Assignment

Yaojia Huang & 01213992

November 12, 2018

1 Floating Point Variables

1.1

The machine epsilon is the smallest number such that

$$1 + \epsilon > 1. \tag{1}$$

If a number is smaller than ϵ, then adding it and 1 still gives 1 due to rounding.

To find the floating point machine epsilon for Python on the macOS Mojave operating system, a recursive function is constructed. The function takes the average of two input floating points(1 and 2 initially) and checks whether the average is equal to 1. If they are not equal, then the function calls itself and uses the average and 1 as new inputs to repeat the above process. Once the average is identical to 1, which means that the larger input is almost the smallest number above 1 such that the average of itself and 1 is rounded to 1, the function returns the value of the difference between the larger input and 1. The final result is $2.220e - 16$, which is the epsilon for double precision. Since the default number type in Python is double precision and macOS Mojave is a 64-bit operating system, the outcome is satisfactory.

1.2

In this part, it is necessary to understand what floating point is. For example, 3.25 can be represented, in single precision format, as

Sign	*Exponent*								*Mantissa*																						
0	1	0	0	0	0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

in the computer system. The formula that coverts the above format is

$$F = (-1)^S 2^{E-129} \cdot M, \tag{2}$$

where F is the floating point, S the sign bit, E the exponent and M the mantissa. When adding two floating points, the exponent of the sum is that of the larger number. To obtain the mantissa of the sum, one has to take the difference, Δ, between the exponents of the two numbers, shift the mantissa of the smaller number Δ positions to the right with the left side filled by 0, and add the mantissas of the two numbers. However, if one of the numbers is so small that the its mantissa is shifted all the way down such that every digit in the mantissa is replaced by 0, the sum would be identical to the larger number. When this case happens, the difference between the exponents must be larger than or equal to the length of the mantissa. Given the exponent of 1 is 129, it is straightforward to estimate the epsilon of each float type using Equation(2).

	Mantissa Length	Theoretical ϵ	Calculated ϵ
single	23	2^{-23}	$1.192e-07 \sim 2^{-23}$
double	52	2^{-52}	$2.220e-16 \sim 2^{-52}$
long double	63	2^{-63}	$1.084e-19 \sim 2^{-63}$

Table 1: Comparison between the theoretical ϵ values and the calculated ϵ values of different float types.

In Python, the float type can be specified by using NumPy(e.g. numpy.float32). The function mentioned in the last section is extended to accept a type input so that it can convert all numbers in the arithmetic operations to the designated type and return a value of that type. Table(1) shows the values of the theoretical ϵ and the calculated ϵ. These two values are compared in Python and the boolean outcomes show they are indeed identical.

When the program is executed on a department computer, the epsilon for long double is the same as that of double precision, so one float type has different number of bits on different systems.

2 Matrix Methods

2.1

A square matrix can be decomposed via LU decomposition. Take one matrix($N = 3$) for example, it can be decomposed into a lower(L) and an upper(U) triangular matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}. \tag{3}$$

The above can be expressed as N^2 equations, but there are $N^2 + N$ unknown variables remain to be determined, so N unknowns could be arbitrarily set. It would be convenient to fix the diagonal elements of L to 1, then it becomes obvious that $u_{11} = a_{11}$, $l_{21} = a_{21}/u_{11}$ and so on. The idea is that every time an unknown is solved, it can be subsequently substituted into another equation so that the next unknown can also be solved. The generalised equations for this procedure are

$$u_{ij} = a_{ij} - \textstyle\sum_{k=1}^{i-1} l_{ik} u_{kj} \quad \text{for } (i \leq j), \tag{4}$$

$$l_{ij} = u_{jj}^{-1}\left(a_{ij} - \textstyle\sum_{k=1}^{j-1} l_{ik} u_{kj}\right) \text{ for } (i > j). \tag{5}$$

To realise this decomposition in Python, the L and U matrices are created with the diagonal elements of L initialised to be 1 and all other elements 0. A loop is built to transverse the rows of the matrices, and within this loop, there is a nested loop that transverses the columns. The calculation of L and U cannot be placed in separate loops; otherwise, the first matrix being calculated, say, U, uses elements from a zero matrix because the elements of L have not been obtained. Therefore, the calculations of both matrix take place in the same inner loop, and an if-statement that judges whether $i \leq j$ or $i > j$ decides which one of Equation(4) and (5) should be applied. In the final step, the calculated element replaces the 0 in the corresponding position in L or U.

