Mid-term project

Šimun Šopar

Problem 1.

Problem 1. was to calculate numerically sine function using it's Taylor series. Code mid1.cpp containts two functions that calculate sine. Both rely on recurrence relation to calculate the terms in series,

$$c_{n+1} = -1\frac{c_n}{(2n+1)2n}$$

That way we can minimize roundoff error and insure no overflow happens. Function *sin* takes a value of *x* for which a sine is to be calculated, number of steps in the series *N* and also a boolean variable *t*. If *t* is true, then it bounds value of *x* to the interval $\langle 0, \pi \rangle$, which we can do since the sine function is periodic. The function *sin2* also takes value of *x* and boolean value *t*, however it does not take number of steps. Instead, it terminates the series when the next terms is smaller then 10^{-7} of the entire sum.

1.) Values of sine for some *x* using the first function, *sin*, are given here, with the corresponding number of steps. After some term *M* all the other terms become practically zero so they don't contribute to the sum. That means we can generally use fewer terms then what I used here. We can see that for values of *x* smaller then 1 and for high number of terms, numerical results give correct results. Results are shown in Table 1. Analytical results were taken from Python.

x	Analytical value	Numerical value	Number of terms
0	0.0	0.0	1
0.1	0.0998334166468282	0. 0998334166468282	61
0.2	0.1986693307950612	0. 1986693307950612	67
0.3	0.2955202066613396	0. 2955202066613396	72
0.4	0.3894183423086505	0.3894183423086505	75
0.5	0.479425538604203	0.479425538604203	78
0.6	0.5646424733950354	0.5646424733950355	81
0.7	0.644217687237691	0.644217687237691	83
0.8	0.7173560908995228	0.7173560908995229	85
0.9	0.7833269096274834	0.7833269096274834	87
1.0	0.8414709848078965	0.8414709848078965	89

Table 2. Value of *sin x* for large number of steps

2.) The same values of sine can be calculated using the other function, *sin2*, which terminates the series differently for different *x*. Results are shown in Table 2. The results are quite precise (at least 7 digit precision), giving that the number of terms used is no more then 5. More precise results can be achived with using more terms, as seen in Section 1.) This section also provides a good insight into how many terms are actually needed. 60 is too much, 5 is still too little, about 10 would work very well for *x* smaller then 1.

x	Analytical value	Numerical value	Number of terms
0	0.0	0.0	1
0.1	0.09983341664682815	0.0998334166666667	3
0.2	0.19866933079506122	0.1986693333333334	3
0.3	0.29552020666133955	0.2955202066071429	4
0.4	0.3894183423086505	0.3894183415873016	4
0.5	0.479425538604203	0.4794255332341270	4
0.6	0.5646424733950354	0.5646424457142858	4
0.7	0.644217687237691	0.6442176877315009	5
0.8	0.7173560908995228	0.7173560930426808	5
0.9	0.7833269096274834	0.7833269174484375	5
1.0	0.8414709848078965	0.8414710097001764	5

Table 2. Values of *sin x* using using termination condition

3.) We can examine the results for x close to 3π, we know the analytical result is zero. That means finite terms in the series should cancel eachother out, which can be seen in C++ code. The code prints out value of numerical sine for different number of terms. We can see that for few terms the results is very imprecise. However, for about 10 terms, the values cancel out, the result goes to zero.

4.) Functions *sin* and *sin2* have a built-in functionality to use value of *x* between zero and π instead of it's real value for larger *x*. This can be done since sine is a periodic function. The way it works is it takes integer value, $n = \lfloor x/2\pi \rfloor$, and subtracts $n \cdot 2\pi$ from *x*, bounding *x* to interval from 0 to 2π. Then if *x* is larger than π, it subtracts π and changes the value of variable sign from *1* to *-1*. It calculates the sine for such *x* and multiplies it with sign, since $sin(x + \pi) = -sin(x)$. Comparison between results using bounding and not using bounding is given in table below.

