Exercise 2

Problem 1: Numerical Differentiation – First Derivative

→The problem is to create a program that can calculate the first derivative the sin function numerically, using both forwards and central difference formulae. The precision increases as the difference, h, is made smaller, however if it is too small rounding errors (due to the finite bit size of floating point variables) cause large errors. I have examined the differences in precision between the two formulae and looked closely at the effect the value of h has on the calculations.
→My program simply uses a loop which takes x from 0 to 2π in 0.05 steps, calculating intermediate sums and computing the derivative at this point from forwards, central and analytical (cosx) formula, the values are then printed to screen where I then copied them into excel. Its also calculates and prints the difference between the central/difference formula and cosx.

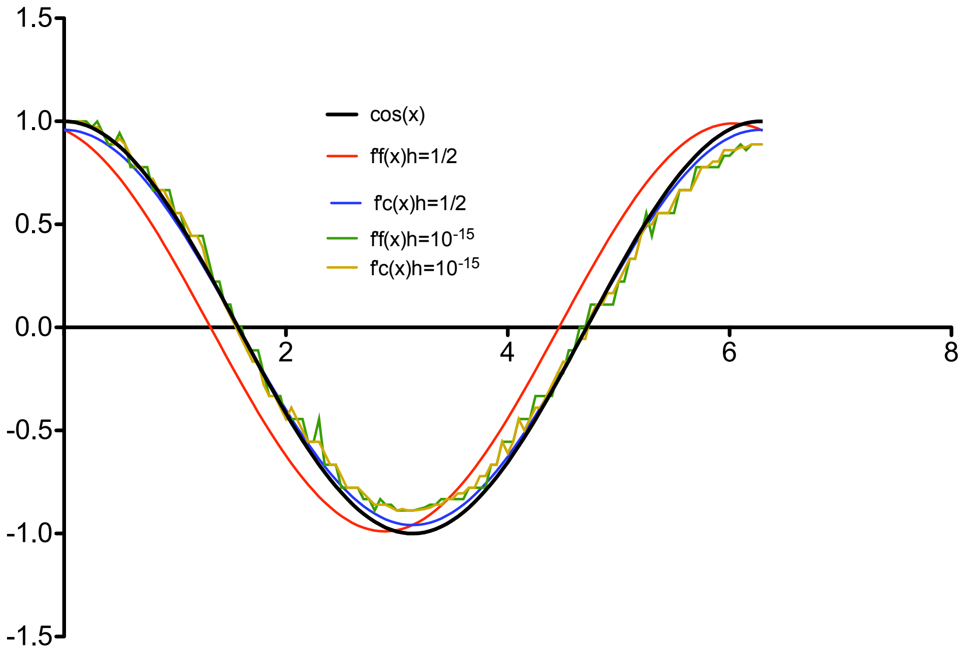

Figure 1.1 shows the effects of too small and too large values of h on both central and forwards difference formulae:

