Programming Project

PX2015 - Assessment 2 - Programming Project

Sidney Pauly
52104132

Abstract

This document contains the solutions to the PX2015 Assessment 2, programming project. All source and output files can be found in the github project linked below.

University of Aberdeen
Scotland
UK

`https://github.com/sidney-pauly/papers`

1 General implementation notes

In solving the assignment some techniques and modifications where made that go beyond what was shown in the course. They where made to have more performant or easier to read code Those include:

1.1 Fast rk solve

A rk solve method was implemented to achieve better performance. The main improvement was achieved by preallocating the output array (rksolve line 12-24):

```
12     % Calculate how many timesteps there will be
13     len = floor((tf - t0)/dt);
14 
15     % Get the lenght of the x vector to create an approprialy shaped array
16     xlen = length(x0);
17 
18     % Preallocate the resulting matrix, with one entry per timestep
19     % and appropriatly sized vectors for x
20     tvec = zeros(1, len);
21     xvec = zeros(xlen, len);
22 
23     tvec(1, 1) = t0;
24     xvec(1:xlen, 1) = x0;
```

this improvement means that matlab only has to change a single value in the result array for each iteration, instead of creating a new array every time. Note that writing to the array works a bit differently as well (See lines 23 and 24)

1.2 Method factories

In the course globals are used to set constant parameters of the differential functions. As globals have the disadvantage of only existing once (they can only be set once and will be used in all the methods) and being less clear from a code standpoint (it is not clear what effect it has to set which global variable), the pattern was changed to use factories. A simple example is the Tout method that gets created as such:

```
function result_fx = make_Tout(tmin, tmax)
    % tmin, tmax: min and max temperature
    % make_TOut creates a new instance of
    % the Tout function with the provided
    % arguments for tmin and tmx

    % Assign the resoulting Tout method as a return value
    result_fx = @Tout;

    % Define the method to be returned
    function y = Tout(x, t)
        % x and y are decimal numbers

        y = tmin + ((tmax-tmin)/2)*(1 + cos(2 * pi * sin(pi * t / 2)^2 ));
    end
end
```

1.3 Saving the plots to the filesystem

To have the plots generated by the matlab scripts available in the typed document, they get saved to the filesystem:

```
saveas(f, '../output/assignment2.png');
```

2 Task 1

The first task is concerned with finding out how a pendulum behaves given different initial angles. A pendulum can be described by the following differential equation (Given in the lecture notes):

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}sin(\theta) = 0 \quad (1)$$

For small angles $\theta << \pi$, $sin(\theta) = \theta$ can be assumed. This makes it possible to get to an analytical solution to equation 1:

$$\theta(t) = \theta_0 cos(\omega t) + \frac{\theta_0'}{\omega} sin(\omega t)$$

this gives an analytical solution to the period of the pendulum which only depends on the physical parameters of the pendulum itself and not it's initial conditions:

$$T = 2\pi\sqrt{\frac{l}{g}} \quad (2)$$

As this solution relies on the small angle approximation it will be less precise as the angle increases. To get a more accurate solution to the pendulum's period a computational approach can be used. This involves setting up a function which returns the new state of the pendulum at the next time step. This method was provided in the assignment:

```
function result_fx = make_pend(g, l)
    % g: gravity
    % l: length of the pendulum
    % Make pend creates a new instance of
    % the pend function with the provided
    % arguments for g and l

    result_fx = @pend;

    function y = pend(x, t)
        % x is a vector , and so is y

        % now we name the components of x
        % so that their meaning is clear
        theta = x(1);   % the 1st comp. of x is theta
        v = x(2);       % the 2nd comp. of x is v

        % now finally we calculate the rhs of
        % the diff. equations, and return those
        % as a row vector
        % IMPORTANT: This was changed so my own rksolve method works with this
        % otherwise the vectors rotate by 90 every trial, which is not desirable
        y = [v, -(g/l)*sin(theta)];
    end
end
```

The pendulum's state over time was to be calculated and plotted on a graph for the given initial angles of $\theta_0 \in \{0.2, 1.0, 2.0, 3.0, 3.14\}$. The periods where then to be compared with the period given by equation 2.

