UCD School of Physics
Scoil na Fisice UCD

Javier Roca García-Valiño
22704709

Computational Scattering

Abstract

The aim of this experiment was to determine if numerical methods, specifically the RK4 method, are sufficiently accurate to model the trajectories of particles in a force field such as gravitational or electrostatic fields. This was done by implementing the fourth-order Runge-Kutta method to solve the equations of motion of a positron in repulsive, attractive, and asymmetrical potentials at different time steps and initial conditions. The effect of different initial energies E and impact parameters b on the solution and its accuracy was studied through the RK4 algorithm and the analysis of the conservation of the total energy of the system. The simulations successfully demonstrated that the RK4 method is sufficiently accurate even for particles under complex potentials resulting in chaotic dynamics, highlighting its applications on the modelling of physical systems from celestial mechanics to charged particle dynamics.

Contents

1 Introduction

The behaviour of a particle under a force field can be studied by solving the equations of motion that describe the particle's conduct. To do that, several numerical methods exist and can be used to work out the particle's motion. The fourth-order Runge-Kutta method (RK4) is a well-studied method with great stability at large time-steps that can be used for simulating the particle's movement under an electrostatic field[1].

The aim of this experiment was to analyse, through the RK4 method, the scattering of particles in an electrostatic field determining if numerical methods, and specifically the RK4 method, are sufficiently accurate to model scenarios like these. An interesting and useful check is to simulate the particle's motion under asymmetrical potentials causing chaotic scattering, i.e. scattering where the small changes in the initial conditions lead to vastly different trajectories.

This was done by implementing the fourth-order Runge-Kutta method to solve the equations of motion of a positron in repulsive, attractive and asymmetrical potentials at different time-steps and initial conditions.

2 Theoretical background and formulae

2.1 Scattering of particles in a potential field

The motion of a particle in a force field $\vec{F}$ can be written as:

$$m\frac{d^2\vec{x}}{dt^2} = \vec{F}(\vec{x}, \frac{d\vec{x}}{dt}, t) \tag{1}$$

with m being the mass of the participle (taken to be 1 in this experiment)[1]. Re-scaled units are used throughout the whole experiment to simplify equations, improve numerical stability by avoiding rounding, facilitate comparison between simulations and enhance universality. In re-scaled units, the force due to an electrostatic field is

$$\vec{F}(\vec{r}) = -\frac{dV}{d\vec{r}} = \frac{\hat{r}}{r^2} = \frac{\vec{r}}{r^3} \tag{2}$$

[1]

Note that in 2d,

$$\begin{cases} \vec{r} = (x, y) \\ \vec{v} = \dot{\vec{r}} = (\dot{x}, \dot{y}) \\ \vec{a} = \dot{\vec{v}} = (\ddot{x}, \ddot{y}) = \left(-\frac{dV}{dx}, -\frac{dV}{dy}\right) \end{cases} \tag{3}$$

By modelling the equations of motion for a positron in a repulsive Coulomb potential field where the particle is incoming parallel and at a distance b from the x-axis, a trajectory like the one being represented in Figure 1 will be obtained.

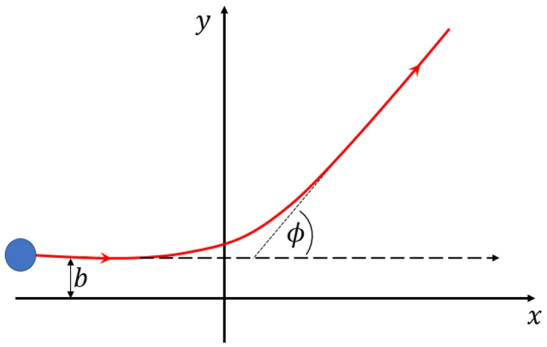


Figure 1: Schematic of position-positron scattering under a repulsive potential[1].

The positron, entering with an initial velocity of $\sqrt{2E}$ in the x direction and 0 in the y direction will interact with the field and be scattered by an angle ϕ (the scattering angle) after a transit time t_t, defined as the time taken to exit the asymptotic region.

2.2 Numerical methods and the RK4 algorithm

It is often that physical systems are described by ordinary differential equations that cannot be solved analytically. This can be the case of the equations of motion describing the trajectory of a particle in an electrostatic potential. Numerical methods provide approximate solutions where they update the system's state at a certain rate h called time-steps.

Amongst the most widely-known methods, one can find the Euler and the Runge-Kutta methods. These have different-order accuracy with the Euler method only reaching first-order and the RK2 and RK4 reaching second and fourth-order accuracy respectively[2].

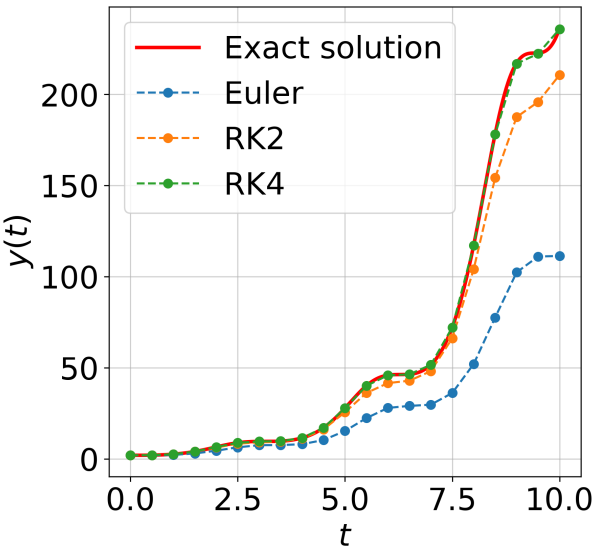


Figure 2: Representation of the accuracy of the Euler, RK2 and RK4 methods (see Appendix for algorithms[2]) modelling $y' = sin(t)^2 \cdot y$ curve.

The fourth-order Runge-Kutta method (RK4) estimates the solution at the next step as

$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

with the base case being $x_0 = x(t_0)$ and the rate of change $dx/dt = f(x,t)$[1]. The k-factors represent the slope of the function through time and are defined as:

$$\begin{cases} k_1 = f(t_n, x_n) \\ k_2 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1) \\ k_3 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2) \\ k_4 = f(t_n + h, x_n + hk_3) \end{cases}$$

This same logic can be applied to obtain the Rung-Kutta method for a 2d problem:

$$v_{n+1} = v_n + \frac{h}{6}(k_{1_v} + 2k_{2_v} + 2k_{3_v} + k_{4_v}) \quad (4a)$$

$$r_{n+1} = r_n + \frac{h}{6}(k_{1_r} + 2k_{2_r} + 2k_{3_r} + k_{4_r}) \quad (4b)$$

where the k-parameters are

$$\begin{cases} k_{1_v} = a(r_i) \\ k_{2_v} = a(r_i + \frac{h}{2}k_{1_r}) \\ k_{3_v} = a(r_i + \frac{h}{2}k_2 r) \\ k_{4_v} = a(r_i + hk_{3_r}) \end{cases} \quad (5)$$

where

$$a = \nabla V(x,y)$$

as defined earlier in equation 3, and

$$\begin{cases} k_{1_r} = v_n \\ k_{2_r} = v_n + \frac{h}{2}k_{1_v} \\ k_{3_r} = v_n + \frac{h}{2}k_{2_v} \\ k_{4_r} = v_n + hk_{3_v} \end{cases} \quad (6)$$

It is important to note that, since the k-parameters for r and for v are dependent on each other, these must be computed simultaneously.

3 Computational method

The fourth-order Runge-Kutta method was implemented through

```
RK4(state, acceleration, rmax,
↪  repulsive=True)
```

- state: numpy array, initial position and velocity vector (x, y, vx, vy).
- acceleration: function, returns the acceleration a(x,y) computed from the potential.
- rmax: float, radius at which integration stops (end of the asymptotic region).
- repulsive: bool, optional; if False, reverses the sign of the potential to simulate attraction.

The function integrated the particle's equations of motion using the fourth-order Runge–Kutta method, updating position and velocity at every step until the particle left the potential region, i.e. reached `rmax`.

3.1 Coulomb scattering

Using this, a positron in the field of a repulsive Coulomb potential was simulated with initial conditions $b = 0.25$, $E = 5$ and a time-step $h = 0.01$ obtaining the scattering angle and the transit time. These were calculated by obtaining the angle between v_x and $v_y = 0$ through `numpy.arctan2(vy,vx)`[3] and by multiplying the number of time-steps taken (i.e. the number of states calculated) by h.

The accuracy of this solution can be evaluated by analysing the conservation of energy throughout the trajectory. For this, the `RK4` function was modified to implement a new optional boolean argument `obtain_energies=False` which. When `obtain_energies=True`, `RK4` obtains and returns not only the states but also the total energy of the system throughout the entire trajectory of the particle.

The conservation of energy was studied through a range of different h values to see the relationship between the time-step and the accuracy of the solution. This assisted the choice of the time-step size such that the solution is accurate enough without causing a large computational time.

The effect on the solution and its accuracy of different initial energies E and different impact parameters b were also studied. A positron in a repulsive and an attractive Coulomb potential were simulated respectively for the range of E and b values allowing us to also observe the behaviour of the positron in the attractive potential.

3.2 Chaotic scattering

For the second part of the experiment, the Coulomb potential was replaced for an asymmetrical potential

$$V = -y^2 \cdot e^{-(x^2+y^2)}$$

This was visualised through a contour plot to observe its structure. Then, just like in the last step of Section 3.1, the path of a positron in the electrostatic force field was modelled with different b values. This time, the initial kinetic energy was set to $E = 1$.

Once the scattering of the positron under the new potential was represented, it was time to try and find a chaotic regime. This

was achieved by scanning through different values of E and b simultaneously obtaining the relative difference in scattering angle and transit time from one value to the next one and selecting those ones that yielded the largest differences of all of them. In order words, values for E and b were selected such that a small variation on this parameters would cause a vastly large difference on the scattering angle and the transit time.

After an appropriate value for E was chosen, neighbouring values of b within a chaotic region were iterated through simulating the distinct trajectories a positron with those initial conditions would follow.

4 Results and discussion

4.1 Coulomb scattering

The first of the plots that were obtained is the one shown in Figure 3 where the scattering of a positron by the force of a repulsive Coulomb potential can be appreciated.

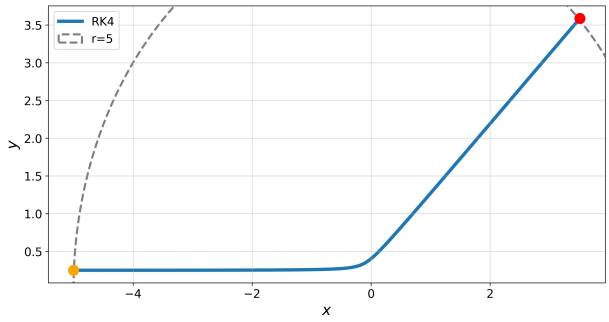


Figure 3: Positron-position scattering under repulsive Coulomb potential with $E = 5$, $b = 0.25$ and $h = 0.01$.

The simulation encompasses the trajectory through the asymptotic region from the initial state $x = -5$, $y = b$ up to when the particle exits the circle $r = \sqrt{x^2 + y^2} = 5$. The scattering angle obtained was of 42.78° with a transit time of 3.27 re-scaled units.

Figure 5 shows the evolution of the total energy $E = T + V$ of the state as a function of time when $h = 0.01$. It can be seen how the maximum deviation is of seven orders of magnitude smaller than the initial energy implying that a time-step size of 0.01 is a reasonable choice for an accurate solution.

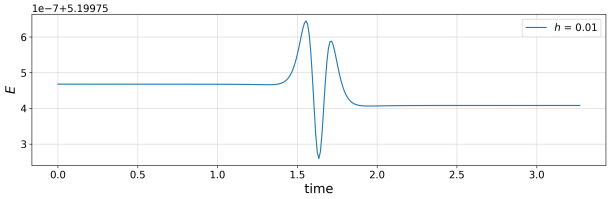


Figure 4: Plot of the total energy of the system $E = T + V$ as a function of time for $h = 0.01$ (where $E = 5$ and $b = 0.25$) used to interpret the goodness of the solution.

More in general, the accuracy of the solution as seen in Figure 5 linearly increases with lower time-step values. This is expected as the lower the step size, the higher is the resolution of the simulation.

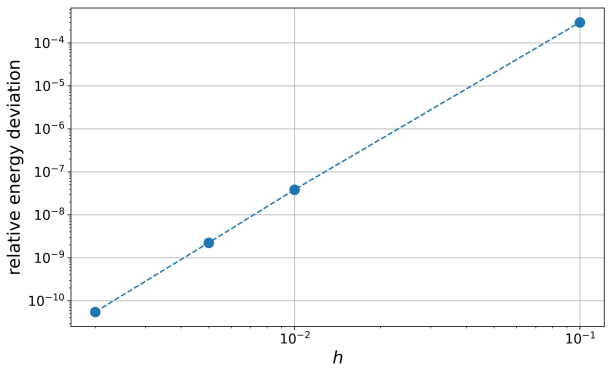


Figure 5: Relationship between the relative maximum energy and the time-step parameter h representing the dependence of the accuracy of the solution on the time-step size.

Since $h = 0.01$ was found to be a good parameter to ensure an accurate solution with-

out causing large computational times, that was the value used throughout the rest of the experiment. Values like $h = 0.1$ would have compromised the accuracy of the experiment since the relative energy deviation is of three significant figures only meanwhile a value of h such as $h = 0.002$ would have supposed unnecessarily long computational times. These are specially relevant with more complex or larger problems where the solution needs to be highly precise or where the number of steps needed is larger.

How accurate the solution is, does not only depend on the time-step h but also on other factors such as the initial energy E and the impact parameter b. This is because the lower the initial kinetic energy is, the lower the velocity along the x-axis as $dx/dt = \sqrt{2E}$ meaning that, with the same time-step, each step will cover less distance therefore improving the resolution. On the other hand, in this case (since the potential is centred at the origin and symmetrical with respect to the x-axis) the larger the absolute value of b, the weaker the interaction with the force field will be making the changes due to the potential smaller and therefore improving the accuracy of the solution.

As Figure 6 represents, a higher initial energy, similarly to a higher time-step, supposes a lower accuracy of the simulation. Regarding the dependence on the impact parameter, the effect of the change on b is weaker and opposite to the previous ones as a larger values are the ones that provide a more accurate solution.

The trajectories yielded by the different values of E and b computed as described in Section 3.1 can be observed in Figure 7. It is only necessary to test the positive values of b since the potential is symmetric with respect to the x-axis meaning the time step will be symmetrical $t(-b) = t(b)$ and the scattering angle anti-symmetrical $\theta(-b) = -\theta(b)$.

These trajectories follow the trends explained earlier where the particles with lower energies spend more time under the influence of the field therefore making both the scattering angle and the transit time larger. The particles with larger impact values have a weaker interaction with the field therefore making the scattering angle and hence the transit time smaller.

