Computational Physics Individual Coursework

November 2020

1 Introduction

The assignment was to model three systems, one from each category of simple, intermediate and complex. I have chosen to analyse the Elastic Pendulum, the Double Pendulum and the Double Well Potential Duffing Oscillator. I used the equations that were provided in the brief, unaltered for my analysis.

2 Modelling Physical Statements

I think I deserve the Excellent Band for having used phase space to analyse the systems.
I have used two different initial conditions to compare how their systems evolve and how they differ from each other for different initial conditions. Through this exploration, I have observed that the systems are chaotic and non linear in nature, having vastly different results for slightly different initial conditions.

Elastic Pendulum

An Elastic Pendulum is one where a regular Pendulum's string is replaced by a spring. This creates some interesting changes in its trajectory and equations of motion. The equations I used to simulate the Elastic Pendulum were -

$$\ddot{x} = (lo + x)\dot{\theta}^2 - kx/m + gcos\theta \tag{1}$$

$$\ddot{\theta} = -gsin\theta/(lo + x) - 2\dot{x}\dot{\theta}/(lo + x) \tag{2}$$

The initial conditions for this plot was
x1 = 0.02, y1 = 0.001, x2 = 0.1, y2 = 0.01, m = 2.5, l = 3.2, k = 19.8, g = 9.8

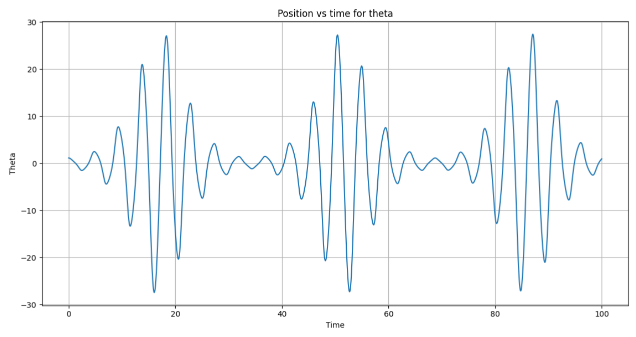

Figure 1: Theta time plot for the pendulum

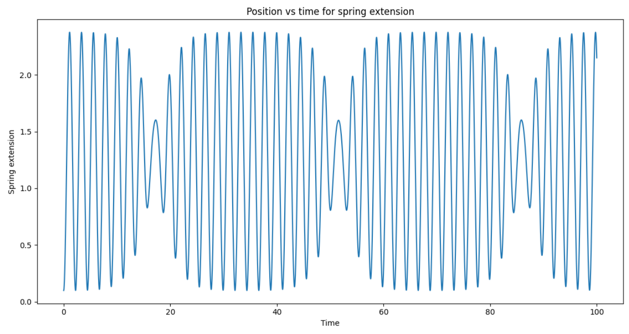

Figure 2: Spring extension vs time plot for the pendulum

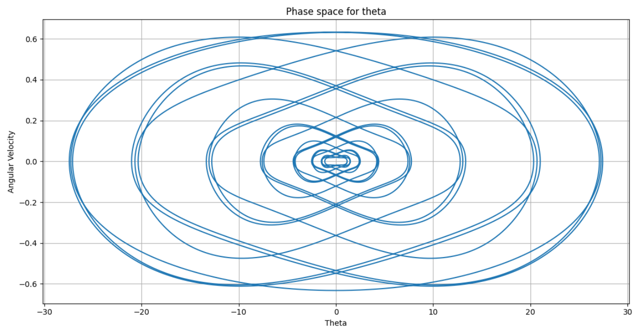

Figure 3: Phase space plot for the theta of pendulum

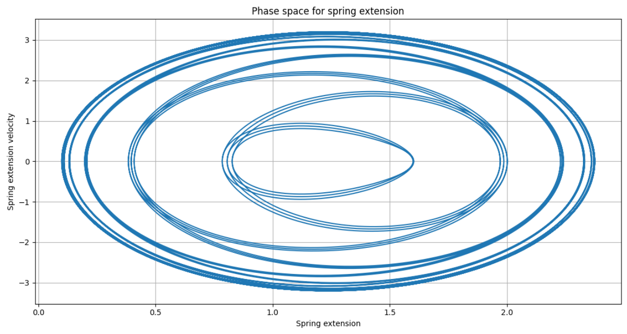

Figure 4: Phase space plot for the spring extension of the pendulum

The initial conditions for this plot was
x1 = 0.3, y1 = 0.03, x2 = 0.2, y2 = 0.01, m = 4.5, l = 4.2, k = 9.8, g = 9.8

