Computational Physics exercise 2

# numerical differentiation of sin[x]

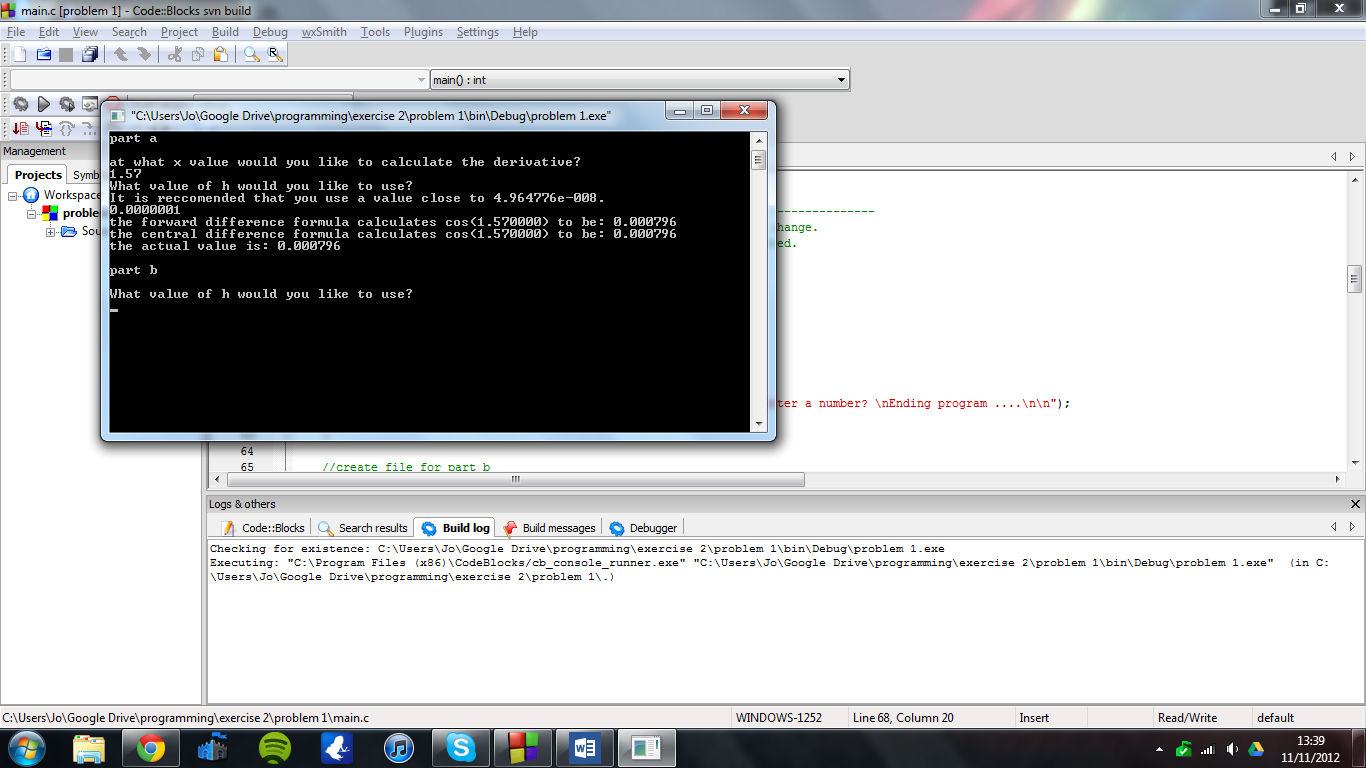
Taking the differential of a function is extremely useful in almost all areas of physics. Generally the analytic derivative will be more accurate, however, occasionally we must use numerical methods. There exist two formula’s the forward and the central difference formula’s for calculating derivatives:

|  |  |
| --- | --- |
| Forward difference formula | Central difference formula |
|  |  |

Where h is a small positive number. If we were to take the limit of h→0 for equation (1) then we would have the analytical derivative. However as we shall see, equation (2) actually tends to be more accurate.

I have written a program that performs three actions:

1. Calculate the derivative of sin[x] at a given x for a given h using both numerical equations and the inbuilt cosine function. A sample output for x=1.57 and h=10-8 is shown below:



1. Investigate how accurate each formula is for various values of x. A text file is produced with the errors in each formula for a range of x. This can then be plotted, as in figure 1.