N	X1 = π + 0.5752220392	X2=10	Exact
1	-0.575222039230621	10.000000000000000	-0.544021110889369
4	-0.544021091946269	-1307.460317460317800	
7	-0.544021110889370	548.965150076261350	
10	-0.544021110889370	-16.811850137411682	
13	-0.544021110889370	-0.462384216212647	
16	-0.544021110889370	-0.544127277005709	
19	-0.544021110889370	-0.544021064696442	

Table 3. Values of *sin 10* (analytically, it is the same as *sin (π+0.5752...)*)

We can see that the bound *x* gives better result. The reason is that **X1** is smaller so Taylor expansion works better (lesser truncation and round-off errors). Mathematically, these two X's should give the same answer, but since we can't you infinite number of terms (truncation error) and for bigger X the terms in sum are bigger (leading to round-off error), results are better for smaller X. What function *sin* actually does when bounding *x* to a smaller interval is it calculates the Taylor series around a point $n \cdot 2\pi$. Since that point is closer to our X than 0, it gives better results.

This is really useful for us because it means we can calculate sine for any *x*, we don't have limitations on too big values of *x* since the program works well for $x \in \langle 0, \pi \rangle$

5.) For different *x* we can see how numerical solution loses accuracy as *x* increases. That is expected since the terms in a finite sum become very big, so we get trunctaion and round-off error.

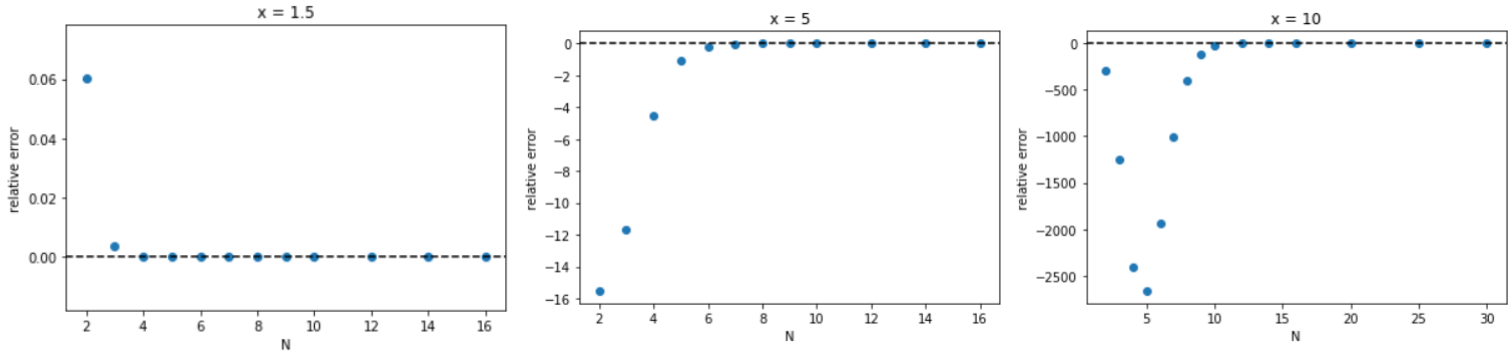
x	Numerical solution	Analytical solution
1	0.841470984807897	0.8414709848078965
2	0.909297426825682	0.9092974268256817
3	0.141120008059867	0.1411200080598672
4	-0.756802495307927	-0.7568024953079282
5	-0.958924274663139	-0.9589242746631385
6	-0.279415498198929	-0.27941549819892586
7	0.656986598718796	0.6569865987187891
8	0.989358246623423	0.9893582466233818
9	0.412118485241764	0.4121184852417566
10	-0.544021110889361	-0.5440211108893698
20	0.912945252478319	0.9129452507276277
30	-0.988008301616039	-0.9880316240928618
40	-60.242038924540282	0.7451131604793488
50	-337533696342.174070	-0.26237485370392877
60	-30894560812706644000.0	-0.3048106211022167
70	-16315595612460545000000000.0	0.7738906815578891
80	-10683434486021721000000000000000	-0.9938886539233752
90	-14213558317646852000000000000000000000	0.8939966636005579
100	-53844709328314655000000000000000000000000	-0.5063656411097588

Table 4. Values of *sin x* for different *x* and 30 steps.