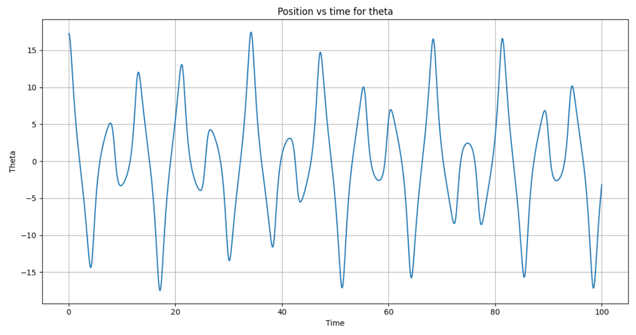

Figure 5: Theta time plot for the pendulum

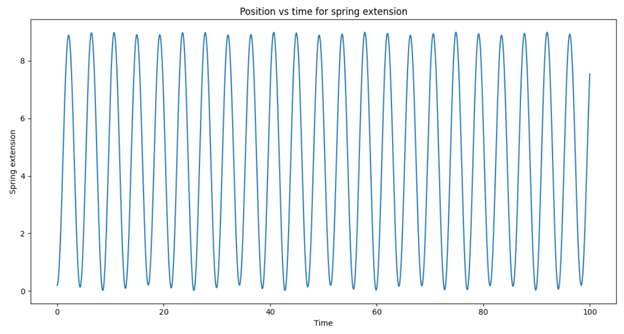

Figure 6: Spring extension vs time plot for the pendulum

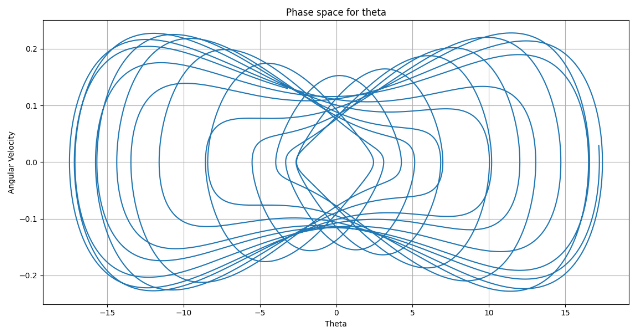

Figure 7: Phase space plot for the theta of pendulum

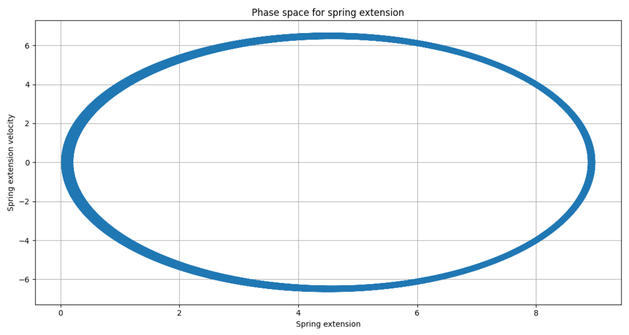

Figure 8: Phase space plot for the spring extension of the pendulum

Double Pendulum

The equations I used for the Double Pendulum system were -

$$(m_1+m_2)l_1\ddot{\theta}+m_2l_2\ddot{\theta}cos(\theta_1-\theta_2)+m_2l_2\dot{\theta}^2sin(\theta_1-\theta_2)+(m_1+m_2)gsin\theta_1 = 0 \quad (3)$$

$$l_2\ddot{\theta}_2 + l_1\ddot{\theta}_1cos(\theta_1-\theta_2)l_1\theta_1^2sin(\theta_1-\theta_2) + gsin\theta_2 = 0 \quad (4)$$