Figure 1: The errors in both numerical derivatives for various x

We can see that the error in the central difference formula is roughly constant whereas the error in the forward difference formula is generally much larger and depends on the value of x. the error in the forward difference formula is maximized when the derivative of sin(x) is maximized.

1. Investigate how accurate each formula is for different values of h, when x=pi/4. Again a text file is produced, this time with the errors for various values of h. This is plotted in figure 2.

Figure 2: The errors in both equations for various values of h.

Working from large h to small h, we can see that the error in the central difference formula reduces faster than in the forward difference formula. However both errors reach a minimum value and then start increasing. I hypothesize this is due to rounding errors. For very small h we can see that the computer calculates the difference between sin[x+h] and sin[x] to be zero and hence a constant error occurs.

# numerical integration used to find the period of a pendulum

The period of a Pendulum can be approximated for small angles by:

|  |  |
| --- | --- |
|  | (3) |

The full equation is:

|  |  |
| --- | --- |
|  | (4) |

Where θ0 is the initial angle of the pendulum. This is difficult to solve analytically. One might think to try numerical methods. However there is a singularity at the upper limit and hence we introduce a new variable ψ, given by:

Which allows us to rewrite the initial equation as:

|  |  |
| --- | --- |
|  | (5) |

I have written a program that first evaluates this equation for a given initial θ0 and secondly evaluates the equation for a range of θ0 and outputs the results to a text file. It also checks a t which value of θ0 equation (3) is no longer a good approximation (i.e. actual T differs by more than 10%).

Figure 3: the time period of a pendulum for various θ0 as calculated by Simpsons rule.

For this pendulum, equation (3) predicts a timeperiod of 2.456920s. This differs significantly from the actual value when θ0 > 1.18 radians.

We can see from the graph that increasing the value of N (i.e. using smaller h) reduces the height of the final upturn as θ0 approaches pi/2. If the pendulum could be perfectly balanced pointing upwards then it would be in unstable equilibrium and would hence have infinite time period. I hypothesize that this is skewing the calculations. As we use smaller and smaller h, this effect skews fewer results and hence the final peak is smaller.

# monte carlo integration used to calculate pi

Some integrals, particularly multidimensional ones are difficult to solve efficiently even using numerical methods. For this type of problem, we can evaluate the function at many points and take the average value In order to estimate the integral. In other words:

|  |  |
| --- | --- |
|  | (5) |

As we use larger values of N, our approximation will improve. We can select the points at which we evaluate f[x] in several ways. In Monte Carlo integration we randomly select our point, but one could simply evaluate the integral every 0.1 units along the x-axis, say. However, often, we will try and evaluate a multidimensional integral which has complicated limits. To perform this integral we can take the ratio of the area of this complicated shape and a simpler shape and then (provided we know the area of the simple shape) we can then calculate the area of the complex shape. Hence we have reduced the problem to one of calculating the ratio of areas, rather than simply calculating an area. This is easier because we can generate random points and easily check whether or not they are within the area we are investigating. We then count up how many points were and weren’t within the area, and we have an approximation for the ratio of the two areas. We will use this in order to calculate the area of the unit circle:

1. It is easy to generate random numbers that are all within the square created by joining the points (1, 1) (-1, 1) (-1, -1) and (1, -1). This square has area 4.
2. It is easy to check whether one of the randomly generated points above is within the unit circle.
3. Hence we can calculate the area of the unit circle:

I have written a program which first performs this operation for a certain number of points. It then investigates how increasing the number of randomly generated points affects the calculation. We expect the answer to be pi and hence I have plotted a graph of (result-pi):

