Year 3 Computational Physics Assignment 2019/20

Name: Suhasan Kanagasabapathy CID: 013383402

Floating Point Number

Machine accuracy or epsilon is meant to be the smallest number understood by the machine. In other words, what is the smallest number, *x* I can add to a number, *A* (e.g. 1) and the answer is not that initial number, *A*? Mathematically *x* can be infinitesimally small, but there is a physical limit in the hardware. I used the exact same understanding into my code. I wrote:

$$x = \frac{7.}{3} - \frac{4.}{3} - 1 \tag{1}$$

The decimal points in 7 and 4 indicate they are floating points. The machine accuracy, *x* of the system was $2.220446049250313 \times 10^{-16}$. This was then made into a function which printed out the accuracy, where the input can be a float of any number of bits. The number of bits correspond to the precision of floating-point variables. For single precision, the float has 32 bits while a double precision float is represented in 64-bits. The function uses similar concept but in a slightly different way. The accuracy, *x* was initialised with a value (e.g. 1). The *x* is added to 1 and checked if the result is not equal to 1. If it isn't equal to 1, *x* is divided by 2 and becomes the new *x*. This was run until the result is too small, the result equals to 1. The result is multiplied by 2 to get the original accuracy *x* because it would have stopped after reaching half of *x*. The number 2 was chosen for division as numbers are stored in base 2.

The result for machine accuracy is summarised in table 1 below. The hardware used was a Google server (Google Colab). The result tells us the machine accuracy depends on the precision of the number. More specifically, the machine accuracy, *x* is given by

$$x = 2^{-p} \tag{2}$$

where *p* is the number of bits in the mantissa (assumed the number is of base 2) [1]. This is what we expect theoretically, and this fits the result obtained from our machine as well. The extended precision refers to the format which gives higher precision than basic floating-point formats, thus has more bits in the mantissa. It should be noted that the machine accuracy of extended precision number, when run on Spyder, gives the same value as the double precision. This could be because of the limitation due to hardware.

Precision	*p*	Machine Accuracy
32-bit	23	1.1920929×10^{-7}
64-bit	52	$2.220446049250313 \times 10^{-16}$
Extended precision	63	$1.084202172485504434 \times 10^{-19}$

Table 1: The machine accuracy with respect to the precision of a floating-point number. *p* is the number of bits in the mantissa. These values of *p* follow the IEEE 754 standard.

Reference:

[1] "David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Computing Surveys, Vol 23, No 1, March 1991" (PDF). Retrieved 11 Apr 2013.

Matrix Methods

In part (a), an arbitrary *N* by *N* matrix needs to be LU decomposed. An arbitrary square matrix, *M* of dimension 3 was created as shown below.

$$M = \begin{pmatrix} 60 & 30 & 20 \\ 30 & 20 & 15 \\ 20 & 15 & 12 \end{pmatrix}$$

LU decomposition is where M is decomposed into a lower triangular matrix, *L* and an upper triangular matrix, *U*. I defined a function, named lu () which used Crout's method of LU decomposition in the algorithm. The matrix *M* was inputted into the function and the results were:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 1/3 & 1 & 1 \end{pmatrix} \quad U = \begin{pmatrix} 60 & 30 & 20 \\ 0 & 5 & 5 \\ 0 & 0 & 1/3 \end{pmatrix}$$

To check if the matrices *L* and *U* were correct, they were multiplied together, and the product returned the matrix *M*. Hence, the algorithm worked.

In part (b), the given matrix A was decomposed into its L and U by using the lu () method. The results were:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1.125 & 1 & 0 & 0 \\ 0 & 0 & -1.419 & 1 & 0 \\ 0 & 0 & 0 & -1.217 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 0 & 8 & 4 & 0 & 0 \\ 0 & 0 & 15.5 & 10 & 0 \\ 0 & 0 & 0 & 45.194 & -25 \\ 0 & 0 & 0 & 0 & 30.575 \end{bmatrix}$$

The values were rounded off to three decimal points. The determinant of A was calculated by multiplying the diagonal entries of U and the answer was 514032.0. The answer was checked using the linalg.det () method in Numpy. It returned the same result, thus verifying the method.