The initial conditions for the following plots are,
x1 = 0.3, y1 = 0.03, x2 = 0.2, y2 = 0.02, m1 = 3.4, m2 = 3.2, l1 = 4.5, l2 = 2.1

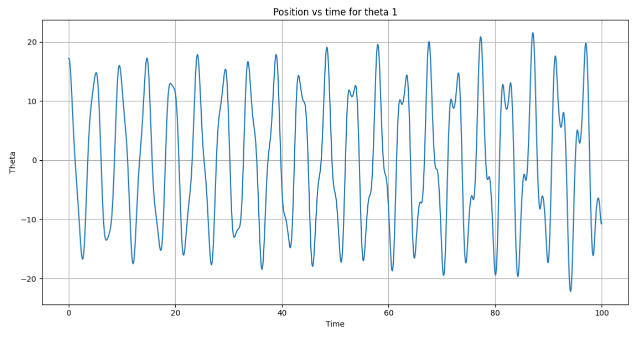

Figure 9: Theta time plot for the first pendulum

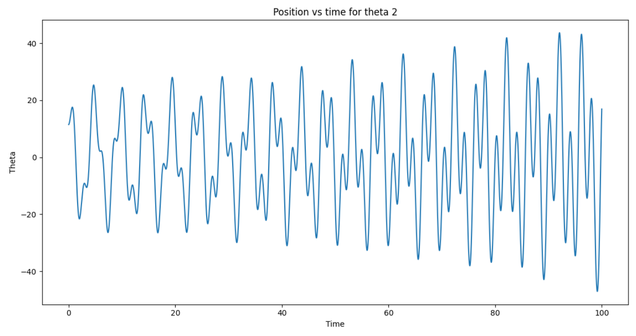

Figure 10: Theta time plot for the second pendulum

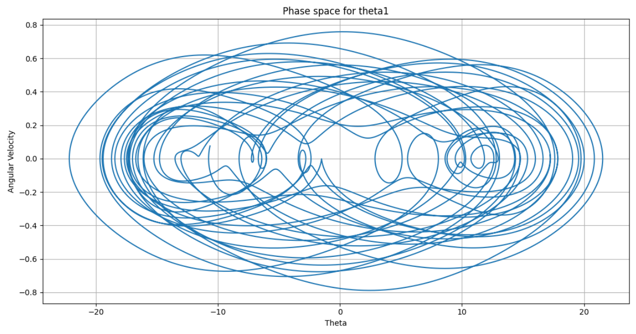

Figure 11: Phase space plot for first pendulum

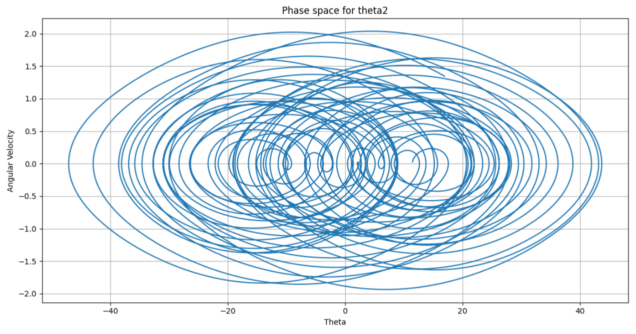

Figure 12: Phase space plot for the second pendulum

The initial conditions for this plot was
x1 = 0.1, y1 = 0.03, x2 = 0.1, y2 = 0.02, m1 = 2.4, m2 = 2.2, l1 = 9.5, l2 = 9.1