2.2

Using the method described in the previous section, the matrix A can be decomposed as

$$\begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 3 & 9 & 4 & 0 & 0 \\ 0 & 9 & 20 & 10 & 0 \\ 0 & 0 & -22 & 31 & 25 \\ 0 & 0 & 0 & -55 & 60 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1.13 & 1 & 0 & 0 \\ 0 & 0 & -1.42 & 1 & 0 \\ 0 & 0 & 0 & -1.22 & 1 \end{bmatrix} \begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 0 & 8 & 4 & 0 & 0 \\ 0 & 0 & 15.5 & 10 & 0 \\ 0 & 0 & 0 & 45.19 & -25 \\ 0 & 0 & 0 & 0 & 29.58 \end{bmatrix}, \tag{6}$$

in which elements with excessive decimals are rounded to the 2nd decimal place. The result was validated by taking the product of L and U using *numpy.matmul()*, which gives the original matrix.

The determinant of matrix A is the product of the determinants of L and U. Since they are both triangular matrices, the determinant of A can be calculated by taking the product of the diagonal elements from both L and U. Since the diagonal of L is populated with 1s, only U needs to be considered. By multiply the diagonal elements of U, it turns out that the determinant is 497220, which agrees with the result calculated by *numpy.linalg.det()*

If the matrix is large, LU decomposition is much more efficient in finding the determinant of a matrix than expansion by minors because the former takes $O(n^3)$ steps, but the later takes $O(n!)$.

2.3

L and U matrices are also helpful in solving a linear system. Consider such an equation:

$$A\mathbf{x} = LU\mathbf{x} = \mathbf{b}, \tag{7}$$

where **b** is the given vector and **x** is the vector that needs to be solved. One way to solve this equation is splitting it into two parts:

$$\begin{aligned} L\mathbf{y} &= \mathbf{b} \quad (8)\\ U\mathbf{x} &= \mathbf{y}. \quad (9) \end{aligned}$$

The procedure of solving these two equations is similar to the one described in Section(2.1). For example, in Equation(8), it is easy to show that $y_1 = b_1/l_{11}$; then $l_{21}y_1 + l_{22}y_2 = b_2$ can also be solved since the values of y_1, b_2, l_{21} and l_{22} are known. Previously solved variables can always be substituted into the next equation and make that equation solvable. The values of y_i and x_i can be obtained by the following equations:

$$\begin{aligned} y_i &= l_{ii}^{-1}\Big(b_i - \textstyle\sum_{j=1}^{i-1} l_{ij}y_j\Big) \quad (10)\\ x_i &= u_{ii}^{-1}\Big(y_i - \textstyle\sum_{j=i+1}^{N} u_{ij}x_j\Big). \quad (11) \end{aligned}$$

In Equation(11), the values of x_i are calculated in the reversed order(from x_N to x_1) because the elements that have higher indices are required to calculate those that have lower indices.

The implementation of the method in Python is also similar. Since L and U have been solved, the **y** vector can be calculated by applying Equation(10) and iterating over i and j indices. **x** can be solved in the same manner. In the implementation, the calculations of both vectors do not have to be in the same loop because calculating **y** does not require any elements from **x**. If both vectors are computed in the same loop, the indices of **x** need to be deliberately manipulated to make the elements iterate reversely. Doing so would involve some perplexing index labelings such as $-i-1$, so constructing another loop for **x** is more optimal, which does not increase the time complexity of the program but improves readability.

