Double pendulum

Barletta Valentina

February 28, 2024

The planar double pendulum, as shown in Fig. 1, is one of the mechanical systems that exhibits chaotic behavior. It consists of two pendulums attached together, with the pivot of the second pendulum located at the end of the first. We will concentrate only on simple double pendulum systems, where the system consists of two point masses, m_1 and m_2, suspended by massless wires of length l_1 and l_2.

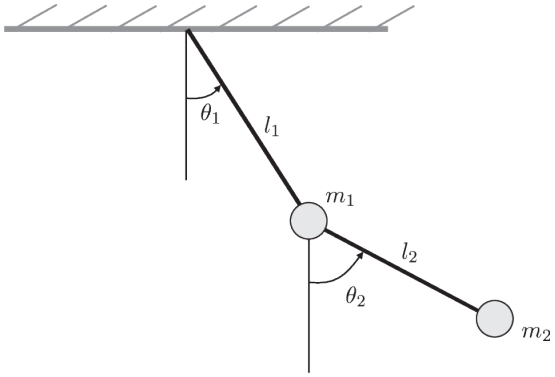

Figure 1: Double pendulum system

We choose a coordinate system with the origin at the top suspension point, the x-axis as a horizontal axis in the plane of motion, and the y-axis pointing down.

We can easily determine the behavior of a non-chaotic system simply by knowing the initial conditions of the problem and the laws of nature governing it. A small change in the initial conditions leads to small changes in its behavior, but this is not true for chaotic systems. The double pendulum becomes chaotic over a certain range of release angles, and one of the aims of the project is to find this range. For this reason, the Lyapunov exponent, which is a measure of how chaotic the system is, is also computed by numerical analysis.

Numerical integration was also used to determine the particular set of initial conditions which will result in either pendulum flipping and, if one does, how long it is before it flips.

This project is organized as follows: in section 1, we present the equations of motion for the double pendulum, computed using the Lagrangian formalism. In section 2, we present the numerical methods we employed to solve the equations of motion. In the remaining section, we present the main results of the dynamic analysis obtained by numerical integration.

1 The Equation of motion

We indicate the upper pendulum by subscript 1, and the lower by subscript 2. Begin by using simple trigonometry to write expressions for the positions x_1, y_1, x_2, y_2 in terms of the angles θ_1, θ_2.

$$\begin{aligned} x_1 &= l_1 \sin(\theta_1) & \qquad y_1 &= -l_1 \cos(\theta_1) \\ x_2 &= x_1 + l_2 \sin(\theta_2) & \qquad y_2 &= y_1 - l_2 \cos(\theta_2) \end{aligned} \tag{1}$$

The velocity is the derivative with respect to time of the position.

$$\dot{x_1} = l_1\dot{\theta_1}\cos(\theta_1) \qquad \dot{y_1} = l_1\dot{\theta_1}\sin(\theta_1)$$
$$\dot{x_2} = \dot{x_1} + l_2\dot{\theta_2}\cos(\theta_2) \qquad \dot{y_2} = \dot{y_1} + l_2\dot{\theta_2}\sin(\theta_2)$$

The total kinetic energy is given by:

$$T = \frac{1}{2}m_1(\dot{x_1}^2 + \dot{y_1}^2) + \frac{1}{2}m_2(\dot{x_2}^2 + \dot{y_2}^2)$$

The potential energy due to gravity force is:

$$U = m_1 g y_1 + m_2 g y_2 \tag{2}$$

Now we can write the Lagrangian $L = T - U$ and its full expression is:

$$L = \frac{1}{2}(m_1 + m_2)l_1^2\dot{\theta_1}^2 + \frac{1}{2}m_2 l_2^2\dot{\theta_2}^2 + m_2 l_1 l_2\dot{\theta_1}\dot{\theta_2}\cos(\theta_1 - \theta_2) + g(m_1 + m_2)l_1\cos\theta_1 + g m_2 l_2\cos\theta_2$$

After applying the Euler-Lagrange equation to $(\dot{\theta_1}, \theta_1)$ and $(\dot{\theta_2}, \theta_2)$ and simplifying a bit, we get the following two equations:

$$(m_1 + m_2)l_1\ddot{\theta_1} + m_2 l_2\ddot{\theta_2}\cos(\theta_1 - \theta_2) + m_2 l_2\dot{\theta_1}\sin(\theta_1 - \theta_2) + (m_1 + m_2)g\sin\theta_1 = 0$$
$$m_2 l_2\ddot{\theta_2} + m_2 l_1\ddot{\theta_1}\cos(\theta_1 - \theta_2) - m_2 l_1\dot{\theta_1}^2\sin(\theta_1 - \theta_2) + m_2 g\sin\theta_2 = 0$$