As for part (c), a function was defined to solve the matrix equation $LU\mathbf{x} = \mathbf{b}$ for *x* using forward and backward substitution. Here *L*, *U* and vector *b* were the input parameters. Hence, the lu () from previous part was used to find *L* and *U*. This function was used to solve the equation in part (d), where the *LU* matrix was *A* and the vector *b* were given. The routine was turned into a function named solve () to be used later. The solution, x obtained up to 7 decimal points was

$$x = \begin{bmatrix} 0.4565708 \\ 0.6302876 \\ -0.5105752 \\ 0.0538915 \\ 0.1961317 \end{bmatrix}$$

The result was verified by multiplying *x* with the matrix *A* and it returned the vector *b*.

In part (e), the inverse of the matrix A was calculated. The matrix A was first LU decomposed into L and U. By definition, $A^{-1} = U^{-1} L^{-1}$ so the inverse of *L* and *U* were found separately. We know $L L^{-1} = I$, where *I* is the identity matrix. I used the solve () function in previous part to solve the L^{-1} and the identity matrix as the vector *b*. *I* solved for each of its column, then the columns of *x* were combined to give *x* as a matrix. Same is done with *U*. The result was

$$A^{-1} = \begin{bmatrix} 0.37951723 & -0.0461839 & 0.00401531 & -0.00474679 & -0.0019454 \\ -0.13855169 & 0.13855169 & -0.01204594 & 0.01424036 & 0.00583621 \\ 0.02710337 & -0.02710337 & 0.02409189 & -0.02848072 & -0.01167243 \\ 0.07048977 & -0.07048977 & 0.06265758 & 0.04414511 & 0.01809226 \\ 0.06355635 & -0.06355635 & 0.05649454 & 0.03980297 & 0.03270614 \end{bmatrix}$$

The inverse A^{-1} was multiplied by A to check if it gave an identity matrix. The result was cross-checked using linalg.solve in Numpy too. Both cases verified the answer, thus the algorithm.

Interpolation

In part (a), an arbitrary tabulated set of *x*-*y* data containing 5 pair of points was created and I performed linear interpolation on the data. Table 2 shows the arbitrary data.

x	*y*
1.5	2.5
2.3	5.0
5.2	7.8
6.9	4.2
7.1	3.5

Table 2: The arbitrary tabulated set of *x*-*y* data used for linear interpolation.

A function was defined to perform linear interpolation. For each consecutive pair of points, the straight-line equation was determined using the expression from the notes and plotted. In part (b) I performed cubic spline interpolation on the tabulated data above. Second derivatives were set as zero at the ends, thus called 'natural' cubic spline. This reduces the number of unknowns to (n-1) with equal number of equations. I used the fundamental expression given in the notes to define the coefficients and built a square matrix. I used the matrix solver functions from previous methods to solve for all second derivatives. Along with the first derivatives (similar to linear interpolation), I could get the equation of curve for a set of three points. The linear and cubic splines interpolations were plotted on the same axes. The result is shown in figure 1 below.

