# Investigating into the basics of Computational Physics

Kei Tsun Yeung (CID: 01054596)

13-11-2017

## Abstract

In this project, five basic aspects of Computational Physics were studied: precision of floating point variables, LU decomposition, interpolation using linear and cubic spline methods, Fourier transforms and convolutions, and random number generation. The results produced mostly matched with theoretical prediction, and in particular, the machine accuracy of Python was found to be $2.22 \times 10^{-16}$ for a 64-bit float, which corresponded to $2 \times 10^{-52}$, showing that the mantissa was 52 bits plus a hidden bit.

## 1 Floating point variables

### 1.1 Introduction and Theory

A floating point variable is stored in single (32-bit), double (64-bit) or extended (128-bit) precision. In each format, there are three parts: a sign bit, an exponent, and a mantissa [1], which contain the number of bits as shown in Table 1 [1].

| Variable type | Sign bit $\pm$ | Mantissa (1).10101 | Exponent $\times 2^n$ |
|---|---|---|---|
| Single (32-bit) | 1 | 23 (+1 hidden) | 8 |
| Double (64-bit) | 1 | 52 (+1 hidden) | 11 |

Table 1: The number of bits for different parts of a floating point variable.

The exponent allows for powers of 2 to be multiplied onto the number. The mantissa stores the actual value of the floating point, and the value in each bit ranges from $2^{-1}$ to $2^{-23}$. Using the IEEE-754 format where the float is stored in scientific notation, there is one extra bit that is not stored (the hidden bit), which is assumed to be 1 as the leading bit ($2^0$) cannot be zero [2].

The machine accuracy is defined as the smallest number that can be meaningfully added to 1. The exponent of the number to be added to 1 has to be changed so that it is the same as the exponent of 1. The mantissa needs to be shifted to keep the float at the same value. Since the maximum number of bits that can be shifted before any meaningful value is lost is equal to the length of the mantissa, machine accuracies of $2^{-23}$ and $2^{-52}$ are expected for single and double precision respectively.

### 1.2 Method

The machine accuracy of floating point variables in Python was found by dividing a test number $i$ by 2 until $1 + i = 1$ due to the finite precision as described above. The number $i$ had an initial value of 1.

The variable types used were `float`, `float32`, `float64` and `longdouble`. Using Python for Windows, however, the extended precision cannot be studied because the `longdouble` variable type is set to double precision (as confirmed below) and the `float128` option is not available [3].

### 1.3 Results and Discussion

The machine accuracies of different variable types were found as shown in Table 2.

| Variable type | Machine accuracy |
|---|---|
| `float` | $2.22 \times 10^{-16} \approx 2^{-52}$ |
| `float32` | $1.19 \times 10^{-7} \approx 2^{-23}$ |
| `float64` | $2.22 \times 10^{-16} \approx 2^{-52}$ |
| `longdouble` | $2.22 \times 10^{-16} \approx 2^{-52}$ |

Table 2: Machine accuracies of four different float variable types.

This confirmed that the default setting of a `float` variable is in double precision. All the above four types had a hidden bit within the mantissa. Without the hidden bit, the machine accuracy was expected to be double the above value, because there would be one less bit. So using the hidden bit, the precision of a floating point variable can be doubled.

## 2 Matrix methods - LU decomposition

### 2.1 Introduction and Theory

LU decomposition is a method in linear algebra to solve a system of linear equations written in matrix form $\mathbf{A}\underline{x} = \underline{b}$, where $\mathbf{A}$ is an $N \times N$ matrix containing the coefficients for each value in unknown vector $\underline{x}$, and $\underline{b}$ is a vector containing the constants. If $\mathbf{A}$ is a square matrix and can be separated into a product of upper and lower triangular matrices $\mathbf{L}$ and $\mathbf{U}$ respectively such that

$$\mathbf{A} \equiv \mathbf{LU}, \tag{1}$$

where $\mathbf{L}$ and $\mathbf{U}$ are represented by

$$\begin{pmatrix} \alpha_{00} & 0 & \cdots & 0 \\ \alpha_{10} & \alpha_{11} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{N-1,0} & \alpha_{N-1,1} & \cdots & \alpha_{N-1,N-1} \end{pmatrix} \tag{2}$$

and

$$\begin{pmatrix} \beta_{00} & \beta_{01} & \cdots & \beta_{0,N-1} \\ 0 & \beta_{11} & \cdots & \beta_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \beta_{N-1,N-1} \end{pmatrix} \tag{3}$$

respectively, then letting

$$\underline{y} = \mathbf{U}\underline{x}, \tag{4}$$

the equation

$$\mathbf{L}\underline{y} = \underline{b} \tag{5}$$

can be obtained [4].