2.4

The linear system

$$A\mathbf{x} = \begin{bmatrix} 2 \\ 5 \\ -4 \\ 8 \\ 9 \end{bmatrix}, \tag{12}$$

where A is the matrix in Section(2.2), can be solved using the above method. The solution(rounded to 3rd decimal place) is

$$\mathbf{x} = \begin{bmatrix} 0.456 \\ 0.631 \\ -0.513 \\ 0.058 \\ 0.203 \end{bmatrix}. \tag{13}$$

The outcome is identical to the result given by *numpy.linalg.solve()*.

2.5

To get the inverse of matrix A, the following equation needs to be solved:

$$AA^{-1} = I, \tag{14}$$

where I is the identity matrix. Let the n-th columns of A^{-1} and I be A_n^{-1} and I_n respectively, it can be shown that

$$AA_n^{-1} = I_n. \tag{15}$$

Equation(15) has exactly the same form as Equation(12). The procedure of getting the inverse matrix is applying the method described in Section(2.3) to Equation(15) with n from 1 to N; each iteration returns one of the columns of the inverse matrix. In the code, I_n is the n-th row of the identity matrix because it is the same as the n-th column, and the rows are more convenient to be extracted. The inverse of the matrix A turns out to be

$$A^{-1} = \begin{bmatrix} 0.379 & -0.046 & 0.004 & -0.005 & -0.002 \\ -0.138 & 0.138 & -0.012 & 0.014 & 0.006 \\ 0.026 & -0.026 & 0.023 & -0.029 & -0.012 \\ 0.072 & -0.072 & 0.064 & 0.045 & 0.019 \\ 0.066 & -0.066 & 0.058 & 0.041 & 0.034 \end{bmatrix}, \tag{16}$$

which matches the result from *numpy.linalg.inv()*. Performing matrix multiplication on A and A^{-1} indeed returns an identity matrix. The errors due to rounding has the magnitude ranging from e^{-19} to e^{-16}.

When it comes to solving a particular equation such as Equation(12), LU decomposition may not be a better practice than Gaussian elimination or Gauss-Jordan elimination; their time complexities are all proportional to N^3, but the complexity of LU decomposition and forward and back substitution has a higher coefficient. However, the two elimination methods pick up an extra power when solving the inverse of a matrix, while LU decomposition still stays at N^3. Therefore, using LU decomposition is a better way to solve this problem.

3 Interpolation

3.1

Linear interpolation is nothing fancy but connecting (x_i, y_i) and (x_{i+1}, y_{i+1}) with a straight line for $i = 0, 1, 2, ..., N-2$, where N is the number of data points. The function of a straight line is

$$f(x) = \frac{(x_{i+1} - x)f_i + (x - x_i)f_{i+1}}{x_{i+1} - x_i}. \tag{17}$$

In order to plot the linear interpolation for a set of data, a loop is constructed to iterate from x_1 to x_{N-1}. In each iteration, the values of x_i and x_{i+1} are substituted into Function(17). By feeding the function an array of x coordinates within (x_i, x_{i+1}), the y values could be calculated and the linear interpolation could also be drawn.

3.2

Linear interpolation is usually not a good description of the reality, but it can be improved by introducing non-zero higher order derivatives to the interpolation function. A cubic spline consists of piecewise quadratic functions, so it fits data points more smoothly. However, more constraints are required because it needs to fix the values of the 2nd derivative at x_i. The piecewise functions are defined as

$$f(x) = A(x)f_i + B(x)f_{i+1} + C(x)f''_i + D(x)f''_{i+1}, \tag{18}$$

with

$$A(x) \equiv \frac{x_{i+1}-x}{x_{i+1}-x_i}, \tag{19}$$
$$B(x) \equiv 1 - A(x), \tag{20}$$
$$C(x) \equiv \frac{1}{6}(A(x)^3 - A(x))(x_{i+1} - x_i)^2, \tag{21}$$
$$D(x) \equiv \frac{1}{6}(B(x)^3 - B(x))(x_{i+1} - x_i)^2. \tag{22}$$