The program starts to lose accuracy for *x* between 30 and 40.

We cannot calculate sine of arbitrarily large number using Taylor expansion around 0. The solution to this problem lies in bounding the values of *x*, as seen in previous section.

6.) Series of error plots were made in Python code 1.iypnb for different *x*. Some of them are below. More can be seen in Python code.



Graph 1. Relative error plots. We can see how drastically bigger errors become with increasing *x*.

Problem 2.

Problem 2. was to calculate integral $\int_0^1 e^{-t}dt = 1 - 1/e$ using trapezoidal and Simpson rule, and also Gaussian quadrature. The code is in mid2.cpp file.

1.) Legendre-Gaussian quadrature

First part of C++ code calculates the integral using Legendre-Gaussian quadrature. It consists of functions *Legendre,* which takes a value of *x* and an integer *N* and calculates the value of N-th Legendre polynomial at position *x* (this function is taken from the textbook). Second function, *Legendre_np*, calculates points where a Legendre polynomial of order *N* is zero. This is needed for the quadrature. Third function, *weight*, calculates the weights for integration using the formula

$$w_i = 2\frac{1 - x_i^2}{N^2\left(x_i L_N(x_i) - L_{N-1}(x_i)\right)^2}$$

where x_i are zero points of the Legendre polynomials of order N. Finally, function *Gauss* calculates zeros and weights for given number of points and returns the solution to the integral:

$$\int_0^1 e^{-x}dx = \int_{-1}^1 \frac{1}{2}e^{-0.5(x+1)} = \frac{1}{2}\sum_{i=1}^{N} w_i e^{-0.5(x_i+1)}$$

This method requires far fewer points then trapezoidal or Simpson. Table below shows results.

Number of points *N*	Numerical solution	Error	Analytical solution
2	0.631978778833793	2.242926492164429E-004	0.632120558828557
5	0.632120540117983	2.959969255830375E-008	
7	0.632120516888703	6.634787182019549E-008	
10	0.632120545904011	2.044632999892299E-008	

Table 5. Results of integration using Legendre-Gaussian quadrature

We can see how for a small number of steps the solution is correct to the precision of 7 digits. Even for only two points, the solution is quite accurate. There isn't much difference in accuracy for N = 5, 7 and 10. Since the zeros of polynomials are calculated numerically, they have some small numerical error which has to be taken into consideration when talking about the error of the method. However, the main error stems from the fact that exponential function cannot be expressed as a finite polynomial – it has infinately many terms in Taylor series. Legendre-Gauss quadrature gives exact results for polynomials of order 2N-1. Since exp is not a polynomial, error occurs.

Trapezoidal and Simpson rule

For trapezoidal and Simpson rule, larger number of steps is required. Table bellow shows relative errors for these methods.