Denoting the elements in the vectors $\underline{b}$, $\underline{x}$ and $\underline{y}$ as $b_i$, $x_i$ and $y_i$ respectively, where $i = 0, 1, 2, \cdots, N-1$, a forward substitution can be used to solve for each element in $\underline{y}$ as in equations 6 and 7.

$$y_0 = \frac{b_0}{\alpha_{00}} \tag{6}$$

$$y_i = \frac{1}{\alpha_{ii}}\left(b_i - \sum_{j=0}^{i-1} \alpha_{ij} y_j\right), i = 1, 2, \cdots, N-1. \tag{7}$$

A backward substitution can then be used to solve for $\underline{x}$ as in equations 8 and 9.

$$x_{N-1} = \frac{y_{N-1}}{\beta_{N-1,N-1}} \tag{8}$$

$$x_i = \frac{1}{\beta_{ii}}\left(y_i - \sum_{j=i+1}^{N-1} \beta_{ij} x_j\right), i = N-2, N-3, \cdots, 0. \tag{9}$$

Each $y_i$ must be solved in ascending order and each $x_i$ must be solved in descending order.

The convention using Crout's method is to fix the diagonal elements of the lower triangular matrix $\alpha_{ii}$ to 1 [5]. Then the other elements can be found as

$$\beta_{ij} = a_{ij} - \sum_{k=0}^{i-2} \alpha_{ik}\beta_{kj}, \tag{10}$$

where $j = 0, 1, 2, \cdots, N-1$, $i = 0, 1, 2, \cdots, j$ and $a_{ij}$ are the elements in matrix $\mathbf{A}$, and

$$\alpha_{ij} = \frac{1}{\beta_{ij}}\left(a_{ij} - \sum_{k=0}^{j-2} \alpha_{ik}\beta_{kj}\right), \tag{11}$$

for $i = j+1, j+2, \cdots, N-1$.

### 2.2 Method

A system of 5 linear equations, expressed as $\mathbf{A}\underline{x} = \underline{b}$ with [6]

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

and

$$\underline{b} = \begin{pmatrix} 2 \\ 5 \\ -4 \\ 8 \\ 9 \end{pmatrix}$$

was solved by decomposing $\mathbf{A}$ using the procedure as described in section 2.1. The unknown values were then calculated using forward and backward substitution.

The determinant of $\mathbf{A}$ was found by multiplying the diagonal elements of $\mathbf{U}$ as the determinant of a triangular matrix is the product of the diagonal entries.

The inverse of $\mathbf{A}$ was also found using this method, by separating the identity matrix $\mathbf{I}$ into five column vectors,

$$\underline{b}_0 = \begin{pmatrix}1\\0\\0\\0\\0\end{pmatrix}, \underline{b}_1 = \begin{pmatrix}0\\1\\0\\0\\0\end{pmatrix}, \underline{b}_2 = \begin{pmatrix}0\\0\\1\\0\\0\end{pmatrix},$$

$$\underline{b}_3 = \begin{pmatrix}0\\0\\0\\1\\0\end{pmatrix}, \underline{b}_4 = \begin{pmatrix}0\\0\\0\\0\\1\end{pmatrix},$$

calculating the corresponding $\underline{x}$ for each column vector and then combining the columns together to give $\mathbf{A}^{-1}$.

## 2.3 Results and Discussion

(In this section, all results are rounded off after 3 significant figures.)

The lower and upper triangular matrices were found to be

$$\mathbf{L} = \begin{pmatrix}1 & 0 & 0 & 0 & 0\\1.5 & 1 & 0 & 0 & 0\\0 & 1.38 & 1 & 0 & 0\\0 & 0 & 1.52 & 1 & 0\\0 & 0 & 0 & -1.54 & 1\end{pmatrix}$$

and

$$\mathbf{U} = \begin{pmatrix}2 & 1 & 0 & 0 & 0\\0 & 6.5 & 4 & 0 & 0\\0 & 0 & 14.5 & 10 & 0\\0 & 0 & 0 & 35.8 & -25\\0 & 0 & 0 & 0 & 21.6\end{pmatrix}$$

These were then checked by multiplying to give $\mathbf{A}$. The results were different compared with the `scipy.linalg.lu` package because there was a permutation matrix $\mathbf{P}$ involved with pivoting. While this was not necessary in this case, decomposition would be impossible without pivoting if one of the rows had a diagonal element of zero. This can be seen from equation 9, as the element $\beta_{ii}$ calculated using the diagonal element is present.