These are non-linear differential equations not solvable analytically. Note that setting $m_2 = 0$, $l_2 = 0$, $\theta_1 = \theta_2$ both equations become the simple pendulum equation $\ddot{\theta} + \omega^2\sin\theta = 0$. In order to solve these equations numerically, we need to turn these two coupled second order equations into a system of four first order equations. After a lot of algebra we get:

$$\begin{aligned}
\omega_1 &= \theta_1 \\
\omega_2 &= \theta_2 \\
\ddot{\theta_1} &= \frac{-g(2m_1 + m_2)\sin\theta_1 - m_2 g\sin(\theta_1 - 2\theta_2) - 2sin(\theta_1 - \theta_2)m_2(\dot{\theta_2}^2 l_2 + \dot{\theta_1}^2 l_1\cos(\theta_1 - \theta_2)}{l_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))} \\
\ddot{\theta_2} &= \frac{2\sin(\theta_1 - \theta_2)(\dot{\theta_1}^2 l_1(m_1 + m_2) + g(m_1 + m_2)\cos\theta_1 + \dot{\theta_2}^2 l_2 m_2\cos(\theta_1 - \theta_2))}{l_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}
\end{aligned} \tag{3}$$

This is now exactly the form needed to plug in to the Runge-Kutta method for numerical solution of the system.

2 Algorithm choice and discussion on Energy

The most basic numeric method for solving dynamical equations is the Euler method. However, as shown in Fig. 2, the fourth-order Runge-Kutta algorithm offers better convergence performance. It provides the best balance between accuracy and computational effort.

For dynamical systems, it is interesting to look for symmetries or, in other words, conserved quantities. For our problem, the obvious conserved quantity is the mechanical energy:

$$E = \frac{1}{2}(m_1 + m_2)l_1^2\dot{\theta_1}^2 + \frac{1}{2}m_2 l_2^2\dot{\theta_2}^2 + m_2 l_1 l_2\dot{\theta_1}\dot{\theta_2}\cos(\theta_1 - \theta_2) - g(m_1 + m_2)l_1\cos\theta_1 - g m_2 l_2\cos\theta_2 \tag{4}$$

So, the double pendulum is a Hamiltonian system, and any Hamiltonian system is symplectic. This property is not automatically satisfied by standard numerical integrators, but there are some methods, such as the Verlet algorithm, that can preserve it. Unfortunately, the double pendulum is governed by a non-separable Hamiltonian, so we should modify the implementation of the symplectic integrator to

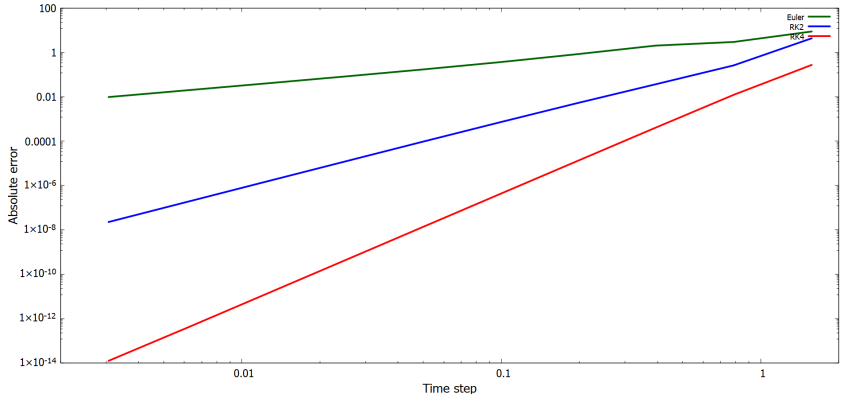

Figure 2: Convergence performance of Euler, second-order and fourth-order Runge-Kutta algorithm. This test was conducted on the simple pendulum, in the case of small angle approximation. The equation of motion was $\ddot{\theta} = -\frac{g}{l}\theta$. It's evident that RK4 as better performance: its slope is greater than the others.

handle it. Therefore, the best choice remains the fourth-order Runge-Kutta method because it is simple to implement, and an appropriate choice of the integration step can improve its performance.

