Computational Physics: Project A

I Kit Cheng
CID: 00941460
Imperial College London
(Dated: 13^{th} November 2017)

This project used Python to investigate the following topics in computational physics: Floating point variables, matrix methods, interpolation, Fourier transforms and random numbers. The algorithms and programming techniques used as well as the results for each task will be summarised below.

I. TASK 1 - FLOATING POINT VARIABLES

A. Aim

Determine the machine accuracy for single, double and extended precision floats.

B. Method

Machine accuracy is the smallest meaningful number that can be added to or subtracted from one [1]. Thus, it was determined using a for loop to calculate $(1 - 2^{-i})$, where $i = 0, 1, 2, ...$, until the program returned one; meaning that the subtraction at index i was not registered by the computer. The value $2^{-(i-1)}$ would be the accuracy of that float type.

C. Results and Discussion

For a 64-bit Windows operating system (OS), the machine accuracy, single, double and extended precision accuracies were found to be 2^{-53}, 2^{-24}, 2^{-53} and 2^{-52} respectively. Theoretically [2], they should be 2^{-52}, 2^{-23}, 2^{-52} and 2^{-63} respectively. The results were not consistent with theory. Subsequently, it was found that $(1+2^{-i})$ returned results which agreed with theory except for extended precision. This was an artefact of rounding error. Extended precision did not agree in both cases due to the limitation of a 64-bit OS which could only produce accuracies up to double precision. The accuracy is determined by the number of explicit bits available in the fractional part of the mantissa (i.e. 2^i for $i < 0$). For single and double precision, the mantissa has 23 and 52 bits for fractions respectively, plus one implicit bit representing the integer part (2^0) which must always be one. Thus, no extra memory is required. Conversely, extended precision does not have an implicit bit in its 64-bit mantissa, leaving 63 bits available for fractions.

D. Conclusion

Machine accuracy is determined by the float precision accuracy and is limited by the architecture of the OS.

II. TASK 2 - MATRIX METHODS

A. Aim

Decompose an arbitrary N x N matrix A into a lower- (L) and upper- (U) triangular matrix. Find the determinant of A and its inverse A^{-1}. Solve a matrix equation of the form $LU\mathbf{x} = \mathbf{b}$.

B. Method

Object-oriented programming was used to create a Matrix class. This technique has the advantage of keeping the code concise and neat as it removed the necessity to store every attribute of a matrix in a separate variable; instead each attribute can be accessed using the 'dot' notation.

The LU-decomposition routine utilised Crout's method to obtain $A = LU$ [1]. For a N x N matrix A, there are N^2 equations, and $N^2 + N$ unknowns, giving a choice of N free entries. The algorithm chooses to set the diagonal of L to 1s. Then, for each row i starting from the top of both L and U, cycle through columns j. If $i \leq j$ (i.e. in the upper triangle), populate U using

$$U_{ij} = A_{ij} - \sum_{k=0}^{i} L_{ik}U_{kj}. \tag{1}$$

If $i \geq j$ (i.e. in the lower triangle), populate L with

$$L_{ij} = \frac{1}{U_{jj}}(A_{ij} - \sum_{k=0}^{j} L_{ik}U_{kj}). \tag{2}$$

In some cases, pivoting would be required to avoid division by zero from $1/U_{jj}$. The resulting U matrix has an useful property in which the product of its diagonal is the determinant of matrix A.

To solve $A\mathbf{x} = \mathbf{b}$, the solver routine required L, U and $\mathbf{b}$ inputs. L and U were obtained via Crout's method. The equation became $LU\mathbf{x} = \mathbf{b}$. By splitting this up into $L\mathbf{y} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$, forward and back substitution were used to solve for $\mathbf{x}$. First, the first entry in $\mathbf{y}$ was obtained simply by $y_0 = b_0/L_{00}$ as the first row in L were all zeroes except for the first entry. Then, the rest of $\mathbf{y}$

were obtained using the equation

$$y_i = \frac{1}{L_{ii}}(b_i - \sum_{k=0}^{i} L_{ik}y_k), \tag{3}$$

for $i = 1, 2, ..., N-1$. Here, the $\mathbf{y}$ vector was validated by checking the multiplication $L\mathbf{y} = \mathbf{b}$. Using $\mathbf{y}$, the last entry in $\mathbf{x}$ was found simply by $x_{N-1} = y_{N-1}/U_{N-1,N-1}$ as the last row in U were all zeroes except for the last entry. Then, the rest of $\mathbf{x}$ were obtained using the equation

$$x_i = \frac{1}{U_{ii}}(y_i - \sum_{k=i+1}^{N} U_{ik}x_k), \tag{4}$$

for $i = N-2, N-3, ..., 0$.