The determinant was then found to be 145180.0, which matched the result found using the matrices generated by the `scipy.linalg.lu` package. The unknown vector $\underline{x}$ in the equation $\mathbf{A}\underline{x} = \underline{b}$ was then solved to be

$$\underline{x} = \begin{pmatrix}0.338\\1.32\\-1.65\\1.71\\1.72\end{pmatrix},$$

which multiplied with matrix $\mathbf{A}$ to give vector $\underline{b}$ as required. The inverse of $\mathbf{A}$ was then found to be

$$\mathbf{A}^{-1} = \begin{pmatrix}0.712 & -0.141 & 0.0464 & -0.0165 & -0.00689\\-0.424 & 0.282 & -0.0929 & 0.0331 & 0.0138\\0.313 & -0.209 & 0.151 & -0.0537 & -0.0224\\-0.245 & 0.164 & -0.118 & 0.0777 & 0.0324\\-0.225 & 0.150 & -0.108 & 0.0712 & 0.0463\end{pmatrix}$$

which multiplied with matrix $\mathbf{A}$ to give the identity matrix. This showed that the LU decomposition is an effective way in solving systems of linear equations.

# 3 Interpolation

## 3.1 Introduction and Theory

Interpolation means finding the value of a function at an arbitrary point in a range, given a set of known data points [7].

### Linear method

The linear method is to connect all adjacent points with straight lines. Let $(x_i, f_i)$ and $(x_{i+1}, f_{i+1})$ be the coordinates of the two adjacent points. The function $f(x)$ at any given point $x$ between these two points can be found as [7]

$$f(x) = \frac{(x_{i+1} - x)f_i + (x - x_i)f_{i+1}}{x_{i+1} - x_i} \tag{12}$$

or

$$f(x) = A(x)f_i + B(x)f_{i+1}, \tag{13}$$

with

$$A(x) \equiv \frac{x_{i+1} - x}{x_{i+1} - x_i} \tag{14}$$

and

$$B(x) \equiv 1 - A(x) = \frac{x - x_i}{x_{i+1} - x_i}. \tag{15}$$

### Cubic spline

The linear interpolation has discontinuities at every data point as the slope is changed abruptly. To match the higher derivatives, the cubic spline method can be used, which matches a polynomial function across two data points.

However, there are four degrees of freedom associated with a cubic function, so four data points are needed for each section, two of which are on either side of the interval [7]. The equation for the cubic spline can then be found to give the general expression

$$f(x) = A(x)f_i + B(x)f_{i+1} + C(x)f''_i + D(x)f''_{i+1}, \tag{16}$$

with

$$C(x) \equiv \frac{1}{6}(A(x)^3 - A(x))(x_{i+1} - x_i)^2 \tag{17}$$

and

$$D(x) \equiv \frac{1}{6}(B(x)^3 - B(x))(x_{i+1} - x_i)^2. \tag{18}$$

By matching the first and second derivatives of each end of the section, the expression

$$\frac{x_i - x_{i-1}}{6} f''_{i-1} + \frac{x_{i+1} - x_{i-1}}{3} f''_i + \frac{x_{i+1} - x_i}{6} f''_{i+1} = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} - \frac{f_i - f_{i-1}}{x_i - x_{i-1}} \tag{19}$$

can be obtained, where $i = 1, \cdots, n-1$ for $n+1$ second derivatives $f''_{0\cdots n}$. This means that two of the second derivatives need to be set, and using the natural spline convention, $f''_0 = f''_n = 0$ is chosen. This can then be converted into a system of $n-1$ linear equations and solved.

## 3.2 Method

A set of 11 data points was studied using linear and cubic spline interpolation. Throughout the whole line, the sample points had a distance of 0.01 on $x$-axis.

The LU decomposition solving function from section 2 was then used to solve for the double derivatives.

## 3.3 Results and Discussion

The interpolation of the 11 data points using linear and cubic spline methods were as shown in Figure 1.