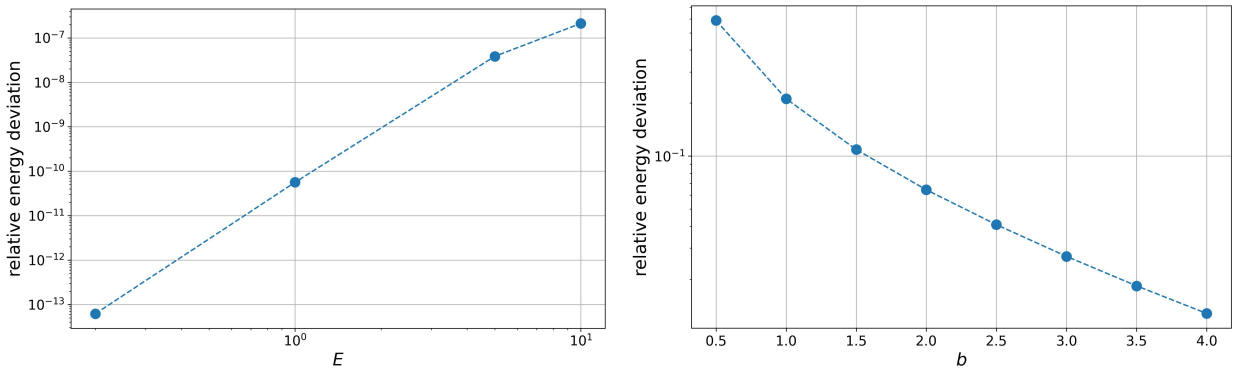


Figure 6: Demonstration of the dependence of the relative energy deviation and therefore the accuracy of the solution on the initial energy E and the impact parameter b. With the accuracy of the solution increasing with lower values of E and/or higher values of b.

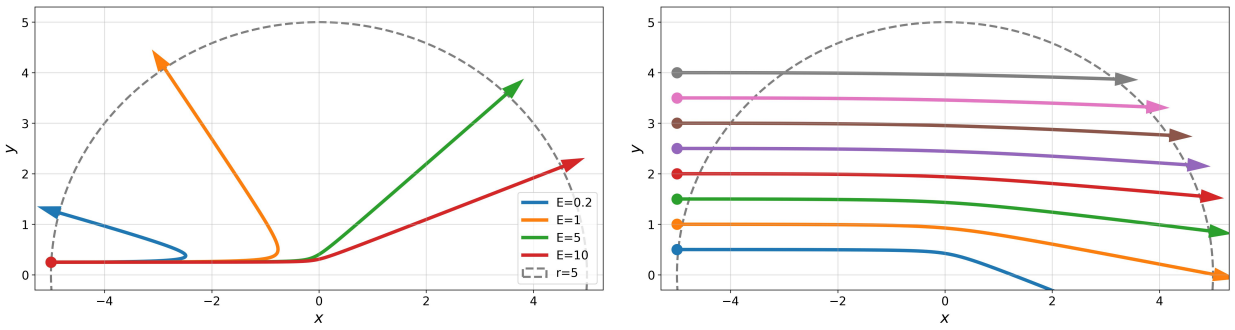


Figure 7: Trajectories of a positron in an electrostatic field where a Coulomb potential is centred at the origin. Figure on the left represents the trajectories of the particle in a repulsive field given different initial energies E. Meanwhile, figure on the right represents the trajectories of the particle in an attractive field given different impact parameters $b = y_0$.

4.2 Chaotic scattering

The asymmetrical potential $V = -y^2 \cdot e^{-(x^2+y^2)}$ can be represented as in Figure 8. This potential causes two symmetrical wells along the y-axis. This can be seen as a simplified model for the scattering of a diatomic molecule or of a star on two other ones[1].

The trajectory traced by a positron under this new potential field at a range of different b values can be seen in Figure 9. Again, due to symmetry with respect to the x-axis, it is not necessary to compute the trajectories for negative values of b. These values will also show the same trend on solution accuracy as the one shown in Figure 5.

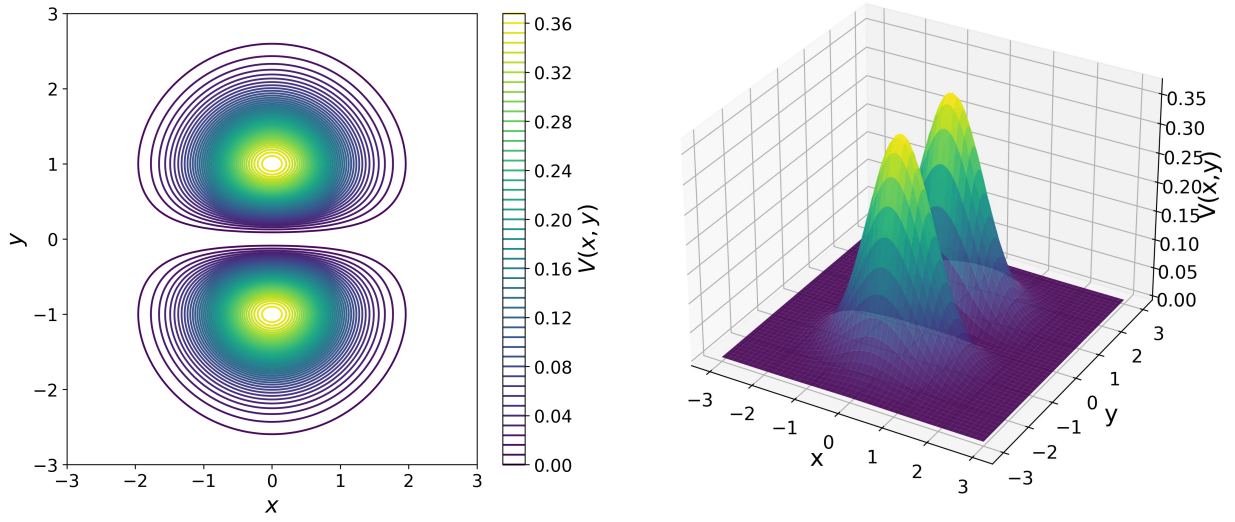


Figure 8: Contour plot and 3D surface plot of the asymmetrical potential $V = -y^2 \cdot e^{-(x^2+y^2)}$ used for the study of chaotic scattering. Plotted with `plt.contour(X, Y, Z, levels)`[4] and `plt.plot_surface(X, Y, Z)`[5].

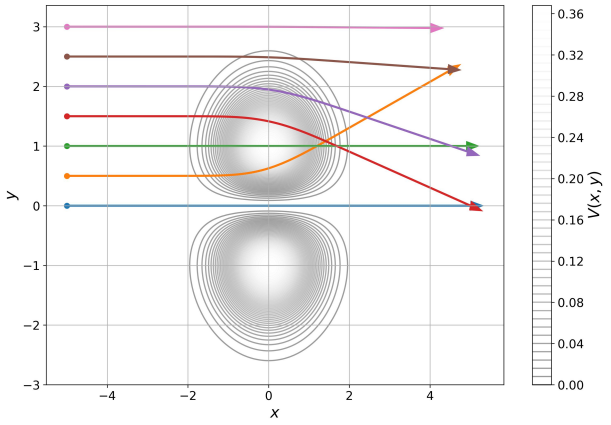


Figure 9: Trajectories of positron particles under electrostatic field $V = -y^2 \cdot e^{-(x^2+y^2)}$ for different impact parameters b.

Here, there are some interesting trajectories to point out such as the one with $b = 1.0$ which has a scattering angle of 0° since it goes through the centre of the potential well. It is also interesting to compare the $b = 0.5$ and $b = 1.5$ trajectories which, although start at the same distance from $b = 1.0$, differ in the scattering angle (21° and -17° for $b = 0.5$ and $b = 1.5$ respectively). Scattering angles and transit times of all trajectories shown in this report can be seen in the Appendix.

When different values of E and b were scanned through on the look for chaotic behaviour as explained in Section 3.2, the results shown in Table 1 were obtained.

Analysing the values obtained, the region $b = (1.7, 2.1)$ for $E = 0.01$ was decided to be studied since the max/mean rates for both the scattering angle and the transit time were both high and at relatively close b values. This resulted in Figure 10:

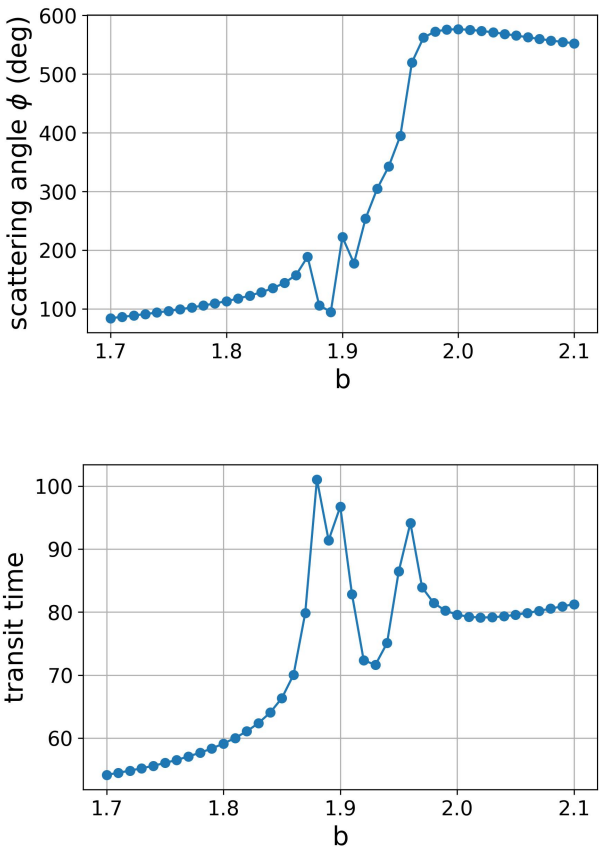


Figure 10: Plots demonstrating chaotic behaviour in scattering angle and transit time for a positron with initial energy $E = 0.01$ and impact parameters $b = [1.87, 1.90]$ computed through the RK4 method.

E	**Scattering Angle ($\Delta\phi$)**			**Transit Time (Δt_t)**		
	Max	Max/Mean	b_ϕ	Max	Max/Mean	b_{t_t}
0.05	120.34	8.26	2.10	27.36	15.13	2.17
0.02	154.03	7.23	2.17	25.30	8.69	2.17
0.01	127.73	9.65	1.89	21.21	10.00	1.87

Table 1: Summary of maximum difference on scattering angles ($\Delta\phi$) and transit times (Δt_t) at different energies (E) along with their corresponding impact parameters (b_ϕ and b_{t_t}).

It can be seen from there that chaos takes over in the $b = [1.87, 1.90]$ region. Simulating the trajectories for a positron in that region, Table 2 and Figure 11 were obtained.

b	ϕ (º)	t_t
1.87	-171.19	79.84
1.88	105.98	101.05
1.89	94.74	91.39
1.90	-137.53	96.74

Table 2: Scattering angles ϕ and transit times t_t for impact parameters 1.87 through 1.90 showing chaotic behaviour.

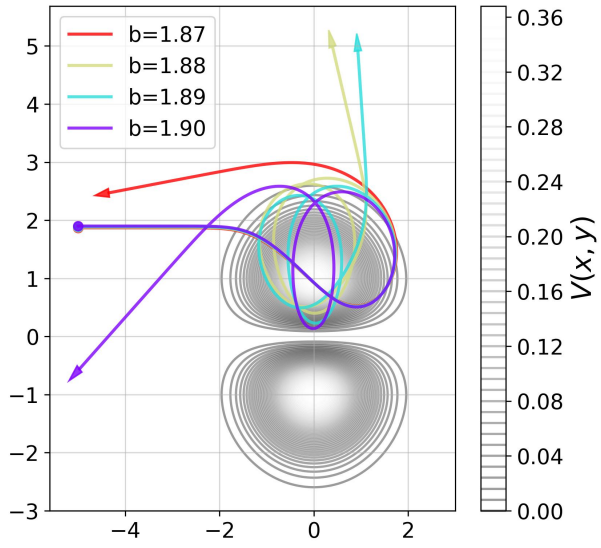


Figure 11: Chaotic trajectories of a positron in the force field of the asymmetrical potential $V = -y^2 \cdot e^{-(x^2+y^2)}$ with initial energy $E = 0.01$ and impact parameters $b = [1.87, 1.90]$ computed at a time-step $h = 0.01$.

The difference in trajectories in Figure 11 is very clearly demonstrating chaotic scattering where very small changes in initial conditions such as b result in a dramatic change on results such as ϕ. To visualize the chaos in transit time, a third dimension would be necessary. This is what is being represented in Figure 12 where the trajectories are traced in the dimensions (x, y) with time in the third dimension along the z-axis.

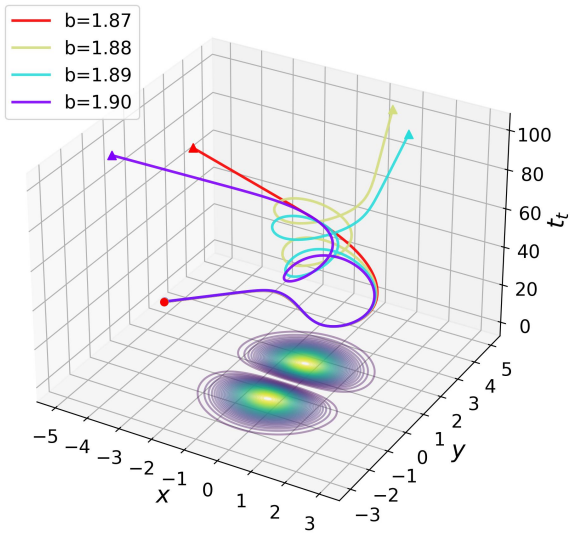


Figure 12: 3D plot of the chaotic trajectories of a positron in the force field of the asymmetrical potential $V = -y^2 \cdot e^{-(x^2+y^2)}$ with initial energy $E = 0.01$ and impact parameters $b = [1.87, 1.90]$ computed at a time-step $h = 0.01$. Plotted with `Axes3D from mpl_toolkits.mplot3d`[6].

Although harder to interpret, Figure 12 represents the chaotic behaviour both in scattering angle and transit time of a positron under said conditions. Whereas in a non-chaotic region, all 4 trajectories would be essentially overlapping in all 3 dimensions from start to finish.

5 Conclusion

With the simulations performed, it was successfully demonstrated that the fourth-order Runge-Kutta method is sufficiently accurate to model the trajectories of particles in a force field such as gravitational or electrostatic fields. It was possible to study the variance of the trajectories due to changes

in initial conditions such as the impact parameter b or the energy E of a positron in a Coulomb and an asymmetrical potential with a time-step $h = 0.01$, which ensured an accurate solution (up to seven significant figures when analysing the conservation of energy). It was also possible to identify several chaotic regions for different values of the initial energy E, modelling the trajectories of the positrons with initial parameters $E = 0.01$ and $b = [1.87, 1.90]$ under the asymmetric potential $V = -y^2 \cdot e^{-(x^2+y^2)}$ allowing to visualise and observe the impact that small changes in b causes on the particle's motion.

These programs could model several real physical experiments. The chaotic potential can represent atomic or molecular collisions in complex potential fields, resulting in chaotic dynamics. Similarly, the simulations can describe chaotic trajectories of small particle systems or the collective motion of active particles. They can also model gravitational interactions between celestial bodies, due to the similarity between gravitational and Coulomb forces, allowing simulation of satellite flybys and asteroid deflections. Further research could compare the efficiency and accuracy of numerical methods for chaotic scattering, such as the Predictor-Corrector method, adaptive Runge-Kutta algorithms via `solve_ivp`, or implicit Runge-Kutta methods like Gauss-Legendre.