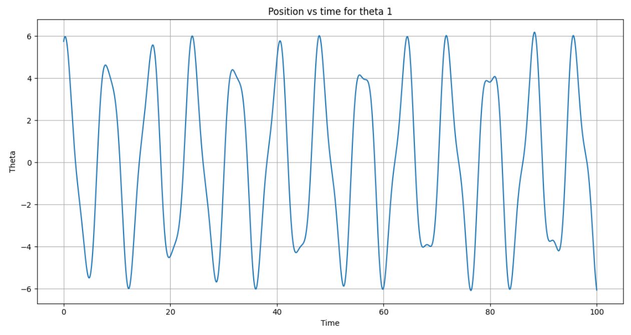

Figure 13: Theta time plot for the first pendulum

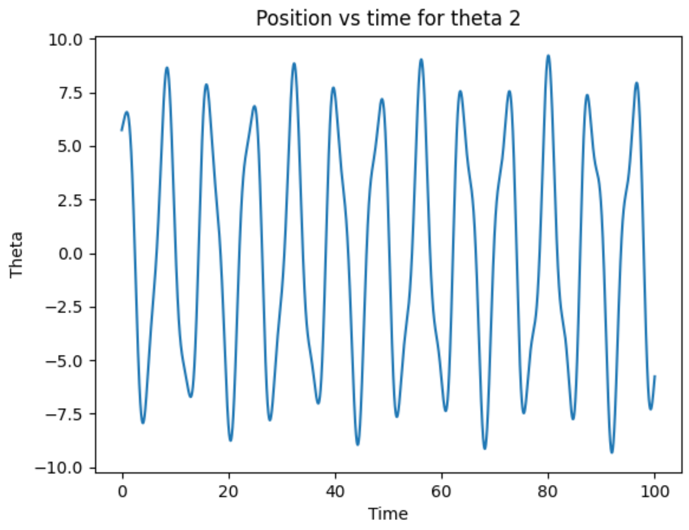

Figure 14: Theta time plot for the second pendulum

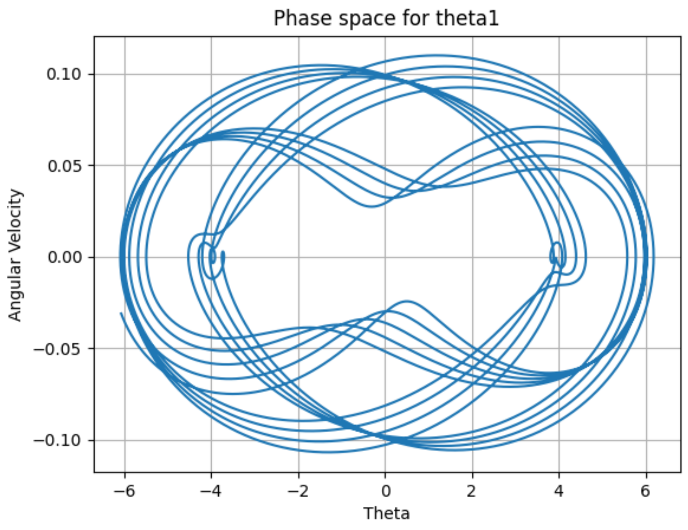

Figure 15: Phase space plot for first pendulum

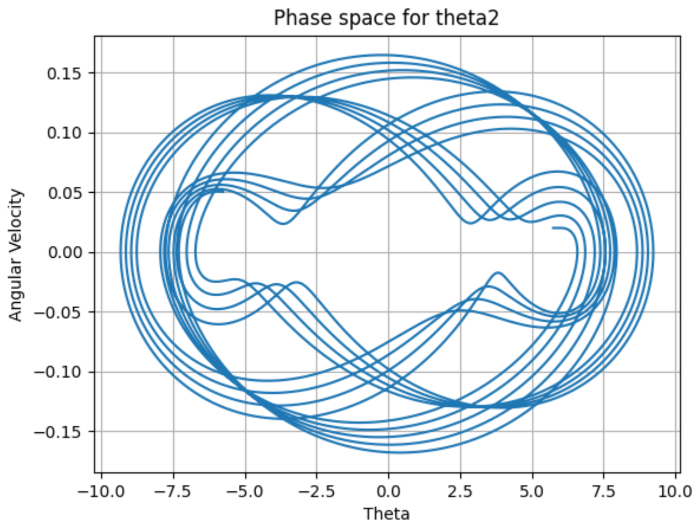

Figure 16: Phase space plot for the second pendulum

Duffing Oscillator

The equations I used for my analysis were

$$\ddot{x} + \delta \dot{x} + \alpha x + \beta x^3 = \gamma \cos(\omega t) \tag{5}$$

