeat the procedure.

Let $f(x) = (1/2)\cos(x/2)$ be the comparator distribution and $g(x) = (2/\pi)\cos^2(x/2)$ is the target distribution. By inspection, we find $M = 4/\pi$. A histogram of these numbers and the corresponding CDF comparison is shown in Fig. 5.3.

We accept all uniformly generated u_i which satisfy $u_i < g(y_i)/Mf(y_i)$ (see Fig. 5.4). The efficiency of the algorithm is given by

$$\epsilon = \frac{1}{\pi}\int_0^{\pi} \cos\left(\frac{x}{2}\right)\mathrm{d}x = \frac{2}{\pi}$$

where $\epsilon = 63.7\%$ is the acceptance rate for any (y_i, u_i) pair.

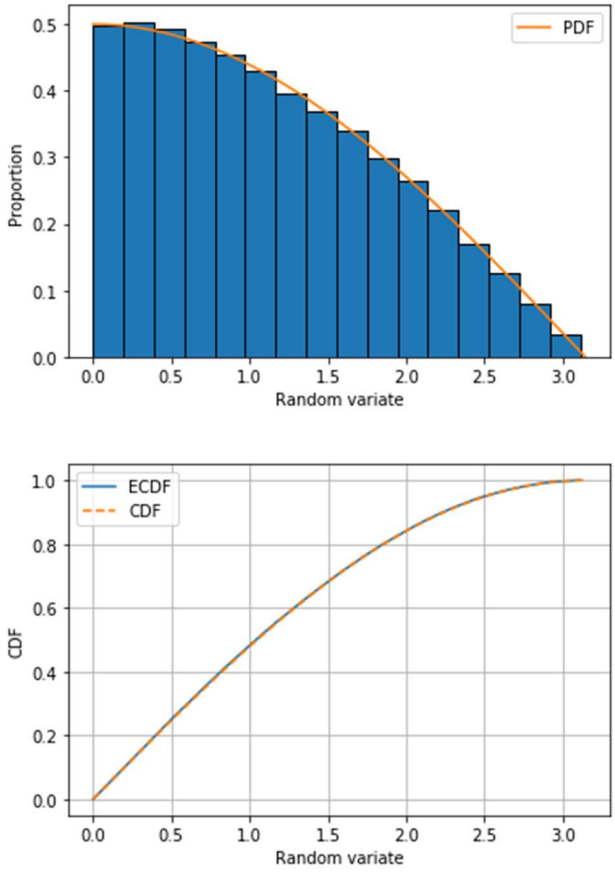

FIG 5.2: A histogram of 10^5 numbers generated using the transformation method according to the distribution $f(x)$, as shown in orange. The second plot shows a strong correlation between the analytic CDF of $f(x)$ (dashed, orange) and the empirical CDF calculated from the random deviates (solid, blue), so the random deviates are distributed according to $f(x)$.

To consider how much longer the rejection method takes, we consider the total number of random numbers that we expect to generate. Initially, we generate 10^5 pairs of numbers (y_i, u_i). Of these pairs, we expect 36.3% pairs to be rejected. We approximate this as a convergent sum and find that we will need to generate a total of 57,080 additional pairs. Therefore, the total number of random numbers generated is 314,160, which is a factor of 3.14 more than with the transformation method. However, an additional computation is required to transform each y_i and this method is not able to make full use of the parallelisation capabilities of `np.random.rand`. Overall, we expect the rejection method to be a factor of six or seven times slower than the transformation method.