References

[1] University College Dublin, School of Physics. Advanced physics laboratories: Computational lab 17 - scattering, 2025.

[2] Niels Warburton. Ordinary differential equations. ACM20030 Computational Science, University College Dublin, 2023.

[3] Numpy - Mathematical functions. `numpy.arctan2`. numpy.org.

[4] Matplotlib - Pyplot. `plt.contour()`. matplotlib.org.

[5] Matplotlib - 3D and volumetric data. `mpl_toolkits.mplot3d.Axes3D.plot_surface()`. matplotlib.org.

[6] Matplotlib Toolkit - Mplot3D - Axes3D. `mpl_toolkits.mplot3d.Axes3D()`. matplotlib.org.

Appendix

Table 3: ϕ and t_t values for 7(a).

E	t_t	ϕ (º)
0.2	12.82	162.92
1.0	7.50	122.32
5.0	3.27	42.78
10.0	2.29	22.37

Table 4: ϕ and t_t values for 7(b).

b	t_t	ϕ (º)
0.5	3.10	-22.96
1.0	3.12	-11.42
1.5	3.10	-7.44
2.0	3.06	-5.38
2.5	2.99	-4.11
3.0	2.89	-3.23
3.5	2.76	-2.56
4.0	2.58	-2.02

Table 5: ϕ and t_t values for 9.

b	t_t	ϕ (º)
0.0	7.09	0.00
0.5	6.76	21.00
1.0	6.82	0.00
1.5	7.07	-17.00
2.0	7.05	-13.00
2.5	6.69	-2.70
3.0	6.39	-0.30

Runge-Kutta methods and higher-order ODEs

Runge-Kutta methods are a broad class of useful ODE solvers. In this notebook we look at a few of them, their convergence rates, and how to apply them to second-order equations

```
import numpy as np
import matplotlib.pyplot as plt

# The below commands make the font and image size bigger
plt.rcParams.update({'font.size': 22})
plt.rcParams["figure.figsize"] = (15,10)
```

We will define an `ODESolve` function that can use different methods. First let's define the stepping functions for the Euler, RK2 and RK4 methods.

```
def EulerStep(f, dx, xi, yi):
    return yi + dx*f(xi, yi)

def RK2Step(f, dx, xi, yi):
    k1 = dx*f(xi, yi)
    k2 = dx*f(xi + dx, yi + k1)

    return yi + 0.5*(k1 + k2)

def RK4Step(f, dx, xi, yi):
        k1 = dx*f(xi,yi)
        k2 = dx*f(xi + 0.5*dx, yi + 0.5*k1)
        k3 = dx*f(xi + 0.5*dx, yi + 0.5*k2)
        k4 = dx*f(xi + dx, yi + k3)

        return yi + 1/6*(k1 + 2*k2 + 2*k3 + k4)
```

The method in the below function can be set using the optional 6th argument.

```
def ODESolve(f, dx, x0, y0, imax, method='RK4', plotSteps=False):

    xi = x0
    yi = y0

    # Create arrays to store the steps in
    steps = np.zeros((imax+1,2))
    steps[0,0] = x0
    steps[0,1] = y0

    stepper = RK4Step
    if(method == 'RK2'):
        stepper = RK2Step
```

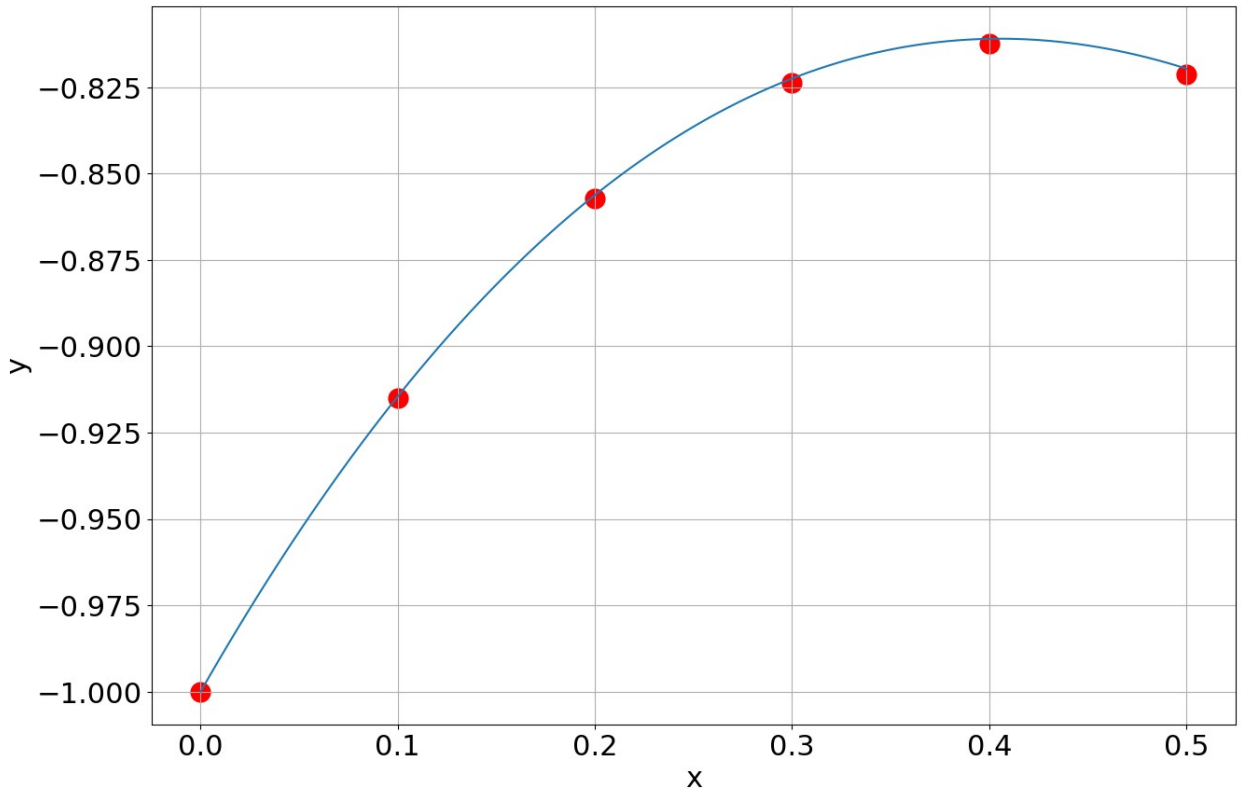
```
    elif(method == 'Euler'):
        stepper = EulerStep

    i = 0
    while i < imax:
        yi = stepper(f, dx, xi, yi)

        xi += dx
        i  += 1

        # Store the steps for plotting
        steps[i, 0] = xi
        steps[i, 1] = yi

    if(plotSteps):
        plt.scatter(steps[:,0], steps[:,1], color='red', linewidth=10)

    return [xi, yi]

def dydx(x,y):
    return -2*x - y

def yExact(x):
    return - 3*np.exp(-x) - 2*x + 2
```

Convergence of the methods

Let's look at the rate of convergence of the three methods: Euler, RK2 and RK4

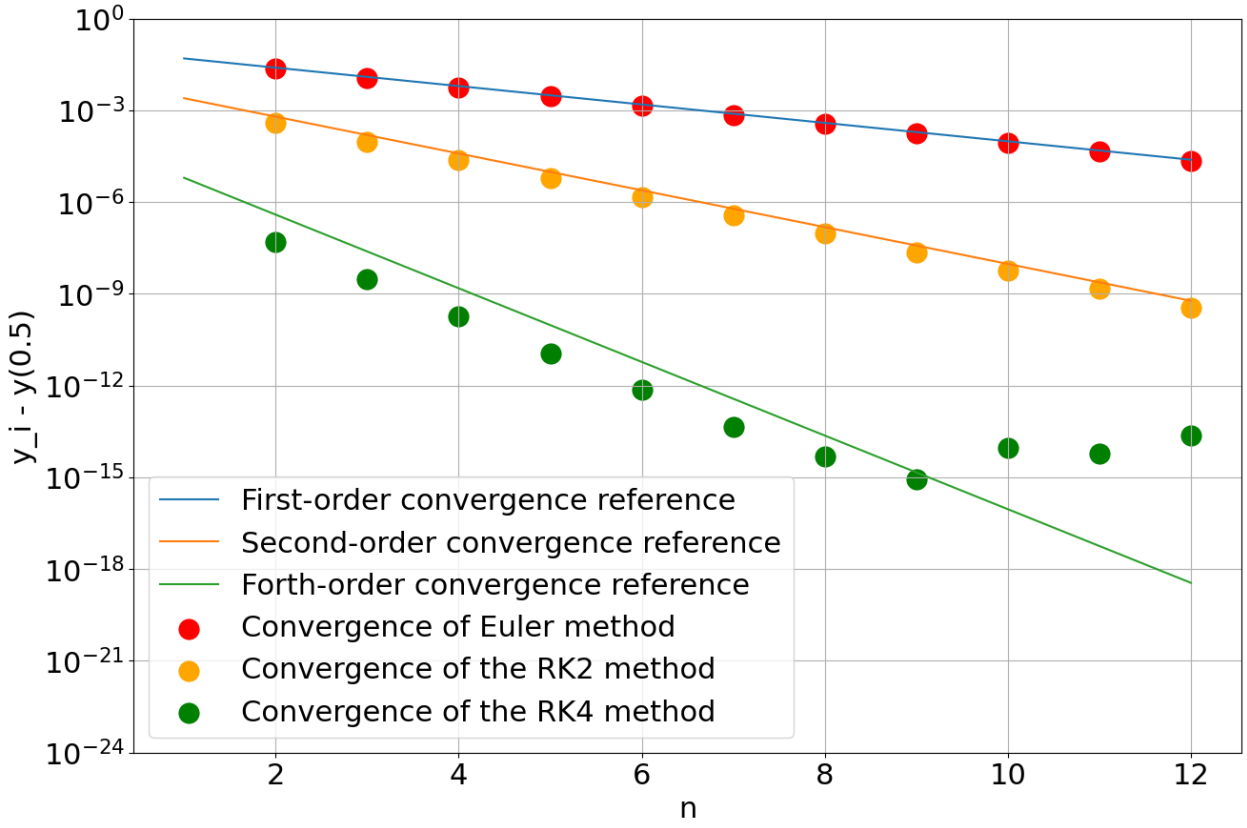
```
x = np.linspace(0, 0.5, 100)
y = yExact(x)
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.plot(x,y)

ODESolve(dydx, 0.1, 0, -1, 5, 'RK2', True)
```

```
[0.5, -0.821227295946875]
```




```
nmax = 12

diffEuler = np.zeros(nmax)
diffRK2 = np.zeros(nmax)
diffRK4 = np.zeros(nmax)

n = 1
while n < nmax:
    deltax       = 0.1/2**n
    nsteps       = 5*2**n
    resEuler     = ODESolve(dydx, deltax, 0, -1, nsteps, 'Euler')
    resRK2       = ODESolve(dydx, deltax, 0, -1, nsteps, 'RK2')
    resRK4       = ODESolve(dydx, deltax, 0, -1, nsteps, 'RK4')

    diffEuler[n] = np.abs(resEuler[1] - yExact(0.5))
    diffRK2[n]   = np.abs(resRK2[1] - yExact(0.5))
    diffRK4[n]   = np.abs(resRK4[1] - yExact(0.5))
    n += 1

# Plot the results
plt.grid(True)
plt.yscale('log')
plt.xlabel('n')
plt.ylabel('y_i - y(0.5)')
plt.ylim([1e-24, 1])
```

```
# Compute and plot reference curves for the convergence rate
x           = np.linspace(1, nmax, 12)
deltax      = (0.1/2**x)
firstOrder  = deltax**1
secondOrder = deltax**2
forthOrder  = deltax**4

plt.plot(x, firstOrder)
plt.plot(x, secondOrder)
plt.plot(x, forthOrder)

plt.scatter(np.arange(1,nmax+1), diffEuler, color='red', linewidth=10)
plt.scatter(np.arange(1,nmax+1), diffRK2, color='orange',
linewidth=10)
plt.scatter(np.arange(1,nmax+1), diffRK4, color='green', linewidth=10)

plt.legend(['First-order convergence reference',
            'Second-order convergence reference',
            'Forth-order convergence reference',
            'Convergence of Euler method',
            'Convergence of the RK2 method',
            'Convergence of the RK4 method'
           ]);
```



Thus we see that the RK4 method is rapidly convergent. It does require 4 evaluations of the right-hand side of the equations. The RK4 method has a good balance between between taking the least number of steps and achieving the highest accuracy.

Second-order ODEs

As we discussed in the lectures we can write any an n^{th}-order ODE as a coupled system of n first-order ODEs. Let's look at implementing it in practice.

Let's consider a second-order ODE. We want to write this in the form:

$$\begin{aligned} y_0'(x) &= f_0(x, y_0, y_1) \\ y_1'(x) &= f_1(x, y_0, y_1) \end{aligned}$$

We thus want to write a function that when passed x and an array $\mathrm{y} = [y_0, y_1]$ returns an array $\mathrm{f} = [f_0, f_1]$. Let's look at a specific example. Consider $y''(x) = -y$ with $y(0) = 1$. This has the analytic solution $y(x) = \cos(x)$. Let's write this in first-order form.

