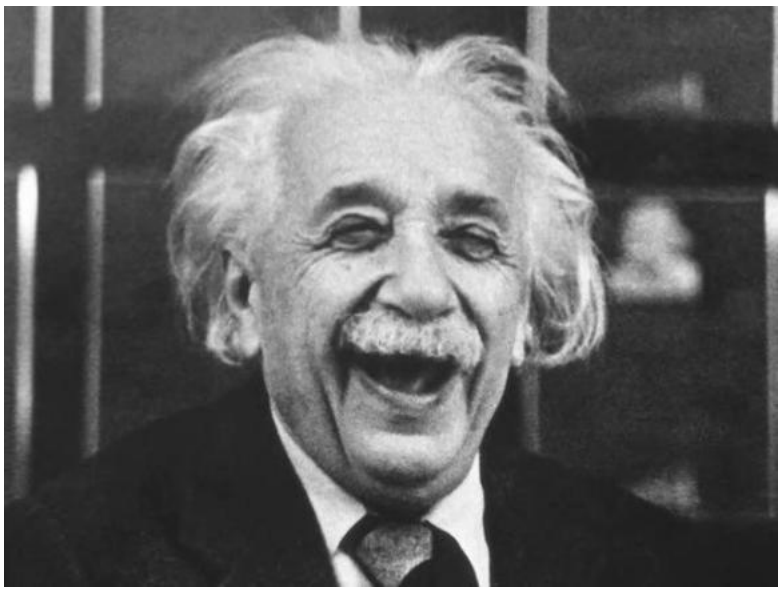
PH388 Project 2: Ordinary Differential Equations

Hamilton, Robert
robert.hamilton.2017@uni.strath.ac.uk

Kennedy, Lewis
lewis.kennedy.2018@uni.strath.ac.uk

Holmes, Aaron
Aaron.holmes.2018@uni.strath.ac.uk

Hunter, Ross
Ross.Hunter.2018@uni.strath.ac.uk



March 12, 2021

Contents

Contributions

A summary of the contributions made by each group member as requested by the supervisors.

Lewis - Developed the RK2 algorithm & collaborated with Robbie on the RKF45 algorithm. Analytical solution to the homogeneous and inhomogeneous differential equation. Section 1 write up, discussion of algorithm performance, transient & steady states, section 2 chaos discussion with Robbie.

Robbie - Developed the RK4 algorithm & collaborated with Lewis on the RKF45 algorithm. Section 1 RKF45 discussion with Lewis, Section 2 write up and discussion, bonus tasks & extra graphing.

Aaron - Finalised the analytical solution to the inhomogeneous ODE using the exponential method and solved the phase. Parts of section 1: helped on transient & steady states, wrote parts of section 1 integral, appendix. Unused RK2 and RK4 solver attempts.

Ross - Wrote parts of section 1 integral, participated in task i, references, style.

1 Damped Driven Oscillator

1.1 Analytic Solution

For a homogeneous 2nd order differential equation of the form:

$$\ddot{x}(t) + 2\zeta\omega_0\dot{x}(t) + \omega_0^2 x(t) = 0$$

We can begin solving analytically by using a trial solution for x(t) which is given by:

$$\text{x(t)} = \text{A}e^{\lambda t}$$

where $\lambda_{1,2}$ are the roots of the characteristic equation of the ODE:

$$\lambda^2 + 2\zeta\omega_0\lambda + \omega_0^2 = 0$$

$$\lambda_{1,2} = -\zeta\omega_0 \pm \omega_0\sqrt{(\zeta^2 - 1)}$$

For $\zeta > \omega_0$, both roots are real and distinct, giving rise to negative exponents, indicating that this system is a purely damped system with no harmonic motion. The general solution is given by:

$$\text{x(t)} = \text{A}e^{(-\zeta\omega_0 - \omega_0\sqrt{(\zeta^2-1)})t} + \text{B}e^{(-\zeta\omega_0 + \omega_0\sqrt{(\zeta^2-1)})t}$$

Similar behaviour is seen for $\zeta = \omega_0$, and therefore $\lambda_1 = \lambda_2$, where there is one real root and the general solution again describes purely damped motion:

$$\text{x(t)} = \text{A}e^{-\zeta\omega_0 t} + Bte^{-\zeta\omega_0 t}$$

For the specific case of $\zeta = 0$, i.e an ODE with no damping term, we have the general solution of a harmonic oscillator with no damping effect:

$$\text{x(t)} = \text{A}e^{iw_0 t} + \text{B}e^{-iw_0 t}$$

$$\text{x(t)} = \text{A}\cos(w_0 t) + \text{B}\sin(w_0 t)$$

For the case where $\zeta < \omega_0$, the roots are complex and we can see we have a system with damped oscillation, due to the negative exponential term and the harmonic terms:

$$\text{x(t)} = \text{A}e^{(-\zeta\omega_0 - i\omega_0\sqrt{(1-\zeta^2)})t} + \text{B}e^{(-\zeta\omega_0 + i\omega_0\sqrt{(1-\zeta^2)})t}$$

$$\text{x(t)} = e^{-\zeta\omega_0 t}\left(A\cos(\omega_0\sqrt{(1-\zeta^2)}t) + B\sin(\omega_0\sqrt{(1-\zeta^2)}t)\right)$$

For the purpose of our project, we will focus on the value of ζ for which we would observe damped oscillatory behaviour which is in the case of $\zeta < \omega_0$, where we have both a damping term and harmonic oscillation in our general solution for x(t). For a 2nd order differential equation of the form:

$$\ddot{x}(t) + 2\zeta\omega_0\dot{x}(t) + \omega_0^2 x(t) = A\sin(\Omega t)$$

We must now consider the addition of the external harmonic forcing term to the ODE, F(t) = Asin(Ωt), which describes an inhomogeneous system. The inhomogeneous solution x_p is found analytically using the variation of constants method with the general solution to the homogeneous ODE:

$$x(t) = e^{\left(-\zeta\omega_o \pm i\sqrt{\omega_o^2 - (\zeta\omega_o)^2}\right)t}$$

$$\lambda_{1,2} = \zeta\omega_o \pm i\omega_o\sqrt{1-\zeta^2} = a \pm ib$$

we then let $\lambda_1 = a + ib$ and $\lambda_2 = a - ib$

In order to find the Wronski determinant, we need to first choose our basis of the solution space given by x = (x_1, x_2). Letting $x_1 = e^{\lambda_1 t}$ and $x_2 = e^{\lambda_2 t}$, the Wronski determinant is given by:

$$W(t) = \begin{vmatrix} x_1 & x_2 \\ x_1' & x_2' \end{vmatrix} = x_1 x_2' - x_1' x_2$$

$$W(t) = (\lambda_2 - \lambda_1) e^{(\lambda_1 + \lambda_2)t}$$

$$W(t) = -2ibe^{(\lambda_1 + \lambda_2)t}$$

With $W_1(t) = -Asin(\Omega t)e^{\lambda_2 t}$ and $W_2(t) = Asin(\Omega t)e^{\lambda_1 t}$ which are determined by solving the determinants of:

$$W_1(t) = \begin{vmatrix} 0 & x_1 \\ F(t) & x_1' \end{vmatrix} = -F(t)x_2(t) \qquad W_2(t) = \begin{vmatrix} x_2 & 0 \\ x_2' & F(t) \end{vmatrix} = F(t)x_1(t)$$

The inhomogeneous solution to the forced ODE is then given by:

$$x_p(t) = x_1(t) \int_{t_0}^{t} \frac{W_1(t)}{W(t)} dt + x_2(t) \int_{t_0}^{t} \frac{W_2(t)}{W(t)} dt$$

$$x_p(t) = e^{\lambda_1 t} \int_{t_0}^{t} \frac{-Asin(\Omega t)e^{\lambda_2 t}}{-2ibe^{(\lambda_1+\lambda_2)t}} dt + e^{\lambda_2 t} \int_{t_0}^{t} \frac{Asin(\Omega t)e^{\lambda_1 t}}{-2ibe^{(\lambda_1+\lambda_2)t}} dt$$

$$x_p(t) = A \left[e^{\lambda_1 t} \int_{t_0}^{t} \frac{(e^{i\Omega t} - e^{-i\Omega t})e^{\lambda_2 t}}{2i(2ib)e^{\lambda_1 t}e^{\lambda_2 t}} dt + e^{\lambda_2 t} \int_{t_0}^{t} \frac{(e^{i\Omega t} - e^{-i\Omega t})e^{\lambda_1 t}}{2i(2ib)e^{\lambda_2 t}e^{\lambda_1 t}} dt \right]$$