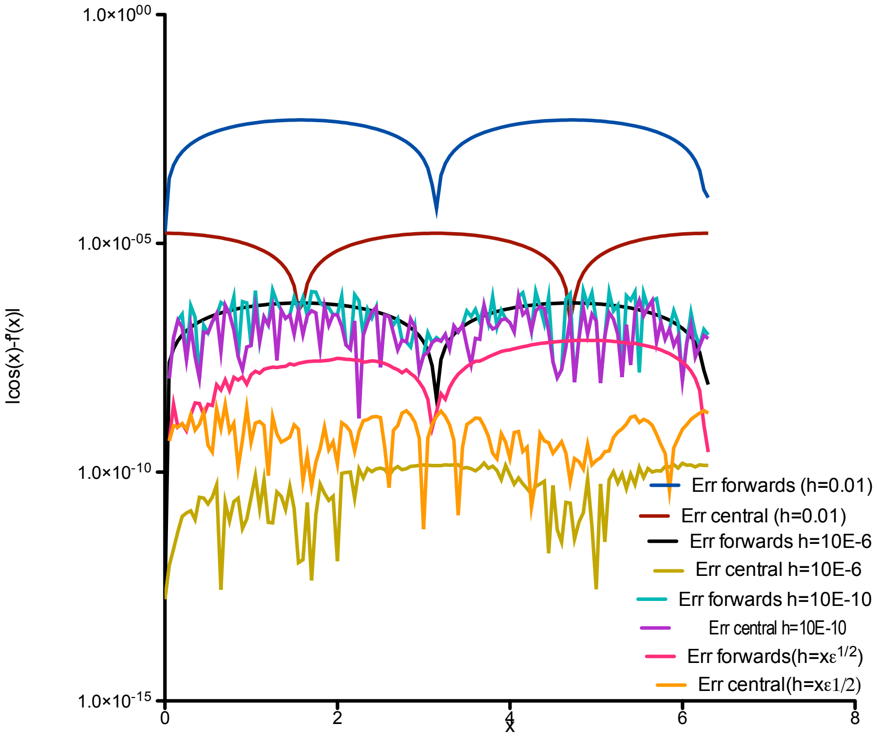

Figure 1.2. This graph shows the modulus of the error (cos(x)-f'x) against x for different values of h, it shows how the error changes from being a smooth (resulting from imprecision in the formula since h is not taken to the limit) to jagged(from imprecision in the rounding of the floating point variables).

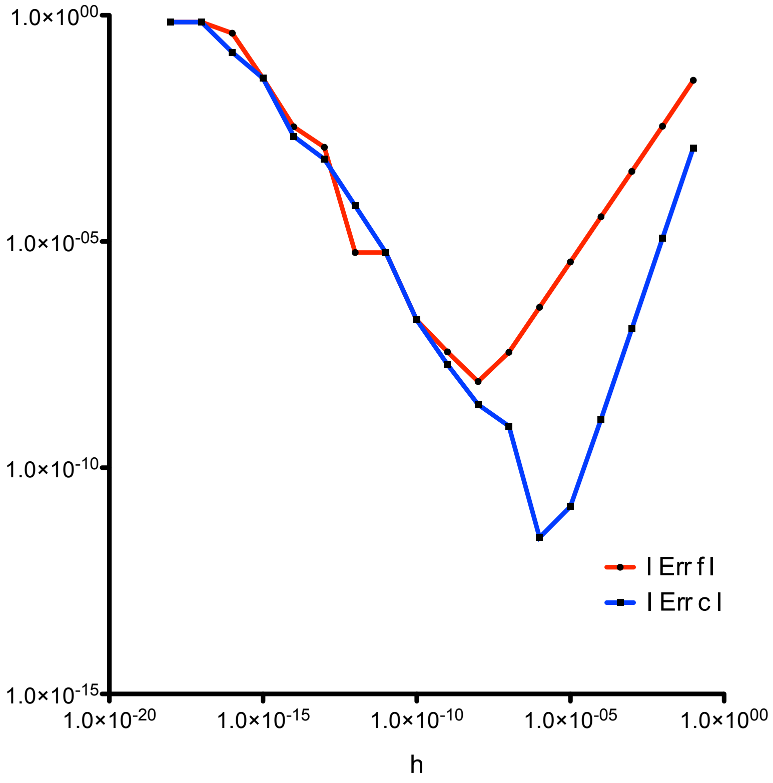

Figure 1.2. shows, for x= π/4, the modulus of the error vs. h. From this we can see that the optimum values of h are: for central difference h=1E-6, and for forwards h=1E-8. The central difference formula is capable of being accurate to about 10^{-12} and the forward formula - to about 1 part in 10^{-7}.

Problem 2: Numerical Integration – Simpson's Rule

→The problem is to write a program that will find the time period of a pendulum when the small angle approximation does not hold. We can find this by evaluating the integral:

$$T = 4\sqrt{\frac{l}{g}}\int_0^{\pi/2} \frac{d\psi}{\sqrt{1-\sin^2\left(\frac{\theta_0}{2}\right)\sin^2\psi}} \qquad (1)$$

To do so I will use Simpsons rule:

$$\int_a^b f(x)dx = \frac{h}{3}\left[f(x_0)+4f(x_1)+2f(x_2)+\ldots+4f(x_{n-1})+f(x_n)\right] \qquad (2)$$

Where $h = (b - a)/n$ and $x_i = a + ih$.