It is usually assumed that a smaller time step, Δt, will lead to smaller integration errors. We decided to test this assumption. For fixed initial conditions, we computed the relative error defined as follows: $err_{rel} = \left|\frac{E-E_0}{E_0}\right|$, where E_0 is the starting energy, and E is the energy evaluated by the Runge-Kutta integration at a given time t. Fig. 3 shows the behavior of err_{rel} for different Δt for a very high number of iterations ($\sim 10^6$). Our results indicate that to achieve an error lower than 1%, the optimal time step should be less than $\Delta t = 0.006$s. As we progress in our study, we will integrate for fewer iterations, so choosing a time step of this magnitude ensures that the energy loss will be even lower.

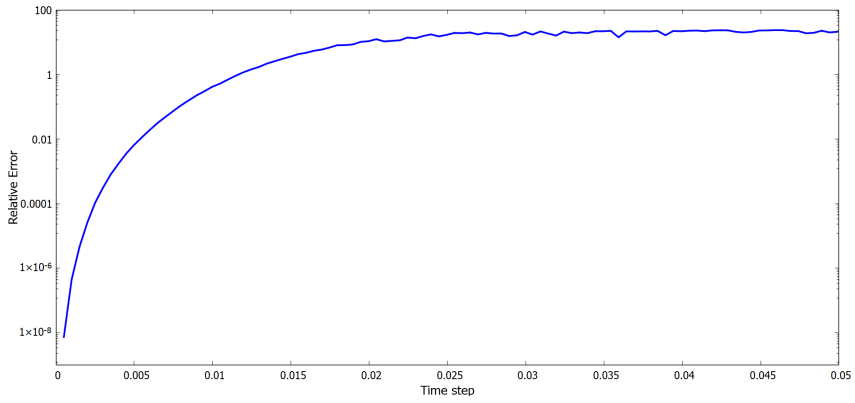

Figure 3: Relative Error on the energy for different time step, using logarithmic y-axis

3 Dynamic analysis

In the preceding section, we explored methods to obtain accurate numerical results for the dynamics of the double pendulum. In this section, our focus shifts to extracting valuable insights from the numerical solution of the system's equations of motion. The parameters of the double pendulum used in the investigations of this project are:

$$m_1 = m_2 = 1\text{kg}$$
$$l_1 = l_2 = 1\text{m}$$

3.1 Trajectories of the Double Pendulum

When the system of equations is solved and graphed, it illustrates the evolution of the angles θ_1 and θ_2 to the vertical of the inner and outer pendulum masses over time. For fixed time step $\Delta t = 0.001$s and number of iteration $N = 10^5$, various trajectories of the double pendulum are displayed.

In Fig.4, the motion of a small-angle-release is shown. It is evident that for small initial release angles, the double pendulum behaves like a linear oscillator, exhibiting relatively regular oscillations with nearly constant amplitude and period.

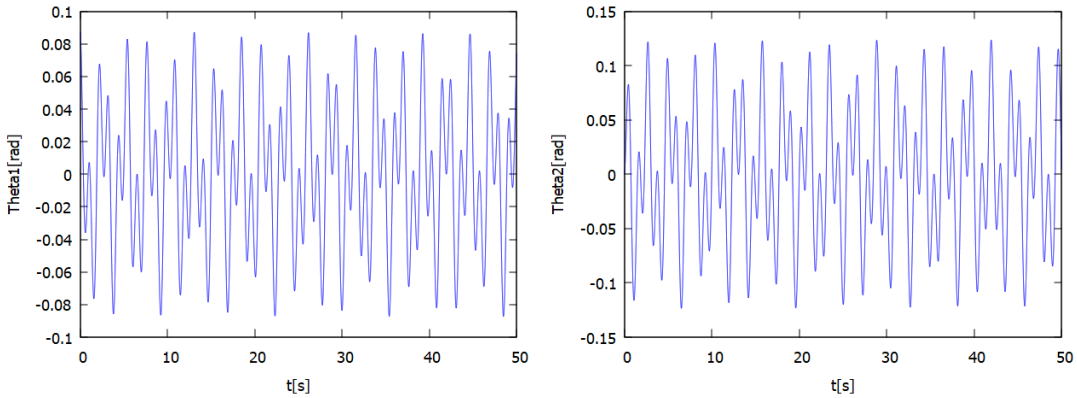

Figure 4: Motion for small-angle-release with initial conditions $\theta_1(0) = 5°$ and $\theta_2(0) = 0°$. Left: shows the behaviour of the time series for θ_1; right: the behaviour for θ_2

Fig.5 shows a large-angle-release, revealing the erratic and unpredictable behavior that corresponds well to the actual motion of the double pendulum.