As shown by the solution given in appendix B, the particular solution to the inhomogeneous differential equation is then:

$$x_p(t) = \frac{Asin(\Omega t - \phi)}{M}$$

Where:

$$M = \sqrt{(\omega_o^2 - \Omega^2)^2 + (2\zeta\omega_o\Omega)^2} \qquad \phi = tan^{-1}\left(\frac{2\zeta\omega_o\Omega}{\omega_o^2 - \Omega^2}\right)$$

Solving for the constants K_1 and K_2 of the general solution:

$$\text{x(t)} = e^{-\zeta\omega_0 t}\left(K_1 \cos(\omega_0\sqrt{(1-\zeta^2)}t) + K_2 \sin(\omega_0\sqrt{(1-\zeta^2)}t)\right) + \tfrac{A\sin(\Omega t - \phi)}{M}$$

$$\text{x(t=0)} = x_0 = K_1 + \tfrac{A\sin(-\phi)}{M} = 1$$

$$K_1 = 1 - \tfrac{A\sin(-\phi)}{M}$$

$$\text{v(t)} = \tfrac{d}{dt}\left[e^{-\zeta\omega_0 t}\left(K_1 \cos(\omega_0\sqrt{(1-\zeta^2)}t) + K_2 \sin(\omega_0\sqrt{(1-\zeta^2)}t)\right) + \tfrac{A\sin(\Omega t - \phi)}{M}\right]$$

$$\text{v(t=0)} = v_0 = K_2 + \tfrac{aK_1}{b} + \tfrac{A\Omega\cos(-\phi)}{bM} = 0$$

$$K_2 = -\tfrac{aK_1}{b} - \tfrac{A\Omega\cos(-\phi)}{bM}$$

1.2 Algorithms Developed

1.2.1 2nd Order Runge-Kutta Solver

A numerical solver based on the 2nd order Runge-Kutta (RK2) algorithm was developed as shown in appendix C to solve the ODE from task 1 using numerical methods to compare with our analytical solution. For the case outlined in task 2, the ODE can be rewritten as a system of ODEs:

$$\dot{x} = v \tag{1}$$
$$\dot{v} = -2\zeta\omega v(t) - \omega_0^2 x(t) + F(t) \tag{2}$$

We can then solve for x(t) by integrating using the RK2, which takes an ODE of the form $\frac{dy}{dt}$ = f(t, y) and solves for y(t). This is done by a series of approximations for y_{i+1}, which is a function of the current iteration solution y_i, since $y_{i+1} = y_i + hf(t_i, y_i)$ where the function f(t, x) is the first derivative of the variable to be solved.

Hence in order to solve our system of ODEs for x(t), we must first solve for v(t) by passing the ODE described by equation (2) into our RK2 solver to solve for v(t) using the already known first derivative $\dot{v}$, before repeating the process to solve the ODE described by equation (1) for x(t). This is done by using a series of approximations given by the k values which are then used in the final calculation of the solution y_{i+1}:

$$\begin{aligned} K_1 &= hf(t_i, y_i) \\ K_2 &= hf(t_i + \frac{h}{2}, y_i + \frac{K_1}{2}) \\ y_{i+1} &= y_i + K_2 + \mathcal{O}(h^3) \end{aligned}$$

This process is then repeated with the latest value y_i being used in the approximation of the value at y_{i+1} on each iteration, integrating over the desired time grid using the step size h.

1.2.2 4th Order Runge-Kutta Solver

A 4th Order Runge-Kutta method (RK4) was then developed as shown in appendix D. The RK4 method is similar to the RK2 method in that it integrates using a series of approximations but it differs from the RK2 in that there are more intermediate k steps. The extra intermediate steps and calculations help on improving on the accuracy of the approximation made with each iteration.

$$\begin{aligned} K_1 &= hf(t_i, y_i) \\ K_2 &= hf(t_i + \frac{h}{2}, y_i + \frac{K_1}{2}) \\ k_3 &= hf(t_i + \frac{h}{2}, y_i + \frac{K_2}{2}) \\ k_4 &= hf(t_i + h, y_i + k_3) \end{aligned}$$

With the solution for each y_{i+1} given by

$$y_{i+1} = y_i + \tfrac{K_1}{6} + \tfrac{K_2}{3} + \tfrac{k_3}{3} + \tfrac{k_4}{6} + \mathcal{O}(h^5)$$

By including more intermediate steps, we reduce the risk of overstepping and potentially introducing unwanted uncertainty propagation in each step thus allowing us to achieve increased accuracy over more coarse time grids in comparison to the RK2. A comparison of the performance of the RK2 and RK4 can be seen in Figure 1a, where it is shown that the RK2 fails to produce an accurate solution for step sizes of magnitude h = 1.

1.2.3 Runge-Kutta-Fehlberg (RKF45) solver

We can make further improvements to the previous algorithms by making slight adaptions to the Runge-Kutta methods as shown in appendix E. These include using an adaptive step size and utilising both the 4th and 5th order Runge-Kutta methods in one solver. By including the calculation of an extra k value, we can calculate the error of the solution and by imposing error thresholds on our solution, we can adjust the step size if the error exceeds this threshold value. This method ensures an optimized step size is used on every iteration, greatly improving accuracy but incurring greater processing costs.

The process for the RKF45 adaptive solver with optimal step size approximates by calculating k values as follows:

$$
\begin{aligned}
k_1 &= hf(t_i, y_i) \\
k_2 &= hf(t_i + \alpha_2 h, y_i + \beta_{21} k1) \\
k_3 &= hf(t_i + \alpha_3 h, y_i + \beta_{31} k1 + \beta_{32} k2) \\
k_4 &= hf(t_i + \alpha_4 h, y_i + \beta_{41} k1 + \beta_{42} k2 + \beta_{43} k3) \\
k_5 &= hf(t_i + \alpha_5 h, y_i + \beta_{51} k1 + \beta_{52} k2 + \beta_{53} k3 + \beta_{54} k4 + \beta_{55} k5) \\
k_6 &= hf(t_i + \alpha_6 h, y_i + \beta_{61} k1 + \beta_{62} k2 + \beta_{63} k3 + \beta_{64} k4 + \beta_{65} k5 + \beta_{66} k6)
\end{aligned}
$$

The constants α_x and β_{xy} are defined by the Butcher Tableau for the Runge-Kutta-Fehlberg method:

0						
α_2	β_{21}					
α_3	β_{31}	β_{32}				
α_4	β_{41}	β_{42}	β_{43}			
α_5	β_{51}	β_{52}	β_{53}	β_{54}		
α_6	β_{61}	β_{62}	β_{63}	β_{64}	β_{65}	
	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6
	γ_1^*	γ_2^*	γ_3^*	γ_4^*	γ_5^*	γ_6^*

0						
$\frac{1}{4}$	$\frac{1}{4}$					
$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$				
$\frac{12}{13}$	$\frac{1932}{2197}$	$\frac{-7200}{2197}$	$\frac{7296}{2197}$			
1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$\frac{-845}{4104}$		
$\frac{1}{2}$	$\frac{-8}{27}$	2	$\frac{-3544}{2565}$	$\frac{1859}{4104}$	$\frac{-11}{40}$	
	$\frac{25}{216}$	0	$\frac{1408}{2565}$	$\frac{2197}{4104}$	$\frac{-1}{5}$	0
	$\frac{16}{135}$	0	$\frac{6656}{12825}$	$\frac{28561}{56430}$	$\frac{-9}{50}$	$\frac{2}{55}$

Then the new solutions to the ODEs are calculated using both 4th and 5th order coefficients γ_i and γ_i^*, respectively:

$$
\begin{aligned}
y_{i+1} &= y_i + \gamma_1 K_1 + \gamma_2 K_2 + \gamma_3 k_3 + \gamma_4 k_4 + \gamma_5 k_5 + \gamma_6 k_6 + \mathcal{O}(h^5) \\
y_{i+1}^* &= y_i + \gamma_1^* K_1 + \gamma_2^* K_2 + \gamma_3^* k_3 + \gamma_4^* k_4 + \gamma_5^* k_5 + \gamma_6^* k_6 + \mathcal{O}(h^5)
\end{aligned}
$$

The estimation of the truncation error ϵ and the optimal step size h_{opt} are then given by

$$
\epsilon = y_{i+1} - y_{i+1}^* = \sum_{n=1}^{6} (\gamma_n - \gamma_n^*) k_n
$$

$$
h_{opt} = \lambda h \left(\frac{\epsilon_0}{\epsilon} \right)^{\frac{1}{p+1}} \text{ where } 0.8 \leq \lambda \leq 0.9 \text{ and } p = 4 \text{ is the order.}
$$

This process is repeated with every iteration ensuring the magnitude of the error falls within the desired error threshold and if not, the step size is adapted and the iteration is repeated until the error conditions are met. Once a smaller error value is obtained, the solution values of the current iteration are written to file and the solver moves to the next time step to compute the solution of the next iteration. Using this method, we have little control over the step size and as such this method is prone to high computational costs due to generating a large number of data points when extremely small step sizes and error thresholds are set. For this reason it is important to find a workaround by writing our data to a text file rather than appending values to an array, helping to reduce the huge processing times and costs incurred without including this step. In some cases, text files of approximately 50Mb were generated using the RKF45 method and we found by storing these values in output files then reading them in later the processing times were reduced significantly.