The j^{th} column of A^{-1} was obtained by solving for $\mathbf{x}$ in $LU\mathbf{x} = \mathbf{b}$ using the solver above, with $\mathbf{b} = \mathbf{e}_j$ (i.e. the j^{th} column of the identity matrix).

C. Results and Discussion

Given matrix A and vector $\mathbf{b}$

$$A = \begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 3 & 8 & 4 & 0 & 0 \\ 0 & 9 & 20 & 10 & 0 \\ 0 & 0 & 22 & 51 & -25 \\ 0 & 0 & 0 & -55 & 60 \end{bmatrix} \mathbf{b} = \begin{bmatrix} 2 \\ 5 \\ -4 \\ 8 \\ 9 \end{bmatrix}, \tag{5}$$

LU-decomposition of A returned

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1.50 & 1 & 0 & 0 & 0 \\ 0 & 1.38 & 1 & 10 & 0 \\ 0 & 0 & 1.52 & 1 & 0 \\ 0 & 0 & 0 & -1.54 & 1 \end{bmatrix} \tag{6}$$

$$U = \begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 0 & 6.50 & 4 & 0 & 0 \\ 0 & 0 & 14.5 & 10 & 0 \\ 0 & 0 & 0 & 35.8 & -25 \\ 0 & 0 & 0 & 0 & 21.6 \end{bmatrix}. \tag{7}$$

Its determinant and inverse were

$$det(A) = 145180.0 \tag{8}$$

and

$$A^{-1} = \begin{bmatrix} 0.712 & -0.141 & 0.046 & -0.017 & -0.007 \\ -0.424 & 0.282 & -0.093 & 0.033 & 0.014 \\ 0.313 & -0.209 & 0.151 & -0.054 & -0.022 \\ -0.245 & 0.164 & -0.118 & 0.078 & 0.032 \\ -0.225 & 0.150 & -0.108 & 0.071 & 0.046 \end{bmatrix}. \tag{9}$$

Finally, the solution to $A\mathbf{x} = \mathbf{b}$ was

$$\mathbf{x} = \begin{bmatrix} 0.338 \\ 1.32 \\ -1.65 \\ 1.71 \\ 1.72 \end{bmatrix}. \tag{10}$$

These results were validated with built-in linear algebra functions in Python.

D. Conclusion

Crout's method was successfully implemented to perform LU-decomposition. It provided an efficient technique to solve matrix equations with the same matrix but different $\mathbf{b}$, and further be used for computing the determinant and inverse.

III. TASK 3 - INTERPOLATION

A. Aim

Perform linear and natural cubic spline interpolation on a tabulated set of x-y data.

B. Method

Both the linear and cubic interpolation routines required three inputs: x-data, y-data, and a value x for interpolation. Using a for loop to cycle through x-data (which must be sorted in ascending order), the index at which $x_i > x$ revealed the two 'sandwiching' data points (x_i,x_{i-1}) either side of x. Using s straight line equation connecting x_i and x_{i-1}, the linear interpolated value at x was found using

$$f = \frac{(x_i - x)y_{i-1} + (x - x_{i-1})y_i}{x_i - x_{i-1}}. \tag{11}$$

The spline algorithm required that the cubic function f was continuous in its value, its first and second derivative $f^{''}$ at each data point. These constraints left over two degrees of freedom. Hence, choosing $f^{''} = 0$ for the start and end data points as boundary conditions resulted in a natural cubic spline (i.e. no curvature at the ends). $f^{''}$ was found by solving the matrix equation $M\mathbf{f}^{''} = \mathbf{F}$ using the solver in *Task 2* where M is a tri-diagonal matrix containing elements a_i (in lower diagonal), b_i (in main diagonal) and c_i (in upper diagonal) given by $\frac{1}{6}(x_i - x_{i-1})$, $\frac{1}{3}(x_{i+1} - x_{i-1})$ and $\frac{1}{6}(x_{i+1} - x_i)$ respectively, and $\mathbf{F}$ is a vector containing the difference in gradient between adjacent pairs of data points. Thus, the spline interpolated value at x was found using

$$f = Ay_{i-1} + By_i + Cf^{''}_{i-1} + Df^{''}_i, \tag{12}$$

where $A = (x_i - x)/(x_i - x_{i-1})$, $B = 1 - A$, $C = \frac{1}{6}(A^3 - A)(x_i - x_{i-1})^2$ and $D = \frac{1}{6}(B^3 - B)(x_i - x_{i-1})^2$.