The initial conditions for this plot was
x1 = 0.1, y1 = 0.03, x2 = 0.1, y2 = 0.02, m1 = 2.4, m2 = 2.2, l1 = 9.5, l2 = 9.1

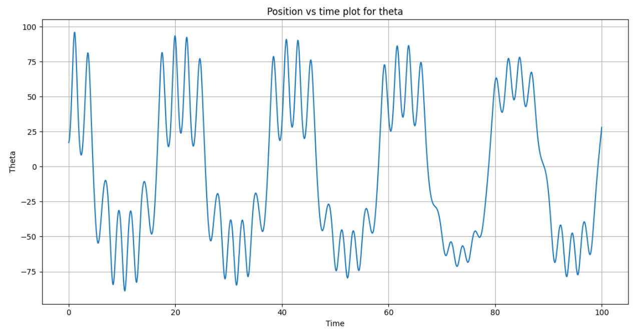

Figure 17: Theta time plot for the pendulum

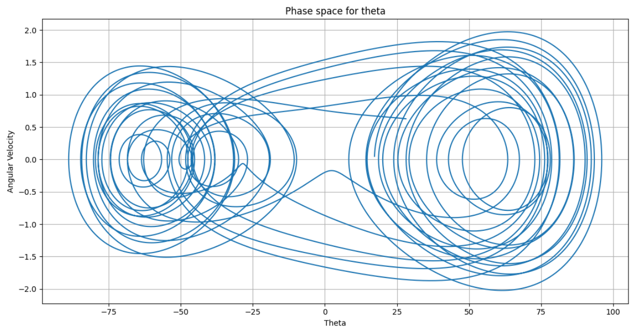

Figure 18: Phase Space

The initial conditions for this plot was
x1 = 0.1, y1 = 0.03, x2 = 0.1, y2 = 0.02, m1 = 2.4, m2 = 2.2, l1 = 9.5, l2 = 9.1

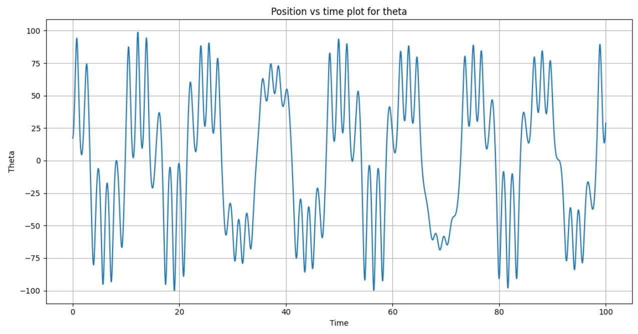

Figure 19: Theta time plot for the pendulum

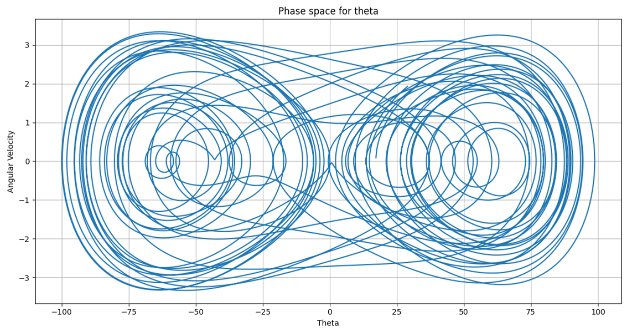

Figure 20: Phase Space

The plots for the specific initial conditions match the ones that are on wikipedia, scholarpedia and wolfram alpha.

3 Numerical Methods

I have used four methods of Numerical Integration to solve the following systems. The methods I used were Euler, adaptive step Euler, RK4 and adaptive stepping RK4. They were separated from each system, such that they were all included in a separate file called numerical integration, so that all the systems could use them for their specific function. I think I deserve the excellent band for Numerical Methods.