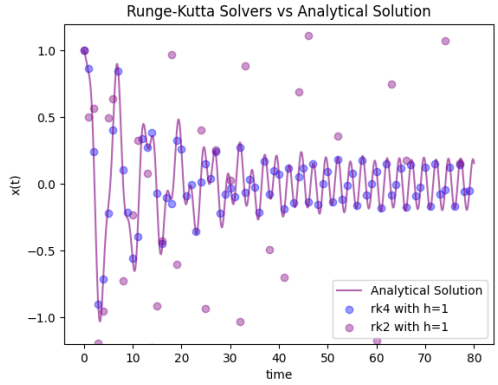
1.3 Discussion

As shown in figure 1, the general solution of any oscillating system has 2 characteristic states known as transient and steady states [1]. The transient state is characterised by the initial period where the system undergoes motion due to both oscillation and damping effects resulting in an exponentially decaying period of oscillation. The transient state is a period of complex instability governed by the solution to the homogeneous differential equation and the initial conditions imposed on the system. Once the system has reached stability, it enters a steady state phase where it oscillates at resonance frequency which can be seen as a steady harmonic oscillation around the equilibrium point. The steady state in this case is a direct result of the harmonic driving force F(t) = Asin(Ωt) which keeps the system oscillating at the resonance frequency around the equilibrium point. The steady state motion is governed by the solution

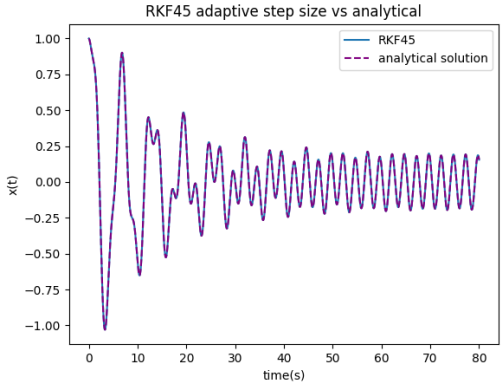
to the inhomogeneous differential equation $x_p(t) = \frac{Asin(\Omega t - \phi)}{M}$ and is independent of any initial conditions of the system, that is regardless of any initial conditions of the system, the system will return to its unique steady state determined by the driving force as a function of it's amplitude, frequency and phase[2]. Let us assume the differential equation describes a mass on a spring, we can then see that for a higher displacement in x_0 a greater restoring force F = -kx will be exerted on the mass. This would suggest that how quickly the system transitions from a transient to a steady state, the systems transient time, is determined by the initial condition x_0 and the resulting relative dominance of the restoring force against the driving force.

When integrating over coarse time grids with a step size of around 1, both the RK2 and RK4 solvers struggle to produce an accurate solution as the step size is too large. This is especially true for the RK2 solver, which suffers from decreased accuracy as shown by the exploding nature of the graph for the RK2 in figure 1a. This is due to the nature of how the RK2 integrates as while it does take intermediate steps, it takes less intermediate steps than the RK4 so for larger step sizes it is prone to overshooting and it never recovers and even for step sizes of the order 1×10^{-3} this method doesn't produce an accurate solution. The RK4 however takes an extra intermediate step and makes extra approximations, allowing it to benefit from a more steady stepping regime less susceptible to overstepping. For decreased step size, we see improved accuracy for both solvers with the RK4 producing more accurate results at larger steps relative to the RK2. It was noted that both solvers struggle to achieve 100% accuracy when compared against the analytical solution to the ODE unless a fine time grid was used and this is especially noticeable during the transient motion state. For step sizes of h = 1, the RK4 achieves a solution close to the expected value but not fully accurate.

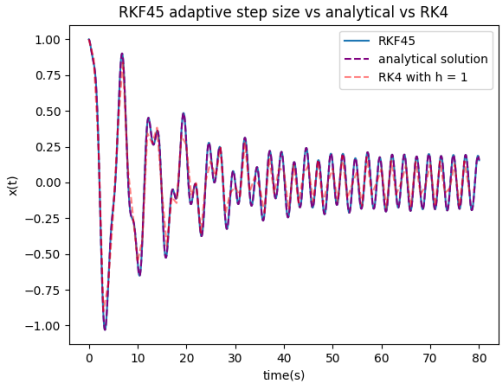
Finally, the performance of the RKF45 method was compared with the analytical solution as shown in figure 1b. With this method we see a significant improvement in the accuracy of the numerical solution as it achieves a highly accurate result comparable with the analytical solution. This is what should be expected as this method makes a greater number of approximations and uses an improved stepping method, adapting to error conditions in order to minimize the error and ensure the optimized step size is always used. The RKF45 was compared against the RK4 for various step sizes and it was found that for a relatively fine time grid, both algorithms produce an accurate solution as seen in figure 1d.



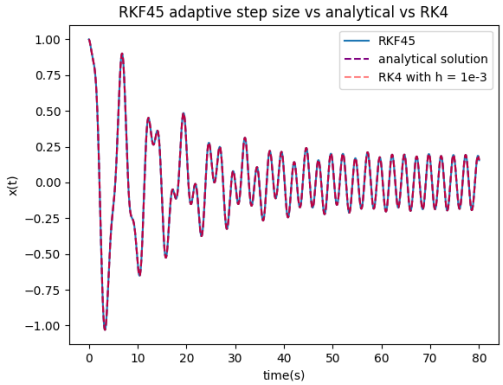
(a) RK2 and RK4 against analytical solution



(b) Runge-Kutta Fehlberg solution against analytical solution



(c) RKF45 vs RK4 h=1 vs analytical



(d) RKF45 vs RK4 h=1×10^{-3} vs analytical

Figure 1: Comparison of methods for the solution to the damped driven oscillator.

Comparing the run times of the 2 solvers capable of achieving an accurate solution, i.e the RK4 and RKF45, it was noted that the RK4 arrives at a solution faster than the RKF45 for the differential equation in this section. This is a relatively simple system however, so it is likely that the RKF45 outperforms the RK4 when integrating more complex systems such as nonlinear forced differential equations as seen in the van der Pol oscillator.

2 Van der Pol Oscillator

2.1 VdP Oscillator RKF45 Solution

The van der Pol Oscillator is characterised by the differential equation:

$$\ddot{x(t)} - \mu(1 - x^2(t))\dot{x(t)} + x(t) = 0$$

This second order ordinary differential equation can be re written as a system of first order ODEs by first rearranging the 2nd order ODE for $\ddot{x}$:

$$\ddot{x} = \mu(1 - x^2)\dot{x} - x$$

Next we set $\dot{x} = v$ and write the system of first order ODEs:

$$\dot{x} = v \tag{3}$$
$$\dot{v} = \mu(1 - x^2)v - x \tag{4}$$

Similarly, we can solve using the same method for a driven non-linear ODE:

$$\ddot{x}(t) - \mu(1 - x^2(t))\dot{x}(t) + x(t) = A\sin(\Omega t)$$

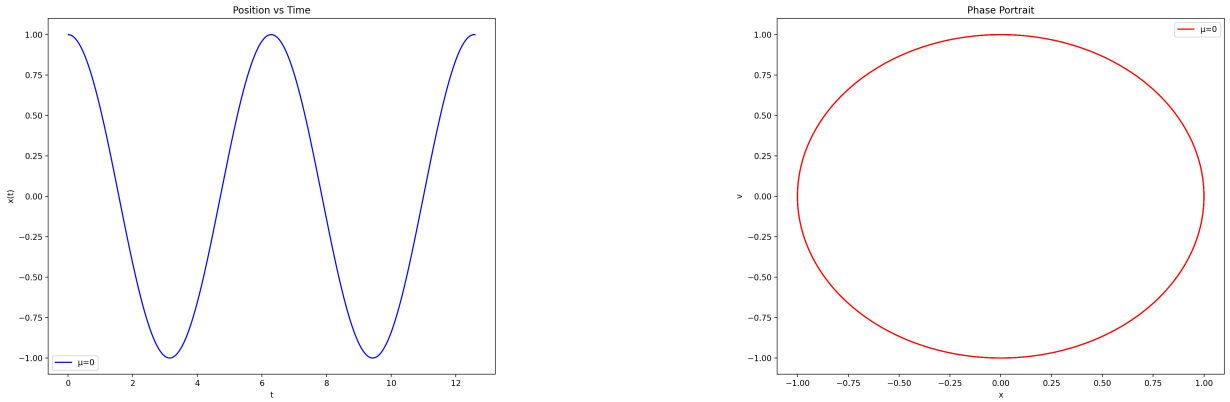
Again we can set $\dot{x} = v$ and express the system of ODEs is follows:

$$\dot{x} = v \tag{5}$$
$$\dot{v} = \mu(1 - x^2)v - x + A\sin(\Omega t) \tag{6}$$