By taking the first derivative of $f(x)$ and imposing continuity at x_i, for N data points, $N-2$ equations can be listed whose format is as follows:

$$\frac{(x_i - x_{i-1})}{6}f''_{i-1} + \frac{(x_{i+1} - x_{i-1})}{3}f''_i + \frac{(x_{i+1} - x_i)}{6}f''_{i+1} = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} - \frac{f_i - f_{i-1}}{x_i - x_{i-1}} \tag{23}$$

This is equivalent to solving an $(N-2) \times N$ linear system with each row containing at most 3 non-zero elements. Since the unknown variables are more than the equations, the boundary condition is imposed such that f''_0 and f''_{N-1}

are both zeros. Therefore, the first and the last columns of the matrix, which contain 0s only, can be eliminated. In Python, a $(N-2) \times N$ zero matrix can be first initialised. By iterating pairs of adjacent data points, the three coefficients on the LHS of Equation(23) can be calculated and placed in the corresponding positions in the matrix, and every constant on the RHS is appended to a list. After the completion of the loop, the first and the last columns are removed so that a $(N-2) \times (N-2)$ linear system is obtained. At this stage, the solution, which is a list of f_i'', can be readily calculated by the program built in Section(2.3). Since the list does not contain f_0'' and f_{N-1}'', two zeros are separately added to the beginning and the end of the solution list.

It is possible to construct a $(N-2) \times (N-2)$ matrix initially and in each iteration, the program checks which coefficient needs to be put in the matrix or treats the first and the last rows separately from the rest. The running time of the former would increase due to repetitive condition checking and both of them are not as concise as simply deleting the first and second columns at the end. Therefore, the method discussed in the last paragraph is adopted.

After that, another loop is created to iterate over adjacent data pairs. In this loop, a number of x values within x_i and x_{i+1} are selected. While iterating the selected x values in a nested loop, the functions are evaluated at these sites using Equation(18). Outside the loops, the x and y values are plotted to visualise the graph of the cubic spline.

3.3

Using the data points from the assignment instruction, Figure(1) is plotted. The turning points of the linear interpolation are the coordinates of the given data. The cubic spline goes through all the data point precisely, which is the correct behaviour of interpolations.

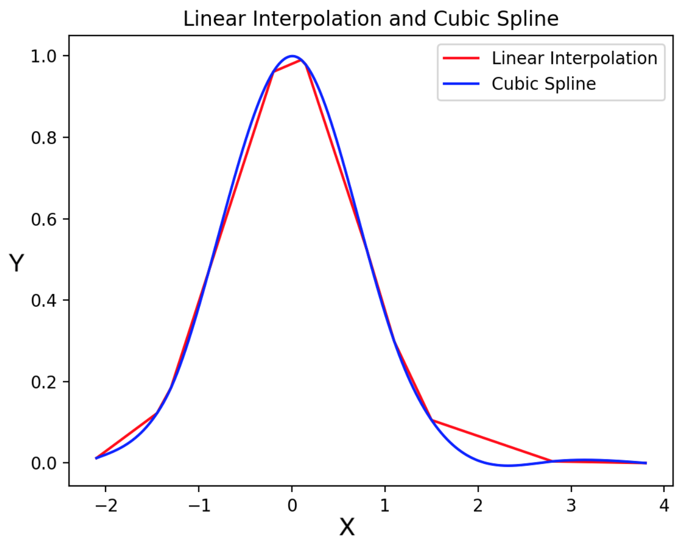

Figure 1: Data Points fitted with linear interpolation(red line) and natural cubic spline(blue line).