Euler Method

```
 public func euler(function: ([Double], Double) ->
    ([Double]), y:[Double], t:Double, dt:Double) ->
    ([Double]){

     return dt * function(y,t)
}
```

Euler Method with an adaptive step

```
/** Take two half steps and compare their difference with a
    full step. Reduce the step size if difference is greater
    than dx_max **/
public func euler_adaptive_step(function: ([Double],
    Double) -> ([Double]), y:[Double], t:Double, dt:Double)
    -> ([Double], Double){

     var h = dt

     /** Fixed step values */
     let h_fixed = 0.01
     let h_min = 0.0001

     var step:[Double] = []

     /** Half step value */
     var half_step:[Double] = []


     /** Tolerable difference ranges */
     let dx_max = 0.01

     step =  dt * function(y, t + dt)
     half_step =  dt/2 * function(y, t + dt/2)
     half_step += dt/2 * (function(y, t + dt) + half_step)
     /** Checking if step size is above defined limit to
     control speed */
     if h < h_min{
         h = h_fixed
     }

     /** Updating step size */
     if (abs(step[2] - half_step[2]) > dx_max || abs(step[3]
     - half_step[3]) > dx_max){
         h = h/2
     }

     return (half_step, h)
 }
```

RK4

```
 /** Fourth order runge kutta method. Taken from
    (https://lpsa.swarthmore.edu/NumInt/NumIntFourth.html) */
public func runge_kutta(function: ([Double], Double) ->
    ([Double]), y:[Double], t:Double, dt:Double) ->
    ([Double]){

    let k1 = function(y,t)
    let k2 = function(y + (dt * 0.5) * k1, t + 0.5 * dt)
    let k3 = function(y + 0.5 * dt * k2, t + 0.5 * dt)
    let k4 = function(y + dt * k3, t + dt)

    return dt * (k1 + (2 * k2) + (2 * k3) + k4) / 6
}
```

RK4 with an adaptive step

```
 /** Take a half step, a full step and a double step.
    If the difference between double step and single step
    is below dx_min double the step size
    If the difference between half step and single step  is
    above dx_max half the step size */
public func runge_kutta_adaptive_stepper(function:
    ([Double], Double) -> ([Double]), y:[Double], t:Double,
    dt:Double) -> ([Double], Double){

    /** Fixed step size */
    var h = dt
    let h_fixed = 0.01
    let h_min = 0.0001

    /** Normal step value */
    var step:[Double] = []

    /** Half step value */
    var half_step:[Double] = []

    /** Double step value */
    var double_step:[Double] = []

    /** Tolerable difference ranges */
    let dx_min = 0.008
    let dx_max = 0.01

    /** Calculating full step based on fixed step size */
    let k1 = function(y, t)
    var k2 = function(y + k1 * h / 2, t + h/2)
    var k3 = function(y + k2 * h / 2, t + h/2)
    var k4 = function(y + k3 * h, t + h)

    step = h * (k1 + 2 * k2 + 2 * k3 + k4) / 6


    /** Calculating half step based on fixed step size */
    k2 = function(y + k1 * h / 4, t + h/4)
    k3 = function(y + k2 * h / 4, t + h/4)
    k4 = function(y + k3 * h, t + h/2)

    half_step = h * (k1 + 2 * k2 + 2 * k3 + k4) / 12



    /** Calculating Double step based on fixed step size */
    k2 = function(y + k1 * h , t + h)
    k3 = function(y + k2 * h , t + h)
    k4 = function(y + k3 * h, t + 2*h)

    double_step = h * (k1 + 2 * k2 + 2 * k3 + k4) / 3


    /** Checking if step size is above defined limit to
    control speed */
    if h < h_min{
        h = h_fixed
    }

    /** Updating step size */
    if (abs(step[2] - half_step[2]) > dx_max || abs(step[3]
    - half_step[3]) > dx_max){
        h = h/2
    }

    else if(abs(double_step[2] - step[2]) < dx_min ||
    abs(double_step[3] - step[3]) < dx_min){
        h = 2 * h
    }


    return (half_step, h)
}
```