As this system of ODEs is of the same form as discussed in section 1.3.1, we can use the RKF45 solver to find x(t) and v(t) using the same method as before by essentially integrating $\dot{v}$ to find v(t) and then repeating for x(t). The quadratic damping term is of particular interest as for $|x| < 1$, the damping force is negative which results in the amplification of the motion of the oscillator. Once the amplified motion reaches $|x| > 1$, the damping force is positive and so the oscillator undergoes damped motion with this cycle repeating. This behaviour is visualised in the phase portraits plotted in figures 3a and 3b. For the purpose of this section, we investigated the impact of changing the initial conditions and damping parameters by generating plots of the behaviour under various conditions and it was found that this system exhibits chaotic behaviour.

2.2 Discussion

A limit cycle is a closed trajectory in 2-D phase space with at least one other trajectory spiralling into or out of the closed trajectory as shown in figure 3. For the case where the trajectory is spiralling into the limit cycle we have a stable or attractive limit cycle which implies a system with self-sustained oscillations - that is any deviation or perturbation from the limit cycle trajectory will cause the system to oscillate before eventually undergoing damping and returning to the limit cycle[3]. For the purpose of our discussion and analysis, we focus only on the stable limit cycle although there are also cases of unstable and semi-stable limit cycles for other input parameters. For the case of the van der Pol oscillator, the system has a unique stable limit cycle for any $\mu > 0$. For the case where $\mu = 0$, it is shown graphically that the system returns to undamped harmonic oscillation which is represented by a circular phase diagram as shown in figure 2b. It is also evident by inspecting the form of the ODE for $\mu = 0$ that we return to the case of a linear unforced differential equation since $\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0$ becomes $\ddot{x} + x = 0$.

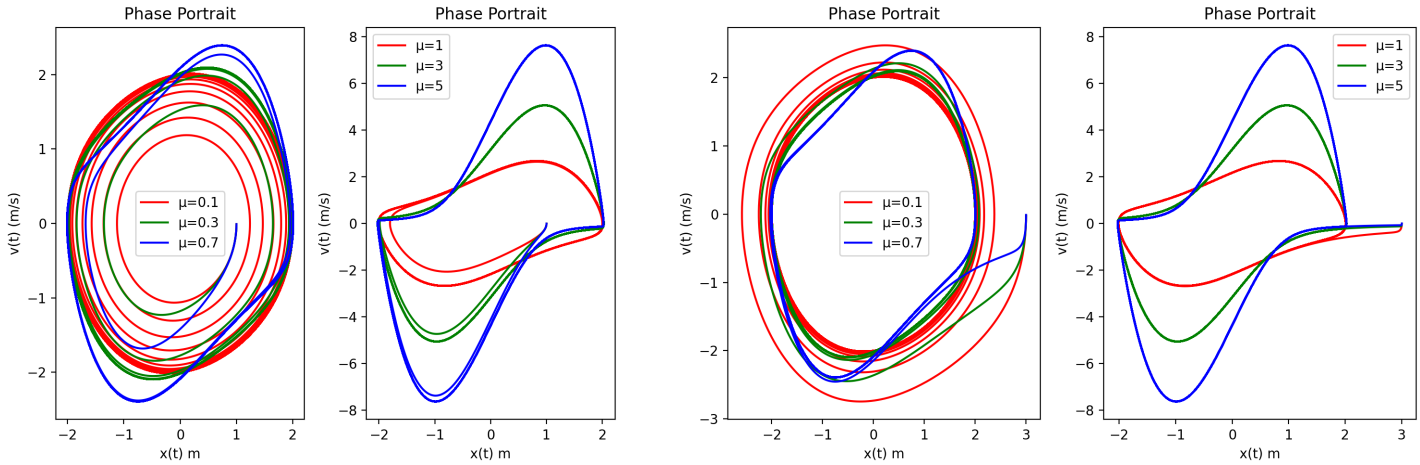


(a) Position vs Time for the vdP Oscillator

(b) Phase Portrait for $\mu = 0$

Figure 2: Position vs time and phase portrait for $\mu = 0$

For the $\mu > 0$ values plotted in figures 3a and 3b, it can be seen that when $x_0 = 1$ the trajectories spiral into the limit cycle from a trajectory originating inside the limit cycle and when $x_0 = 3$, the trajectories spiral into the limit cycle from a trajectory originating outside of the limit cycle thus indicating a stable limit cycle for the vdP oscillator as expected.



(a) $x_0 = 1$ for the vdP Oscillator

(b) $x_0 = 3$ for the vdP Oscillator

Figure 3: Phase portraits for varying μ and x_0 values.

For variations in the damping parameter μ, we found that this parameter essentially determines how quickly the trajectory of the oscillator recovers from perturbations and returns to the limit cycle. For greater values of μ, the system returns to its limit cycle sooner than for cases with a lower value of μ. This is visible by inspecting the plot of position vs time as shown in figure 4, which shows that for greater values of μ, a constant amplitude indicating the system has returned to the limit cycle is reached sooner.

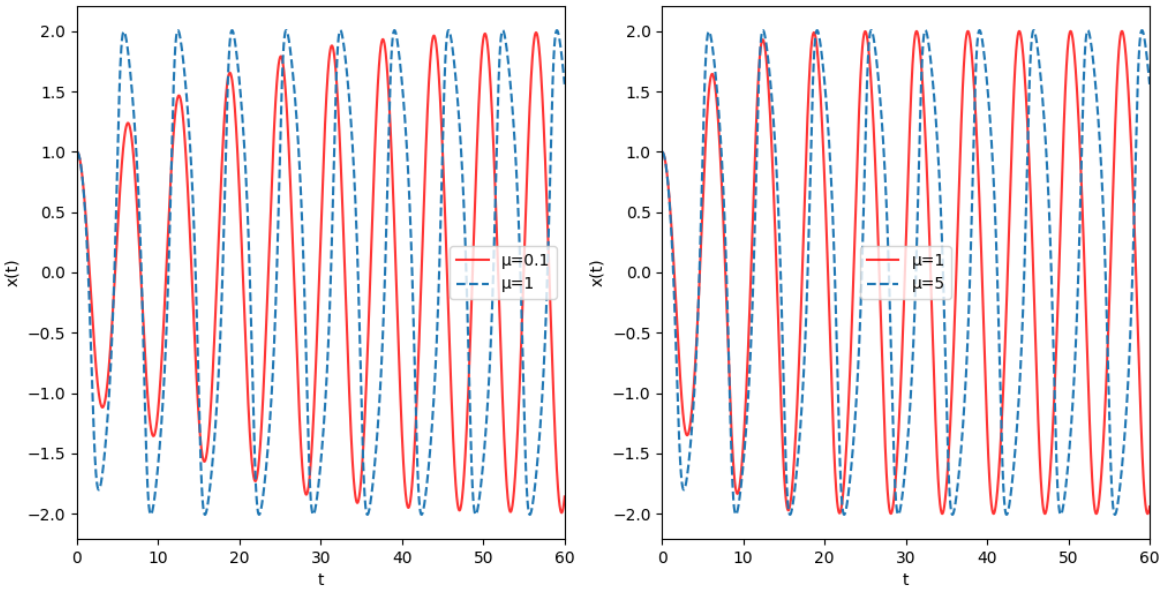
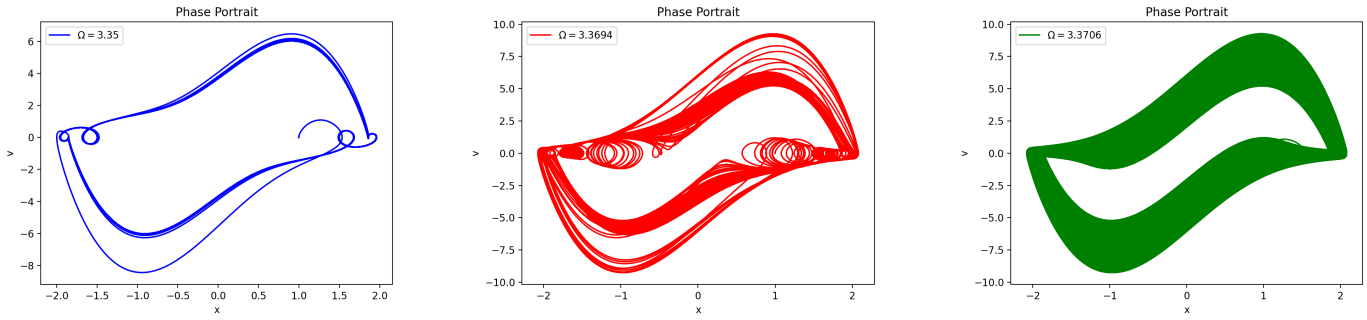