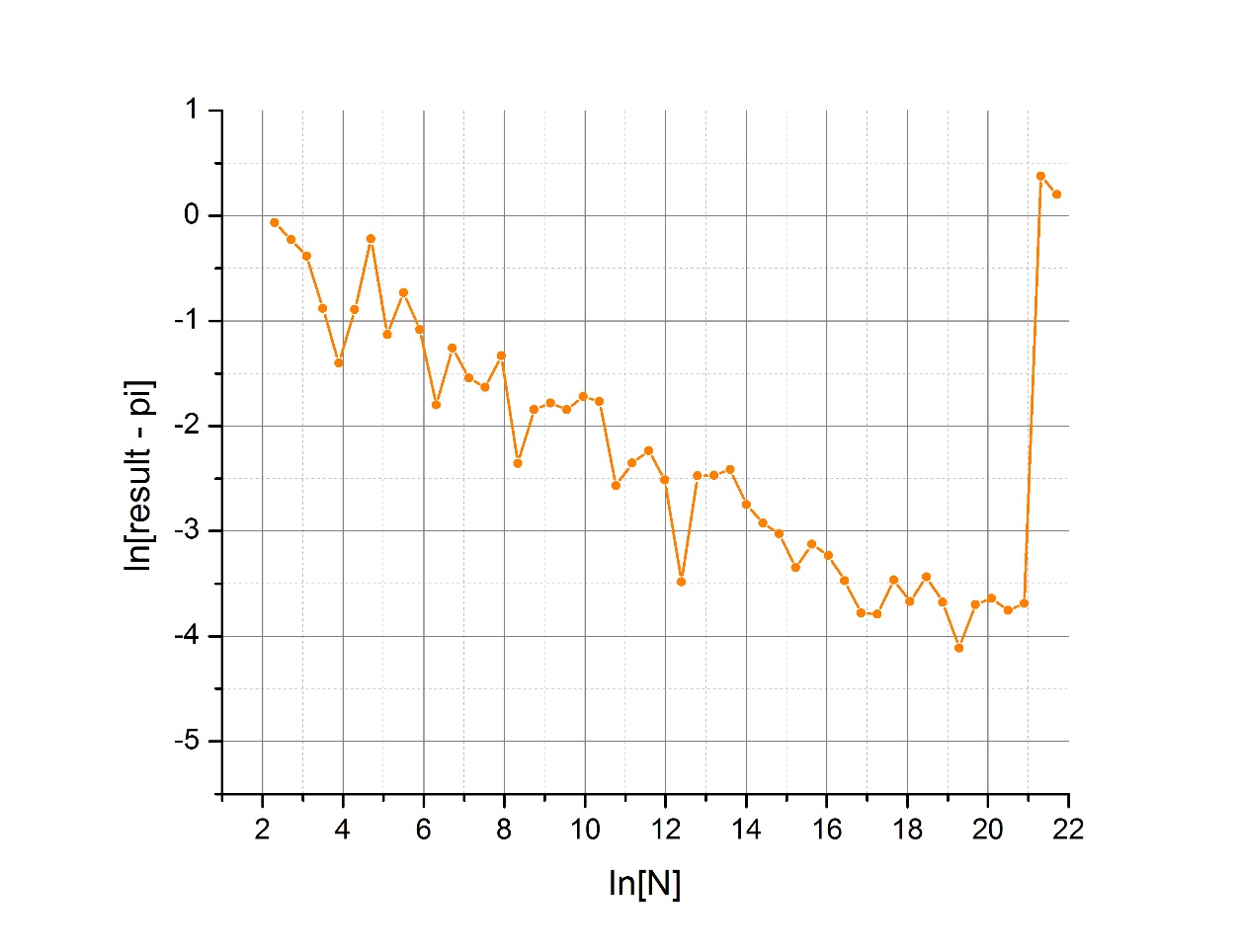


Figure 4: the error in the calculation of pi for increasing numbers of random numbers.

We can see that the error in the calculation of pi reduces for increasing N up to N≈e21. After this point the error is suddenly very large. This is because max\_generate is of type long int and hence it cannot store any number larger than about 2x109, whereas e21.7 ≈ 2.6x109. Regardless we see that ln[error] ln[N]. More specifically, ln[error] ≈ -0.25 x ln[N], which implies that: error ≈ ((constant)N)-0.25 . Hence we see that the error in pi reduces slowly and we notice on the graph that random fluctuations have a massive impact. It seems that this is not a very efficient way of calculating Pi.

My program uses the rand() function to generate random numbers. This is a pseudorandom generator and hence will generate the same sequence of numbers each time. However my program uses the current time as a seed and hence will produce different random numbers each time. Various other pseudo random generators exist:

* The Mersenne twister. This algorithm can quickly generate random numbers in a very long sequence. Despite having a very long period this generator is not suitable for cryptographic applications because only a small number of iterations need to be observed in order to predict the next result.
* Blum Blum Shub. This algorithm is more suitable for cryptographic applications since determining the next number is as difficult as integer factorization. However, blum blum shub is very slow and thus not suitable for my program.

However, for true randomness, physical methods must be used. There exist several quantum mechanical events whose outcome is inherently unpredictable. For example sending a single photon at a semi-silvered piece of glass, there is no way to be certain as to whether the photon will pass through or be reflected. This particular effect is obviously impractical, but there are other processes which can be used: radioactive decay, radio noise and thermal noise for example.