4 Program structure: Logic and features

Due to the approach of separating the numerical integration file, the code is highly reusable and adaptive for new systems. As long as a new physical system adheres to the style of the other systems, there is no problem in integrating it. Thus, I think I deserve the outstanding band.

```
 /** Returns the updated array based on the derivatives
for time step dt */
 private func fz(y:[Double], t:Double) -> [Double]{

     let y2 = ((self.g * cos(self.w * t)) - (self.d *
y[1]) - (self.a * self.y[0]) - (self.b * y[0] * y[0] *
y[0]))
     return [y[1], y2]

 }


 /** Solves the differential equation using the
integration methods in numerical_integration.
  type = 1 - RK4,  2 - Euler,  3 - RK4(Adaptive Step),  4
 - Euler(Adaptive Step)
  Returns the positions and velocites of the generalised
coordinates for (dt * n) time period */
 public mutating func solve(dt:Double, n:Int, type:Int)
-> ([Double], [Double]){

     /** Creating arrays to hold data regarding the
positions, velocities of the generalised coordinates */
     var x1: [Double] = []
     var y1: [Double] = []

     var dt = dt

     /**Initializing the time variable */
     var t = 0.0

     /** Integrating the equation from 0 to n steps,
with a time step of dt */
     for _ in 0...n{

         /** Updating value of the generalised
coordinates in the time step dt */
         if type == 1{
             self.y = self.y + runge_kutta(function:
self.fz, y: self.y, t: t, dt: dt)
         }

         else if type == 2{
                self.y = self.y + euler(function: self.fz,
y: self.y, t: t, dt: dt)
            }

            else if type == 3{
                let val =
runge_kutta_adaptive_stepper(function: self.fz, y:
self.y, t: t, dt: dt)
                self.y = self.y + val.0

                /** Updating the step size */
                dt = val.1
            }

            else if type == 4{
                let val = euler_adaptive_step(function:
self.fz, y: self.y, t: t, dt: dt)
                self.y = self.y + val.0

                /** Updating the step size */
                dt = val.1
            }


            /** Appending the angle in the array in degrees
*/
            x1.append(self.y[0] * 3.14 / 180)

            /** Appending the velocity of the generalised
coordinate to the array */
            y1.append(self.y[1])

            /** Updating the time of the system */
            t += dt

        }

        /** Returning the data for analysis */
        return (x1, y1)

    }
```

For a new system that is to be used through the same program, it just needs to represent the following code in essence. As long as it has a derivative function, and an array that holds the generalised coordinates and velocities, it can be solved using the same method.

5 Program Structure: Maintainability

The file structure is very conducive to new systems to be added to the package. Each system has a separate file that has a constructor, the derivative function(fz) and a solve function. The code is also comprehensively tested for an equilibrium position and two different initial conditions. Thus, I think I deserve the outstanding band.

```
 public init(x:Double, y:Double, a:Double, b:Double,
w:Double, d:Double, g:Double)

 private func fz(y:[Double], t:Double) -> [Double]

 public mutating func solve(dt:Double, n:Int, type:Int)
-> ([Double], [Double])
```

By having these functions, I high degree of abstraction is maintained with the user. An object of the system maintains all its system variables, making it easy for the user to solve the system using the different types of methods of numerical integration present.

```
 /** Updating value of the generalised coordinates in
the time step dt */
     if type == 1{
         self.y = self.y + runge_kutta(function:
self.fz, y: self.y, t: t, dt: dt)
     }

     else if type == 2{
         self.y = self.y + euler(function: self.fz, y:
self.y, t: t, dt: dt)
     }

     else if type == 3{
         let val =
runge_kutta_adaptive_stepper(function: self.fz, y:
self.y, t: t, dt: dt)
         self.y = self.y + val.0

         /** Updating the step size */
         dt = val.1
     }

     else if type == 4{
         let val = euler_adaptive_step(function:
self.fz, y: self.y, t: t, dt: dt)
         self.y = self.y + val.0

         /** Updating the step size */
         dt = val.1
```

```
        }
```

The separation of concern, that is, numerical integration, systems and tests helps maintain a high level of abstraction with the user, and encapsulates the methods for re usability in different systems.