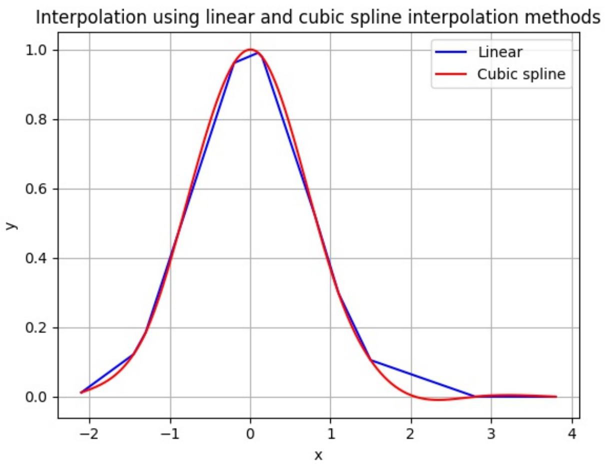

Figure 1: The interpolated curves of 11 data points using linear and cubic spline interpolation methods.

Discontinuities from the linear interpolation were smoothed out on the cubic spline. So, the cubic spline method is effective in generating a smooth curve.

Comparing the cubic spline method with polynomial interpolation, the interpolation error of splines are in general lower, because the interpolation was done in piecewise fashion, instead of taking into account all data points at once as in polynomial. This avoids problems such as Runge's phenomenon [8] which occurs when interpolating between a set of equidistant points using a polynomial to interpolate.

# 4 Fourier transforms

## 4.1 Introduction and Theory

A Fourier transform requires an integration over an infinite time period in order to determine all the frequencies contained in the wave up to infinity [9]. On a computer, however, it is impossible to perform an integration with infinite limits. A discrete calculation can be obtained via the Fast Fourier Transform (FFT).

In FFT, the function is only sampled for a period $T$, after which the function is assumed to repeat the same pattern. This results in a minimum frequency [9]

$$\omega_{min} = \frac{2\pi}{T} \equiv \triangle\omega, \tag{20}$$

which is also the difference between each discrete frequency values. Within the period $T$, the time interval sampled $\triangle t > 0$, so there is a maximum frequency which corresponds to the shortest wave that can fit into $2\triangle t$ of time (3 points are needed to identify a wave with a unique frequency) given by the Nyquist frequency

$$\omega_{max} = \frac{\pi}{\triangle t} = \triangle\omega \frac{N}{2}, \tag{21}$$

where $N$ is the number of sampling points within a period.

The FFT reduces the number of operations required by spliting up the sum into even and odd sections repeatedly, until $N$ functions of period 1 are left [9], which can then be Fourier transformed trivially. As the function is repeatedly halved, the number of sampling points $N$ is best to be a power of 2 to increase the speed of the operation.

The discrete nature of FFT raises two concerns: sampling and aliasing. The time interval between sample points $\triangle t$ limits the bandwidth to which information is stored to the Nyquist frequency $\omega_{max}$. So, this needs to be chosen carefully so that information is not lost. If the function has frequency components $|\omega| > \omega_{max}$ then this will be expressed as a sum of waves with frequency below $\omega_{max}$. This effect is known as aliasing, and causes distortion on the Fourier transformed function by increasing the values on some of the frequencies.

## 4.2 Method

A convolution between a rectangular function defined by [6]

$$\begin{aligned} h(t) &= 5 \quad \text{for} \quad 2 \leq t \leq 4 \\ h(t) &= 0 \quad \text{otherwise} \end{aligned}$$

and a Gaussian response function

$$g(t) = \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}}$$

was studied in the range of -10 to 10 s (i.e. period $T = 20$ s) using FFT provided by the `numpy.fft` package.

Using a number of sample points $N = 2^{10} = 1024$ to optimise the speed of the program, the time between sample points $\triangle t$ was 0.0235 s, $\triangle\omega = 0.314$ rad/s, and $\omega_{max} = 134$ rad/s. In order to test whether the time for Fourier transform with $N = 2^m$ was much shorter, the

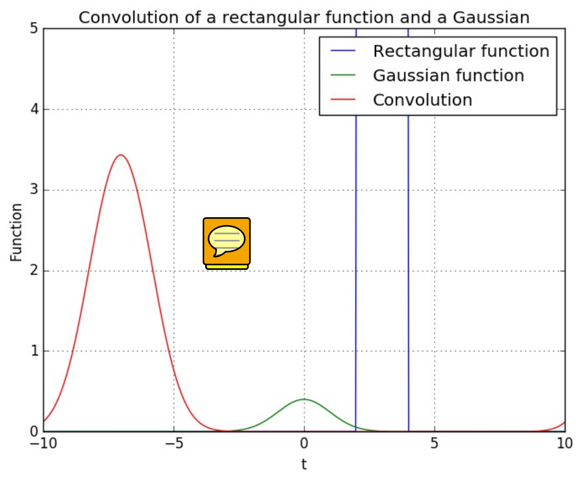

Figure 2: The convolution of a rectangular function and a Gaussian, with the original functions plotted.