A list of its first derivative was approximated discretely based on the data points, then this list was used to obtain another list of the second derivative, F'', using the same method. If the difference between F_i'' and F_{i+1}'' is less than $0.05F_i$, a True was put in a boolean list; otherwise a False was inserted. If 500 steps were taken between each pair of data points, Trues occupied 93.7 percent of the list. When the number of steps increased to 10000, 99.7 percent of the list were Trues. The result showed that the function is indeed a piecewise quadratic function. In this case, the percentage cannot be raised to 100% because the second derivative is not continuous at each x_i.

4 Fourier Transforms

According to Convolution theorem, the convolution of two functions can be expressed as

$$(h * g)(t) = \mathfrak{F}^{-1}(\mathfrak{F}(h(t)) \cdot \mathfrak{F}(g(t))), \tag{24}$$

where $\mathfrak{F}$ and $\mathfrak{F}^{-1}$ are symbols for Fourier and inverse Fourier transform. To find the convolution of

$$h(4) = \begin{cases} 4, & 3 \leq x \leq 5, \\ 0, & \text{otherwise.} \end{cases} \tag{25}$$

$$g(t) = \frac{1}{\sqrt{2\pi}} e^{\frac{-t^2}{2}}, \tag{26}$$

the fast Fourier transform can be used. A step size, Δt, is firstly chosen. Then, data points are sampled from the two functions such that adjacent data points are separated by Δt on the t-axis. The data are stored in two lists and passed to $numpy.fft.fft()$, which returns the discrete Fourier transforms of the two functions. The returned two lists are multiplied together to generate a new list(in a way that the i-th element of the new list is the product of the i-th elements of the returned list).

5 Random Numbers

5.1

For generate uniformly distributed numbers over the range $[0, 1]$, the Random library is imported in Python. It is chosen because it is a Mersenne Twister generator, which has a long period of $2^{19937} - 1$ and runs faster than the random method in NumPy(as explained later).

Construct a loop that iterates $100,000$ times to get $100,000$ uniformly distributed number within $[0, 1]$, group the random numbers in a histogram of 200 bins and normalise the bins, the following plot is obtained:

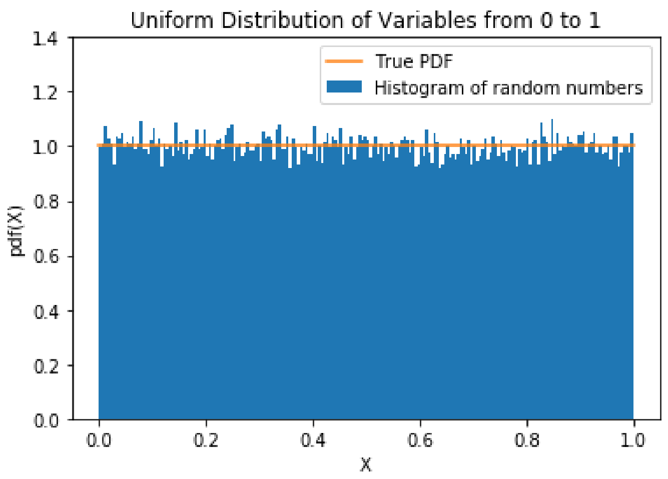

Figure 2: Normalised uniform distribution of $100,000$ random numbers within $[0, 1]$. The standard deviation of the bin heights is 0.039.

The number of bins needs to be large to show the uniformity of the distribution. However, if there are too many bins, the number of samples in each bin is small, which could lead to a great variation across bins. Therefore, the bin number is chosen to be as large as possible with the constraint that the average number of samples in each bin is kept above 500. The standard deviation of the histogram from 1 is 0.039, which is small enough to assume that the distribution is uniform.

5.2

To generate random numbers that obey the distribution:

$$P(Y) = \frac{1}{2}\sin(Y) \tag{27}$$

over the interval $[0, \pi]$, the following steps are taken.