The double pendulum, which reminds two coupled harmonic oscillators, exhibits two types of dynamics for small energies: periodic or quasiperiodic. Periodic motion occurs when the frequencies are commensurable, while quasiperiodic motion is a more general case. In nonlinear systems, the dynamics may go beyond the quasiperiodic motion, exhibiting a very complex behaviour known as chaos.

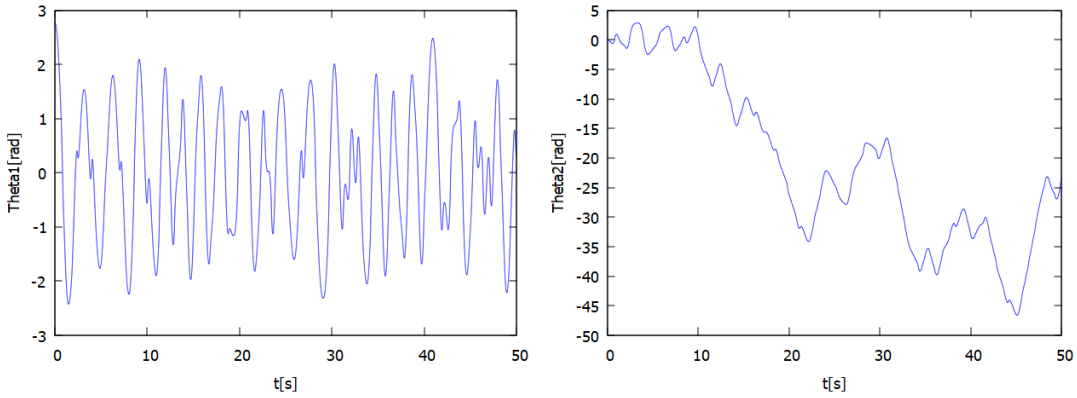

Figure 5: Motion for large-angle-release with initial conditions $\theta_1(0) = 160°$ and $\theta_2(0) = 0°$. Left: shows the behaviour of the time series for θ_1 right: the behaviour for θ_2

In Fig.6 we can observe what happens to the phase space as the angles-release increase.

A traditional way to characterize a chaotic system is through its Lyapunov exponents.

3.2 Lyapunov exponents

If the system is chaotic, then nearby orbits on the phase space will diverge from each other exponentially, i.e., as $e^{\lambda t}$. The exponent λ is called the Lyapunov exponent. The number of these exponents corresponds with the dimension of the phase space, and if any of them is positive, the system is said to be chaotic.