→My code sets l = 1m, g=9.81ms^{-2}. It uses a loop taking θ_0 from 0 to π and calculates the value of T every 0.01. The code inside the loop calculates the sum in (2); it sums up the first term and the last term, then uses loops to sum up all the odd terms, multiplies these by four and adds them to the sum, again for the even terms multiplied by 2. The total is then multiplied by $4(l/g)^{1/2}$ to give the time period. The results are printed in columns on the screen.

→ I have produced a graph to show the effect N on the result:

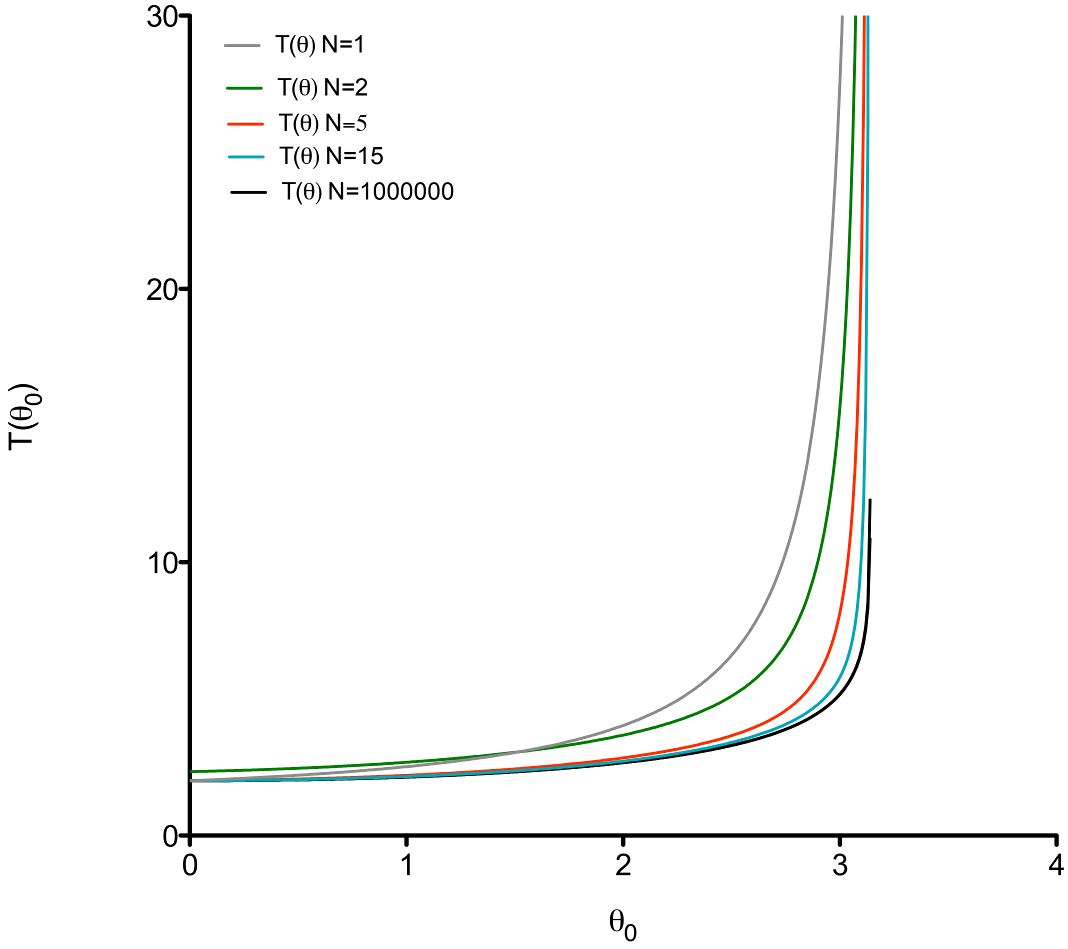

Figure 2.1. This graph shows how the number of points in the sum N, effects $T(\theta_0)$. For low N the time period goes to very high values as θ_0 approaches π e.g. for N=2 the time period for $\theta_0=3.14$ is 1189 seconds which is clearly inaccurate. At higher values of N the time period stays in a smaller range for longer and diverges less at the limit.