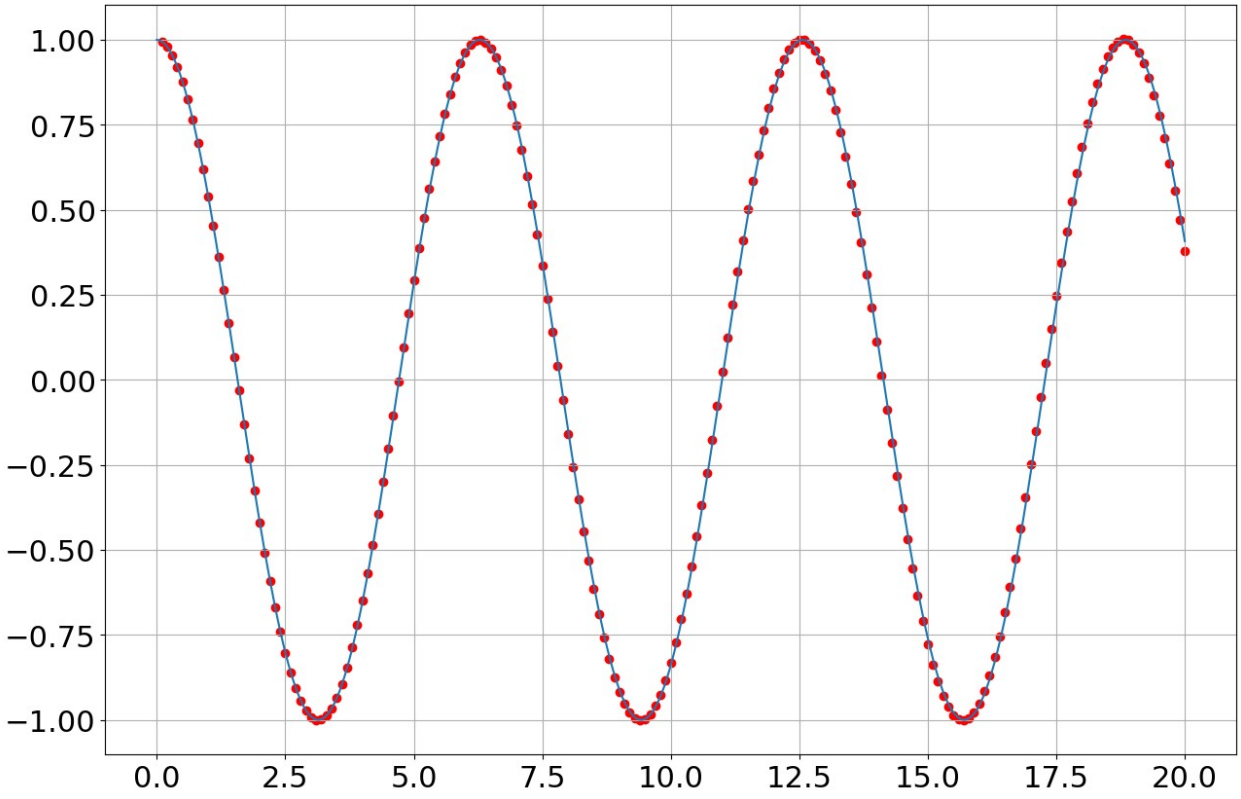
Let $y_0(x) = y(x)$ and $y_1 = y_0'(x)$. Then we have $y_1' = -y_0$. Thus

$$\begin{aligned} y_0'(x) &= y_1 \\ y_1'(x) &= -y_0 \end{aligned}$$

so $\mathrm{f} = [y_1, -y_0]$. Let's define a Python function for this.

```
def f2(x, y):
    return np.array([y[1], -y[0]])

def SecondOrderRK2(f, dx, x0, y0, imax):
    output = np.empty((imax, 3))
    i = 0
    xi = x0
    yi = y0
    while(i < imax):
        k1 = dx*f(xi, yi)
        k2 = dx*f(xi + dx, yi + k1)
        yi = yi + 0.5*(k1 + k2)
        xi += dx
        output[i, 0] = xi
        output[i, 1] = yi[0]
        output[i, 2] = yi[1]
        i += 1
    return output

res = SecondOrderRK2(f2, 0.1, 0, [1,0], 200);

x = np.linspace(0,20,400)
y = np.cos(x)
```

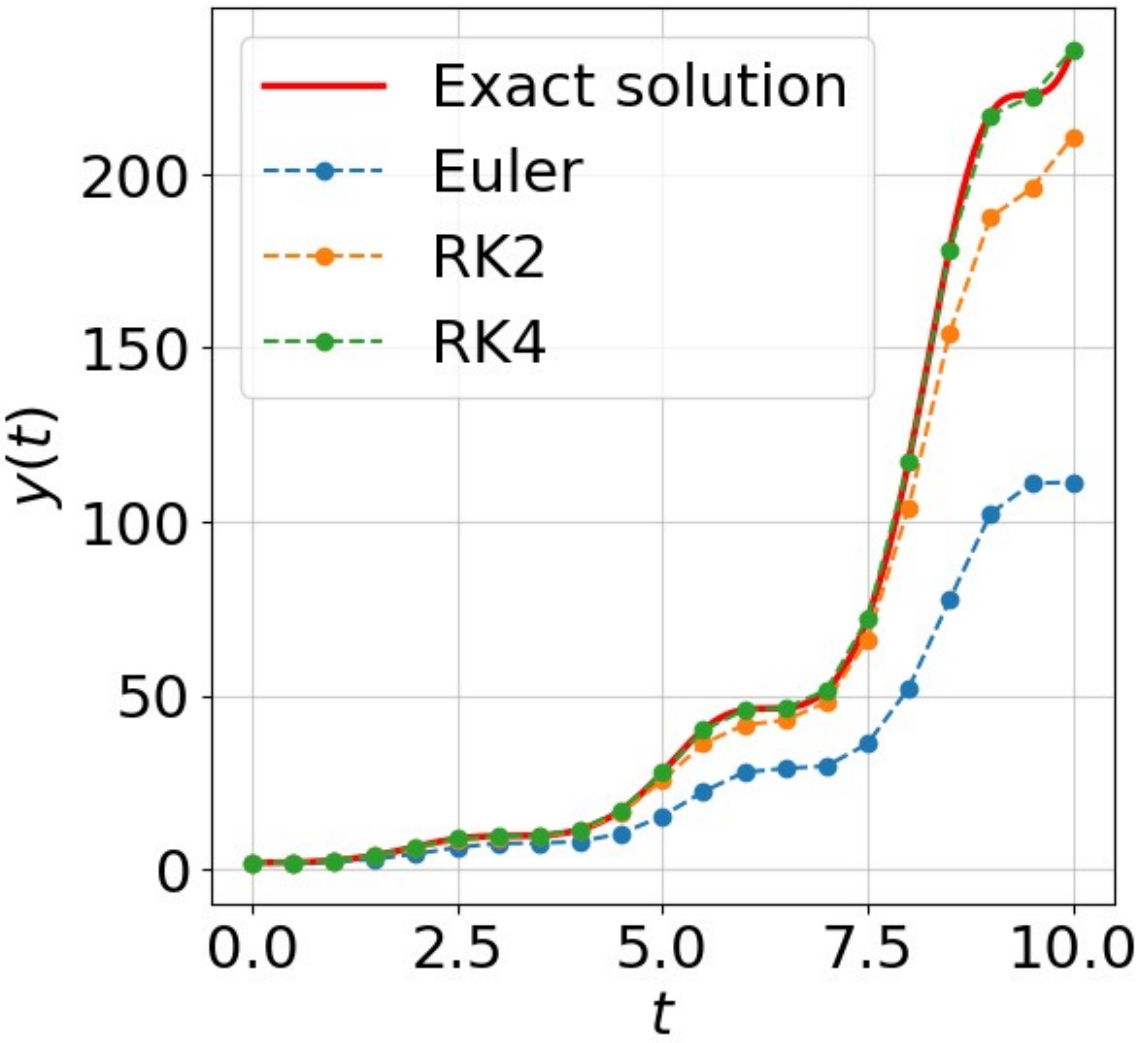
```
plt.grid(True)
plt.scatter(res[:,0], res[:,1], color='r')
plt.plot(x, y);
```



Paths of the methods

```
def dydt(t, y):
    return (np.sin(t))**2 * y

def y_exact(t, y0=2):
    return y0 * np.exp((t/2) - (np.sin(2*t)/4))

h = 0.5   # large time-steps show trend better
t_values = np.arange(0, 10.5, h)

y_euler, y_rk2, y_rk4 = [2], [2], [2]   # Populate with y0=2

for t in t_values[:-1]:
    y_euler.append(EulerStep(dydt, h, t, y_euler[-1]))
    y_rk2.append(RK2Step(dydt, h, t, y_rk2[-1]))
    y_rk4.append(RK4Step(dydt, h, t, y_rk4[-1]))

t_line = np.linspace(0, 10, 500)
y_exact_values = y_exact(t_line)
```

```
plt.figure(figsize=(6, 6))
plt.plot(t_line, y_exact_values, 'r-', linewidth=2.5, label='Exact
solution')
plt.plot(t_values, y_euler, 'o--', label='Euler')
plt.plot(t_values, y_rk2, 'o--', label='RK2')
plt.plot(t_values, y_rk4, 'o--', label='RK4')

plt.xlabel('$t$')
plt.ylabel('$y(t)$')
plt.grid(alpha=0.6)
plt.legend()
plt.savefig("method_comparison.png", dpi=300, bbox_inches='tight')
plt.show()
```



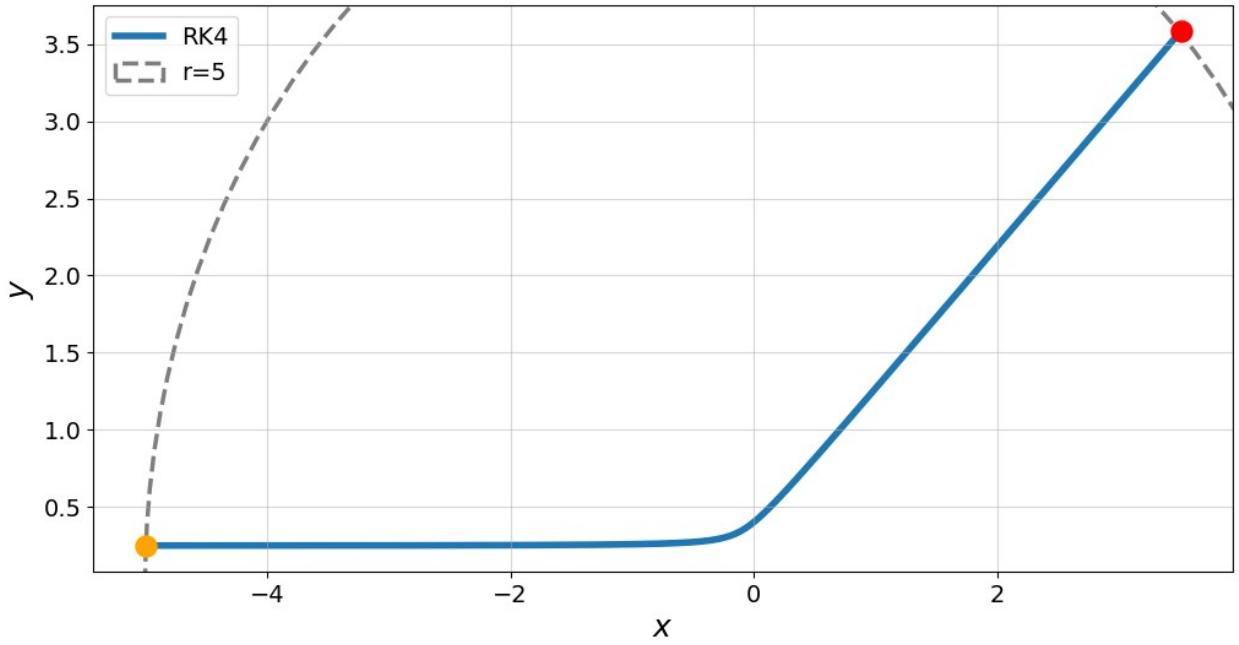
Computational Scattering

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.patches import Circle
from mpl_toolkits.mplot3d import Axes3D

# Size plots & labels
plt.rcParams['figure.figsize'] = (12,6)
plt.rcParams['axes.labelsize'] = 18
plt.rcParams['xtick.labelsize'] = 14
plt.rcParams['ytick.labelsize'] = 14
plt.rcParams['axes.titlesize'] = 14
plt.rcParams['legend.fontsize'] = 14

# Calculation of the acceleration due to the Coloumb potential - eq.
(5) & (2)
def Coloumb_acceleration(x,y):
    r3 = (x**2 + y**2)**(3/2)
    return np.array([x/r3, y/r3])

# Calculation of the acceleration due to the chaotic potential - eq.
(8)
def chaotic_acceleration(x,y):
    dVdx = 2*x * y**2 * np.exp(-x**2 - y**2)
    dVdy = (-2*y + 2*y**3) * np.exp(-x**2 - y**2)
    return np.array([-dVdx, -dVdy])

# Calculation of the next step through RK4
def RK4_step(state_n, h, acceleration, repulsive=True):
    def k(params):
        x, y, vx, vy = params
        ax, ay = acceleration(x,y) if repulsive else -
acceleration(x,y)   # else, attractive potential i.e. attractive
acceleration
        return np.array([vx, vy, ax, ay])
    k1 = k(state_n)   # state_n = r_n, v_n when wanting to calculate
r_(n+1), v_(n+1)
    k2 = k(state_n + k1*h/2)
    k3 = k(state_n + k2*h/2)
    k4 = k(state_n + k3*h)
    return state_n + h/6 * (k1 + 2*k2 + 2*k3 + k4)   # returns
state_(n+1) i.e. r_(n+1), v_(n+1)

# Obtain all the states at each step through RK4
def RK4(state, acceleration, rmax, obtain_energies=False,
repulsive=True):
    data = [state.copy()]
```

```
    if obtain_energies: energies = [total_energy(state)]

    inside = False   # Only start counting once inside the circle of
r=rmax
    while True:
        r = np.sqrt(state[0]**2 + state[1]**2)
        if r <= rmax: inside = True
        elif inside: break   # particle has exited the circle

        state = RK4_step(state, h, acceleration, repulsive)
        data.append(state.copy())
        if obtain_energies: energies.append(total_energy(state))
    return np.array(data) if not obtain_energies else (np.array(data),
energies)

# Calculate the total energy
def total_energy(state, repulsive=True):
    x, y, vx, vy = state
    r = np.sqrt(x**2 + y**2)
    V = 1/r if repulsive else -1/r   # else, attractive potential
    K = 1/2 * (vx**2 + vy**2)
    return K + V
```

1. Positron-positron scattering

```
# Set initial conditions
E, b, h = 5, 0.25, 0.01
state = np.array([-5, b, np.sqrt(2*E), 0])

# Initialize and record the data using RK4
data = RK4(state, Coloumb_acceleration, rmax=5)

# Plot
plt.plot(data[:,0], data[:,1], zorder=0, linewidth=4, label='RK4')
plt.scatter(data[0,0], data[0,1], color='orange', s=150)
plt.scatter(data[-1,0], data[-1,1], color='r', s=150)
circle = plt.Circle((0,0), 5, fill=False, color='k', linestyle='--',
linewidth=2.5, alpha=0.5, label='r=5', zorder=0)
plt.gca().add_artist(circle)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend()
plt.grid(alpha=0.5)
plt.savefig("positron_scattering.jpg", dpi=300, bbox_inches='tight')
plt.show()
```



a) Transit time and scattering angle of the particle

The transit time is the time step times the number of steps taken meanwhile the angle of scattering is the angle between the direction of the final velocity and the direction of the initial velocity (which was all along its x component).