The interpolation was performed using a high density of points ($n = 1000$) across the data range for a smooth curve.

C. Results and Discussion

Figure 1 shows the expected straight lines between data points for linear interpolation and a smooth curve for cubic spline. With sufficient magnification on a data point, the linear line missed the point more than the cubic spline. The spline was not truly smooth either due to a finite number of interpolation points. These errors could be reduced by increasing the density of points.

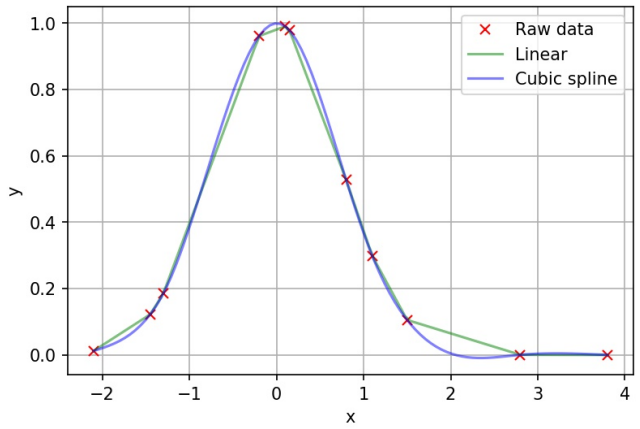

FIG. 1. A plot of the tabulated x-y data with linear and natural cubic spline interpolation evaluated at 1000 uniformly spaced points between the range of x-data.

D. Conclusion

Cubic spline interpolation provided a more physical function than linear interpolation for finding estimates of values within the range of known data points.

IV. TASK 4 - FOURIER TRANSFORM

A. Aim

Perform convolution of top-hat signal with Gaussian response using fast Fourier transform (FFT) and convolution theorem.

B. Method

The signal $h(t)$ and response $g(t)$ were sampled $N = 2^8$ times over a period of $T = 8s$ from $t = -T/2$ to $t = T/2$, with sample spacing $dt = T/N$. N was chosen to be a power of two to optimise the FFT algorithm since it works by continually splitting the discrete Fourier transform (DFT) sum containing N terms into two halves until a single term is reached. Since h and g were non-band-limited functions (i.e. contains infinite frequencies), their FFT would suffer from aliasing regardless of sample size. Thus, N was chosen to be sufficiently large such that the maximum frequency ($\omega_{max} = \pi/dt$) was relatively high to minimise the effect of having too high an amplitude at lower frequency bins due to aliasing. Before performing FFT on h and g, the arrays were padded with $N/2$ zeroes on each side (i.e. total of $2N$ samples- still power of two) to prevent circular convolution [3] where the convolution output repeats. Padding ensures the FFT is long enough to prevent this. Using convolution theorem, backward FFT was performed on the product in the frequency domain to obtain the convolution in the time domain. The output was shifted using *numpy.fft.fftshift* to centre around the signal and was multiplied by dt for normalisation.

C. Results and Discussion

Figure 2 shows that the convolution resulted in a smoothed-out signal with lower amplitude than the original top-hat signal as expected from convoluting with a Gaussian response.

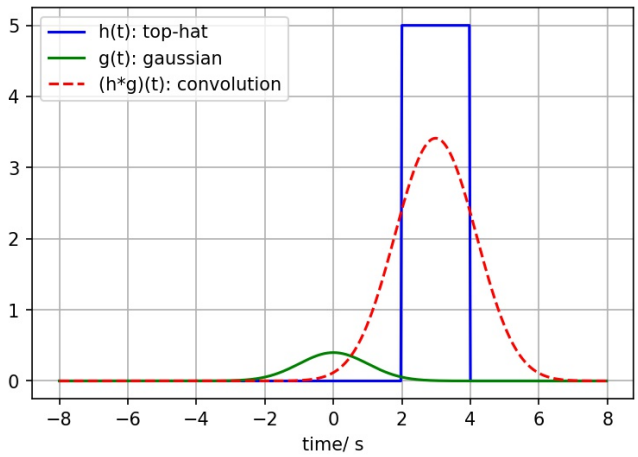

FIG. 2. A plot showing the convolution between a top-hat function h(t) and a normalised Gaussian g(t) performed via Fast Fourier Transform using *numpy.fft* and *numpy.fft.ifft*.

D. Conclusion

FFT convolution provided an easy and efficient way to perform convolutions, but careful consideration of sampling, aliasing and padding was needed.