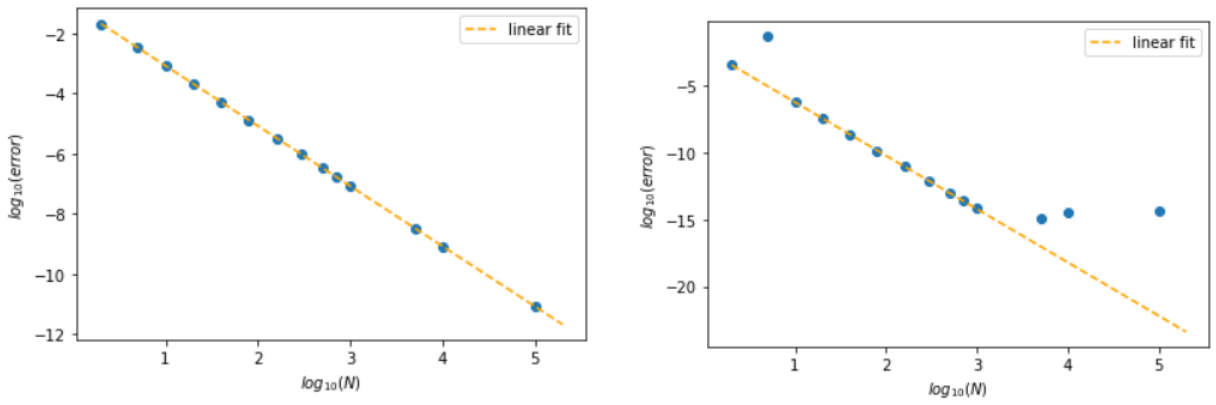
N	Error for trapezoidal	Error for Simpson
2	2.0747041268E-002	3.3715273476E-004
5	3.3311132254E-003	4.2656563806E-002
10	8.3319447750E-004	5.5489487462E-007
20	2.0832465329E-004	3.4711891843E-008
40	5.2082790806E-005	2.1699780908E-009
80	1.3020799426E-005	1.3563213798E-010
160	3.2552062155E-006	8.4784137012E-012
300	9.2592575601E-007	6.8813674065E-013
500	3.3333331335E-007	9.0627503362E-014
700	1.7006802183E-007	2.4413222805E-014
1000	8.3333334879E-008	6.6741184647E-015
5000	3.3333380600E-009	1.0538081786E-015
10000	8.3333763252E-010	3.3370592323E-015
50000	8.3382572134E-012	4.0395980181E-015

Table 6. Results of integral using trapezoidal and Simposon method

For trapezoidal rule about 1000 steps gets the same precision as 10 steps for Gaussian quadrature. For Simpson it's about 30. However there is difference in calculation time. While for Simpson and trapezoidal rule the time is almost zero, the Gaussian rule took 3 seconds to complete. That makes sence since we have to find the zeros of Legendre polynomials and to do that precisly we have to calculate the value of polynomials for a lot of points, which takes time. This is the trade-off of precision. (It's important to note that the code is not optimized. The computation time can be signifactly reduced with proper algorithms. This however ilustrates the difference between methods for raw, un-optimized code).

2.) Errors for all three methods are calculated and printed in the C++ code. They can be seen in the code and also in the tables above (Table 5 and Table 6.) . Plots for relative error are given in the next section.

3.) Relative errors were plotted in Python code 2.ipynb for trapezoidal and Simpson rule (there weren't enough points to plot for Gaussian quadrature).



Graph 2. Relative error plots for trapezoidal (left) and Simpson rule (right)

We can see that the logarithms of the values have a linear tendencies for most of the points. By excluding the points that significantly deviate from this linearity, we can calculate the equation of the corresponding line.

4.) Logarithm of relative errors seem to have a sort of linear dependence on logarithm of N. There dependences can be determined, they are shown with an orange line in plots above. These lines represent truncation error, and we can see how it becomes smaller for larger *N*. The reason point deviate from the line in Simpson rule for large *N* is because of round-off error which becomes more visible as *N* increases and truncation error decreases. These deviations aren't noticable for trapezoidal rule since the truncation error is still dominant to the round-off error for these number of points.

From the error plots we can determine the number of decimal places of precision in numerical solution. For each point, the absolute value of *y* coordinate corresponds to precision. So for N = 10000, solution is precise to 10 decimals. As N increases, precision also increases, which is as expected.

Problem 3.

Problem 3. was to calculate the solution of aharmonic oscillator using RK4 method. The aharmonic oscillator equation is

$$\ddot{x} = -\frac{k}{m}x^{p-1}$$

where *p* is an even positive integer. The reason *p* has to be even is so the force would always have a sign opposite from position, it's a restoring force. The problem has a constant of motion, energy. That is because this is a time independent ordinary differential equation. We expect, and will show, the motion to be oscillatory, because of the restoring force.

The problem is solved in code mid3.cpp. The code takes value of *p* which will be from interval 2-12. It also takes amplitude of oscillations (which is initial position, since initial velocity is zero) and number of steps. It than calculates position and velocity using RK4 method.