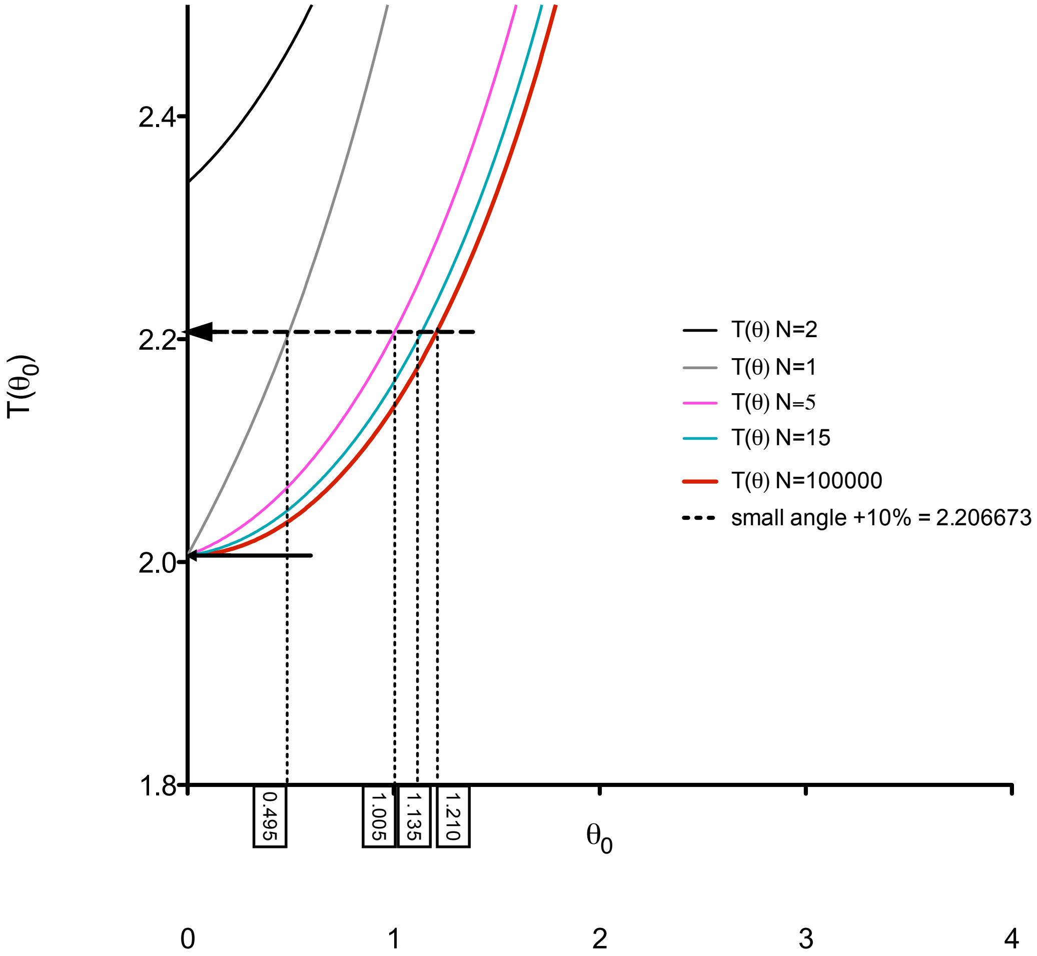

Figure 2.2. In this graph the bottom arrow shows the value of the small angle approximation 2.002, the dashed arrow show that value +10%. The values of θ_0 for at which the time period diverges by 10% from the small angle limit are displayed for a range of N.

If we wished to improve the program, we could alter it to ask the user to input the accuracy required and then set N accordingly e.g. by repeating the sum with increasing N and testing the diffence between subsequent values (which is a estimate of the error) untill it reaches the requirement.