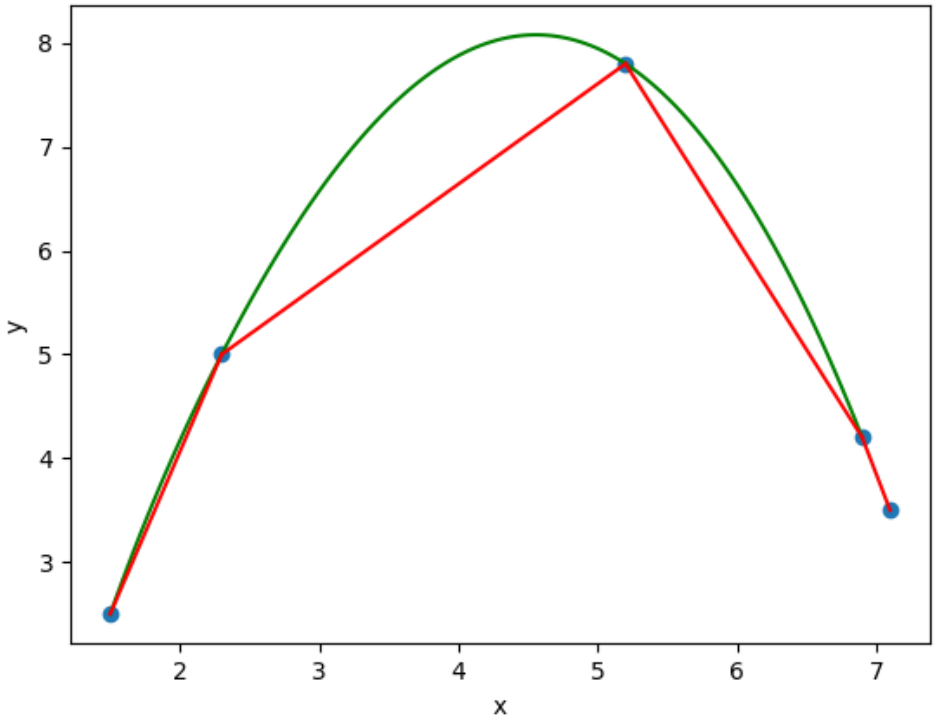

Figure 1: The plot of *y* against *x*. The blue points represent the scatter plot of the data. Red lines are the linear interpolations whereas the green curve is the cubic splines interpolation. The linear and cubic interpolations differ the most at sharp turns. In this case, the cubic interpolation is more suitable since it might describe the physics in a more natural way.

In part (c) a table of 10 x-y points was given. I used the previously defined functions to plot the linear and cubic spline interpolations on same axes as shown in figure 2. Here both interpolate the data mostly well and linear interpolation fails at some sharp turns, whereas cubic spline captures the turns well.

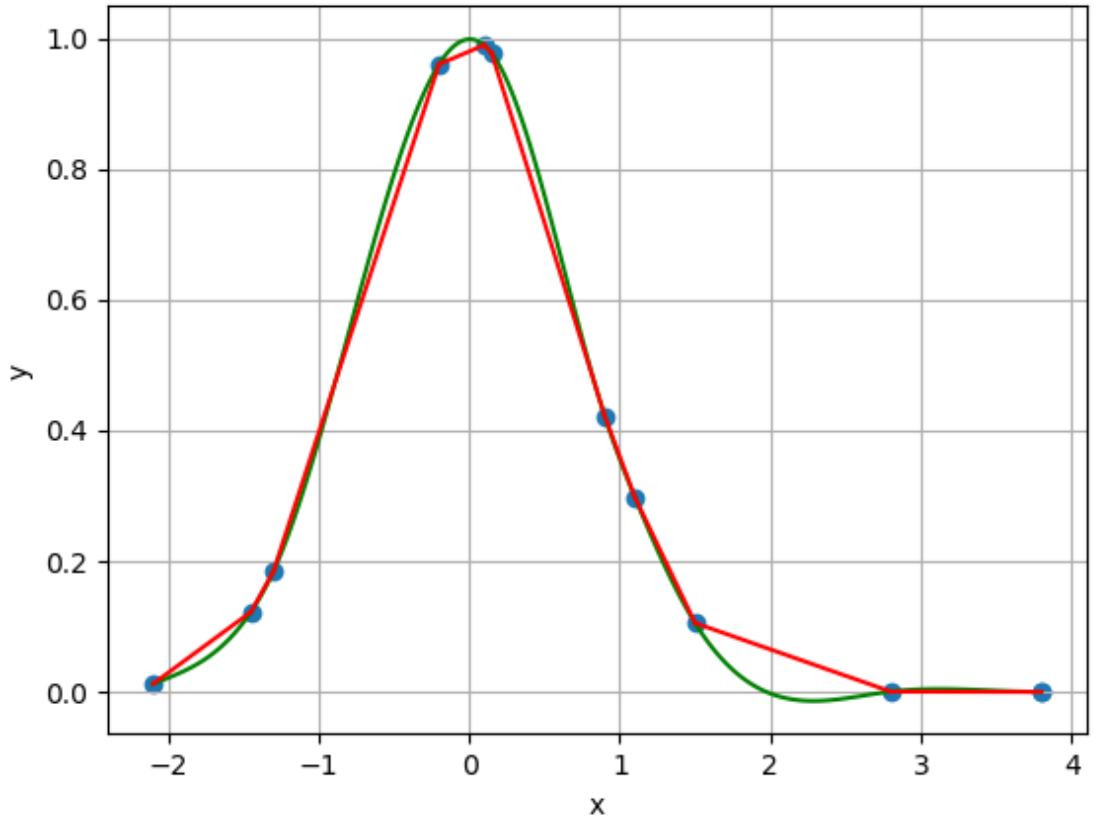

Figure 2: The plot of *y* against *x*. As before red lines represent the linear interpolations and green curve is the cubic spline interpolation.