Figure 4: Shows the time taken for return to limit cycle for increasing μ is reduced.

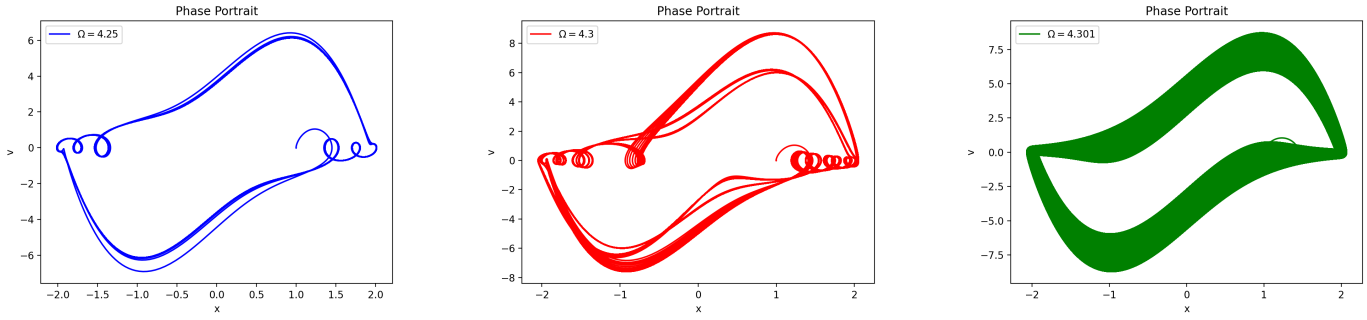
As explained in section 2.1, there is both a positive and negative damping force acting on the oscillator depending on the position $x(t)$. From figures 3a and 3b it can be seen that as μ increases so does the rate at which this damping effect occurs. This can also be seen by inspecting the form of the ODE where $\ddot{x} = \mu(1 - x^2)\dot{x} - x$. When μ is increased then the magnitude of the non linear damping force is increased, whether negative or positive overall, the magnitude of the damping will increase resulting in greater acceleration in the phase portraits.

When a driving force is applied to the van der Pol oscillator, over longer time intervals the system begins to exhibit behaviour satisfying the requirements of a chaotic system which are that the projected trajectories of the system are sufficiently dense and that the system is extremely sensitive to the initial conditions[3] as shown in figure 5. While $\Omega = 3.35$ leads to a mostly periodic closed loop as shown in figure 5a, altering the value of Ω to 3.3694 as shown in figure 5b shows an increase in period-doubling cascades. These period-doubling cascades are a source of chaos in dynamical systems[4] and are a result of infinitesimal changes in the initial conditions causing new periodic trajectories to emerge from existing trajectories with a period double that of the original trajectory, meaning that over time the system will become increasingly more chaotic. Incrementing further to $\Omega = 3.3706$, the number of period-doubling cascades is significantly greater than in 5b, creating a phase portrait with an extremely dense coverage of the loop. Since the trajectories for the infinitesimally incremented values of Ω are so significantly different and the case shown in figure 5 is sufficiently dense, we can say that the van der Pol oscillator is a system which exhibits chaotic behaviour. We can see that going from figure 5a to 5b we have a jump in initial value of Ω of double an order of magnitude greater than in figure 5b to 5c but for the latter, more chaos is generated. This sensitivity to infinitesimal changes in the parameters is demonstrated again in figure 6 with an even smaller incremented Ω value generating significantly more chaos than the previous examples. Another feature of figure 6a to note, is that while Ω is now higher than in figure 5c, the occurrence of period-doubling cascades has decreased indicating that the oscillator switches back and forth between order and chaos as Ω increases, and doesn't just get more chaotic.



(a) $\Omega = 3.35$ (b) $\Omega = 3.3694$ (c) $\Omega = 3.3706$

Figure 5: Phase Portraits for Varying Ω with $t_0 = 0, x_0 = 1, v_0 = 0, \mu = 5, A = 5$



(a) $\Omega = 4.25$ (b) $\Omega = 4.3$ (c) $\Omega = 4.301$

Figure 6: Phase Portraits for Varying Ω with $t_0 = 0, x_0 = 1, v_0 = 0, \mu = 5, A = 5$

References

[1] Fotso HF, Dohner E, Kemper A, Freericks JK. Bridging the Gap Between the Transient and the Steady State of a Nonequilibrium Quantum System. arXiv e-prints. 2021 Jan:arXiv:2101.00795.

[2] Fowler M. Oscillations III;. Available from: http://galileo.phys.virginia.edu/classes/152.mf1i.spring02/Oscillations4.htm.

[3] Tsatsos M. The Van der Pol Equation; 2008.

[4] Perez R, Glass L. Bistability, period doubling bifurcations and chaos in a periodically forced oscillator. Physics Letters A. 1982;90(9):441–443. Available from: https://www.sciencedirect.com/science/article/pii/0375960182903917.

Appendices

A Table of Values

Quantity	Name	SI Unit
x	Position	m
ζ	damping ratio	
ω_o	Angular Frequency	$rads^{-1}$
A	Amplitude	m
Ω	Frequency	H_Z
F	Force	N
t	Time	s
v	Velocity	ms^{-1}
μ	Non-Linear Damping Parameter	
T	Period	s^{-1}