```
transit_time = len(data) * h
scattering_angle = np.arctan2(data[-1,3], data[-1,2])   # np.arctan() does element-wise, np.arctan2() not

print(f"The transit time is {transit_time:.4g} dimensionless units of time.\nThe scattering angle is {scattering_angle:.4g} rad, {np.degrees(scattering_angle):.4g}º")
```

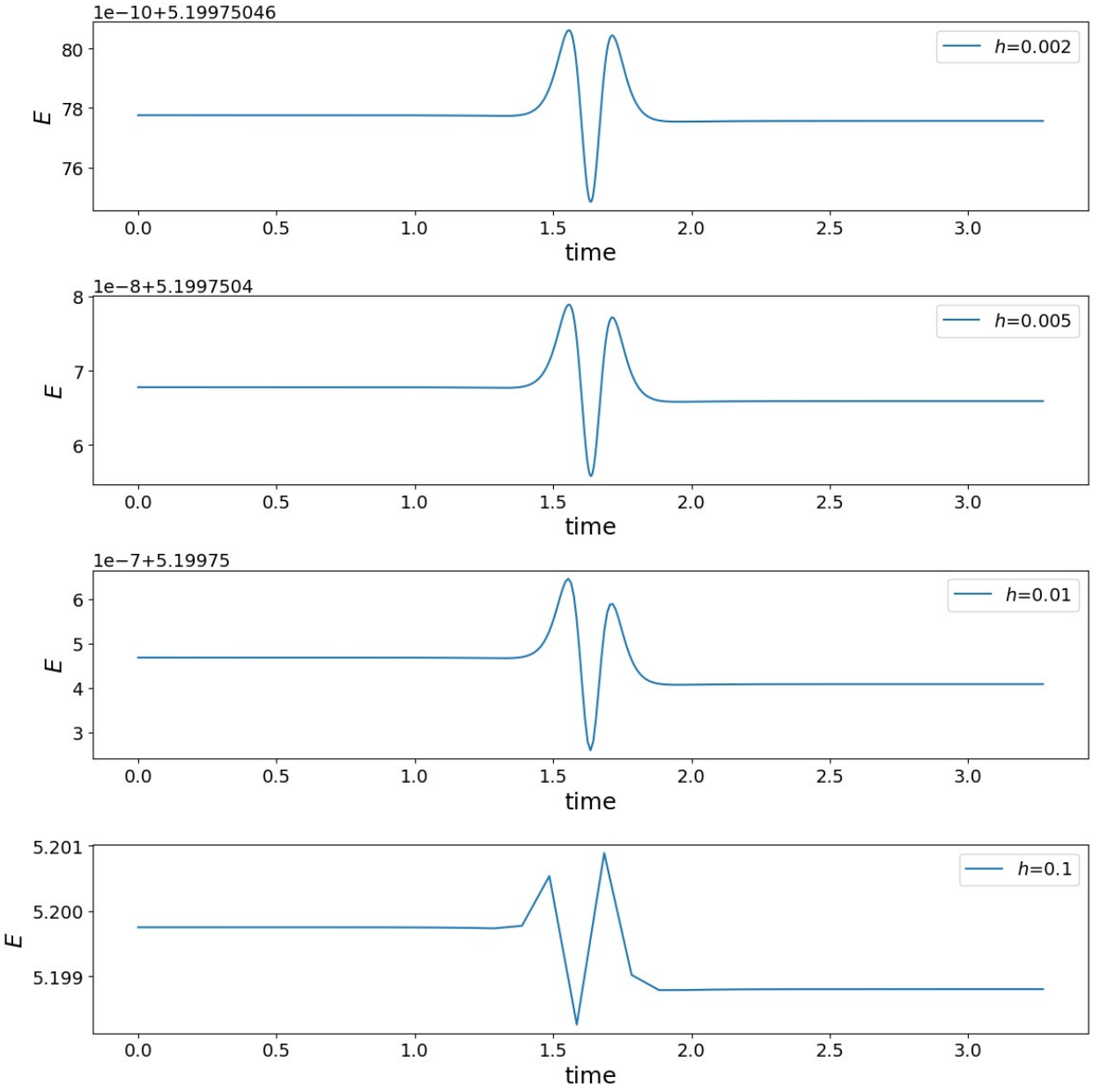
```
The transit time is 3.27 dimensionless units of time.
The scattering angle is 0.7466 rad, 42.78º
```

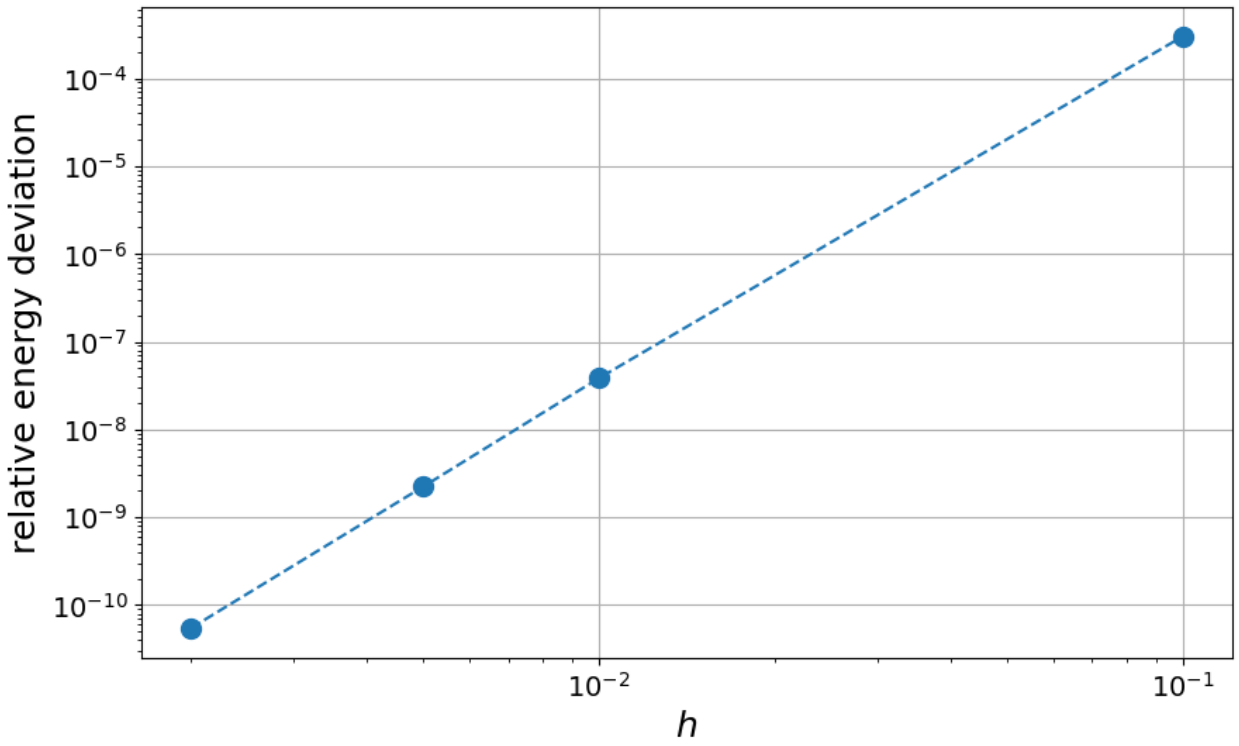
b) How the accuracy of the solution improves as the time step is reduced

With a lower time step the accuracy improves but it also causes the computation time to increase. Although in this example computation time is not an issue due to the small size of the region studied, a timestep of $h=\Delta t=0.01$ is a great choice as it has a maximum deviation on the energies of -7 orders of magnitude with respect to the energy itself while keeping a low computational time.

```
# Set initial conditions
E, b = 5, 0.25
state = np.array([-5, b, np.sqrt(2*E), 0])
```

```
# Iterate through different values of h obtaining the energies
h_values = [0.002, 0.005, 0.01, 0.1]
fig, ax = plt.subplots(len(h_values), 1, figsize=(12, 12))
energy_errors = []

for i, h in enumerate(h_values):
    data, energies = RK4(state, Coloumb_acceleration, rmax=5,
obtain_energies=True)
    energy_errors.append( (max(energies)-
np.mean(energies))/energies[0] )
    times = np.linspace(0, transit_time, len(data))
    ax[i].plot(times, energies, label=f'$h$={h}')
    ax[i].legend()
    ax[i].set_xlabel('time')
    ax[i].set_ylabel('$E$')

plt.tight_layout()
plt.show()

plt.figure(figsize=(10,6))
plt.loglog(h_values, energy_errors, 'o--', ms=10)
plt.xlabel('$h$')
plt.ylabel('relative energy deviation')
plt.grid()
plt.savefig("accuracy_solution.jpg", dpi=300, bbox_inches='tight')
plt.show()
```


1e−10+5.19975046
80
78
76
E
h=0.002
0.0
0.5
1.0
1.5
2.0
2.5
3.0
time
1e−8+5.1997504
8
7
6
E
h=0.005
0.0
0.5
1.0
1.5
2.0
2.5
3.0
time
1e−7+5.19975
6
5
4
3
E
h=0.01
0.0
0.5
1.0
1.5
2.0
2.5
3.0
time
5.201
5.200
5.199
E
h=0.1
0.0
0.5
1.0
1.5
2.0
2.5
3.0
time



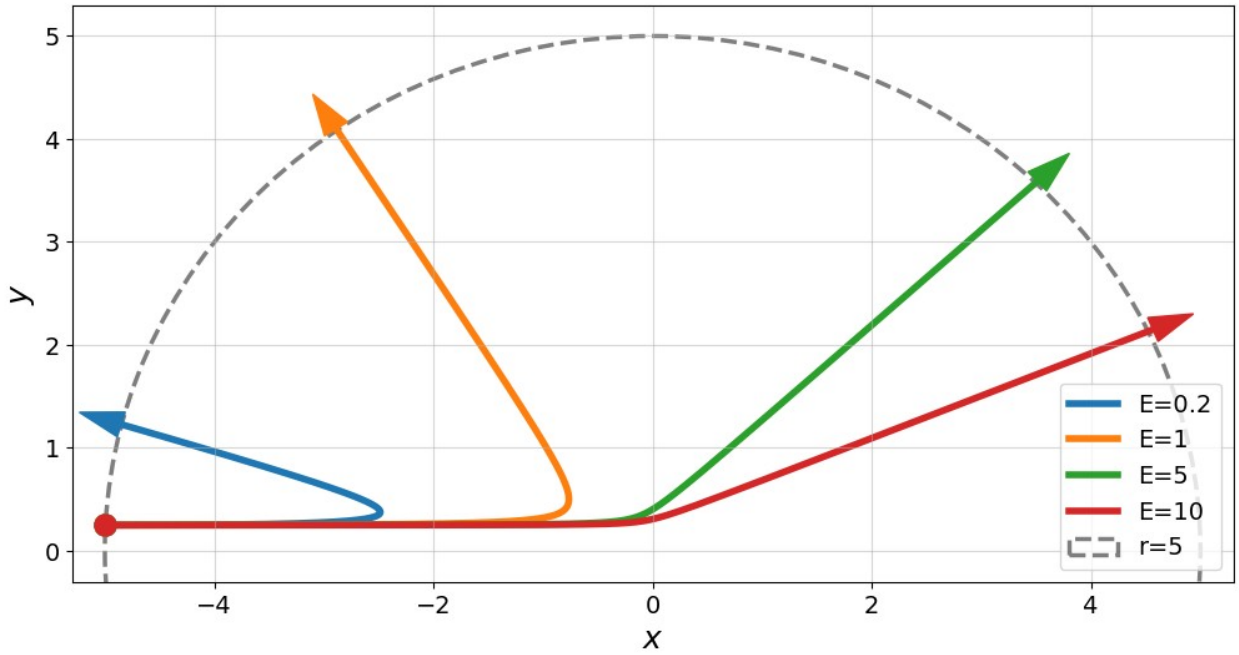
c) What happens as the initial E is reduced

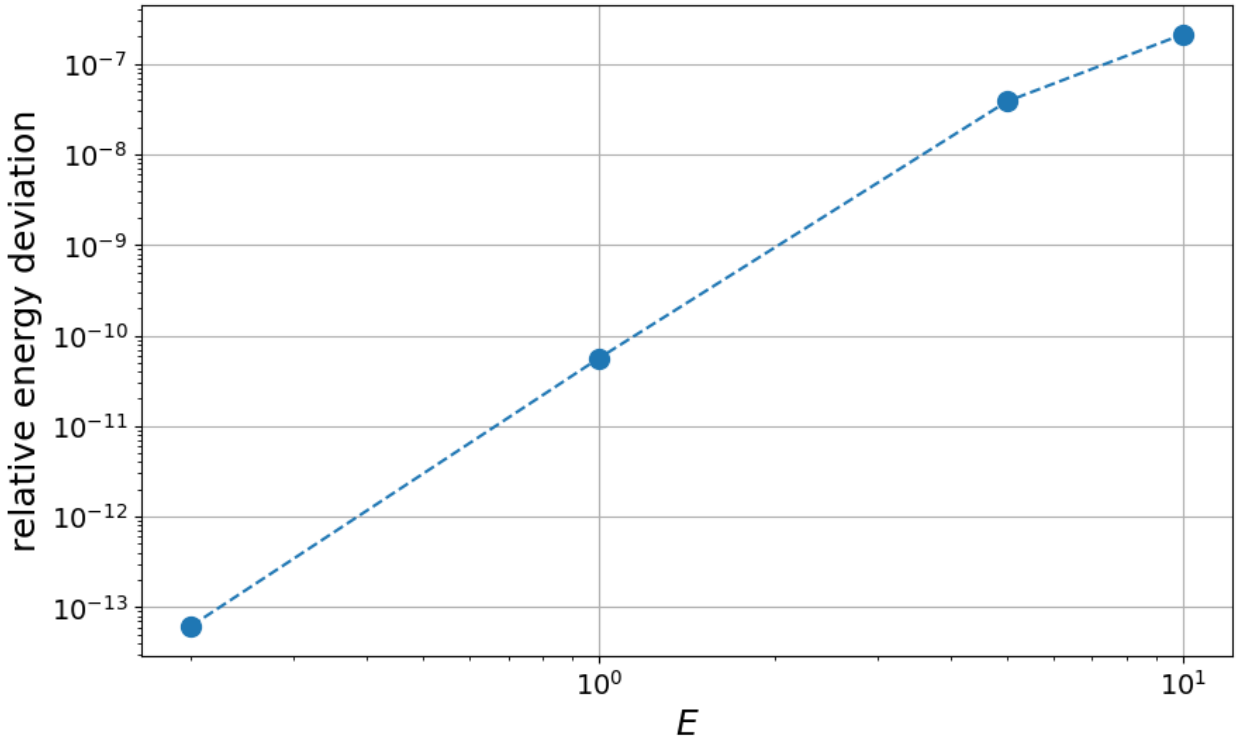
Lower E means lower v_x and therefore a lower initial velocity. This causes an increase in the transition time also causing an increase in scattering angle as the particle spends a longer time under the potential field. A decreased E also improves the accuracy of the solution significantly since the initial speed is lower and each step cover a lower length of the path the paricle has gone through.