The following matlab code accomplishes this:

```
% Task 1

% Define a method calulating the pendiulum's period with the small
% angle approximation
```

```
small_angle_apprx = @(g, l) (2*pi*sqrt(l/g));

% Define the constants to be used
initial_angles = [0.2, 1.0, 2.0, 3.0, 3.14]; % The list of intial angles to be examined
g = 9.8; % gravity
l = 2; % length

% The number of initial angles to be examinend
len = length(initial_angles);

% Define a result matrix with one row for each value and one column for each
% intital angle plus one for the titles
comparison_result = cell(len+1, 4);

% Set the title for the output data
comparison_result{1, 1} = "Initial angle (rad)";
comparison_result{1, 2} = "Period (numerical solution)";
comparison_result{1, 3} = "Period (analytical solution)";
comparison_result{1, 4} = "Error";

% Iterate over all initial angles and examine how the pendulum behaves over time
for i = 1:len

    % Run rk solve with the different initial angles
    [times, pos] = rksolve(make_pend(g, l), 0, 30, [initial_angles(i), 0], 0.01);

    % Create a new figure for each run
    f = figure(i);

    f.Name = sprintf('Initial angle: %f rad', [initial_angles(i)]);

    angles = pos(1, :); % Select the first row of the data containg all angles
    velocities = pos(2, :); % Select the second row of the data containing all agular velocities

    % Plot the angle
    subplot(2, 1, 1)
    plot(times, angles, 'LineWidth', 2);
    axis([0, 30, min(angles)*1.5, max(angles)*1.5])
    title(sprintf('The angle of a pendulum with initial angle %.2f rad over time', initial_angles(i)))
    xlabel 'Time (s)';
    ylabel 'Angle (rad)';

    % Plot the velocity
    subplot(2, 1, 2)
    plot(times, velocities, 'LineWidth', 2);
    axis([0, 30, min(velocities)*1.5, max(velocities)*1.5])
    title(sprintf('The velocity of a pendulum with initial angle %.2f rad over time', initial_angles(i)))
    xlabel 'Time (s)';
    ylabel 'Velocity (rad/s)';

    % Save the plot to hte file system for later use
    saveas(f, sprintf('../output/assignment1/%.2f_rad.png', [initial_angles(i)]));

    % Find out the zeros with the zero crossing function using
    % the angles
    calc_zeros = zerocrossing(times, angles);

```

```
62      % Calculate the average distance between the zeros
63      % This will give 0.5*period as the pendulum goes through zero
64      % twice for every swing
65      % 1. Add all the distances up
66      len_zeros = length(calc_zeros)-1;
67      T = 0;
68      for j = 1:(len_zeros)
69          T = T + (calc_zeros(j+1)-calc_zeros(j));
70      end
71
72      % 2. Devide through the differences to get the average and
73      % multiply by two to get the actual period
74      T_numerical = (T / len_zeros) * 2;
75      T_apprx = small_angle_apprx(g, l);
76      error = abs(T_numerical - T_apprx);
77
78      % Assign the result column and limit each number to two
79      % significant digits
80      comparison_result{i+1, 1} = sprintf('%.2f', initial_angles(i));
81      comparison_result{i+1, 2} = sprintf('%.2f', T_numerical);
82      comparison_result{i+1, 3} = sprintf('%.2f', T_apprx);
83      comparison_result{i+1, 4} = sprintf('%.2f', error);
84  end
85
86  % Write the result to the file system as a .csv file
87  writecell(comparison_result,'../output/assignment1/comparison_table.csv');
```

First a simple helper method gets defined which is equivalent to equation 2 [1]. This method can be called later to obtain the analytical solution.

In the subsequent lines 7 to 10 the initial conditions θ_0, constants g (gravity) and l (length of the pendulum) get defined. Next a cell (like a matrix but can contain any datatype), is created to hold the resulting results for the period. To later create a well formatted csv file, titles for each column get assigned added to the cell.
With the setup being done the pendulums behavior can now be calculated and the result plotted. This is done by looping over all the initial angles θ_0, adding results to the aforementioned *comparison_result* cell and a graph for each iteration.