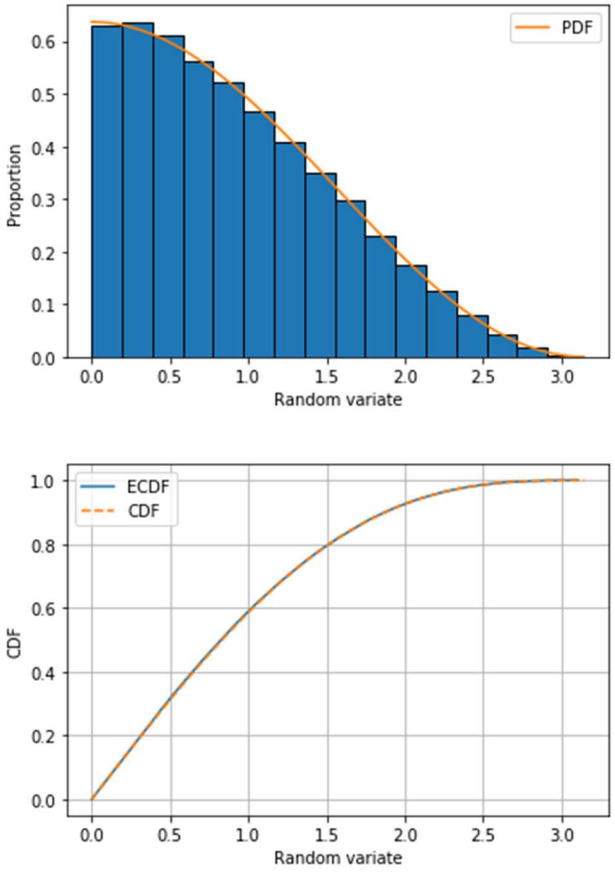

FIG 5.3: A histogram of 10^5 numbers generated using the rejection method according to the distribution $g(x)$, as shown in orange. The second plot shows a strong correlation between the analytic CDF of $g(x)$ (dashed, orange) and the empirical CDF calculated from the random deviates (solid, blue), so the random deviates are distributed according to $g(x)$.

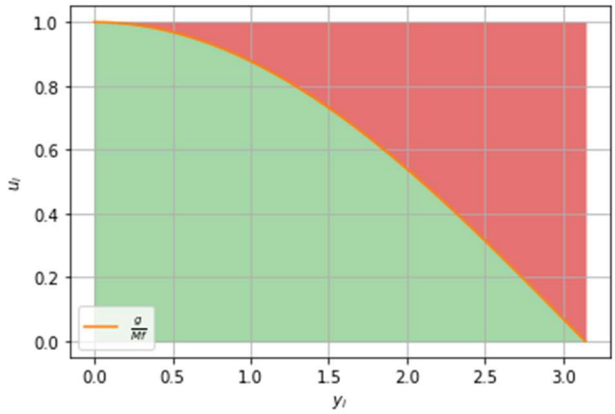

FIG 5.4: A plot of the acceptance area (green) and rejection area (red) for a pair of random deviates (y_i, u_i). The efficiency of the rejection method, ϵ, is the ratio of the acceptance area to the total area.

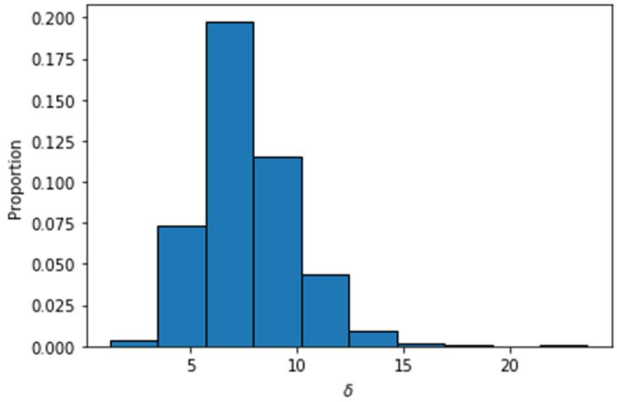

FIG 5.5: A histogram of the retardation factor $\delta = t_2/t_1$ calculated from 1000 trials in generating 1000 numbers using the transformation and rejection methods. The mean retardation factor is $\delta = (7.4 \pm 2)$.

To find the retardation factor $\delta = t_2/t_1$, we compute the times taken, t_1 and t_2, to generate 1000 numbers using the transformation method and rejection method, respectively. The values of δ are plotted as a histogram, as shown in Fig. 5.5. The overall retardation factor is $\delta = (7.4 \pm 2)$, so the rejection method is approximately 7.4 times slower than the transformation method.

REFERENCES

[1] W. Press, Numerical recipes in C, 2nd ed. Cambridge: Cambridge Univ. Press, 2002.

[2] D. Zuras and M. Cowlishaw, "IEEE Standard for Floating-Point Arithmetic", IEEE, p. 13, 2008. Available: https://ieeexplore.ieee.org/document/4610935. [Accessed 6 November 2019].

[3] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator", ACM Transactions on Modeling and Computer Simulation, vol. 8, no. 1, pp. 3-30, 1998. Available: 10.1145/272991.272995.

APPENDIX

The convolution of the top-hat signal $h(t)$ with a Gaussian response $g(t)$ is given by

$$\begin{aligned} h * g &= \int_{-\infty}^{\infty} h(\tau) g(t-\tau)\,\mathrm{d}\tau \\ &= 4\int_{3}^{5} g(t-\tau)\,\mathrm{d}\tau \\ &\overset{k=t-\tau}{\Longrightarrow} 4\int_{t-5}^{t-3} g(k)\,\mathrm{d}k \\ &= \frac{4}{\sqrt{2\pi}}\int_{t-5}^{t-3} \exp\left(-\frac{k^2}{2}\right) dk \\ &= 2\left[\mathrm{erf}\left(\frac{t-3}{\sqrt{2}}\right) - \mathrm{erf}\left(\frac{t-5}{\sqrt{2}}\right)\right] \end{aligned}$$

where we have used the standard result to compute the integral.