Fourier Transforms

I was given a pulse function, *h* and a gaussian response function, *g* and required to find the convolution of those functions. The convolution theorem was used, and the Fourier transforms *(FT)* of each function were determined using Fast Fourier Transform *(FFT)* module from Numpy. Firstly, appropriate time range of (-10, 10) with a period of $T = 20$ was chosen. These are taken as dimensionless for simplicity. The number of samples, N was chosen as 1024, since it is a power of 2 and gives no padding. This gives time interval Δt of 20/1024 which is reasonably small. This corresponds to the maximum frequency or Nyquist frequency, w_0 of about 160.8. So, there will be aliasing for frequencies higher than this value. *FFT* gives aliased *FT* and thus needs correction to shift it to provide the expected plot. So, I used np.fft.fftshift to shift the zero-frequency component to the centre of the spectrum.

Since *FFT* does Discrete Fourier Transform, the *FT* was multiplied by Δt to give approximate continuous transform. I took the absolute values of *FT*s to consider real and imaginary components. The plots of *h, g*, their *FT*s and the convolution functions are shown on the next page.

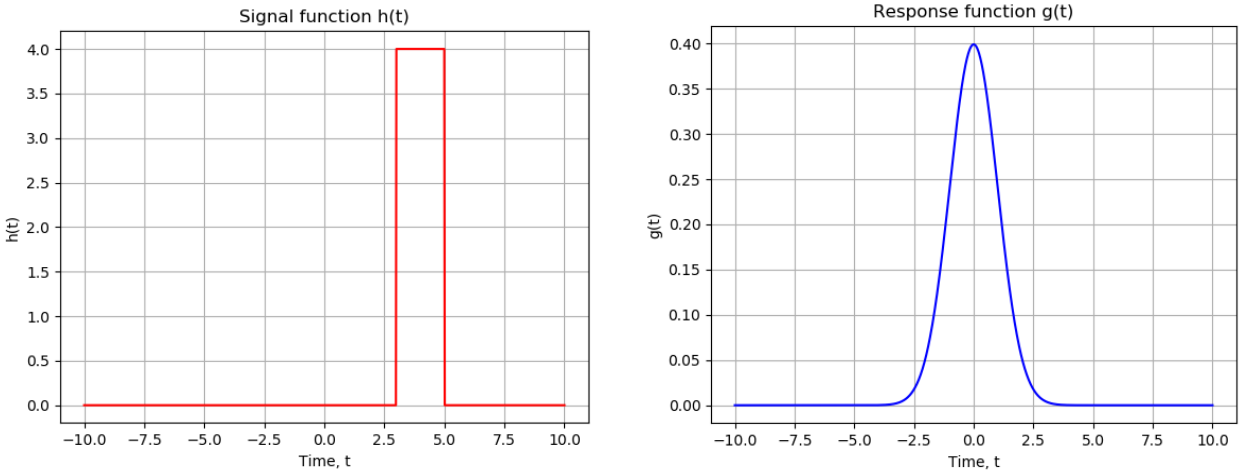

Figure 3: Left is the plot of *h* and right is the plot of *g* in the time range of (-10, 10).