B Integral

$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i}$$

$$\begin{aligned}
x_p(t) &= e^{\lambda_1 t}\int_{t_0}^{t} \frac{-A sin(\Omega t) e^{\lambda_2 t}}{-2ibe^{(\lambda_1+\lambda_2)t}}dt + e^{\lambda_2 t}\int_{t_0}^{t} \frac{A sin(\Omega t) e^{\lambda_1 t}}{-2ibe^{(\lambda_1+\lambda_2)t}}dt \\
&= A\left[e^{\lambda_1 t}\int_{t_0}^{t} \frac{(e^{i\Omega t} - e^{-i\Omega t})e^{\lambda_2 t}}{2i(2ib)e^{\lambda_1 t}e^{\lambda_2 t}}dt + e^{\lambda_2 t}\int_{t_0}^{t} \frac{(e^{i\Omega t} - e^{-i\Omega t})e^{\lambda_1 t}}{2i(2ib)e^{\lambda_2 t}e^{\lambda_1 t}}dt\right] \\
&= A\left[e^{\lambda_1 t}\int_{t_0}^{t} \frac{(e^{i\Omega t} - e^{-i\Omega t})}{2i(2ib)e^{\lambda_1 t}}\frac{e^{\lambda_2 t}}{e^{\lambda_2 t}}dt + e^{\lambda_2 t}\int_{t_0}^{t} \frac{(e^{i\Omega t} - e^{-i\Omega t})}{2i(-2ib)e^{\lambda_2 t}}\frac{e^{\lambda_1 t}}{e^{\lambda_1 t}}dt\right] \\
&= \frac{A}{4b}\left[-e^{\lambda_1 t}\int_{t_0}^{t} e^{(i\Omega-\lambda_1)t} - e^{(-i\Omega-\lambda_1)t}dt + e^{\lambda_2 t}\int_{t_0}^{t} e^{(i\Omega-\lambda_2)t} - e^{(-i\Omega-\lambda_2)t}dt\right] \\
&= \frac{A}{4b}\left[-e^{\lambda_1 t}\left[\frac{e^{(i\Omega-\lambda_1)t}}{(i\Omega-\lambda_1)} - \frac{e^{(-i\Omega-\lambda_1)t}}{(-i\Omega-\lambda_1)}\right]_{t_0}^{t} + e^{\lambda_2 t}\left[\frac{e^{(i\Omega-\lambda_2)t}}{(i\Omega-\lambda_2)} - \frac{e^{(-i\Omega-\lambda_2)t}}{(-i\Omega-\lambda_2)}\right]_{t_0}^{t}\right] \\
&= \frac{A}{4b}\left[\frac{-e^{i\Omega t}}{i\Omega-\lambda_1} + \frac{e^{-i\Omega t}}{-i\Omega-\lambda_1} + \frac{e^{i\Omega t}}{i\Omega-\lambda_2} - \frac{e^{-i\Omega t}}{-i\Omega-\lambda_2}\right] \\
&= \frac{A}{4b}\left[\frac{-e^{i\Omega t}(i\Omega-\lambda_2) + e^{i\Omega t}(i\Omega-\lambda_1)}{i^2\Omega^2 + i\Omega(-\lambda_1-\lambda_2) + \lambda_1\lambda_2}\frac{e^{-i\Omega t}(-i\Omega-\lambda_2) - e^{-i\Omega t}(-i\Omega-\lambda_1)}{i^2\Omega^2 - i\Omega(\lambda_1+\lambda_2) + \lambda_1\lambda_2}\right] \\
&= \frac{A}{4b}\left[\frac{-e^{i\Omega t}(i\Omega-\lambda_2-i\Omega+\lambda_1)}{-\Omega^2 + i\Omega(2a) + a^2 + b^2} + \frac{e^{-i\Omega t}(-i\Omega-\lambda_2+i\Omega+\lambda_1)}{-\Omega^2 - i\Omega(2a) + a^2 + b^2}\right] \\
&= \frac{A}{4b}\left[\frac{-(2ib)e^{i\Omega t}}{-\Omega^2 + i\Omega(2a) + a^2 + b^2} + \frac{(2ib)e^{-i\Omega t}}{-\Omega^2 - i\Omega(2a) + a^2 + b^2}\right] \\
&= \frac{Ai}{2}\left[\frac{-e^{i\Omega t}}{-\Omega^2 + 2ia\Omega + a^2 + b^2} + \frac{e^{-i\Omega t}}{-\Omega^2 + 2ia\Omega + a^2 + b^2}\right] \\
&= \frac{Ai}{2}\left[\frac{-e^{i\Omega t}(-\Omega^2 + 2ia\Omega + a^2 + b^2) + e^{-i\Omega t}(-\Omega^2 + 2ia\Omega + a^2 + b^2)}{(-\Omega^2 + 2ia\Omega + a^2 + b^2)(-\Omega^2 + 2ia\Omega + a^2 + b^2)}\right] \\
&= \frac{-Ai}{2}\left[\frac{e^{i\Omega t}(-\Omega^2 - 2ia\Omega + a^2 + b^2)}{(a^2 + b^2 - \Omega^2) + 4a^2\Omega^2} - \frac{e^{-i\Omega t}(-\Omega^2 + 2ia\Omega + a^2 + b^2)}{(a^2 + b^2 - \Omega^2) + 4a^2\Omega^2}\right] \\
&= \frac{A}{2i}\left[\frac{(e^{i\Omega t} - e^{-i\Omega t})(a^2 + b^2 - \Omega^2)}{(a^2 + b^2 - \Omega^2) + 4a^2\Omega^2} - \frac{(e^{i\Omega t} + e^{-i\Omega t})(2ia\Omega)}{(a^2 + b^2 - \Omega^2) + 4a^2\Omega^2}\right]
\end{aligned}$$

We can then begin making substitutions of the form:

$$sin(x) = \frac{e^{ix} - e^{-ix}}{2i} \qquad and \qquad cos(x) = \frac{e^{ix} + e^{-ix}}{2i}$$

$$\begin{aligned}
x_p(t) &= A\left[\frac{(e^{i\Omega t} - e^{-i\Omega t})}{2i}\frac{(a^2 + b^2 - \Omega^2)}{(a^2 + b^2 - \Omega^2)^2 + 4a^2\Omega^2} - \frac{(e^{i\Omega t} + e^{-i\Omega t})}{2}\frac{(2ia\Omega)}{i(a^2 + b^2 - \Omega^2)^2 + (A^2\Omega^2)}\right] \\
&= A\left[\frac{sin(\Omega t)(a^2 + b^2 - \Omega^2)}{(a^2 + b^2 - \Omega^2)^2 + 4a^2\Omega^2} - \frac{cos(\Omega t)(2a\Omega)}{(a^2 + b^2 - \Omega^2)^2 + 4a^2\Omega^2}\right]
\end{aligned}$$

here we insert values for a and b:

$$\begin{aligned}
x_p(t) &= A\left[\frac{sin(\Omega t)(\zeta^2\omega_o^2 + \omega_o^2(1-\zeta^2) - \Omega^2)}{(\zeta^2\omega_o^2 + \omega_o^2(1-\zeta^2) - \Omega^2)^2 + 4\zeta^2\omega_o^2\Omega^2} - \frac{cos(\Omega t)(2\zeta\omega_o\Omega)}{(\zeta^2\omega_o^2 + \omega_o^2(1-\zeta^2) - \Omega^2)^2 + 4\zeta^2\omega_o^2\Omega^2}\right] \\
&= A\left[\frac{sin(\Omega t)(\omega_o^2 - \Omega^2)}{(\omega_o^2 - \Omega^2)^2 + 4\zeta^2\omega_o^2\Omega^2} - \frac{Cos(\Omega t)(2\zeta\omega_o\Omega)}{(\omega_o^2 - \Omega^2)^2 + 4\zeta^2\omega_o^2\Omega^2}\right]
\end{aligned}$$

This can be simplified using:

$$sin(a)cos(b) - cos(a)sin(b) = sin(a-b) \qquad and \qquad z = u + iv$$

The numerator of x_p can be put into the form of the trigonometric identity:

$$sin(m)cos(n) - cos(m)sin(n) = sin(m-n)$$

In order to this we can compare it to the complex equivalent complex exponential and find the real and imaginary components:

$$z = u + iv = Me^{i\phi}$$

The complex exponential can be put into polar form to give:

$$Me^{i\phi} = (cos\phi + isin\phi)M$$

$$Re_z = u = Mcos\phi \qquad\qquad Im_z = v = Msin\phi$$

From the numerator of x_p and the desired form we can compare the components to see what the real and imaginary parts are:

$$sin(m)MCos(n) - cos(m)MSin(n)$$
$$sin(\Omega t)(\omega_o^2 - \Omega) - cos(\Omega t)(2\zeta\omega_o\Omega)$$

From this we can see that:

$$m = \Omega t, \qquad Mcos(n) = \omega_o^2 - \Omega^2, \qquad Msin(n) = 2\zeta\omega_o\Omega$$

This can then be used to find the angle, ϕ, and the magnitude, M, of the complex number z:

$$\phi = tan^{-1}\left(\frac{Im_z}{Re_z}\right)$$
$$\phi = tan^{-1}\left(\frac{2\zeta\omega_o\Omega}{\omega_o^2 - \Omega^2}\right)$$

$$M = \sqrt{Re_z^2 + Im_z^2}$$
$$M = \sqrt{(\omega_o^2 - \Omega^2)^2 + (2\zeta\omega_o\Omega)^2}$$

Subbing these values back into x_p gives:

$$x_p(t) = A\left[\frac{sin(\Omega t)Mcos(\phi) - cos(\Omega t)Msin(\phi)}{(\omega_o^2 - \Omega^2)^2 + (2\zeta\omega_o\Omega)^2}\right]$$

The denominator ofx_p is equal to M^2 therefore x_p becomes:

$$x_p(t) = A\left[\frac{sin(\Omega t)Mcos(\phi) - cos(\Omega t)Msin(\phi)}{M^2}\right]$$