1.) We can check that the amplitude remains constant. From conservation of energy, we know amplitude will be maximal when the velocity is zero and that velocity will be maximal when position is zero. Using a simple search algorithm, I determined the maximal values of *x* and *v* from calculated solutions. Than I determined moments for which $x = 0$ and compared the velocity at that moment to the maximal velocity. This was easy to do. Since solutions are saved in an array, I simply went through the array in a *for* loop comparing elements $(i - 1)$ and i, this corresponds to positions at time $h*(i - 1)$ and $h*i$. If the adjacent elements have differents signs, we have found our zero. We then print velocity at time $h*i$, that is the i-th element of velocity array, and compare it to maximal velocity. I have chosen to take position at $h*i$ as zero as opposed to $h*(i - 1)$. It would make no difference to choose the other one, since they are both close to zero. Since our time variable i discrete, we are actually looking at a position close to zero, but not zero. This means the velocity we are looking at is not maximal velocity, rather something very close. That is why there is some difference between maximal velocity and velocity for certain zero-positions. To get better solutions, we can use smaller *h*, but not too small so we can avoid round-off error.

We can do the same for velocity, we find the zero of velocity and print out the corresponding position, comparing it to maximal position.

We can say the amplitude is conserved (This can also be seen graphically).

We can also check time difference between two consecutive moments at which position is zero. We can see that this difference is the same for any two consecutive moments, that is, the motion is periodic (for further explanation see Section 4.)

2.) It is easy to varify that for $p \neq 2$ the motion is nonisochronus. We simply take different values of amplitude and the program will print out the periods, for which we can see are not the same. For a more visual proof, see Section 5.). Here is a Table comparing some periods for p = 4.

Amplitude	**Period**
1	5.327333
1.2	4.442833
1.4	3.802333
1.6	3.334667

Table 7. Period of oscillations for different amplitudes for $p = 4$

For larger amplitudes, the period is smaller.

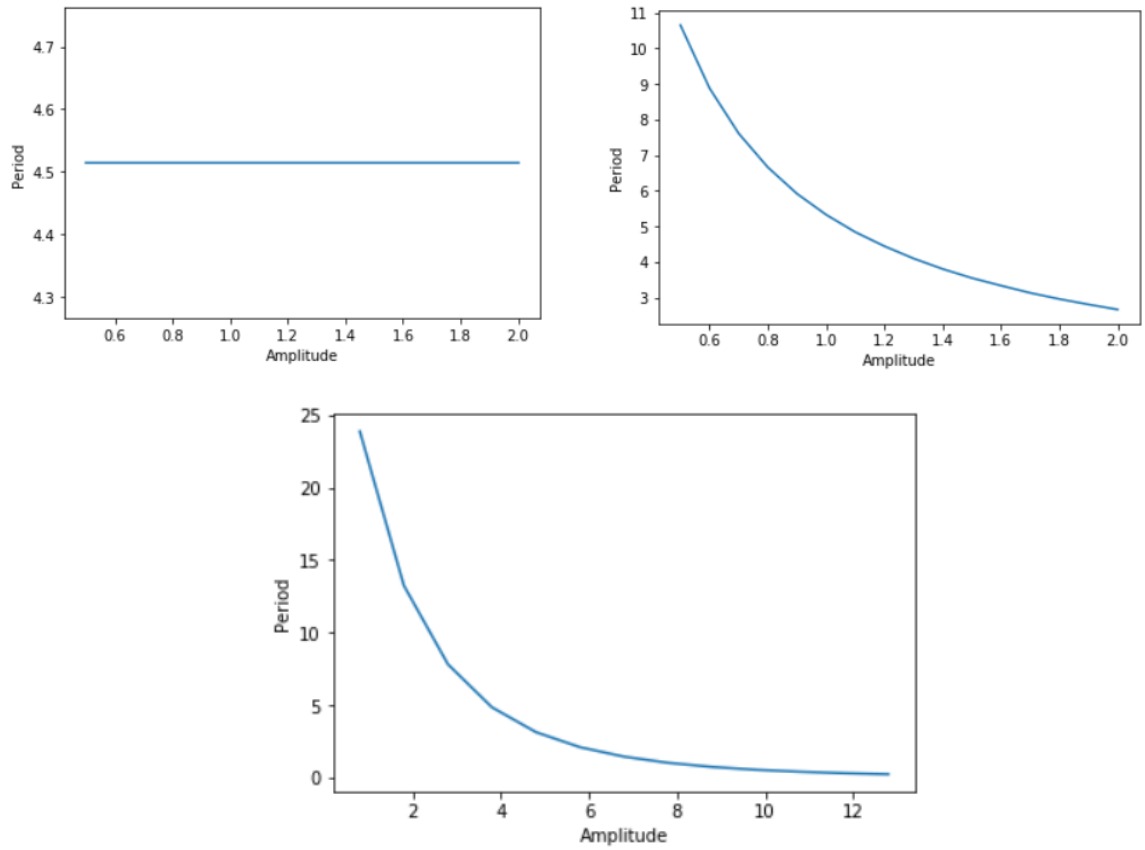
For $p = 2$, the problem is a harmonic oscillator, the period does not depend on the amplitude, as expected.