V. TASK 5 - RANDOM NUMBERS

A. Aim

Generate $N = 10^5$ pseudo-random numbers based on a single seed using uniform deviate generator, transformation method and rejection method for different probability density functions (pdf).

B. Method

The built-in Python uniform deviate generator *numpy.random.uniform* was used to generate a uniform distribution over the interval $x \in [0, 1]$, with seed *numpy.random.seed(2)*.

For $pdf(y) = 0.5\sin(y)$ over interval $y \in [0, \pi]$, the transformation method was used to map the uniform pdf onto the new one based on the seed from above. By definition the pdf's area is one. Thus, the area of the new pdf from 0 to y must equal to the area under the uniform pdf from 0 to x. Mathematically,

$$\int_0^y 0.5\sin(\tilde{y})d\tilde{y} = \int_0^x d\tilde{x}. \tag{13}$$

This was rearranged to

$$y(x) = \arccos(1 - 2x). \tag{14}$$

For $pdf(y) = \frac{2}{\pi}\sin^2(y)$ over interval $y \in [0, \pi]$, the rejection method was used based on a seed defined outside of a while loop with condition ($counter < 1e10$) to prevent an infinite loop. A random number y_i was generated using the pdf from transformation method as their similarity in shape would reduce the number of rejections. A comparison function $f(y) = \sin(y)$ was defined such that $f(y) > pdf(y)\ \forall y$. Another random number z_i was generated over interval $[0, f(y_i)]$ using *numpy.random.uniform*. If $pdf(y_i) < z_i$, then y_i was rejected, otherwise accepted. This process was repeated under a while loop until 10^5 accepted values were produced. This algorithm works because regions where $pdf(y)$ is much smaller than $f(y)$ would be rejected more often than regions with similar values, giving the desired distribution.

Using *time.clock* at the start and end of each method gave the time taken to produce 10^5 random numbers.

The resulting distributions were plotted in a histogram with 100 bins and was normalised.

C. Results and Discussion

Figure 3 shows good agreement between the obtained distributions and the desired pdfs. The ratio of time taken: Rejection/Transformation ≈ 400, which was much higher than the theoretical value of 2 calculated from the inverse of the area ratio $A_{pdf(y)}/A_{f(y)}$ [1]. It represents the average number of times the rejection algorithm is performed to obtain an accepted y_i. However, this was an unfair comparison since the transformation method did not require any loops or if-statements which are relatively slow.

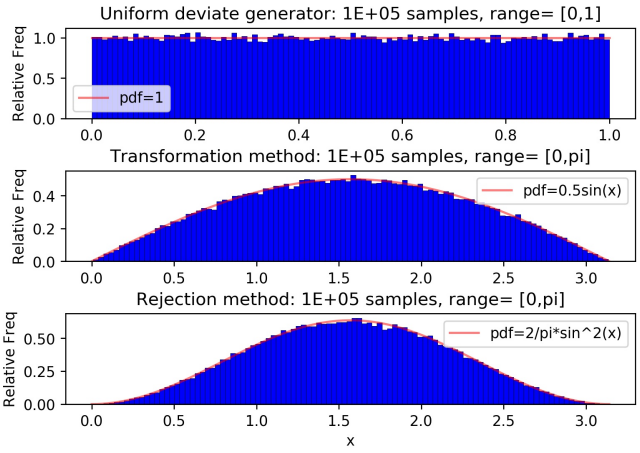

FIG. 3. Three subplots showing the distribution of random numbers with different pdf: uniform distribution using uniform deviate generator (top), $pdf = 0.5sin(x)$ using transformation method (middle), and $pdf = \frac{2}{\pi}\sin^2(x)$ using rejection method (bottom). N.B. (no. of samples / bin) = (relative freq) x N / (no. of bins)

D. Conclusion

The three pseudo-random number generators produced distributions which followed closely to the desired pdfs. The rejection method was inferior in efficiency compared to the transformation method but has the advantage of mapping to any pdf.

Word Count = 1684

[1] Uchida Y., Scott P. (2017) *Lecture Notes 2017*. Blackboard Imperial. Available from: `https://bb.imperial.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_12315_1content_id=_1216038_1` [Accessed 10th November 2017]

[2] IEEE Standard for Floating-Point Arithmetic (2008), *IEEE Std 754-2008* , vol., no., pp.1-70, doi: 10.1109/IEEESTD.2008.4610935

[3] Smith S. (2011) *Chapter 18: FFT Convolution*. The Scientist and Engineer's Guide to Digital Signal Processing. Available from: `http://www.dspguide.com/ch18/2.htm` [Accessed 12th November 2017]