To get the numerical result for the equation 1 a numerical differential solver method is used. In this case the Runge-Kutta equation is used as requested in the assignment. The solving method takes the method to be solved (*pend*), the initial time $t_0 = 0s$ the final time $t_{final} = 30s$, the initial conditions θ_0 (different per iteration) and $\omega_0 = 0rad/s$ as well as the time step $\Delta t = 0.01s$. The result will be a matrix containing a resulting position θ and velocity ω at each time t as parameters. The resulting (angular) positions and (angular) velocities are then plotted:

[1] Note that this is being done by defining a inline function. An inline function is a function that is defined within one line, instead of putting it into a separate file. The definition of such a method works by providing typing an @ followed by all required parameters in delimitating brackets "(x)". The function content is then directly put after the bracket. Instead of assigning the result the value produced by the expression directly acts as the return value

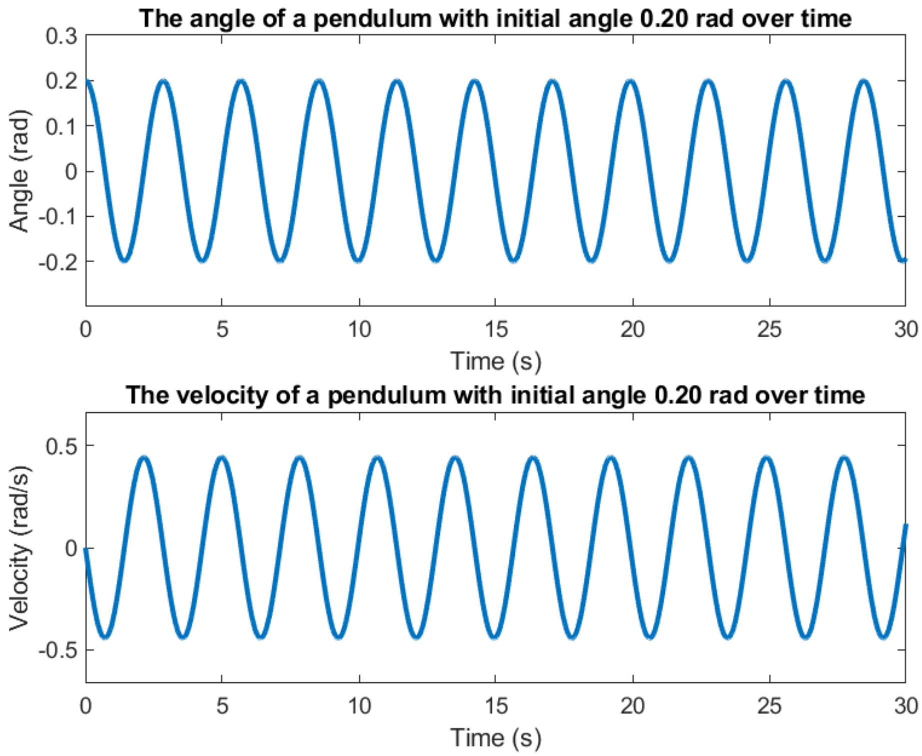

Figure 1: Behavior of the pendulum with initial angle of 0.2 rad

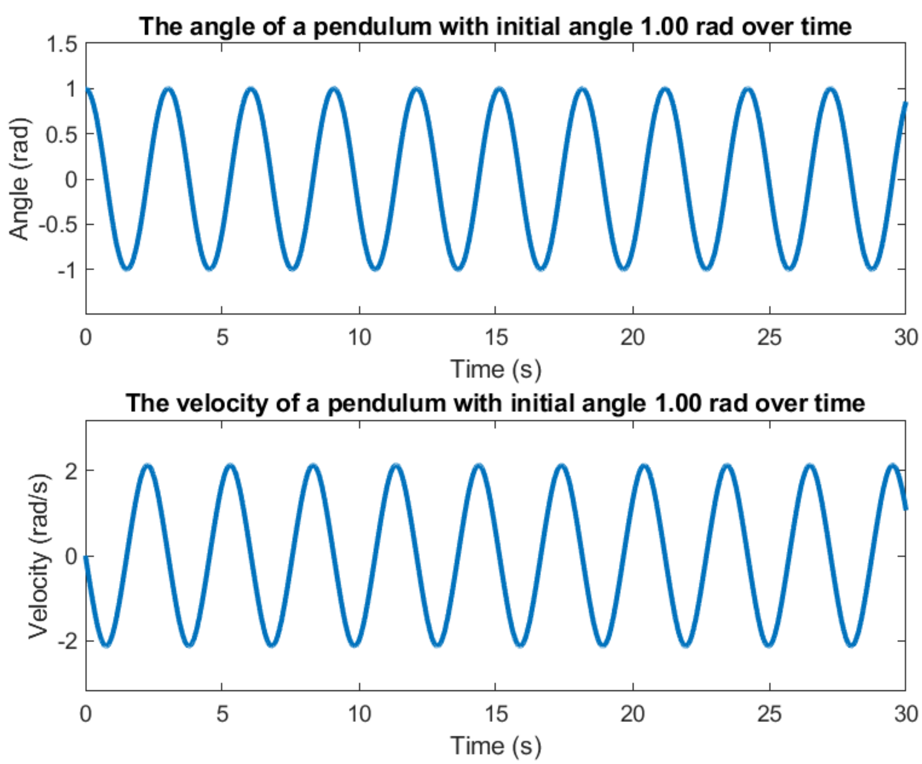

Figure 2: Behavior of the pendulum with initial angle of 1 rad

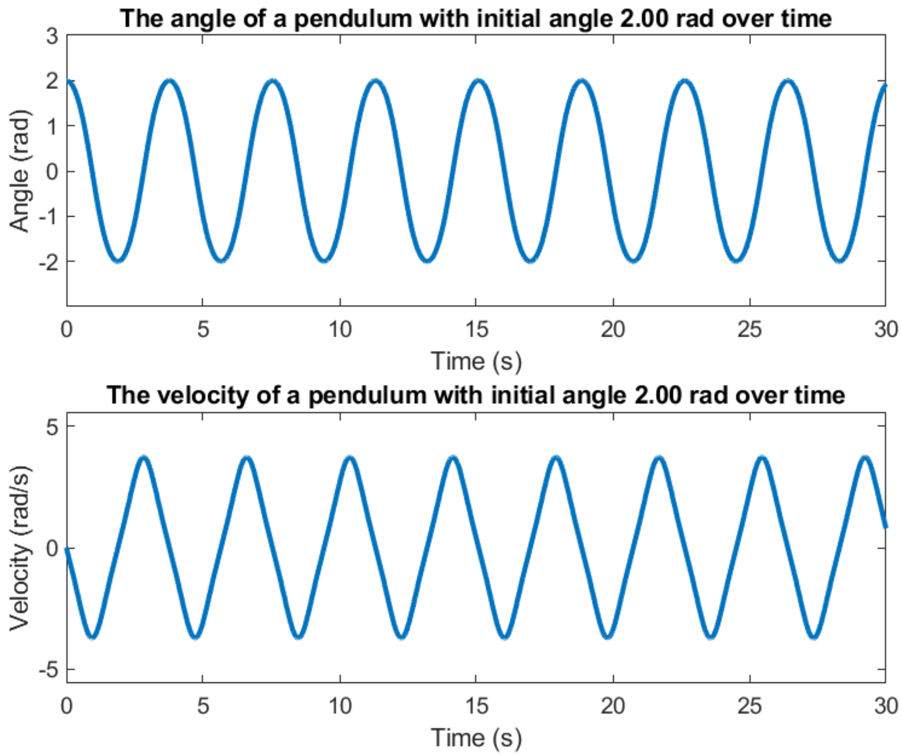

Figure 3: Behavior of the pendulum with initial angle 2 rad

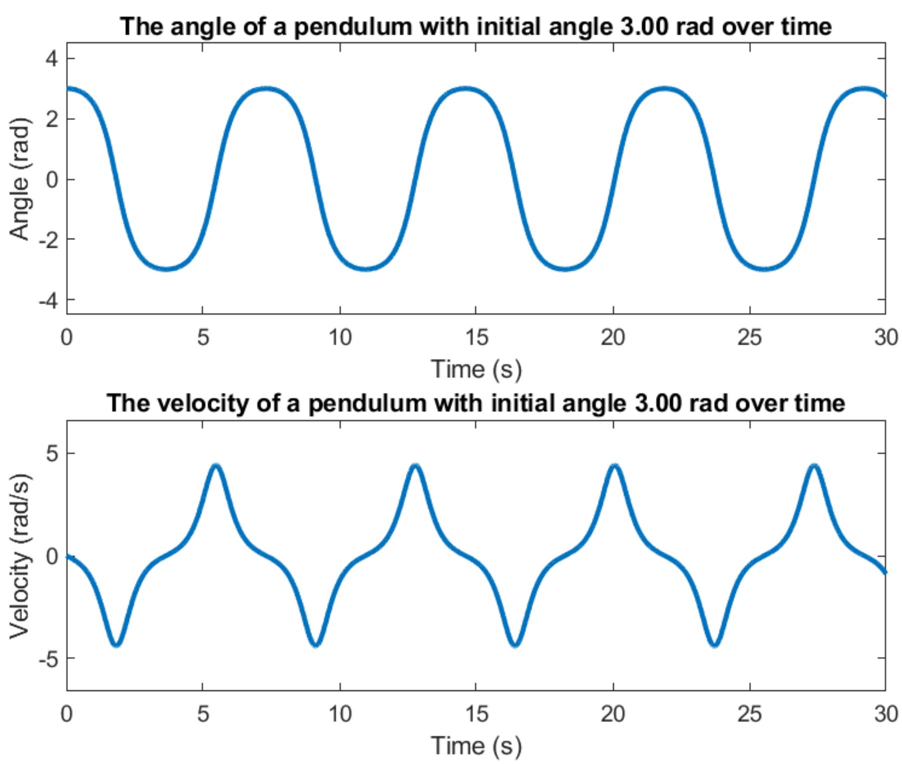

Figure 4: Behavior of the pendulum with initial angle 3 rad

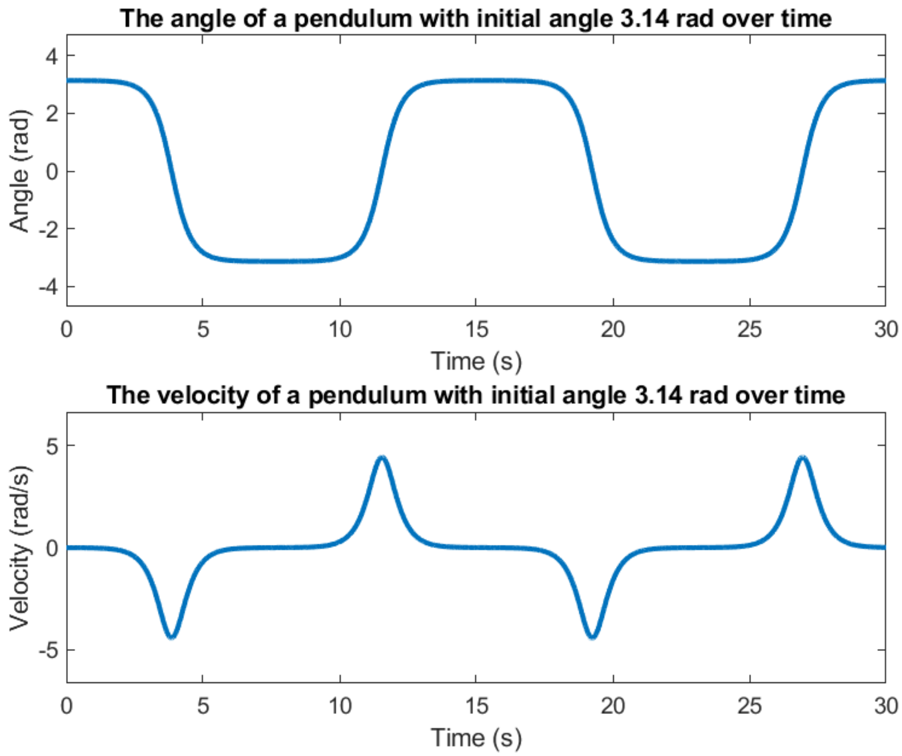

Figure 5: Behavior of the pendulum with initial angle 3.14 rad

Using this data one can calculate the period of the pendulum under the given initial angle θ_0. The period describes how often a oscillation repeats itself. Therefore it can be calculated by looking at how often the system reaches the same conditions. As both the position and angle have the same period it is enough to look at one of them. In the code this is done by getting all the times t where the position of the pendulum is at $\theta = 0rad$. The provided "zerocrossing.m" method is used to get these times. To get the actual period the time between these points is taken and averaged. They then have to be divided by 2 as the pendulum is at point $\theta = 0$ twice in it's period (once with positive and once with negative velocity). These results are then written into the *comparison_result* cell together with the analytical solution as well as the the error between them. This result is later exported as a csv producing the following table:

Initial angle (rad)	Period (numerical solution)	Period (analytical solution)	Error
0.20	2.85	2.84	0.01
1.00	3.03	2.84	0.19
2.00	3.77	2.84	0.93
3.00	7.30	2.84	4.46
3.14	15.40	2.84	12.56

From the table it can be seen that the error between the two methods is very small at an angle of $\theta = 0rad$