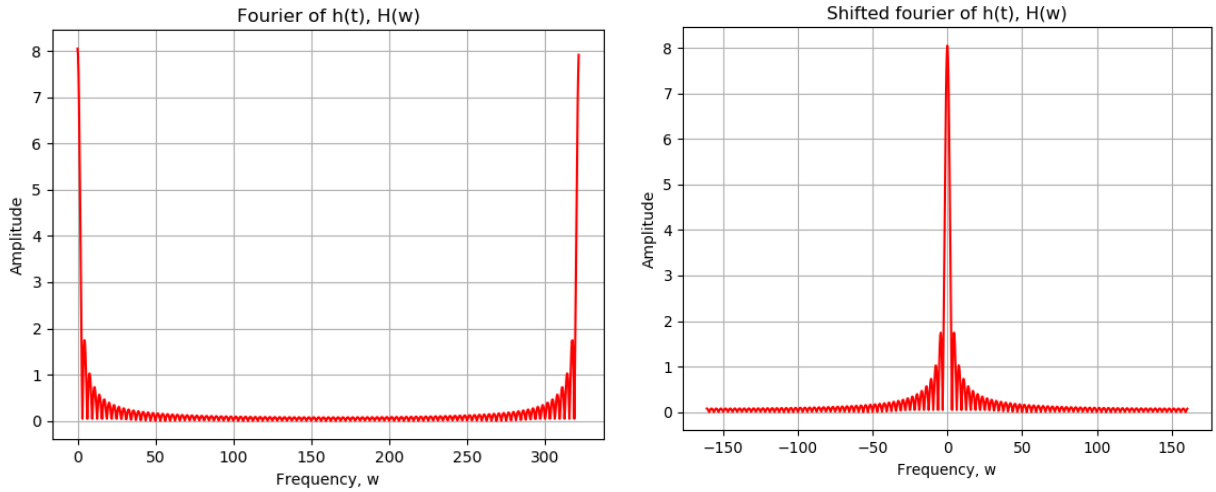

Figure 3: Left is the plot of *FT* of *h* on a range of (0, $2w_0$) and right is the plot of shifted *FT* of *h* on the range of ($-w_0$, w_0) which gives a more familiar shape.

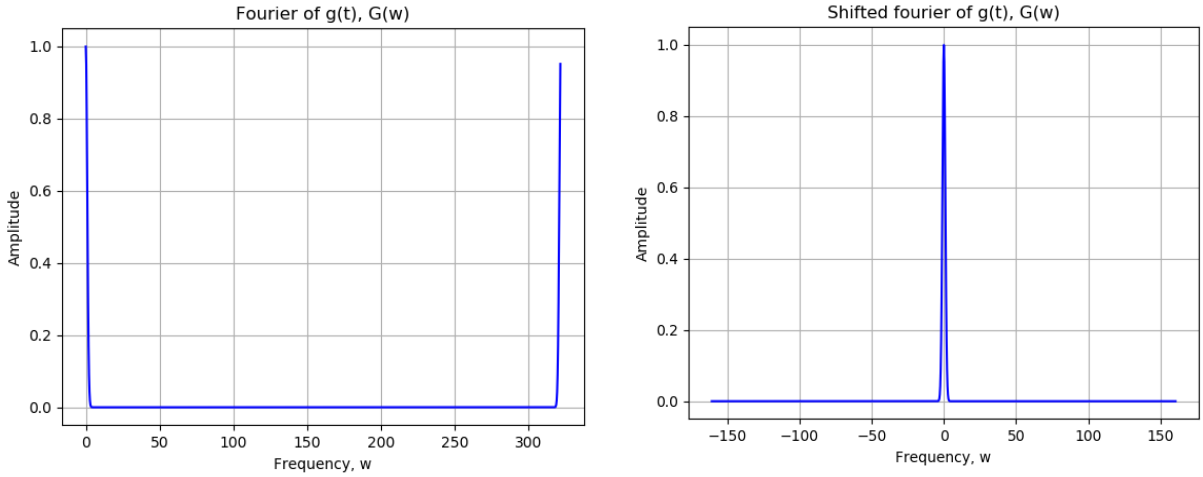

Figure 4: Left is the plot of *FT* of *g* on a range of (0, $2w_0$) and right is the plot of shifted *FT* of *g* on the range of ($-w_0$, w_0) which gives more familiar shape. The width is narrower than in figure 3.