Fourier transform times were measured for $N = 1023, 1024, 1025$ using the `time.clock` package.

The normalisation of the convoluted curve had to be done manually because two Fourier transforms were done, however only one inverse was performed after convolution. The factor required was found to be $\triangle t$ and this was multiplied to the result.

## 4.3 Results and Discussion

Figure 2 shows the combined diagram of the rectangular function, the Gaussian and their convolution.

The area of the convoluted function should match the product of the areas of the two functions before convolution. To check this, trapezoidal rule (`np.trapz`) was used to integrate the three areas over the region of interest and the areas of the convolution and the rectangular function were equal while the Gaussian was normalised, so the convolution had a correct normalisation factor.

Aliasing can be seen at the start and the end of the convolution, where there is an unexplainable rise.

The functions were run for $N = 1023, 1024$ and 1025 and the time required to run for $N \neq 1024$ were about 2.2 times and 2.6 times the time required for $N = 1024$ for the rectangular and Gaussian functions respectively. So, a number of sample points with a power of 2 is preferred.

# 5 Random number generation

## 5.1 Introduction and Theory

A uniform variation is the situation in which all values within the range are equally likely to occur. From a uniform variate generator, non-uniform distributions can be generated using transformation or rejection methods [10].

### Transformation method

The transformation method transforms the uniform variate $x$ into a non-uniform random variable $y$ using cumulative distribution function (CDF), which can be defined as

$$\begin{aligned} F(y) &= \int_{-\infty}^{y} P(y')dy' && (22) \\ &= \int_{-\infty}^{y} \frac{dx'}{dy'}dy' && (23) \\ &= \int_{0}^{x} dx' = x && (24) \end{aligned}$$

Separating out $y$ in terms of $x$ will yield a relation to transform into the non-uniform distribution.

### Rejection method

The rejection method is used when the above cannot be inverted. In this case, a comparison function $f(y)$ is determined which has all $f(y) > P(y)$ for all $y$ within the range of interest. A random number $y_i$ is picked within the range of $y$ allowed (say from $y_{min}$ to $y_{max}$) using the normalised PDF of $f(y)$. The probability density of this number $P(y_i)$, and also the probability density on the comparison function $f(y_i)$ are then calculated. A random number $p_i$ between 0 and $f(y_i)$ is then picked. If $P(y_i) > p_i$ this value is accepted. Over a huge amount of trials, the resultant distribution should tend towards the desired PDF.

## 5.2 Method

Two modules were tested in generating in a uniform variate between 0 and 1, `random.random` and `np.random.uniform` . It was found that the `numpy` module was a lot more troublesome as only integer inputs were allowed for the seed

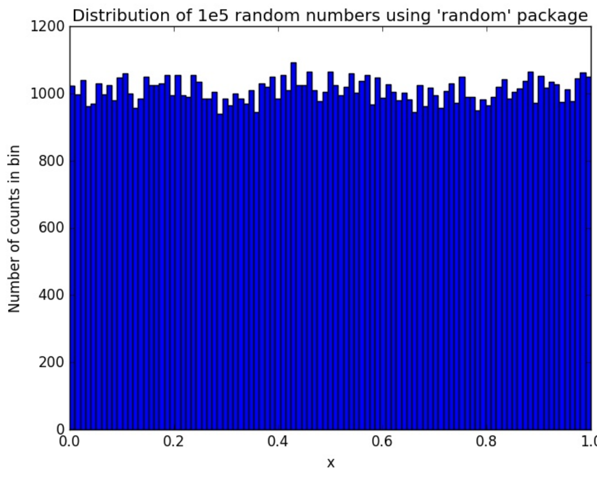

Figure 3: A uniform distribution in the range $0 \leq x \leq 1$ generated using `random.random` package.

(i.e. only 0 and 1). Therefore it was decided that the `random` package would be used.

Using a seed value of 1.0, $10^5$ samples uniformly distributed in the range of 0 to 1 were generated for this and all distributions below. This was then converted to a sinusoidal distribution given by

$$\text{PDF}(y) = \frac{1}{2}\sin y.$$

Using the method described, the CDF was found to be

$$F(y) = -\frac{1}{2}(\cos y - 1) = x$$

which was inverted to give

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

This was then used as the comparison function of the rejection method being investigated for the sine squared distribution given by

$$\text{PDF}(y) = \frac{2}{\pi}\sin^2 y,$$

changing the scaling factor from $1/2$ to $2/\pi$.