3 Task 2

This task uses the same building blocks as the first one. The goal is now to precisely analyze the period of the pendulum depending on the initial angle θ_0 by producing a plot. The code to accomplish this looks like this:

```
initial_angles = [0.01:0.04:0.9, 0.9:0.001:0.999]; % The initial angles to be examined (as a factor of pi)
initial_angles = arrayfun(@(x) x*pi, initial_angles); % Muliply each of the elements by pi to obtain the in
g = 9.8; % gravity
l = 2; % length

len = length(initial_angles);
periods = [];
```

```
8
9   for i = 1:len
10
11      % Run rk solve with the different initial angles
12      [times, pos] = rksolve(make_pend(g, l), 0, 30, [initial_angles(i), 0], 0.01);
13
14      % Find out the zeros with the zero crossing function. pos(2, :) selects
15      % the second row of the data (i.e. theta)
16      zeros = zerocrossing(times, pos(2, :));
17
18      % Calculate the average distance between the zeros
19      % This will give 0.5*period as the pendulum goes through zero
20      % twice for every swing
21      % 1. Add all the distances up
22      len_zeros = length(zeros)-1;
23      T = 0;
24      for j = 1:(len_zeros)
25          T = T + (zeros(j+1)-zeros(j));
26      end
27
28      % 2. Devide through the differences to get the average and
29      % multiply by two to get the actual period
30      T_numerical = (T / len_zeros) * 2;
31
32      periods(i) = T_numerical;
33  end
34
35  f = figure();
36
37  plot(initial_angles, periods, 'LineWidth', 2);
38  hold on
39  plot(initial_angles, periods, 'o');
40  hold on
41
42  plot([initial_angles(1), initial_angles(len)], [periods(1), periods(1)], ':', 'LineWidth', 2)
43
44  axis([0, pi, 0, max(periods)*1.1])
45  lgd = legend('T', 'Sampling points', 'T_0');
46  lgd.Location = 'northwest';
47  title('Period vs. Initial Angle')
48  xlabel 'Initial angle (rad)';
49  ylabel 'Period (s)';
50
51  saveas(f, '../output/assignment2.png');
```

First the initial conditions have to be defined again. This time a lot more initial angles θ_0 are analyzed to make a continuous plot out of the results. To get the different angles the shorthand operator in line 1 is used to easily get values between $0.01\pi < \theta_0 < 0.9\pi$ with $\Delta\theta_0 = 0.04\pi$ and $0.9\pi < \theta_0 < 0.999\pi$ with $\Delta\theta_0 = 0.001\pi$. The values are first created as multiples of pi. They are then each multiplied by π using the standard matlab function *arrayfun*[2]

Next the the result matrix *periods* is initialized to store the produced periods in.The code from lines 9 to 33 follows the same pattern as task 1. Instead of plotting each of the pendulums behaviors instead the numerically determined period is put into the *periods* result matrix.

This data is then plotted. The period compared to the angles is plotted twice once as a continuous line and

[2]The function uses the function provided in the first argument on all the elements in the provided array (second parameter) In the shown use case an inline method is again used for easy readability

once as dots, to show the sampling points (line 39). Additionally a plot at $\theta_0 = 0.01$ is added representing T_0[3] to compare the results with:

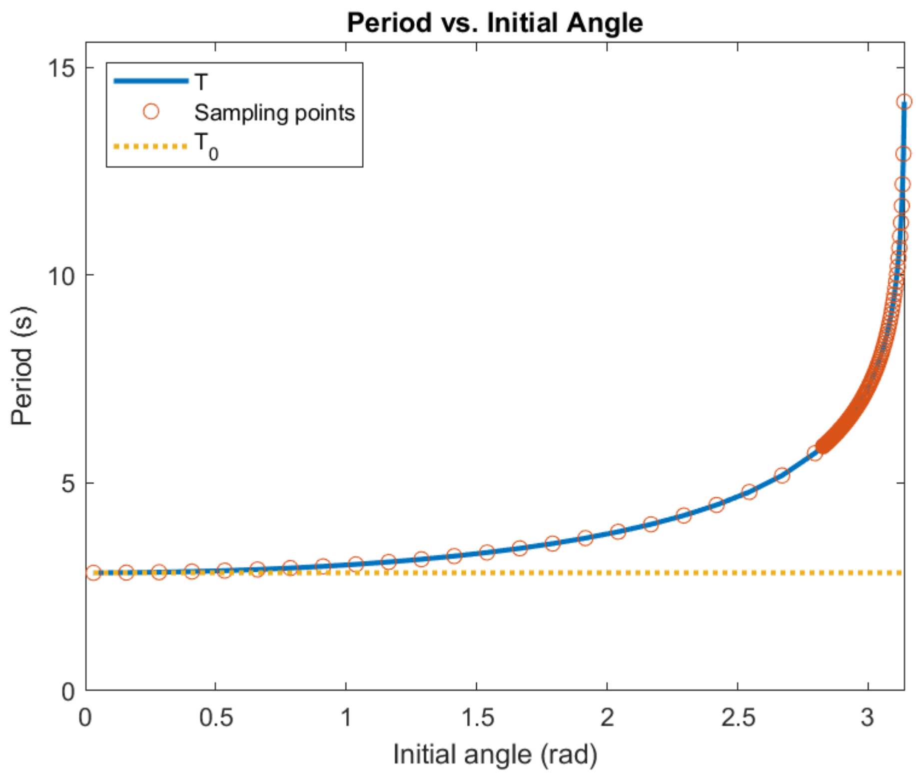

Figure 6: Period of the pendulum vs. initial angle

From the chart it can be seen that the period T asymptotically approaches T_0 for decreasing values of θ_0. On the other hand when θ_0 gets approaches π the period T seems to rise exponentially towards positive infinity. From initial appearance the the relationship seems to be exponential. This makes intuitive physical sense considering the extreme case when $\theta_0 = \pi$. In this case the pendulum would stand exactly upright. Assuming there are no external forces it would stay in that position as there is no component of the gravitational force acting lateral to the pendulum and thus no acceleration would be applied to the pendulum ($a_x = 0$). If the pendulum does not oscillate at all, the period is effectively infinite. This is confirmed by the data in figure 6 as the period seems to be infinity at $\theta_0 = \pi$

4 Task 3

Here the task is to implement a new custom differential equation to be solved by the same methods discussed in the two previous tasks. The differential equation in question describes a the temperature over time in an indoor room. It is defined by the following equation:

$$\frac{dT_{in}(t)}{dt} = -\alpha(T_{in}(t) - T_{out}(t)) + c \tag{3}$$

with T_{out} given as:

$$T_{out}(t) = T_{min} + \frac{T_{max} - T_{min}}{2}(1 + cos(2\pi sin^2(\pi t/2))) \tag{4}$$

To analyze these differential equations the first step is to implement the given formulas as matlab methods. The implementation for equation 4 looks like this:

[3] $\theta_0 = 0.01$ can be as the error is very small to the analytical solution we want to actually compare to

```
1 function result_fx = make_Tout(tmin, tmax)
2     % tmin, tmax: min and max temperature
3     % make_TOut creates a new instance of
4     % the Tout function with the provided
5     % arguments for tmin and tmx
6 
7     % Assign the resoulting Tout method as a return value
8     result_fx = @Tout;
9 
10    % Define the method to be returned
11    function y = Tout(x, t)
12        % x and y are decimal numbers
13 
14        y = tmin + ((tmax-tmin)/2)*(1 + cos(2 * pi * sin(pi * t / 2)^2 ));
15    end
16 end
```

As can be seen the method is again implemented in the factory pattern to avoid having to hand around global variables. The factory method gets as an input the two constants T_{in} and T_{out}

The method itself is then quite straight forward as the equation is only a first order differential equation and only one decimal number has to be returned opposed to a vector like in task 1. The full implementation of the equation can be done in one line and is essentially just a translation of the equation into matlab code (see line 14).

The heating method is implemented in a similar way:

```
function result_fx = make_heating(alpha, c, t_out)
    % alpha: Scaling constant
    % c: heating constant
    % t_out: the t_out method to be used
    % Make pend creates a new instance of
    % the Tout function with the provided
    % arguments for alpha, c and t_out

    result_fx = @heating;

    function y = heating(x, t)
        % x and y are decimal numbers

        t_in = x; % Reassign x to t_in for clarity

        y = -alpha * (t_in - t_out(x, t)) + c;
    end
end
```

Again we provide the two constants α and c in the factory method. Additionally an instance of the T_out method also needs to be provided as it is needed as part of the heating method. As equation 3 is already given in the form $\frac{dx}{dt}$. Thus it already describes a change within one time step, which is the value needed for numerical integration. This again makes the implementation quite straight forward and the equation can be translated into matlab code directly.