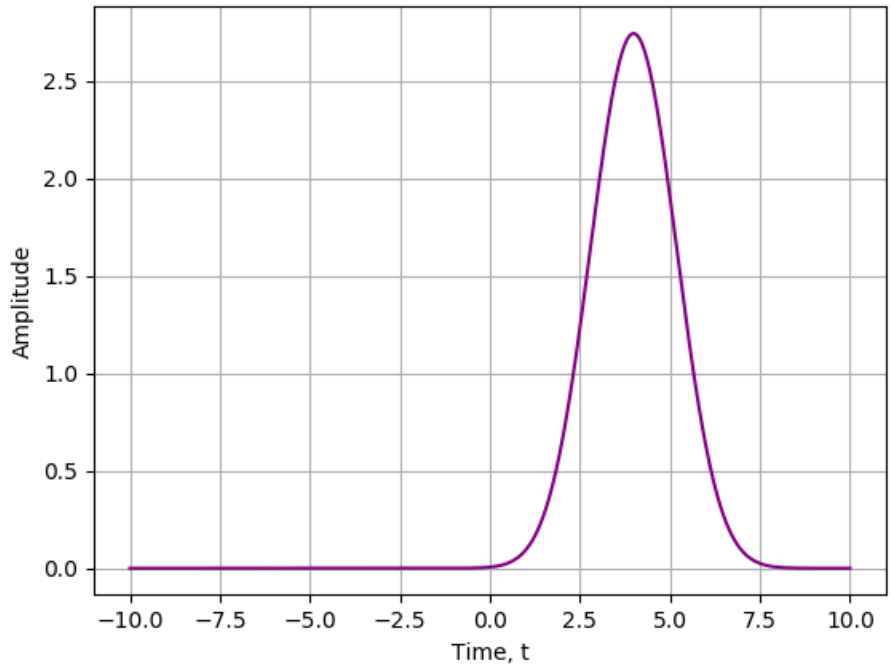

Figure 5: The convolution of signal and response function, *(h∗g)(t)*. The centre of the peak corresponds to the pulse, *h*. The amplitude also looks reasonable.

Random Numbers

In part (a), I used a random number generator class in numpy called RandomState which creates a container for the Mersenne Twister pseudo-random number generator. Via this, I could initialise the generator with the same seed (seed = 1) which ensures reproducibility throughout the algorithm. A list of 10^5 uniformly distributed random numbers, *x* was generated and plotted on a histogram as shown in figure 6.

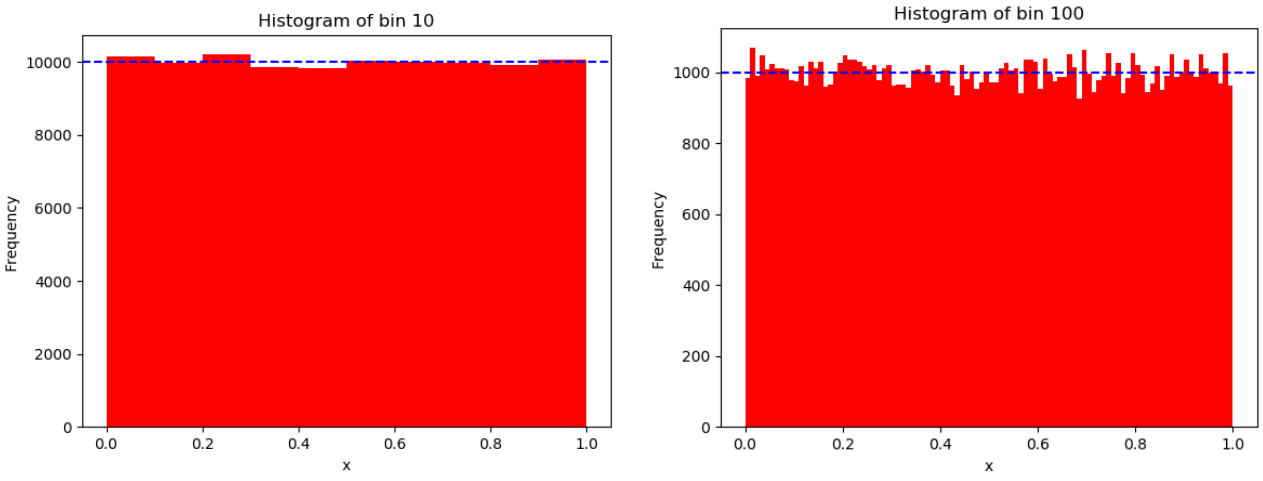

Figure 6: The histogram on the left is with 10 bins and the distribution is strongly uniform. Whereas the histogram with 100 bins on the right shows the fluctuations more clearly.

In part (b), I used transformation method to create 10^5 samples of random numbers generated with the given pdf. Firstly, the Cumulative Distribution Function (*CDF*) was found by integrating pdf from 0 to *x*. The inverse of *CDF* was found as $y = 2\sin^{-1} x$. A uniformly generated *x* was mapped into the

inversed *CDF* to give y the desired pdf. The histogram of the transformed distribution is shown in figure 7.