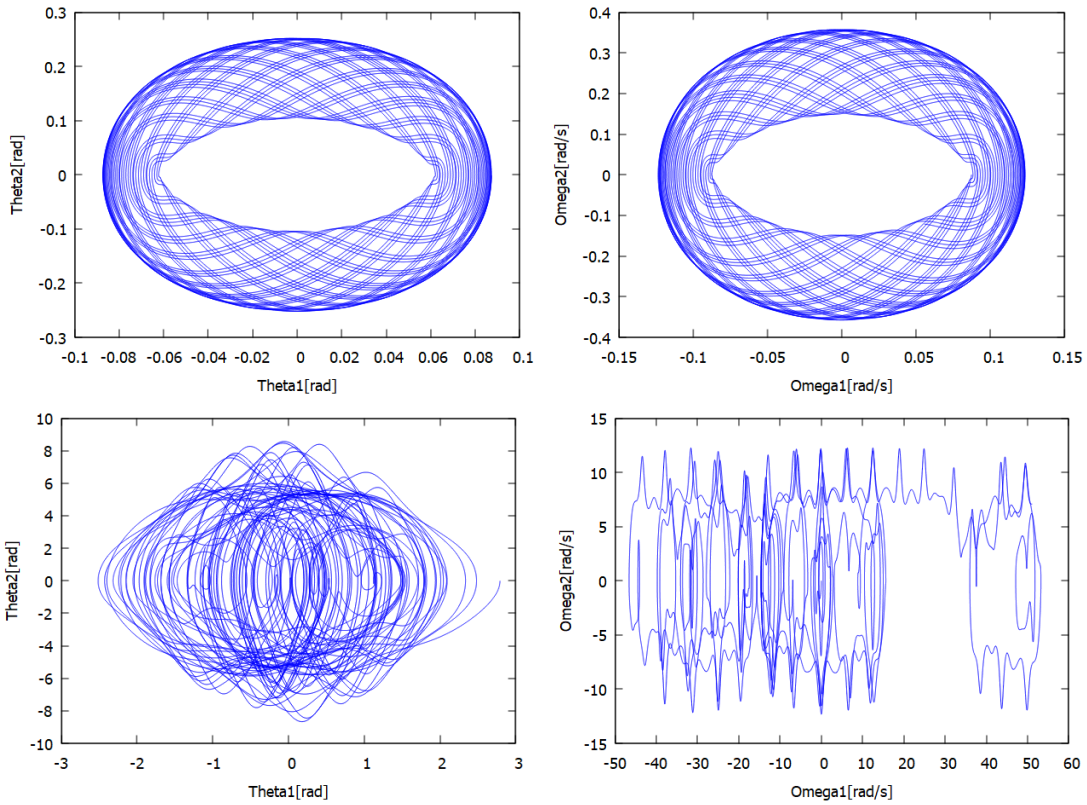

Figure 6: Up: phase space for small angle release $\theta_1(0) = 5°$ and $\theta_2(0) = 5°$; down: phase space for large angle release $\theta_1(0) = 160°$ and $\theta_2(0) = 0°$

This behavior indicates a high sensitivity to initial conditions and this is one of the critical aspects of chaos. Lyapunov exponents are defined as:

$$\lambda = \lim_{t\to\infty} \lim_{\delta(t_0)\to 0} \frac{1}{t} \log \left| \frac{\delta(t)}{\delta(t_0)} \right| \tag{5}$$

where $\delta(t_0)$ is the infinitesimal distance of the two orbits at the starting time t_0 and $\delta(t)$ is the distance after a time t.

A naive Lyapunov exponent estimation is to use a numerical method to integrate two very close trajectories, and then calculate how the orbits diverge from each other. In practical terms, this method is not appropriate; the divergence will grow exponentially and the finite precision of computer digits would lead to a wrong estimate of the exponent.

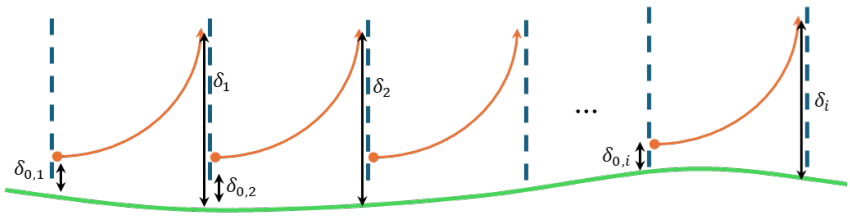

Figure 7: Lyapunov estimation scheme

Let $\delta_{0,i}$ be the initial Cartesian distance at iteration i and δ_i be the final Cartesian distance, obtained by Runge-Kutta algorithm, at the same iteration. A schematic figure is represented in Fig.7. To find the

Lyapunov exponent we must average the Cartesian distances $\delta_{0,i}$ and δ_i many times along a trajectory. The final expression is:

$$\lambda = \frac{1}{t} \log \left(\sum_{i=0}^{N} \frac{\delta_i}{\delta_{0,i}} \right) \tag{6}$$

where t is the integration time and N is the number of iterations.

We identified four distinct ranges of initial conditions leading to chaos for the parameters θ_1, θ_2, ω_1, and ω_2. We explored the range $[0, \pi]$ for each angular parameter and the range $[0, 2\pi]$ for the velocity parameters, while setting the others to zero. To ensure infinitesimal separation between trajectories, we created a second nearby trajectory by adding $d_0 = 10^{-8}$ rad to the initial one. Subsequently, we integrated the system to generate a new point on the reference orbit and calculated the new Cartesian distance. This process was repeated 100 times and the integration time was fixed to 1000s.

The final results are depicted in Fig. 8.