Using the trigonometric identity we then get:

$$x_p(t) = \frac{Asin(\Omega t - \phi)}{M}$$

C Second Order Runge-Kutta (RK2) Code

```
def rk2(f, start, stop, x0, v0, w0, A, zeta, omega, step):
    '''
    Runge-Kutta solver, order 2.
    '''

    # Initial setup
    t_grid = np.arange(start, stop, step)

    x = np.zeros(len(t_grid))
    v = x.copy()
    v[0] = v0
    x[0] = x0

    # Loop through specified steps
    for i in range(len(x) - 1):

        # k is the first equation of the system of ODEs, l is the second.

        k1 = step * f(t_grid[i], x[i], v[i])[0]
        l1 = step * f(t_grid[i], x[i], v[i])[1]

        k2 = step * f(t_grid[i] + step/2, x[i] + k1/2, v[i] + l1/2)[0]
        l2 = step * f(t_grid[i] + step/2, x[i] + k1/2, v[i] + l1/2)[1]

        # k vals give x, l vals give v
        x[i+1] = x[i] + k2
        v[i+1] = v[i] + l2

    return x, t_grid, v
```

D Fourth Order Runge-Kutta (RK4) Code

```
def rk4(f, start, stop, x0, v0, w0, A, zeta, omega, step):
    '''
    Runge-Kutta solver, order 4.
    '''

    # Initial setup
    t_grid = np.arange(start, stop, step)

    x = np.zeros(len(t_grid))
    v = x.copy()
    v[0] = v0
    x[0] = x0

    # Loop through specified steps
    for i in range(len(x) - 1):

        # k is the first equation of the system of ODEs, l is the second.
        k1 = step * f(t_grid[i], x[i], v[i])[0]
        l1 = step * f(t_grid[i], x[i], v[i])[1]

        k2 = step * f(t_grid[i] + step/2, x[i] + k1/2, v[i] + l1/2)[0]
        l2 = step * f(t_grid[i] + step/2, x[i] + k1/2, v[i] + l1/2)[1]

        k3 = step * f(t_grid[i] + step/2, x[i] + k2/2, v[i] + l2/2)[0]
        l3 = step * f(t_grid[i] + step/2, x[i] + k2/2, v[i] + l2/2)[1]

        k4 = step * f(t_grid[i] + step, x[i] + k3, v[i] + l3)[0]
        l4 = step * f(t_grid[i] + step, x[i] + k3, v[i] + l3)[1]

        # k vals give x, l vals give v
        x[i+1] = x[i] + k1/6 + k2/3 + k3/3 + k4/6
        v[i+1] = v[i] + l1/6 + l2/3 + l3/3 + l4/6

    return x, t_grid, v
```

E Runge-Kutta-Fehlberg (RK45) Code

```
def rkf45(f, t0, t1, x0, v0, A, omega, filename1, filename2, mu=0):
    """
    Adaptive Runge-Kutta Fehlberg algorithm.
    """

    # Initial conditions
    h = 0.01
    t = t0
    x = x0
    v = v0
    lam = 0.9
    e0 = 1E-12

    # Using name 'fi' if you use standard 'f' then it tries to call the function dxdt

    fi = open(f'{filename1}', 'w')
    fi2 = open(f'{filename2}', 'w')

    # Butcher tableau coefficients for beta and alpha (only the ones that are fractions,
       for better speed)
    c0 = 1/4
    c1 = 3.0/8.0
    c2 = 3.0/32.0
    c3 = 9.0/32.0
    c4 = 12.0/13.0
    c5 = 1932.0/2197.0
    c6 = 7200.0/2197.0
    c7 = 7296.0/2197.0
    c8 = 439.0/216.0
    c9 = 3680.0/513.0
    c10 = 845.0/4104.0
    c11 = 0.5
    c12 = 8.0/27.0
    c13 = 3544.0/2565.0
    c14 = 1859.0/4104.0
    c15 = 11.0/40.0

    # Coefficients to calculate error and delta x and delta v
    ch = np.array([25.0/216.0, 0.0, 1408.0/2565.0, 2197.0/4104.0, -1.0/5.0, 0.0])
    ck = np.array([16.0/135.0, 0.0, 6656.0/12825.0, 28561.0/56430.0, -9.0/50.0, 2.0/55.0])
    ct = ck-ch

    # k is the vector for the first ODE, l is for the second.
    k = np.zeros(6)
    l = np.zeros(6)

    while t <= t1:

        # Compute each k and l value, the order of this matters.
        k[0] = h * f(t, x, v, mu)[0]
        l[0] = h * f(t, x, v, mu)[1]

        k[1] = h * f(t + c0*h, x + c0*k[0], v + c0*l[0], mu)[0]
        l[1] = h * f(t + c0*h, x + c0*k[0], v + c0*l[0], mu)[1]

        k[2] = h * f(t + c1*h, x + c2*k[0] + c3*k[1], v + c2*l[0] + c3*l[1], mu)[0]
        l[2] = h * f(t + c1*h, x + c2*k[0] + c3*k[1], v + c2*l[0] + c3*l[1], mu)[1]
```

```
        k[3] = h * f(t + c4*h, x + c5*k[0] - c6*k[1] + c7*k[2], v + c5*l[0] - c6*l[1] + c7*
            l[2], mu)[0]
        l[3] = h * f(t + c4*h, x + c5*k[0] - c6*k[1] + c7*k[2], v + c5*l[0] - c6*l[1] + c7*
            l[2], mu)[1]

        k[4] = h * f(t + h, x + c8*k[0] - 8.0*k[1] + c9*k[2] - c10*k[3], v + c8*l[0] - 8.0*
            l[1] + c9*l[2] - c10*l[3], mu)[0]
        l[4] = h * f(t + h, x + c8*k[0] - 8.0*k[1] + c9*k[2] - c10*k[3], v + c8*l[0] - 8.0*
            l[1] + c9*l[2] - c10*l[3], mu)[1]

        k[5] = h * f(t + c11*h, x - c12*k[0] + 2.0*k[1] - c13*k[2] + c14*k[3] - c15*k[4], v
             - c12*l[0] + 2.0*l[1] - c13*l[2] + c14*l[3] - c15*l[4], mu)[0]
        l[5] = h * f(t + c11*h, x - c12*k[0] + 2.0*k[1] - c13*k[2] + c14*k[3] - c15*k[4], v
             - c12*l[0] + 2.0*l[1] - c13*l[2] + c14*l[3] - c15*l[4], mu)[1]


        # Calculate the error and change in x, and v.
        error = 0.0
        delta_x = 0.0
        delta_v = 0.0
        for i in range(len(ct)):
            error += ct[i]*k[i]
            delta_x += ck[i]*k[i]
            delta_v += ck[i]*l[i]

        # If the error is too large, adjust step size and repeate iteration.
        error = abs(error)
        if error > e0:
            h = lam * h * (e0 / error)**0.2
            continue

        # Compute; x, v, t and write them to file. Then compute optimal step size
        # and move to next loop.
        x += delta_x
        v += delta_v
        t += h
        print(Fore.RED + '    ', t, end='\r')
        fi.write('%f\n %f\n' % (x, t))
        fi2.write('%f\n %f\n' % (v, t))
        if error <= e0:
            h = lam * h * (e0 / error)**0.2
```

F Task 2

```python
import numpy as np
import matplotlib.pyplot as plt
# Importing our function from odesolvers.py
from odesolvers import rk2


def dxdt(t, x, v):
    return v, -2*zeta*w0*v-w0**2*x + A*np.sin(omega*t)


# Initial conditions
t0 = 0
x0 = 1
v0 = 0
zeta = 0.07
w0 = 1
A = 1
omega = 2.5

# Compute x, t and v.
x, t, v = rk2(dxdt, t0, 80, x0, v0, w0, A, 0.07, omega, 10000)

# Plot
plt.figure()
plt.plot(t, x, '-', label='rk2')
plt.legend()
plt.show()
```

G Task 3

```python
import numpy as np
import matplotlib.pyplot as plt
# Importing our functions from odesolvers.py
from odesolvers import rk2, rk4

def dxdt(t, x, v):
    return v, -2.*zeta*w0*v-w0**2.*x + A*np.sin(omega*t)

# Initial conditions
t0 = 0.
x0 = 1.
v0 = 0.
zeta = 0.07
w0 = 1.
A = 1.
omega = 2.5

# Compute; x, t and v using both RK2 and RK4 methods
x, t, v = rk2(dxdt, t0, 80, x0, v0, w0, A, 0.07, omega, 1000)
x2, t2, v2 = rk4(dxdt, t0, 80, x0, v0, w0, A, 0.07, omega, 1000)

# Analytical solution for comparison to RK methods.
phi = np.tan((2.*zeta*w0*omega) / (w0**2. - omega**2.))

M = np.sqrt((w0**2. - omega**2.)**2. + (2.*zeta*w0*omega)**2.)

xp = A*np.sin(omega*t - phi) / M

a = zeta*w0
b = w0*(np.sqrt(1.-zeta**2.))
lambda1 = -a + (1j * b)
lambda2 = -a - (1j * b)
xh = np.exp(lambda1*t) + np.exp(lambda2*t)

sol = xh.real + xp
# Plot; RK2, RK4 and analytical solution.
plt.figure()
plt.plot(t, x, '-', label='rk2')

plt.plot(t, sol, '-', label='Analytical Solution', alpha=0.5)
plt.plot(t2, x2, '-', label='rk4')
plt.legend()
plt.show()
```