First assume that the Y is a function of X, which has a uniform distribution from 0 to 1, such that

$$|P(Y)dY| = |P(X)dX|. \tag{28}$$

$P(x)$ is equals to 1 because of the way Xs are distributed. If $dY/dX \leq 0$, then Equation(28) can be transformed to

$$P(Y) = \frac{dX}{dY}. \tag{29}$$

Since $P(X)$ is a probability density function, it can be shown that

$$C(Y) = \int_0^Y P(Y')dY' = \int_0^Y \frac{dX'}{dY'}dY' = \int_0^X dX' = X, \tag{30}$$

where $C(Y)$ is the cumulative probability distribution function. Then substitute $P(Y') = \frac{1}{2}\sin(Y')$ to get

$$X = \quad \tfrac{1}{2} - \tfrac{1}{2}\cos(Y) \tag{31}$$
$$\Rightarrow Y = \quad \arccos(1 - 2X). \tag{32}$$

To test whether Y is distributed according to $P(Y)$, $100,000$ samples are obtained by substituting uniformly and randomly generated X in Equation(32). The outcomes are put in 200 bins within $[0, \pi]$. The histogram is

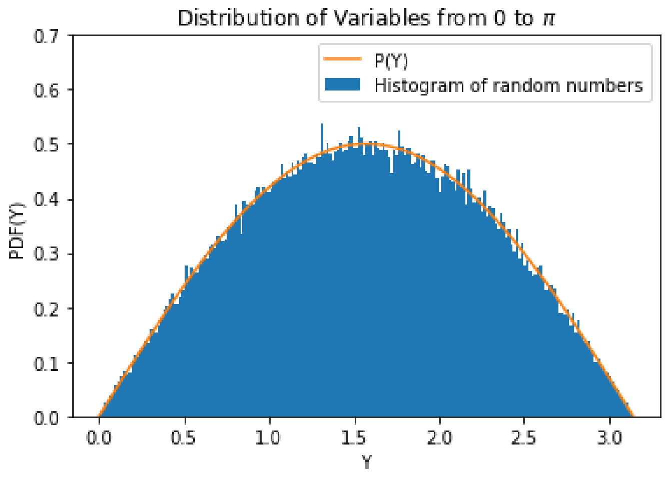

Figure 3: Histogram with 200 bins of $100,000$ randomly generated Ys within $[0, \pi]$. The standard deviation of the bin heights is 0.014.

The standard deviation of the bin heights from the expected PDF is 0.014; therefore it is fairly confident to say that the distribution satisfied Equation(27).

5.3

The above method cannot be used to make a generator that produces random numbers according to

$$P(x) = \frac{2}{\pi}\sin^2(x) \tag{33}$$

with $x \in [0, \pi]$ because its cumulative distribution is not invertible. Therefore, the rejection method is used here. The comparison function is chosen to be $\frac{2}{\pi}\sin(x)$ as shown in Figure(4).
The procedure has five steps:
1.) Randomly generate a variable, x_i, according to the PDF in the previously section.
2.) Substitute x_i in $\frac{2}{\pi}\sin(x)$ and obtain the result, y_i.
3.) Pick another random number, r_i , whose PDF has a uniform distribution over $[0, y_i]$
4.) Compare r_i with $P(x_i)$. If $P(x_i) \geq r_i$, x_i is accepted, otherwise it is rejected.
5.) Repeat from the first step until the number of accepted variables reaches a certain value.