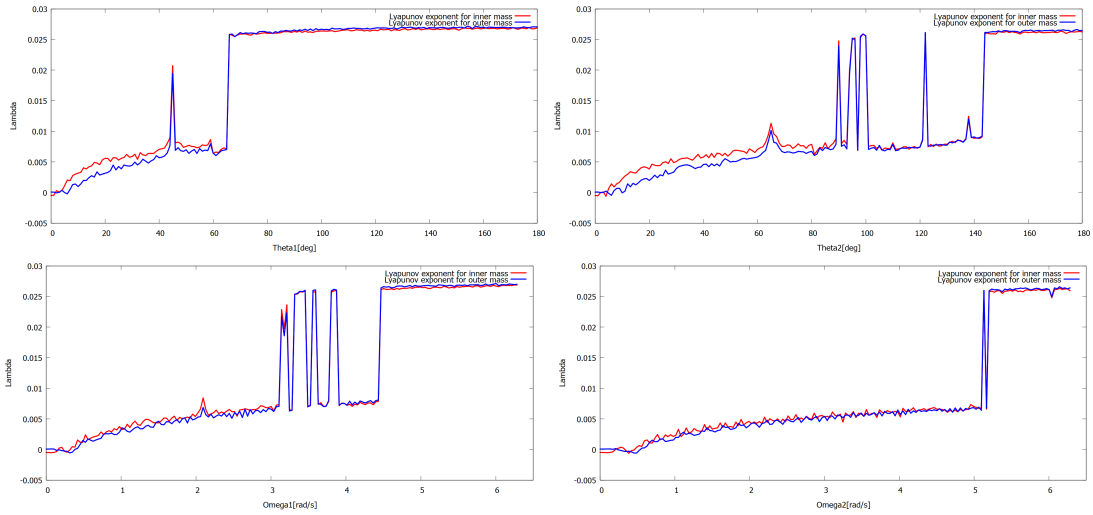

Figure 8: Lyapunov exponents for the double pendulum. It is shown how the Lyapunov exponent depends on the initial conditions.

Chaos begins when:

- θ_1 has a release angle greater than 66°;
- θ_2 has a release angle greater than 157°;
- ω_1 has an initial angular velocity greater than 4.7 rad/s;
- ω_2 has an initial angular velocity greater than 5.4 rad/s.

As observed, there is a general increase in λ with the release angle, indicating that a larger release angle results in greater sensitivity to initial conditions for the double pendulum. Fluctuations of λ can be avoided by increasing the time of integration.

3.3 Time for First Flip

We are currently focused on determining the Time for First Flip (TFF), representing the duration it takes for a pendulum mass to flip for the first time since its release at $t = 0$.

By applying energy considerations, we can establish the theoretical criteria for the flipping of both the inner and outer pendulum. The minimum condition for pendulum 1 to flip is met when $\theta_1 = \pi$ and $\theta_2 = 0$. Similarly, for the outer pendulum (pendulum 2), the minimum condition is when $\theta_1 = 0$

and $\theta_2 = \pi$. Substituting these minimum conditions into the general potential energy expression defined earlier by equation (2), we obtain the minimum energy, U_{min1}, required for the inner pendulum to flip:

$$U_{min1} = g(l_1(m_1 + m_2) - l_2 m_2)$$

A similar equation can be derived for pendulum 2. For the respective pendulum masses to flip, their potential energy U must be greater than or equal to their U_{min}. Thus, the following conditions are derived:

$$l_1(m_1 + m_2)(\cos\theta_1 - 1) + l_2 m_2(\cos\theta_2 + 1) \leq 0$$
$$l_1(m_1 + m_2)(\cos\theta_1 + 1) + l_2 m_2(\cos\theta_2 - 1) \leq 0.$$

Assigning a unit to each parameter, we obtain:

$$2\cos\theta_1 + \cos\theta_2 \leq -1$$
$$2\cos\theta_1 + \cos\theta_2 \leq 1.$$

Initial release angles θ_1 and θ_2 not satisfying these constraints will result in the respective pendulum masses not having enough energy to flip.

After extensive CPU run-time, initial starting positions of the inner and outer pendulums are systematically looped through to find the Time for First Flip (TFF) for each set of initial conditions. Starting at $-180°$, initial conditions are iterated through to $180°$ with increments of half a degree. This is done for the two pendulum masses with respect to each other, setting $\omega_1 = \omega_2 = 0$.

A 3-dimensional plot can be obtained by representing starting points $\theta_1(0)$ and $\theta_2(0)$ on the x and y axes, and the corresponding TFF values on the z-axis. The two plots are shown in Fig.9.