The code is comprehensively tested to see if it analyses the systems correctly,

```
/** Testing and returning the values for two initial
    conditions */
public func ElasticPendulumTests() -> (([Double], [Double],
    [Double], [Double]), ([Double], [Double], [Double],
    [Double])){


    /** Checking if system oscillates at equilibrium */
    var t0 = ElasticPendellum(l: 3.2, x1: 0, y1: 0, x2: 0,
    y2: 0, mass: 3.4, spring_constant: 30.3)
    let val0 = t0.solve(dt: 0.001, n: 10000, type: 1)

    for i in val0.0{
        if i != 0{
            print("The system is unbalanced at equilibrium!
    Faulty simulation! ")
        }
    }


    /** Checking the system for two initial conditions */
    var t1 = ElasticPendellum(l: 2.1, x1: 0.1, y1: 0.001,
    x2: 0.3, y2: 0.02, mass: 5.5, spring_constant: 18.9)
    var t2 = ElasticPendellum(l: 3.1, x1: 0.21, y1: 0.001,
    x2: 0.3, y2: 0.002, mass: 3.5, spring_constant: 8.9)

    let val1 = t1.solve(dt: 0.01, n: 10000, type: 3)
    let val2 = t2.solve(dt: 0.01, n: 10000, type: 3)

    return (val1, val2)
}


/** Testing and returning the values for two initial
    conditions */
public func DoublePendulumTests() -> (([Double], [Double],
    [Double], [Double]),([Double], [Double], [Double],
    [Double])){

    /** Checking if system oscillates at equilibrium */
    var t0 = DoublePendellum(m1: 2.4, m2: 2.3, l1: 2.3, l2:
    2.1, x1: 0, x2: 0, y1: 0, y2: 0)
    let val0 = t0.solve(dt: 0.001, n: 10000, type: 1)
    for i in val0.0{
        if i != 0{
            print("The system is unbalanced at equilibrium!
    Faulty simulation! ")
        }
    }

    /** Checking the system for two initial conditions */
    var t1 = DoublePendellum(m1: 0.24, m2: 2.5, l1: 0.3,
    l2: 4.5, x1: 0.3, x2: 0.02, y1: 0.003, y2: 0.002)
    var t2 = DoublePendellum(m1: 2.5, m2: 0.24, l1: 3.1,
    l2: 2.5, x1: 0.3, x2: 0.2, y1: 0.0, y2: 0.01)

    let val1 = t1.solve(dt: 0.01, n: 1000, type: 1)
    let val2 = t2.solve(dt: 0.01, n: 1000, type: 3)

    return (val1, val2)

}

/** Testing and returning the values for two initial
    conditions */
public func DuffingOscialltorTests() -> (([Double],
    [Double]),([Double], [Double])){

    /** Checking if system oscillates at equilibrium */
    var t0 = DuffingOscillator(x: 0, y: 0, a: 1, b: 1, w: 2
    , d: 1, g: 3)
    let val0 = t0.solve(dt: 0.001, n: 10000, type: 1)

    for i in val0.0{
        if i != 0{
            print("The system is unbalanced at equilibrium!
    Faulty simulation! ")
        }
    }

    /** Checking the system for two initial conditions */
    var t1 = DuffingOscillator(x: 0.02, y: 0.001, a: 1, b: 5
    , w: 0.5, d: 0.02, g: 8)
    var t2 = DuffingOscillator(x: 0.03, y: 0.002, a: 1 , b:
    -1, w: 1, d: 0.2, g: 0.3)

    let val1 = t1.solve(dt: 0.01, n: 1000, type: 1)
    let val2 = t2.solve(dt: 0.01, n: 1000, type: 3)

    return (val1, val2)
}
```