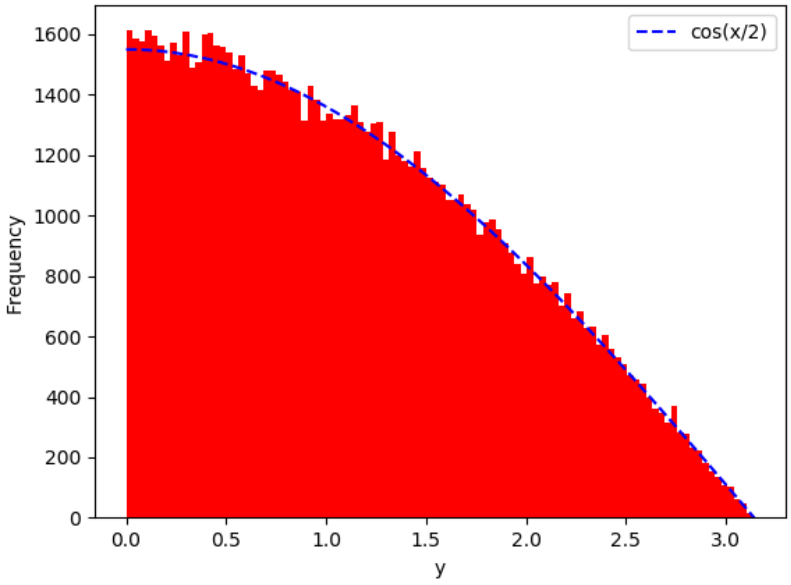

Figure 7: The histogram with 100 bins of 10^5 random numbers generated through transformation method in range (0, π). The blue curve is the expected pdf with an adjusted amplitude, thus not normalised, but shows the distribution follows the shape strongly.

In part (c), I used rejection method to generate random numbers distributed with new pdf. The pdf_1 in previous part was used as a comparison function but with altered amplitude, so it is always bigger in the interval. To get the number of samples close to 10^5, I made a list containing random numbers from pdf with a size of around 127 373. This value is obtained from the ratio of areas of pdf_1 and pdf_2 and this gave about 100018 samples in the new pdf. The histogram is shown on figure 8.

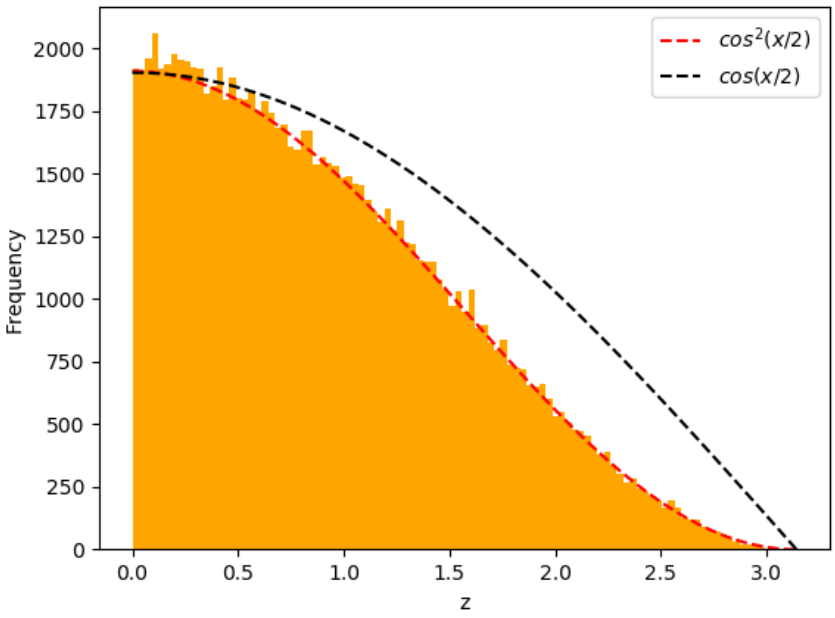

Figure 8: The histogram with 100 bins of about 100 018 random numbers generated through rejection method in range (0, π). The red curve shows the distribution follows the $\cos^2 x$ shape strongly. The black curve shows the comparison function used. These curves are not normalised.

The ratio of time taken to produce the samples in this part to previous one was calculated, using built-in timeit module, to be about 3.2. Theoretically, the ratio should be $4/\pi \sim 1.27$ which is the ratio of areas. This difference could be due to numerical error or inefficient algorithm.