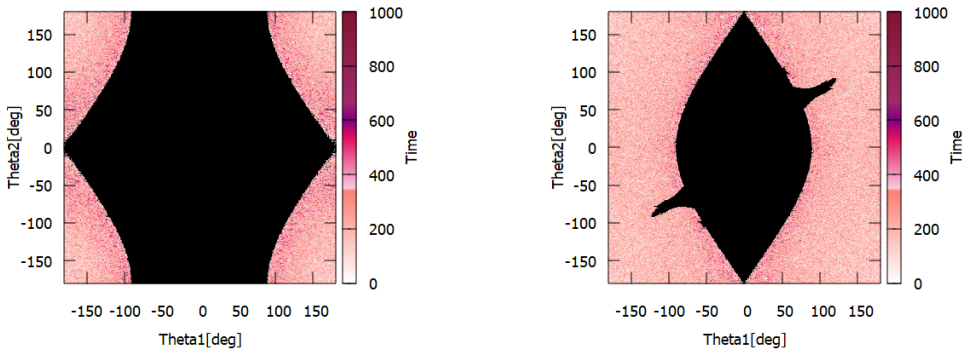

Figure 9: Contour plots for the time-for-first-flip of the respective pendulum masses. Left: the plot for the inner pendulum; right: the outer pendulum. Axis shows initial conditions $\theta_1(0)$ along the x-direction and $\theta_2(0)$ along the y-direction. Clear region needs less time to flip, dark region needs more time, black regions do not flip at all in the 1000 seconds evaluated.

To limit computational costs, the step size used in integration was $\Delta t = 0.01$s, with a total number of iterations $N = 10^5$. Notice that the curvature of the main black region of the pattern, where the pendulum masses never have enough energy to flip, corresponds exactly with the constraint equations derived earlier.

4 Conclusion

We explored the dynamics of the double pendulum, a straightforward mechanical system that displays chaotic behavior.

The fourth-order Runge-Kutta algorithm, selected for its superior convergence performance, was employed to integrate the equations of motion. Energy considerations led to the identification of an optimal time step ($\Delta t = 0.006$s) to maintain energy loss below 1% over a significant number of iterations. With the reliability of the fourth-order Runge-Kutta algorithm confirmed, we used it to integrate the equations of motion and explore the system's dynamics. Our focus included evaluating the Lyapunov exponent to

identify initial conditions triggering chaotic behavior. Additionally, we generated a color map illustrating the instances and occurrences of pendulum flips.

Future research could focus on studying the double pendulum using symplectic methods and subsequently comparing the results. This could provide valuable insights into the system's behavior and enhance our understanding of its complex dynamics.

5 Code

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
#include <fstream>
#include <stdlib.h>

using namespace std;

// Function prototypes
void dYdt(double t, double *Y, double *R);
void RK4Step(double t, double *Y, double h, void (*RHSFunc)(double, double *, double *), int neq);
double Energy(double theta1, double theta2, double omega1, double omega2);

double G_ACCg = 9.81;  // Acceleration due to gravity

// Masses and lengths of the pendulums
# define M1 1
# define M2 1
# define L1 1
# define L2 1

// SESSION 1 (Is energy conserved?), SESSION 2 (GIF), SESSION 3 (Chaos), SESSION 4 (Flip)
#define SESSION 1

// PARAMETER == 1 select theta1, PARAMETER == 2 select theta2, PARAMETER == 3 select omega1, PARAMETER == 4 select omega2
#define PARAMETER 4

int main() {

  // Parameters and variables for computation
  double h = 0.;        // Step size
  double tb = 0.;       // Initial time
  double t = tb;        // Current time
  double t_end = 1.e2;  // End time for simulation (100s)
  double E_0, E, relErr; // Energy variables

  // Initial conditions for the double pendulum
  double x1, x2, y1, y2;  // Cartesian coordinates for the masses
  int neq = 4;    // Number of equations
  int end = 1.e6; // Total number of iterations
  double Y[neq];  // Initial state vector

#if SESSION == 1

  ofstream file; // Output file stream

  file.open("Double_pendulum_energy.txt"); // Open a file to write energy data

  // Declaration and initialization of variables
  double hbeg = 1.e-4; // Initial step size
  double hend = 0.05; // Final step size
  double npoints = 100; // Number of points for step size variation
  double IC[neq]; // Array to hold initial conditions

  // Assign initial condition
  Y[0] = 1.6;
  Y[1] = 1.;
  Y[2] = 2.;
  Y[3] = -3.;

  // Save the intial conditions
  for(int i = 0; i < neq; i++) IC[i] = Y[i];