```
plt.figure(figsize=(12,6))
# Set initial conditions
b, h = 0.25, 0.01

# Iterate through different values of E
E_values = [0.2, 1, 5, 10]
energy_errors = []
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
[:len(E_values)]

for E, color in zip(E_values, colors):
    state = np.array([-5, b, np.sqrt(2*E), 0])

    # Initialize and record the data using RK4
    data, energies = RK4(state, Coloumb_acceleration, rmax=5,
obtain_energies=True)
    energy_errors.append( (max(energies)-
np.mean(energies))/energies[0] )
```

```
    # Calculate the transit time and the scattering angle
    transit_time = len(data) * h
    scattering_angle = np.arctan2(data[-1,3], data[-1,2])
    print(f"E={E}: transit time={transit_time:.2f} dimensionless units
of time\n      scattering angle={np.degrees(scattering_angle):.2f}º\
n")

    # Plot scattering
    plt.plot(data[:,0], data[:,1], color=color, zorder=1, linewidth=4,
label=f'E={E}')
    plt.scatter(data[0,0], data[0,1], color=color, s=150)
    plt.arrow(data[-1,0], data[-1,1], 0.001*data[-1,2], 0.001*data[-
1,3], head_width=0.25, head_length=0.4, color=color)

circle = plt.Circle((0,0), 5, fill=False, color='k', linestyle='--',
linewidth=2.5, alpha=0.5, zorder=0, label='r=5')
plt.gca().add_artist(circle)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.xlim(-5.3,5.3)
plt.ylim(-0.3,5.3)
plt.legend(loc='lower right')
plt.grid(alpha=0.5)
plt.savefig("E_values.jpg", dpi=300, bbox_inches='tight')
plt.show()

# Plot the accuracy of the solution
plt.figure(figsize=(10,6))
plt.loglog(E_values, energy_errors, 'o--', ms=10)
plt.xlabel('$E$')
plt.ylabel('relative energy deviation')
plt.grid()
plt.savefig("accuracy_solution_E.jpg", dpi=300, bbox_inches='tight')
plt.show()
```

```
E=0.2: transit time=12.82 dimensionless units of time
      scattering angle=162.92º

E=1: transit time=7.50 dimensionless units of time
      scattering angle=122.32º

E=5: transit time=3.27 dimensionless units of time
      scattering angle=42.78º

E=10: transit time=2.29 dimensionless units of time
      scattering angle=22.37º
```





d) How the impact parameter b affects the transit time and the scattering angle on an attractive potential

A higher value of b means the particle starts further away from the origin of the potential therefore being less affected by it (since $V = -1/r$) and having a smaller deviation and smaller absolute value of the scattering angle. It also means the accuracy of the solution is improved

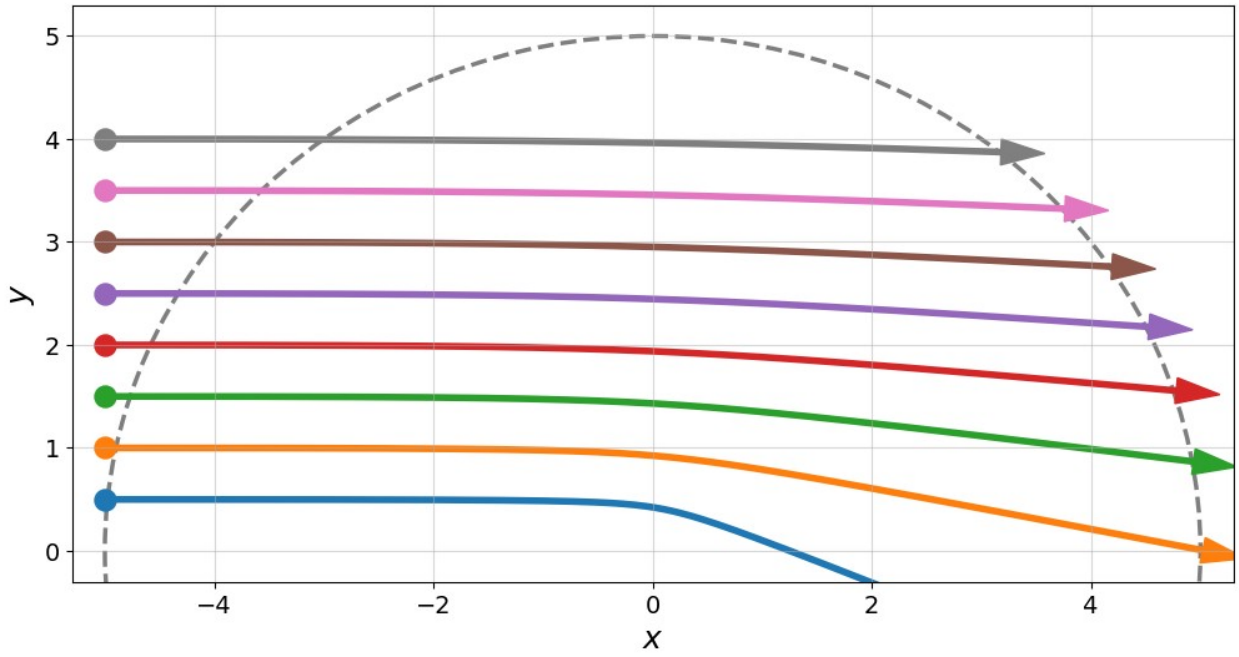
although this is mostly affected by the time step and the length covered in each step (i.e. the initial velocity, dependeant on E).

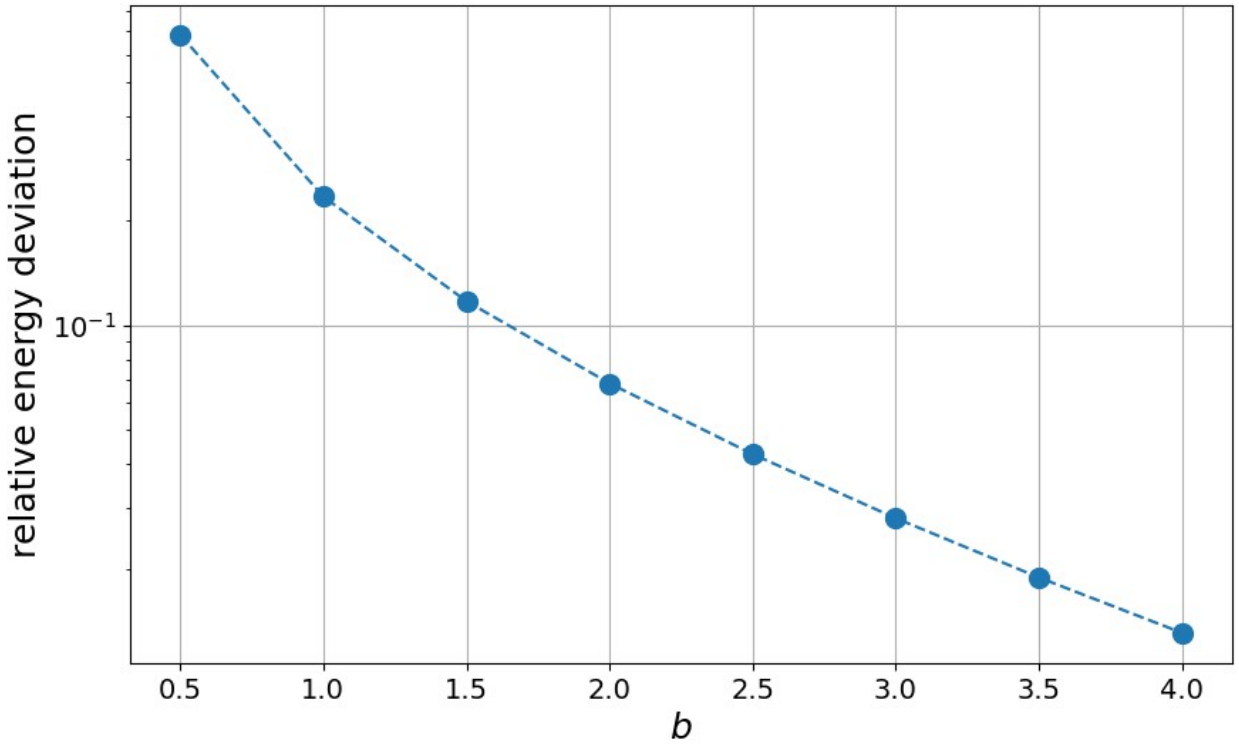
It is only necessary to test the positive values of b since the potential is symmetric with respect to the x-axis meaning the time step will be symmetrical $t(-b)=t(b)$ and the scattering angle will be antisymmetrical $\theta(-b)=-\theta(b)$.

```
plt.figure(figsize=(12,6))
# Set initial conditions
E, h = 5, 0.01

# Iterate through different values of b
b_values = np.arange(0.5, 4.5, 0.5)
energy_errors = []
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
[:len(b_values)]

for b, color in zip(b_values, colors):
    state = np.array([-5, b, np.sqrt(2*E), 0])

    # Initialize and record the data using RK4
    data, energies = RK4(state, Coloumb_acceleration, rmax=5,
obtain_energies=True, repulsive=False)
    energy_errors.append( (max(energies)-
np.mean(energies))/energies[0] )

    # Calculate transit time and scatttering angle
    transit_time = len(data) * h
    scattering_angle = np.arctan2(data[-1,3], data[-1,2])   #
np.arctan() does element-wise, np.arctan2() not
    print(f"b={b}: transit time={transit_time:.2f} dimensionless units
of time\n     scattering angle={np.degrees(scattering_angle):.2f}º\n")

    # Plot scattering
    plt.plot(data[:,0], data[:,1], color=color, zorder=1, linewidth=4,
label=f'b={b}')
    plt.scatter(data[0,0], data[0,1], color=color, s=150)
    plt.arrow(data[-1,0], data[-1,1], 0.001*data[-1,2], 0.001*data[-
1,3], head_width=0.25, head_length=0.4, color=color)

circle = plt.Circle((0,0), 5, fill=False, color='k', linestyle='--',
linewidth=2.5, alpha=0.5, zorder=0, label='r=5')
plt.gca().add_artist(circle)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.xlim(-5.3,5.3)
plt.ylim(-0.3,5.3)
#plt.legend(loc='lower left')
plt.grid(alpha=0.5)
plt.savefig("b_values.jpg", dpi=300, bbox_inches='tight')
```

```
plt.show()

# Plot the accuracy of the solution
plt.figure(figsize=(10,6))
plt.semilogy(b_values, energy_errors, 'o--', ms=10)
plt.xlabel('$b$')
plt.ylabel('relative energy deviation')
plt.grid()
plt.savefig("accuracy_solution_b.jpg", dpi=300, bbox_inches='tight')
plt.show()
```

```
b=0.5: transit time=3.10 dimensionless units of time
     scattering angle=-22.96º

b=1.0: transit time=3.12 dimensionless units of time
     scattering angle=-11.42º

b=1.5: transit time=3.10 dimensionless units of time
     scattering angle=-7.44º

b=2.0: transit time=3.06 dimensionless units of time
     scattering angle=-5.38º

b=2.5: transit time=2.99 dimensionless units of time
     scattering angle=-4.11º

b=3.0: transit time=2.89 dimensionless units of time
     scattering angle=-3.23º

b=3.5: transit time=2.76 dimensionless units of time
     scattering angle=-2.56º

b=4.0: transit time=2.58 dimensionless units of time
     scattering angle=-2.02º
```

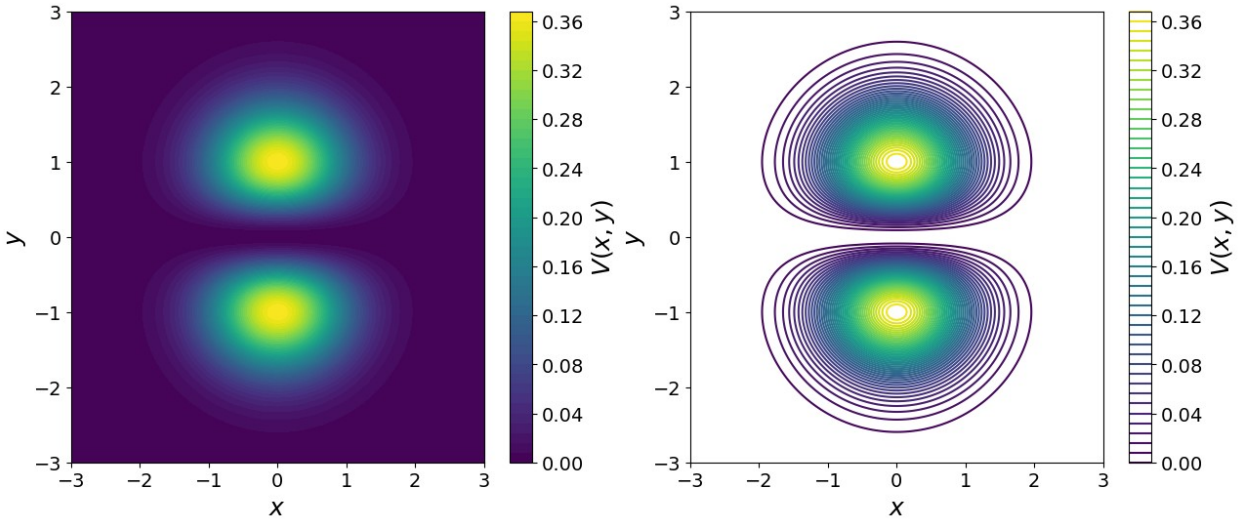


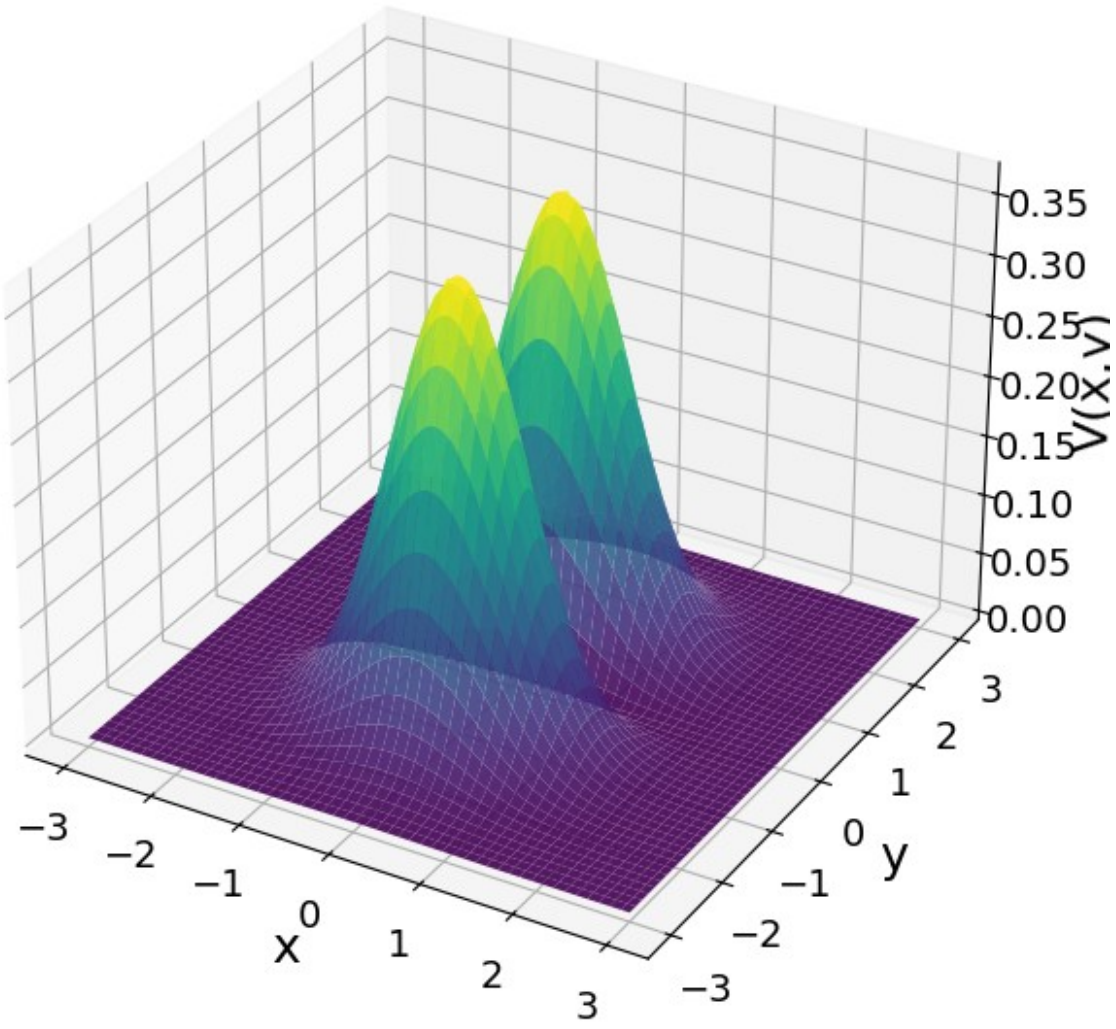


2. Chaotic Scattering