Problem 3 Problem 3: Monte Carlo integration – calculating π

→ The program needs to generate random points in the square: $-1 \le x \le 1$, $-1 \le y \le 1$, most of the points fall inside the circle $x^2+y^2=1$. The ratio of points where $x^2+y^2 \le 1$ to the total number of points, N, is equal to $\pi/4$.

→My code calculates random numbers between -1 and 1 by first generating a number between 0 and 1 by using rand()/RAND_MAX, then assigning that number a + or a - depending on whether a second random number between 0 and 1 is > 0.5, a 50/50 chance.

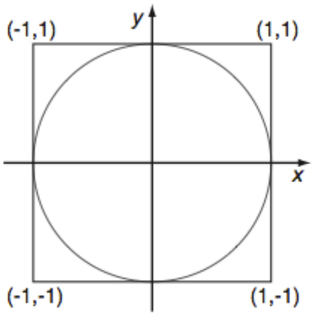

Figure 3.1

Two of these numbers are squared and summed and if this sum is ≤ 1 then the total is incremented. π is found from the ratio multiplied by four. This is repeated N times the result is printed at N=10,10^2, 10^3…10^9. The process begins to take a extremely long time as N increases exponentially.

These are the results:

N=10, Pi=3.600000

N=1E2, Pi=3.200000

N=1E3, Pi=3.092000

N=1E4, Pi=3.147600

N=1E5, Pi=3.146600

N=1E6, Pi=3.141508

N=1E7, Pi=3.141576

N=1E8, Pi=3.141495

N=1E9, Pi=3.141592

As you can see the value has converged to 6 figures of accuracy by N=10^9.

Here is a graph of the results:

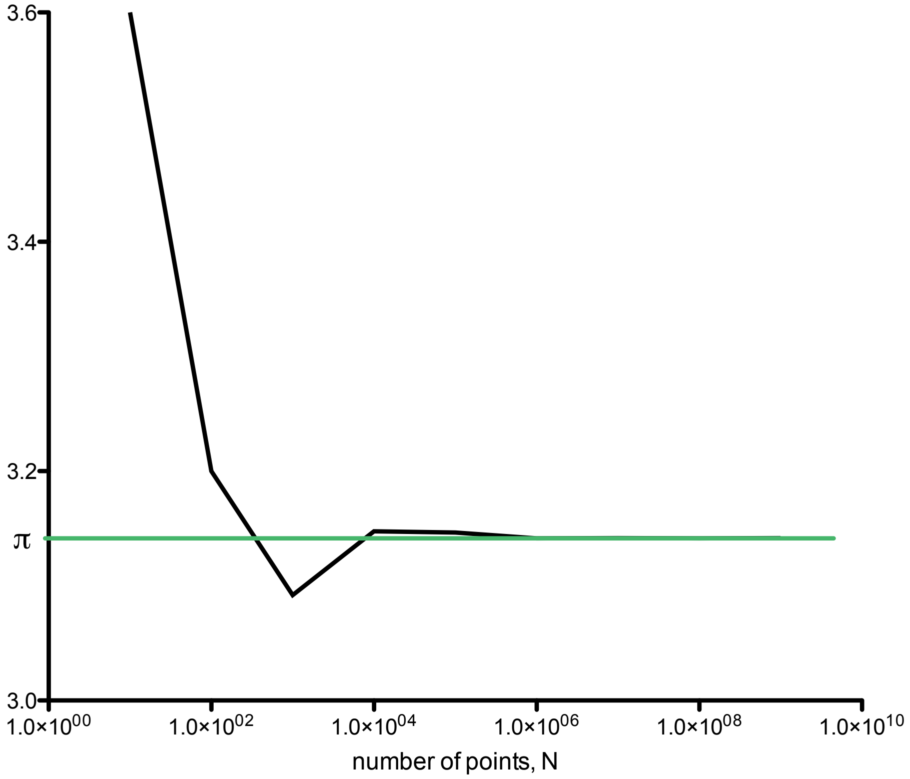

Figure 4.2 shows how the value of converges on π as the number of points increases,