H Task 4

```
import numpy as np
import math
import matplotlib.pyplot as plt
import pandas as pd
# Importing our function from odesolvers.py
from odesolvers import rkf45


def dxdt(t, x, v, mu):
    return v, -2*zeta*w0*v-w0**2*x + A*np.sin(omega*t)

# Initial conditions:
t0 = 0
x0 = 1
v0 = 0
w0 = 1
A = 1
omega = 2.5
zeta = 0.07
t1 = 80

# Call adaptive RKF45 method.
function = rkf45(dxdt, t0, t1, x0, v0, A, omega, 'data/task4_1.txt', 'data/task4_2.txt')

# Read data from file, to Pandas dataframe.
df = pd.read_csv('data/task4_1.txt', header=None)

# Get; x and t from dataframe.
x = df.iloc[::2,:]
t = df.iloc[1::2,:]

# Plots
fig, ax1 = plt.subplots(1, 1, figsize=(10,8))
ax1.plot(t, x, 'k--', label = 'zeta < w, dampened oscillations')
ax1.set(title='Dampening', xlabel='time(s)', ylabel='x(t)')
ax1.legend(loc='best')
plt.show()
```

I Task 5

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
# Importing our function from odesolvers.py
from odesolvers import rkf45

def dxdt(t, x, v, mu):
    '''
    Input function for runge-kutta, 2 first order ODEs for the van der Pol equation.
    '''
    return v, mu*(1-x**2)*v-x


# Initial conditions:
t0 = 0
x0 = 1.0
v0 = 0
A = 0
omega = 0
t1 = 2*np.pi
mu = 0

# Call adaptive RKF45 method for the van der Pol oscillator.
function = rkf45(dxdt, t0, t1, x0, v0, A, omega, 'data/task5_1.txt', 'data/task5_2.txt', mu
    )

# Read file into Pandas dataframe.
df = pd.read_csv('data/task5_1.txt', header=None)
df2 = pd.read_csv('data/task5_2.txt', header=None)

# Read dataframe into arrays for; x, t and v
x = df.iloc[::2,:]
t = df.iloc[1::2,:]
v = df2.iloc[::2,:]

# Plots
fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True)
ax1.plot(t, x, 'k--', label = '\u03bc=0')
ax1.set(title='Trajectory vs Time', xlabel='Time (s)', ylabel='x (m)')
ax1.legend(loc='best')

ax2.plot(x, v, 'b-', label='\u03bc=0')
ax2.set(title='Phase Portrait', xlabel='x (m)', ylabel='v (m/s)')
plt.show()
```

J Task 6

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
# Importing our function from odesolvers.py
from odesolvers import rkf45

def dxdt(t, x, v, mu):
    '''
    Input function for runge-kutta, 2 first order ODEs for the van der Pol equation.
    '''
    return v, mu*(1-x**2)*v-x

# Initial conditions:
t0 = 0
x0 = 1.0
v0 = 0
A = 0
omega = 0

# Different mu values for x0 = 1 or x0 = 3
x1_mu0_1 = rkf45(dxdt, t0, 500, x0, v0, A, omega, 'data/task6_1_mu_0_1.txt', 'data/
    task6_2_mu_0_1.txt', 0.1)
x1_mu0_3 = rkf45(dxdt, t0, 500, x0, v0, A, omega, 'data/task6_1_mu_0_3.txt', 'data/
    task6_2_mu_0_3.txt', 0.3)
x1_mu0_7 = rkf45(dxdt, t0, 500, x0, v0, A, omega, 'data/task6_1_mu_0_7.txt', 'data/
    task6_2_mu_0_7.txt', 0.7)
x1_mu1 = rkf45(dxdt, t0, 500, x0, v0, A, omega, 'data/task6_1_mu_1.txt', 'data/task6_2_mu_1
    .txt', 1)
x1_mu3 = rkf45(dxdt, t0, 500, x0, v0, A, omega, 'data/task6_1_mu_3.txt', 'data/task6_2_mu_3
    .txt', 3)
x1_mu5 = rkf45(dxdt, t0, 500, x0, v0, A, omega, 'data/task6_1_mu_5.txt', 'data/task6_2_mu_5
    .txt', 5)

# Datasets
df = pd.read_csv('data/task6_1_mu_0_1.txt', header=None)
df2 = pd.read_csv('data/task6_2_mu_0_1.txt', header=None)

df3 = pd.read_csv('data/task6_1_mu_0_3.txt', header=None)
df4 = pd.read_csv('data/task6_2_mu_0_3.txt', header=None)

df5 = pd.read_csv('data/task6_1_mu_0_7.txt', header=None)
df6 = pd.read_csv('data/task6_2_mu_0_7.txt', header=None)

df7 = pd.read_csv('data/task6_1_mu_1.txt', header=None)
df8 = pd.read_csv('data/task6_2_mu_1.txt', header=None)

df9 = pd.read_csv('data/task6_1_mu_3.txt', header=None)
df10 = pd.read_csv('data/task6_2_mu_3.txt', header=None)

df11 = pd.read_csv('data/task6_1_mu_5.txt', header=None)
df12 = pd.read_csv('data/task6_2_mu_5.txt', header=None)

# Arrays
x = df.iloc[::2,:]
t = df.iloc[1::2,:]
v = df2.iloc[::2,:]
```

```
x2 = df3.iloc[::2,:]
t2 = df3.iloc[1::2,:]
v2 = df4.iloc[::2,:]

x3 = df5.iloc[::2,:]
t3 = df5.iloc[1::2,:]
v3 = df6.iloc[::2,:]

x4 = df7.iloc[::2,:]
t4 = df7.iloc[1::2,:]
v4 = df8.iloc[::2,:]

x5 = df9.iloc[::2,:]
t5 = df9.iloc[1::2,:]
v5 = df10.iloc[::2,:]

x6 = df11.iloc[::2,:]
t6 = df11.iloc[1::2,:]
v6 = df12.iloc[::2,:]

# Plots
fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True)
ax1.plot(x, v, 'r-', label = '\u03bc=0.1')
ax1.plot(x2, v2, 'g-', label='\u03bc=0.3')
ax1.plot(x3, v3, 'b-', label='\u03bc=0.7')
ax1.set(title='Phase Portrait', xlabel='x(t) m', ylabel='v(t) (m/s)')
ax1.legend(loc='best')

ax2.plot(x4, v4, 'r-', label = '\u03bc=1')
ax2.plot(x5, v5, 'g-', label='\u03bc=3')
ax2.plot(x6, v6, 'b-', label='\u03bc=5')
ax2.set(title='Phase Portrait', xlabel='x(t) m', ylabel='v(t) (m/s)')
ax2.legend(loc='best')

plt.show()
```

K Task 7

```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
# Importing our function from odesolvers.py
from odesolvers import rkf45
from numba import jit

# Large datasets -> use this:
mpl.rcParams['agg.path.chunksize']=50001

#jit(nopython=True)
def dxdt(t, x, v, mu):
    '''
    Input function for runge-kutta, 2 first order ODEs for the van der Pol equation with
        forcing term included.
    '''
    return v, mu*(1-x**2)*v-x + A*np.sin(omega*t)



# Initial conditions:
t0 = 0
x0 = 1.0
v0 = 0
A = 5.0
omega = 3.35
mu = 5.0
t1 = 50000.0

# Call RKF45 method for van der Pol oscillator
function = rkf45(dxdt, t0, t1, x0, v0, A, omega, 'data/task7_1.txt', 'data/task7_2.txt', mu
    )

# Datasets
df = pd.read_csv('data/task7_1.txt', header=None)
df2 = pd.read_csv('data/task7_2.txt', header=None)

# Arrays
x = df.iloc[::2,:]
t = df.iloc[1::2,:]
v = df2.iloc[::2,:]

# Plots
plt.figure()
plt.plot(x, v, 'b-', label=rf'$\Omega={omega}$')
plt.title('Phase Portrait')
plt.xlabel('x (m)')
plt.ylabel('v (m/s)')
plt.show()
```