Finally the differential equation has to be solved numerically for different parameters and the result plotted:

```
t_min = 5;
t_max = 10;
initial_t_in = 22;
t_out = make_Tout(t_min, t_max);
heating_off = make_heating(2, 0, t_out);
heating_on = make_heating(2, 25.5, t_out);
```

```
% Run rk solve with the different initial angles
[times, result_off] = rksolve(heating_off, 0, 20, initial_t_in, 0.01);
[times, result_on] = rksolve(heating_on, 0, 20, initial_t_in, 0.01);

len_time = length(times);

t_out_values = arrayfun(@(t)  t_out(0, t), times);

f = figure();

plot(times, result_off, 'LineWidth', 2);
hold on

plot(times, result_on, 'LineWidth', 2);
hold on

plot([0, 20], [t_min, t_min])
hold on

plot([0, 20], [t_max, t_max])
hold on

plot(times, t_out_values)

title('Temperature vs. Time')
legend('Heating off', 'Heating on', 'T_{min}', 'T_{max}', 'T_{out}')
xlabel 'Time (days)';
ylabel 'Temperature (C°)';

% This save the plot to the filesytem
saveas(f, '../output/assignment3.png');
```

First all the constants T_{min}, T_{max} and T_{in0} (Initial value of T_{in}). Then the differential equations are defined. As T_{out} is required as part of the heating equation is defined first. As both T_{min}, T_{max} are the same for all experiments only one instance needs to be created. In contrast two heating methods need to be defined as the equation has to be solved once for the heating being on and once for the heating being off: $c = 0$ and $c = 25.5$[4]. After obtaining the methods the equations can be solved.
To obtain the T_{out} values the *arrayfun* is used again to call the method for each of the timestamps. The last thing left is to plot the results which yields the following plot:

[4]The value of 25.5 was estimated by adjusting and observing at what point the oscillations land around 21 $C°$

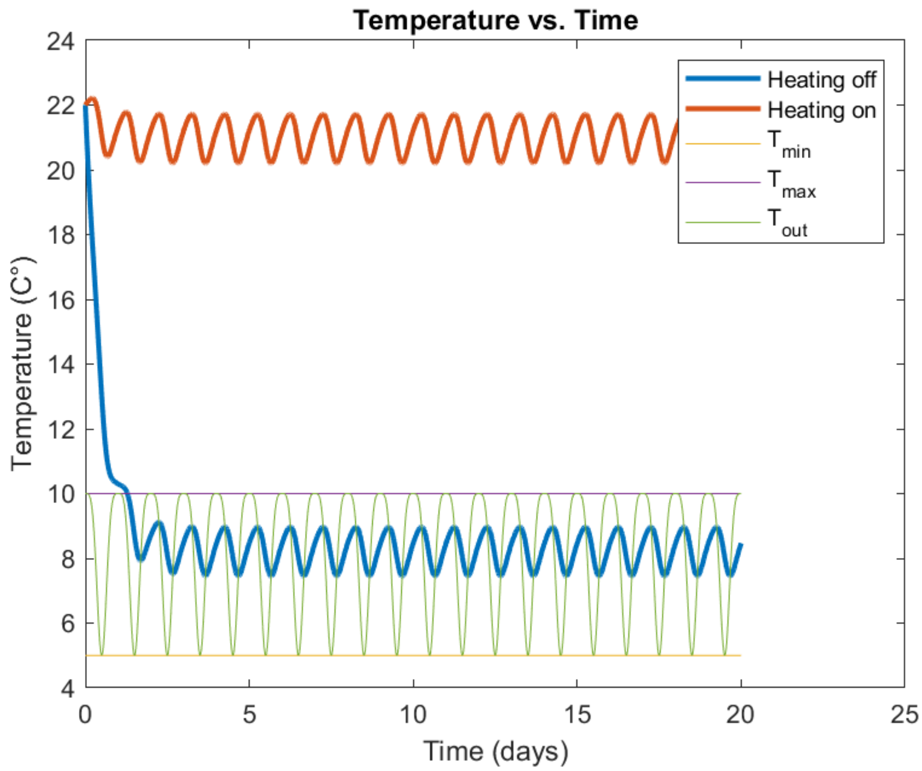

Figure 7: The temperature of the room over time