```
def chaotic_potential(x, y):
    return y**2 * np.exp(-x**2 - y**2)

x = np.linspace(-3, 3, 200)
```

```
y = np.linspace(-3, 3, 200)
[X, Y] = np.meshgrid(x, y)
Z = chaotic_potential(X, Y)

fig, ax = plt.subplots(1, 2, figsize=(15, 6))

# Filled contour
contourf = ax[0].contourf(X, Y, Z, levels=50)
fig.colorbar(contourf, label='$V(x,y)$', ax=ax[0])
ax[0].set_xlabel('$x$')
ax[0].set_ylabel('$y$')

# Non-filled contour
contour= ax[1].contour(X, Y, Z, levels=50)
fig.colorbar(contour, label='$V(x,y)$', ax=ax[1])
ax[1].set_xlabel('$x$')
ax[1].set_ylabel('$y$')

plt.show()

# 3D plot
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.9)   # Surface is for
3D what contour map is for 2D
ax.set_xlabel('x'); ax.set_ylabel('y'); ax.set_zlabel('V(x,y)')
plt.tight_layout()
plt.savefig("chaotic_surface.jpg", dpi=300, bbox_inches='tight',
pad_inches=0.3)
plt.show()
```





a) Attractive chaotic potential with $E=1$

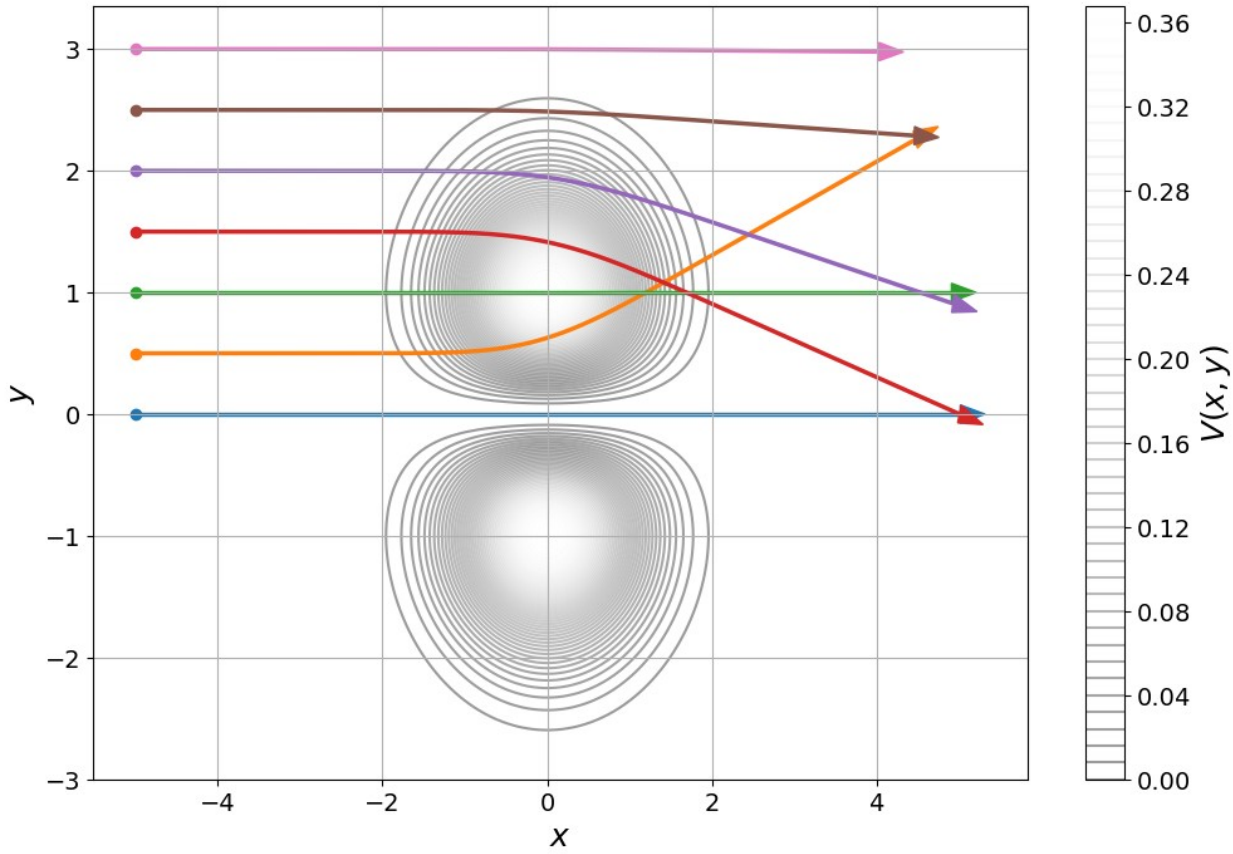
Due to the chaotic potential with two wells, the change in b doesn't suppose a constant increase or decrease in transit time and scattering angle anymore. However, the way the parameter b affects the accuracy of the solution remains the same.

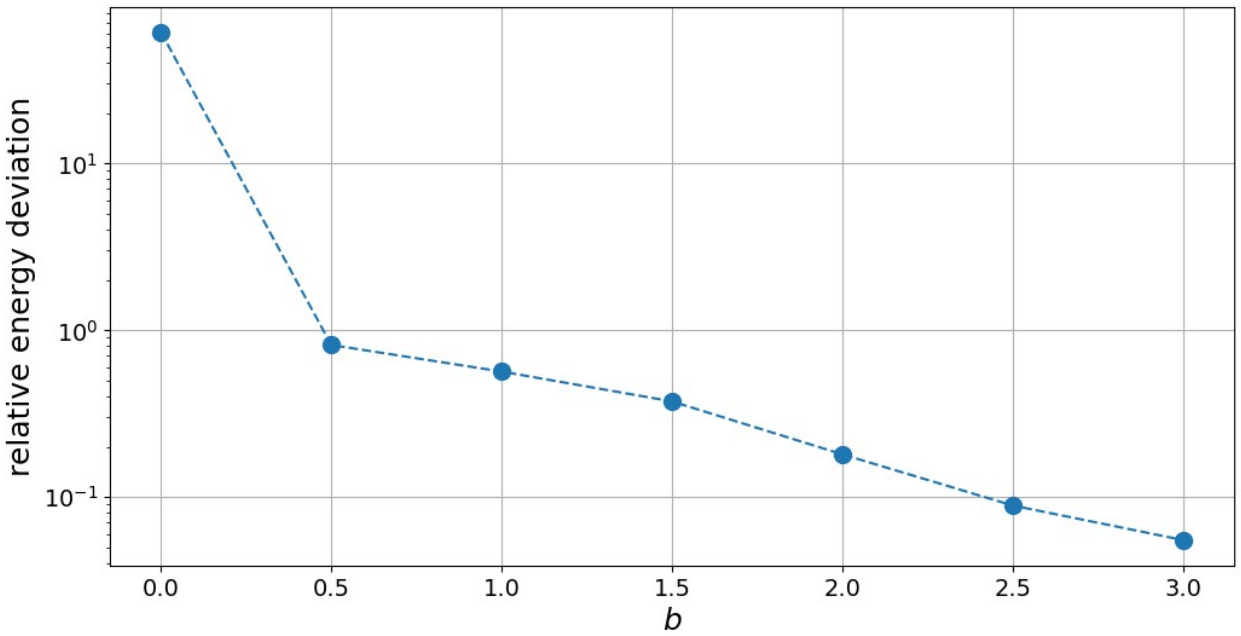
```
plt.figure(figsize=(12,8))
# Set initial conditions
E, h = 1, 0.01

# Iterate through different values of b
b_values = np.arange(0, 3.5, 0.5)
energy_errors = []
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
[:len(b_values)]
```

```
for b, color in zip(b_values, colors):
    state = np.array([-5, b, np.sqrt(2*E), 0])

    # Initialize and record the data using RK4
    data, energies = RK4(state, chaotic_acceleration, rmax=5,
obtain_energies=True)
    energy_errors.append( (max(energies)-
np.mean(energies))/np.mean(energies) )

    # Calculate transit time and scatttering angle
    transit_time = len(data) * h
    scattering_angle = np.arctan2(data[-1,3], data[-1,2])   #
np.arctan() does element-wise, np.arctan2() not
    print(f"b={b}: transit time={transit_time:.2f} dimensionless units
of time\n     scattering angle={np.degrees(scattering_angle):.2g}º\n")

    # Plot scattering
    plt.plot(data[:,0], data[:,1], color=color, zorder=1,
linewidth=2.5, label=f'b={b}')
    plt.scatter(data[0,0], data[0,1], color=color)
    plt.arrow(data[-1,0], data[-1,1], 0.001*data[-1,2], 0.001*data[-
1,3], head_width=0.15, head_length=0.3, color=color)

plt.contour(X, Y, Z, levels=50, alpha=0.4, cmap='Greys_r', zorder=0)
plt.colorbar(label='$V(x,y)$')
plt.xlabel('$x$')
plt.ylabel('$y$')
#plt.legend(loc='lower left')
plt.grid()
plt.savefig("chaotic_b_values.jpg", dpi=300, bbox_inches='tight')
plt.show()

# Plot the accuracy of the solution
plt.semilogy(b_values, energy_errors, 'o--', ms=10)
plt.xlabel('$b$')
plt.ylabel('relative energy deviation')
plt.grid()
plt.show()
```

```
b=0.0: transit time=7.09 dimensionless units of time
     scattering angle=0º

b=0.5: transit time=6.76 dimensionless units of time
     scattering angle=21º

b=1.0: transit time=6.82 dimensionless units of time
     scattering angle=0º

b=1.5: transit time=7.07 dimensionless units of time
```

```
     scattering angle=-17º

b=2.0: transit time=7.05 dimensionless units of time
     scattering angle=-13º

b=2.5: transit time=6.69 dimensionless units of time
     scattering angle=-2.7º

b=3.0: transit time=6.39 dimensionless units of time
     scattering angle=-0.3º
```





b) Finding chaotic transits

```
# Set initial conditions
E_values = [1.0, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]
h = 0.01

# Iterate through different values of b
b_values = np.arange(0.1, 2.5, 0.1)
energy_errors = []
for E in E_values:
    angles, times = [], []
    for b in b_values:
        state = np.array([-5, b, np.sqrt(2*E), 0])

        # Initialize and record the data using RK4
        data = RK4(state, chaotic_acceleration, rmax=5)

        # Calculate transit time and scatttering angle
        transit_time = len(data) * h
        scattering_angle = np.arctan2(data[-1,3], data[-1,2])   #
np.arctan() does element-wise, np.arctan2() not
        times.append(transit_time), angles.append(scattering_angle)

    # Calculate change in scattering angles and transit times w.r.t.
previous parameters (i.e. variability due to small changes in initial
conditions)
    angles, times = np.array(angles), np.array(times)
    angles_diff, times_diff = np.diff(np.unwrap(angles)),
np.diff(times)
    print(f"E={E:.3g} max:
```

```
Δθ={np.degrees(np.max(np.abs(angles_diff))):.2f},
Δt={np.max(np.abs(times_diff)):.2f}")
    print(f"max/mean rate:
Δθ={np.degrees(np.max(np.abs(angles_diff)))/np.degrees(np.mean(np.abs(
angles_diff))):.2f},
Δt={np.max(np.abs(times_diff))/np.mean(np.abs(times_diff)):.2f}")
    print(f"Where the maxima occur, b0:
{b_values[np.argmax(np.abs(angles_diff))]:.1f}, bt:
{b_values[np.argmax(np.abs(times_diff))]:.1f}\n")
```

```
E=1 max: Δθ=7.90, Δt=0.10
max/mean rate: Δθ=2.67, Δt=2.17
Where the maxima occur, b0: 0.1, bt: 0.1

E=0.5 max: Δθ=5.81, Δt=0.26
max/mean rate: Δθ=1.29, Δt=2.52
Where the maxima occur, b0: 2.1, bt: 0.1

E=0.2 max: Δθ=16.85, Δt=0.80
max/mean rate: Δθ=2.32, Δt=3.33
Where the maxima occur, b0: 2.3, bt: 2.3

E=0.1 max: Δθ=49.56, Δt=3.11
max/mean rate: Δθ=4.44, Δt=5.81
Where the maxima occur, b0: 2.0, bt: 2.3

E=0.05 max: Δθ=154.92, Δt=21.88
max/mean rate: Δθ=9.03, Δt=8.67
Where the maxima occur, b0: 2.0, bt: 2.2

E=0.02 max: Δθ=141.44, Δt=18.15
max/mean rate: Δθ=7.90, Δt=6.42
Where the maxima occur, b0: 2.3, bt: 2.2

E=0.01 max: Δθ=109.32, Δt=37.61
max/mean rate: Δθ=6.23, Δt=6.76
Where the maxima occur, b0: 1.8, bt: 1.8
```

It can be seen that chaos is bigger at lower energies. Let's reduce the step size in `b_values` to see when chaos occurs with even smaller changes in the initial conditions. We can reduce `E_values` and `b_values` so that computational time isn't spent with parameters we are not that interested on.

Same code, reduced lists of parameters E and b

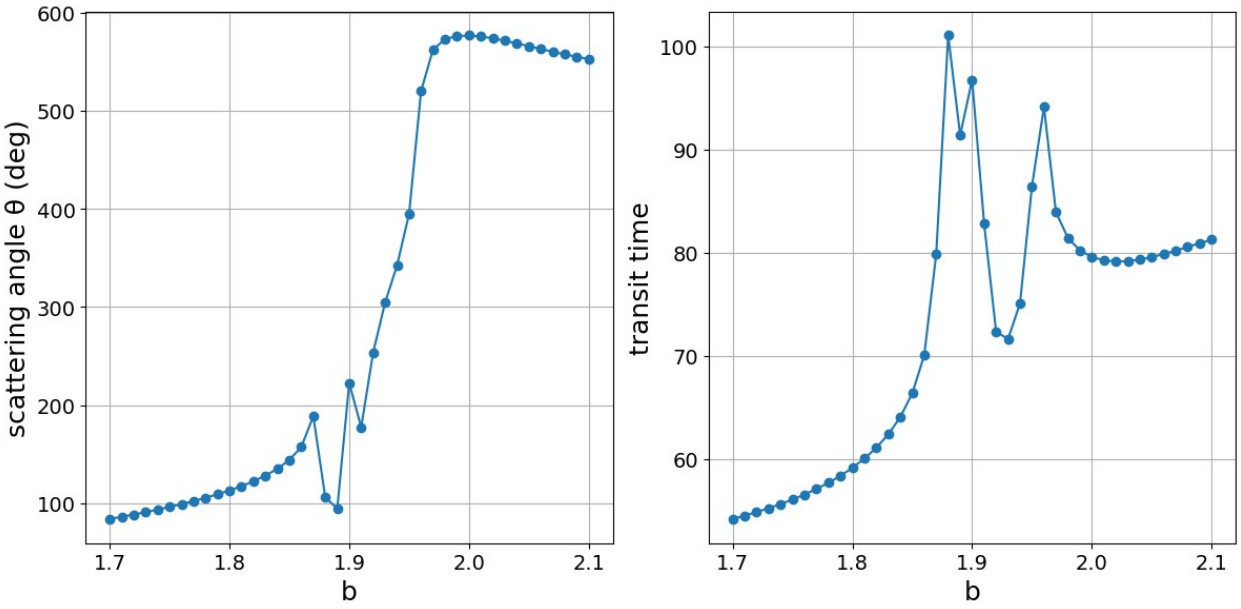
```
# Set initial conditions
E_values = [0.05, 0.02, 0.01]
h = 0.01
```