3.) The solution of this problem for any of the *p* is oscillatory, however, the shape of oscillations change for different powers. This is expected since the differential equation changes for every problem. For p=2 solution is harmonic oscillator, it can be caluclated analytically. For other p, solution isn't harmonic. We can see mathematically why the solution changes. If we exapand the solution in a power series, then from plugging that series into the equation we get a system of algebraic equations for the coefficients of the series by comparing coefficients next to the same powers of *t*. On the left, we get simple terms, while on the right we raise the power of an infinite series, leading to a lot of complex terms. Since that power is different for every *p*, the coefficients will be different so the solutions will be different (Another approach would be to exand solution in a Fourier series, the conclusion is the same). Physically, larger *p* mean stronger spring. Meaning that for the same initial displacement, higher *p* spring will achieve greater acceleration for a small position change and will have greater impulse in the origin. This will lead to more „pointy“ (or triangular) look of the solution. Change of velocity, which is a tangent to the solution, will change more rapidly in the same time interval for larger powers. Also, because of a larger restoring force, the oscillations will be quicker, the mass will have a greater tendency to return to origin (see Section 6.)

4.) The algorithm for detemining period is similar to that for checking amplitude conservation (see 1.) We look for moments when x = 0 and where the slope of the function (the velocity) is positive. Difference between two consecutive such moments is the period. We are looking for moments where *x* is aproximately zero, and *x* is zero between two points of opposite signs. We take the positive one to be

zero-point. That means the real period is $T \pm h$, where T is the calculated period. That is also the reason why sometimes calculated period differ slightly. This can be easily proven. Let's imagine a discrete time mash. We find two adjecent moments t_1 and t_2 on that mash that have opposite signs of calculated position, meaning the moment at which x = 0, let's call it τ, is somewhere in between those moments. The program takes t_2 as aproximate τ. Then it finds next two such moments t_1' and t_2' and calculated the difference $t_2' - t_2$ as the period. We can determine the smallest possible value of T the program can calculate. If the value of t_1 and t_2' are $t_1 = \tau - \epsilon, t_2 = \tau' + \epsilon$, then $T = t_2' - t_2 = T_{real} - h$ (where T_{real} is the analytical period and $\tau' = \tau + T_{real}$) for $\epsilon \to 0$. The same can be determined for highest value of T.