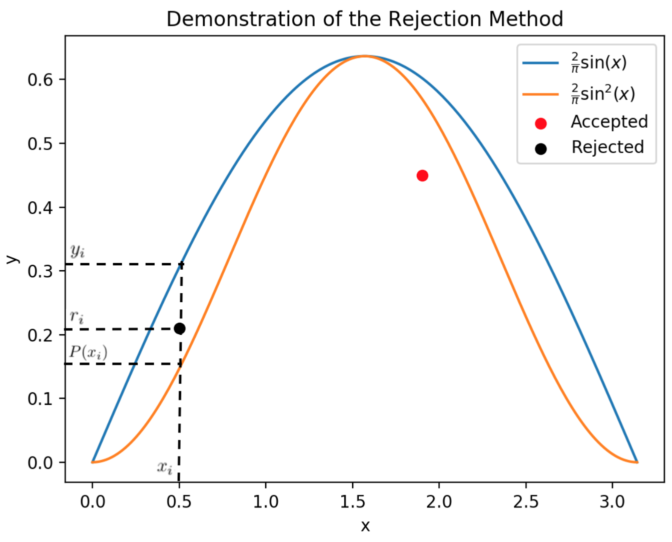

Figure 4: A plot that shows how the original PDF and the comparison function determine whether a random number is accepted or rejected.

Figure(33) shows the concept of the method visually; the x value of a randomly selected coordinate is accepted only if the coordinate is below or on the PDF. The program can be made more efficient if

$$G(x) = \quad 1 \quad \text{replaces} \quad \frac{2}{\pi}\sin(x), \tag{34}$$

$$H(x) = \frac{P(x)}{\frac{2}{\pi}\sin(x)} \quad \text{replaces} \quad P(x) \tag{35}$$

before Step 1. Although within $[0, \pi]$, the area ratio of $G(x)$ to $H(x)$ is more than that of $\frac{2}{\pi}\sin(x)$ to $P(x)$, which increases the probability of rejection, the reduction of calculations in the loop has a greater effect on the running time. Therefore, this optimisation is adopted.

The above comparison is chosen because its shape closely mimics the PDF of interest. If the comparison function is a constant, the program would produce more variables that are above the PDF and the time taken to accept, say, $100,000$ numbers will be longer.

The resultant histogram is shown in Figure(5). The standard deviation is 0.015, which is also small enough to believe the distribution has the shape described by Equation(33).

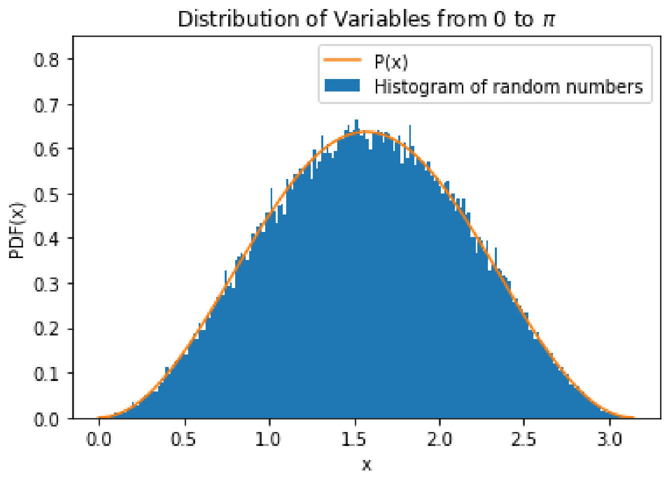

Figure 5: Histogram with 200 bins of $100,000$ randomly generated xs within $[0, \pi]$. The standard deviation of the bin heights is 0.014.

In the last section, the sampling process iterates $100,000$ times and in each loop, one random number is selected. In this section, there are more than $100,000$ loops and two random numbers are selected in each one. The number of loops is estimated to be $100,000 \times \pi/2$, where $\pi/2$ is the area ratio of $G(x)$ to $H(x)$ within $[0,\pi]$. Therefore, it is expected that the time taken for this question should be more than $2 \times \pi/2 = \pi$ times of the time spent in the previous question. The ratio turned out to be around 3.52, which is very close to the expectation. $x_i, y_i, r_i, P(x_i)$

References

[1] Kragh, H. (2000). Max Planck: the reluctant revolutionary. Physics World, [online] 13(12), pp.31-36. Available at: http://iopscience.iop.org/article/10.1088/2058-7058/13/12/34 [Accessed 5 Feb. 2018].