```
# Iterate through different values of b
b_values = np.arange(1.8, 2.4, 0.01)
energy_errors = []
for E in E_values:
    angles, times = [], []
    for b in b_values:
        state = np.array([-5, b, np.sqrt(2*E), 0])

        # Initialize and record the data using RK4
        data = RK4(state, chaotic_acceleration, rmax=5)

        # Calculate transit time and scatttering angle
        transit_time = len(data) * h
        scattering_angle = np.arctan2(data[-1,3], data[-1,2])   #
np.arctan() does element-wise, np.arctan2() not
        times.append(transit_time), angles.append(scattering_angle)

    # Calculate change in scattering angles and transit times w.r.t.
previous parameters (i.e. variability due to small changes in initial
conditions)
    angles, times = np.array(angles), np.array(times)
    angles_diff, times_diff = np.diff(np.unwrap(angles)),
np.diff(times)
    print(f"E={E:.3g} max:
Δθ={np.degrees(np.max(np.abs(angles_diff))):.2f},
Δt={np.max(np.abs(times_diff)):.2f}")
    print(f"max/mean rate:
Δθ={np.degrees(np.max(np.abs(angles_diff)))/np.degrees(np.mean(np.abs(
angles_diff))):.2f},
Δt={np.max(np.abs(times_diff))/np.mean(np.abs(times_diff)):.2f}")
    print(f"Where the maxima occur, bθ:
{b_values[np.argmax(np.abs(angles_diff))]:.2f}, bt:
{b_values[np.argmax(np.abs(times_diff))]:.2f}\n")
```

```
E=0.05 max: Δθ=120.34, Δt=27.36
max/mean rate: Δθ=8.26, Δt=15.13
Where the maxima occur, bθ: 2.10, bt: 2.17

E=0.02 max: Δθ=154.03, Δt=25.30
max/mean rate: Δθ=7.23, Δt=8.69
Where the maxima occur, bθ: 2.17, bt: 2.17

E=0.01 max: Δθ=127.73, Δt=21.21
max/mean rate: Δθ=9.65, Δt=10.00
Where the maxima occur, bθ: 1.89, bt: 1.87
```

$E=0.01$, although it would take cause a slightly larger computational time, would be a great case to study. Let's study it around the b values where the chaos takes over $b=[1.87,1.89)$

```
# Set initial conditions
E, h = 0.01, 0.01

# Iterate through different values of b
b_values = np.arange(1.70, 2.10, 0.01)
angles, times = [], []
for b in b_values:
    state = np.array([-5, b, np.sqrt(2*E), 0])

    # Initialize and record the data using RK4
    data = RK4(state, chaotic_acceleration, rmax=5)

    # Calculate transit time and scatttering angle
    transit_time = len(data) * h
    scattering_angle = np.arctan2(data[-1,3], data[-1,2])   #
np.arctan() does element-wise, np.arctan2() not
    times.append(transit_time), angles.append(scattering_angle)

times, angles = np.array(times), np.array(angles)

plt.figure(figsize=(12,6))
plt.subplot(1,2,1)
plt.plot(b_values, np.degrees(np.unwrap(angles)), marker='o')
plt.xlabel('b')
plt.ylabel('scattering angle θ (deg)')
plt.grid()

plt.subplot(1,2,2)
plt.plot(b_values, times, marker='o')
plt.xlabel('b')
plt.ylabel('transit time')
plt.grid()

plt.tight_layout()
plt.savefig("chaos.jpg", dpi=300, bbox_inches='tight')
plt.show()
```

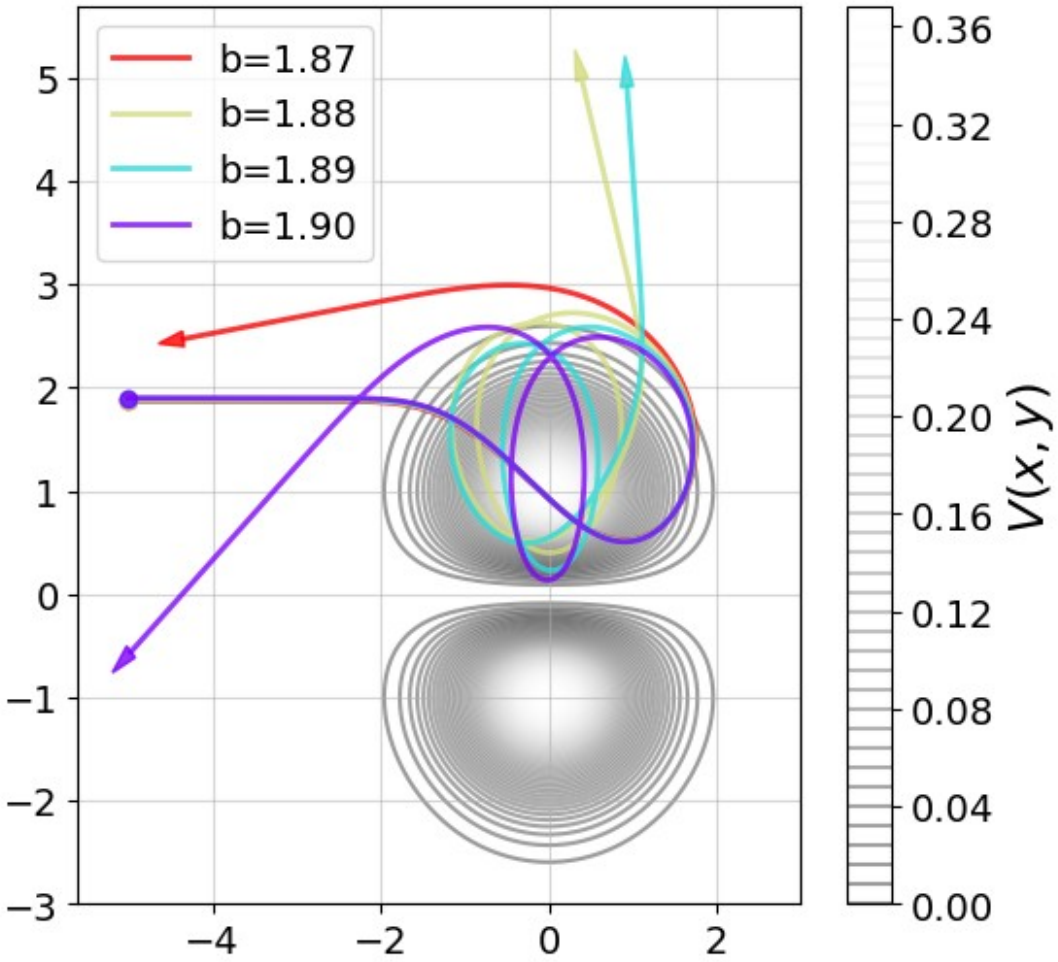


It can be seen how chaos takes over in the $b = [1.87, 1.90)$ region so let's visualize the path

```
plt.figure(figsize=(6,6))
# Set initial conditions
E, h = 0.01, 0.01

# Iterate through different values of b
b_values = np.arange(1.87, 1.91, 0.01)
colors = plt.cm.rainbow(np.linspace(1, 0, len(b_values)))

for color, b in zip(colors, b_values):
    state = np.array([-5, b, np.sqrt(2*E), 0])

    # Initialize and record the data using RK4
    data = RK4(state, chaotic_acceleration, rmax=5)

    # Calculate transit time and scatttering angle
    transit_time = len(data) * h
    scattering_angle = np.arctan2(data[-1,3], data[-1,2])   #
np.arctan() does element-wise, np.arctan2() not
    print(f"b={b:.2f}: transit time={transit_time:.2f} dimensionless
units of time\n     scattering
angle={np.degrees(scattering_angle):.2f}º\n")

    # Plot
    plt.plot(data[:,0], data[:,1], color=color, linewidth=2,
alpha=0.8, label=f'b={b:.2f}')
    plt.scatter(data[0,0], data[0,1], color=color, alpha=0.8,
zorder=2)
    plt.arrow(data[-1,0], data[-1,1], 1e-3*data[-1,2], 1e-3*data[-
1,3], head_width=0.15, head_length=0.3, color=color, alpha=0.8)
```

```
# PLot contour
plt.contour(X,Y,Z,levels=50, zorder=0, alpha=0.4, cmap='Greys_r')
plt.colorbar(label='$V(x,y)$')
plt.legend()
plt.grid(alpha=0.5)
plt.savefig("final2d.jpg", dpi=300, bbox_inches='tight')
plt.show()
```

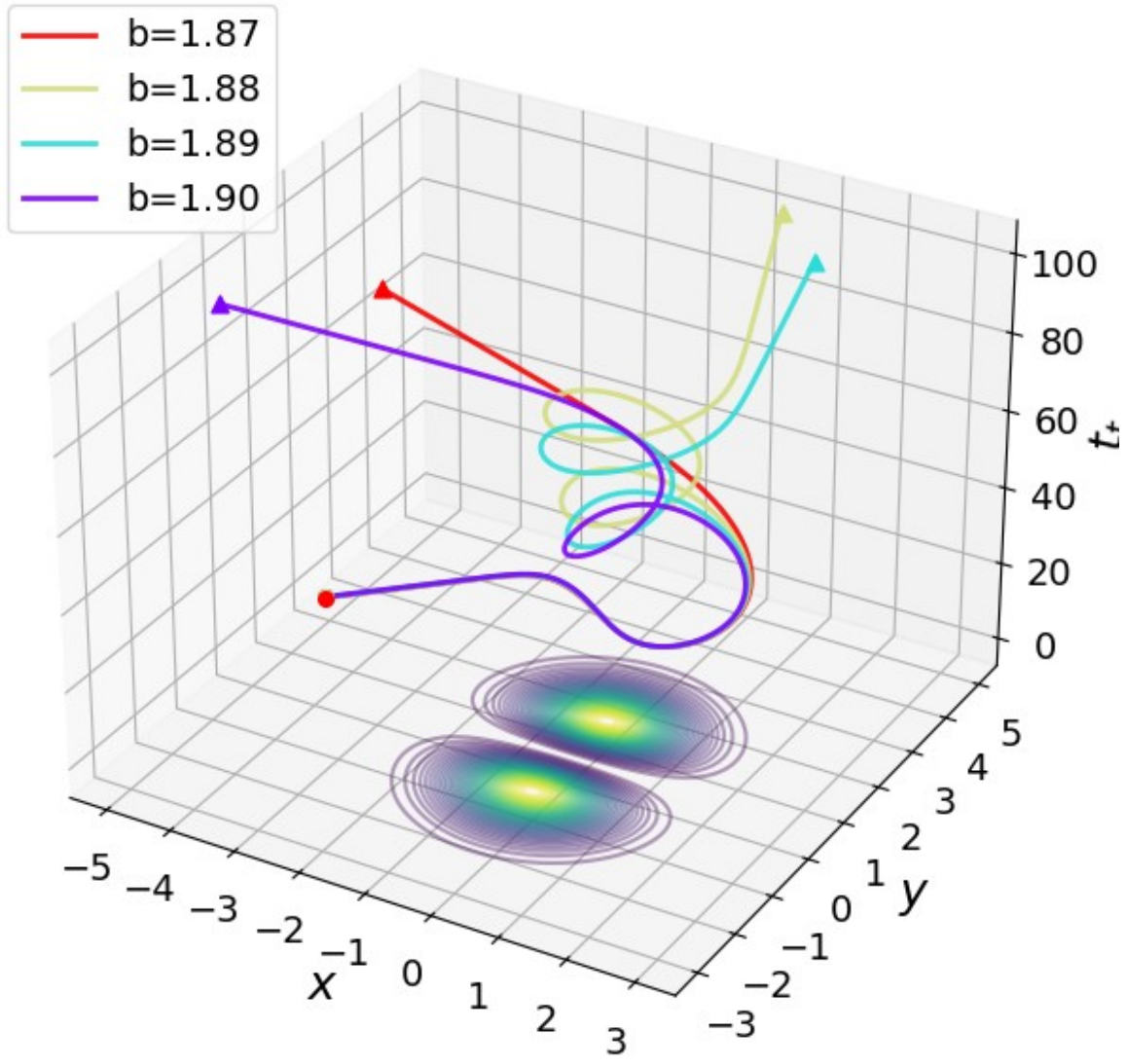
```
b=1.87: transit time=79.84 dimensionless units of time
     scattering angle=-171.19º

b=1.88: transit time=101.05 dimensionless units of time
     scattering angle=105.98º

b=1.89: transit time=91.39 dimensionless units of time
     scattering angle=94.74º

b=1.90: transit time=96.74 dimensionless units of time
     scattering angle=-137.53º
```



Below is plotted in 3D so that the time spent in the potential before leaving and reaching $r = 5$ can be seen, being able to observe a chaotic behaviour in that as well.. Here both the angle and the timestep can be seen as a function of b (consistent with the θ vs b and t vs b from above).

```
# Set initial conditions
E, h = 0.01, 0.01

# Iterate through different values of b
b_values = [1.87, 1.88, 1.89, 1.90]
colors = plt.cm.rainbow(np.linspace(1, 0, len(b_values)))

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

for color, b in zip(colors, b_values):
    state = np.array([-5, b, np.sqrt(2*E), 0])

    # Initialize and record the data using RK4
    data = RK4(state, chaotic_acceleration, rmax=5)
    t = np.arange(len(data))*h   # time as the z coordinate
```

```
    # Plot 3D trajectory
    ax.plot3D(data[:,0], data[:,1], t, color=color, linewidth=2,
alpha=0.9, label=f"b={b:.2f}")
    ax.scatter(data[0,0], data[0,1], t[0], color=color, marker='o',
s=30)
    ax.scatter(data[-1,0], data[-1,1], t[-1], color=color, marker='^',
s=40)

ax.contour(X,Y,Z,levels=50, zorder=0, alpha=0.4)
ax.set_xlabel('$x$'); ax.set_ylabel('$y$'); ax.set_zlabel('$t_t$')
ax.legend()
plt.tight_layout()
plt.savefig('final3d.jpg', dpi=300, bbox_inches='tight',
pad_inches=0.3)
plt.show()
```


b=1.87
b=1.88
b=1.89
b=1.90
100
80
60
40
20
0
t_+
5
4
3
2
1
0
−1
−2
−3
y
−5
−4
−3
−2
−1
0
1
2
3
x