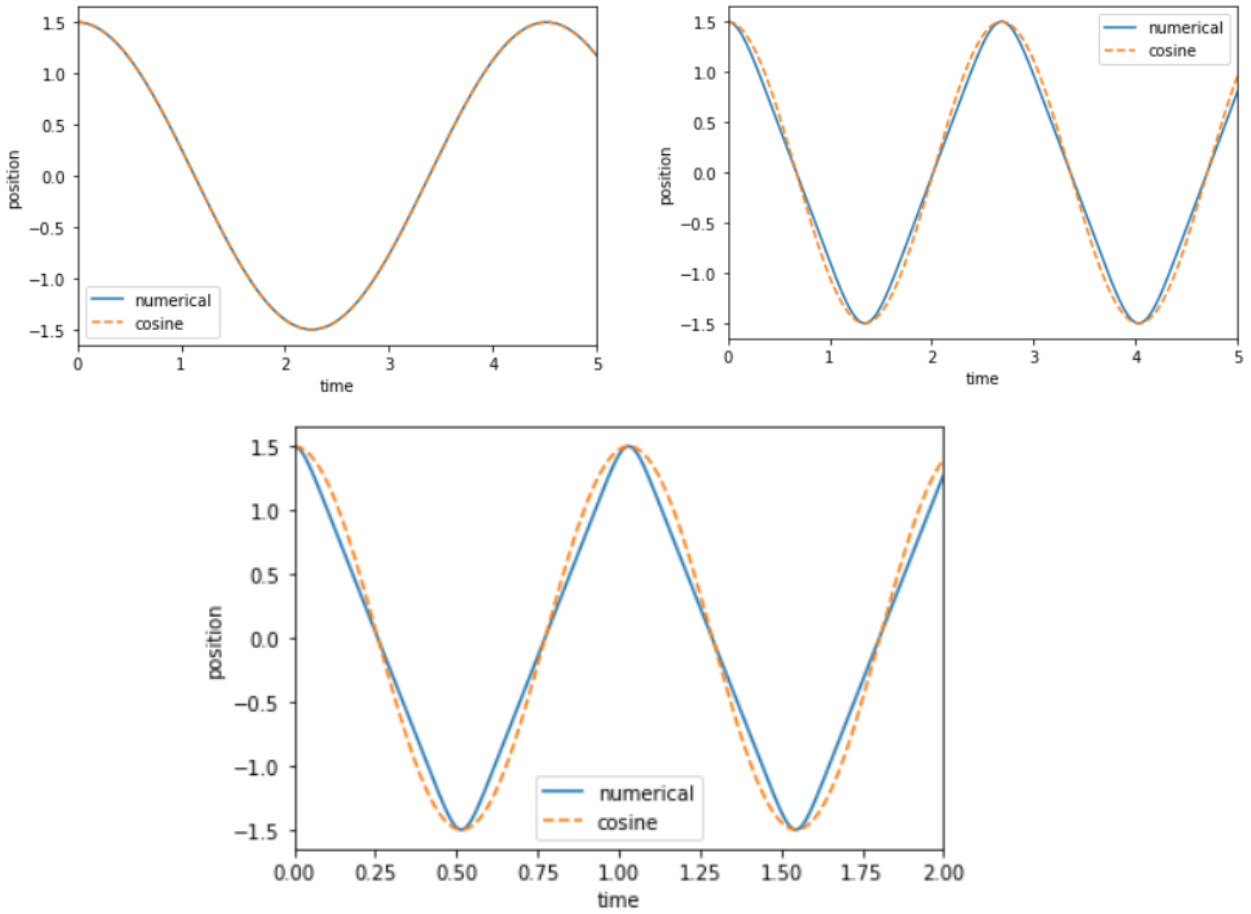
5.) Graphs of period as a function of amplitude are shown below. The graphs were plotted in Python. Number of points used for plotting was around 10, that is the reason the graphs look rough. We can get finer, more continues graphs by using more points.



Graph 3. Graphs of period depending on amplitude. We can see for p = 2 (left) motion is isochronous, while for p = 4 (right) and p = 12 (below) period gets smaller for bigger amplitudes

As already seen in Table 7., period gets smaller with larger amplitudes. Period also gets bigger with increasing *p* (for the same amplitude).

6.) Harmonic motion is sinusoidal. A graphical way to verify if motion is harmonic or not is simple. We have calculated the period and we know the amplitude, so we can plot the motion against a cosine function and see if the motion is harmonic.



Graph 4. Graphs of position plotted against cosine function of same amplitude and period for p = 2 (left), p = 4 (right) and p = 12 (below)

As expected, motion is harmonic only for *p* = 2.

It is interesting to note that for incresing *p* the motion gets more and more triangular (shaped like a triangle). Also, for the same values of *k* and *m*, and for the same amplitude, larger powers oscillate quicker.