The running times of these two methods were then timed and the ratio of the times was expected to be the efficiency in the rejection method, i.e. the fraction of points accepted over the number of trial points.

## 5.3 Results and Discussion

The uniform distribution was generated as shown in Figure 3.

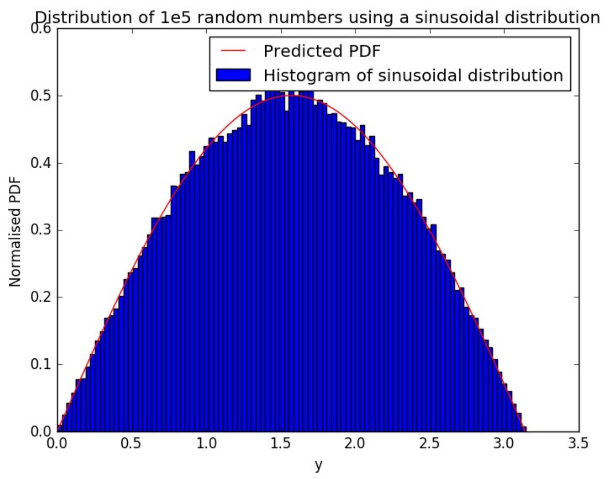

Figure 4: A normalised sinusoidal distribution in the range $0 \leq x \leq \pi$.

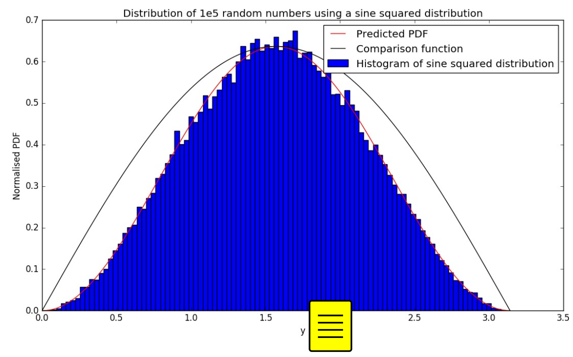

Figure 5: A normalised sine squared distribution in the range $0 \leq x \leq \pi$.

100 bins were used in this histogram and as a result, the predicted mean number of counts was around 1000, which was confirmed in the diagram.

The sinusoidal distribution using the transformation method is as shown in Figure 4. The distribution had been normalised in order to compare to the predicted PDF. The distribution followed closely with the predicted PDF. The main reasons for the difference from theoretical is the finite bin width instead of 0 and the finite number of points which should tend to infinity.

The sine squared distribution is as shown in Figure 5. The times for these two methods was calculated and the ratio of times used for transformation to rejection methods was found to be 0.000297. Another measure of the efficiency is the ratio of accepted points to trial points, which was found using a counter to be 0.785. A third measure of the efficiency was the ratio of areas between the desired PDF and the comparison function, which was $\pi/4 = 0.785$. The discrepancy in the times may be due to `time.clock` measurements were affected by other programs.

## 6 Conclusion

Five basic aspects of Computational Physics were studied. The results produced with the data mostly matched with theoretical prediction, and in particular, the machine accuracy of Python was found to be $2.22 \times 10^{-16}$ for a 64-bit float, which corresponded to $2 \times 10^{-52}$, showing that the mantissa was 52 bits plus a hidden bit.

Number of words: 2348

## References

[1] Scott P. *Introduction to Numerical Calculations.* Imperial College London; 2017.

[2] Bonten JHM. *Binary floats with hidden bit*; 2009. Available from: `http://home.kpn.nl/jhm.bonten/computers/bitsandbytes/wordsizes/hidbit.htm`.

[3] *Data types.* The Scipy community.; 2017. Available from: `https://docs.scipy.org/doc/numpy-dev/user/basics.types.html`.

[4] Uchida Y. *Matrix Algebra - LU Decomposition.* Imperial College London; 2017.

[5] Press WH. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press; 2007.

[6] Uchida Y, Scott P. *Project A [Guidelines], Computational Physics.* Imperial College London; 2017.

[7] Uchida Y. *Interpolation (Linear interpolation and Cublic splines).* Imperial College London; 2017.

[8] *Runge's phenomenon.* University of Tallinn;.

[9] Uchida Y. *Fourier Transforms.* Imperial College London; 2017.

[10] Uchida Y. *Random Numbers and Monte Carlo Methods.* Imperial College London; 2017.