  // Compute initial energy of the system
  E_0 = Energy(Y[0], Y[1], Y[2], Y[3]);
```

```
  E = E_0;

  // Loop for evaluating energy preservation over various step sizes
  for(int j = 1; j <= npoints; j++){

    // Incrementing step size
    h += fabs(hbeg - hend) / npoints;
    t = tb; // Resetting time

    // Restoring initial conditions
    for(int k = 0; k < neq; k++) Y[k] = IC[k];

    // Performing integration using Runge-Kutta method
    for(int i = 0; i < end; i++){
      RK4Step(t, Y, h, &dYdt, neq);
      t += h;
    }

    t -= h; // Correcting time

    // Computing energy and relative error
    E = Energy(Y[0], Y[1], Y[2], Y[3]);
    relErr = fabs((E_0 - E) / E_0);

    // Writing step size and relative error to file
    file << h << "\t" << relErr << endl;

    // Clear output and print the progression of the process
    system("clear");
    cout << j << endl;
  }

  // Closing the file
  file.close();

#endif

#if SESSION == 2

  ofstream file; // Output file stream
  file.open("Motion_small_angles.txt"); // Open a file to write motion data

  // Setting initial conditions for the double pendulum system
  Y[0] = double(5) * M_PI / 180.; // Initial angle of the first pendulum arm in radians
  Y[1] = double(0) * M_PI / 180.; // Initial angle of the second pendulum arm in radians
  Y[2] = 0.; // Initial angular velocity of the first pendulum arm
  Y[3] = 0.; // Initial angular velocity of the second pendulum arm

  h = 0.001; // Selected step size based on the analysis of the previous session
  end = 1.e5;

  // Iterating over the system dynamics
  for(int i = 0; i < end; i++){
    RK4Step(t, Y, h, &dYdt, neq);

    // Computing positions of pendulum masses
    x1 = L1 * sin(Y[0]); // x-coordinate of the first pendulum mass
    y1 = - L1 * cos(Y[0]); // y-coordinate of the first pendulum mass
    x2 = x1 + L2 * sin(Y[1]); // x-coordinate of the second pendulum mass
    y2 = y1 - L2 * cos(Y[1]); // y-coordinate of the second pendulum mass

    // Computing energy and relative error
    E = Energy(Y[0], Y[1], Y[2], Y[3]);
    relErr = fabs((E_0 - E) / E_0);

    // Writing time, positions, angles, and velocities to file
    file << t << " " << x1 << " " << y1 << " " << x2 << " " << y2 << " "
       << Y[0] << " " << Y[1] << "\t" << Y[2] << "\t" << Y[3] << endl;

```

```
    t += h; // Updating time
  }

  file.close();

#endif


#if SESSION == 3

  // Setting step size
  h = 0.005; // Selected step size based on the analysis of the previous session

  // Array for initial conditions
  double IC[neq];

  // Declaration of variables for Lyapunov exponent computation
  double Ly1, Ly2; // Lyapunov exponents
  int n_iter = 100; // Number of iterations on a single trajectory
  double d1, d2, d0_1, d0_2; // Variables for computing distances
  double delta = 1.e-8; // Perturbation
  double x1_a, x2_a, y1_a, y2_a; // Auxiliary variables for pendulum positions

  double theta_end = M_PI; // Range for theta
  double omega_end = 2 * M_PI; // Range for omega
  double N_point = 180; // Number of points
  double step; // Step for parameter
  t = 0.;
  t_end = 1.e3;
  end = t_end / h; // Total number of iterations

  // Setting initial conditions
  Y[0] = 0.;
  Y[1] = 0.;
  Y[2] = 0.;
  Y[3] = 0.;

  int param; // The index of the parameter of interest
  double Deviation[] = {delta, delta, 0., 0.}; // Perturbation

  // Determining chaos search based on parameter type

#if PARAMETER == 1 // Search chaos for theta1

  ofstream file_chaos;
  file_chaos.open("Theta1_100.txt");
  file_chaos << setiosflags(ios::scientific);
  file_chaos << setiosflags(ios::scientific);
  step = theta_end / N_point;
  param = 0;

#elif PARAMETER == 2 // Search chaos for theta2

  ofstream file_chaos;
  file_chaos.open("Theta2_100.txt");
  file_chaos << setiosflags(ios::scientific);
  file_chaos << setiosflags(ios::scientific);
  step = theta_end / N_point;
  param = 1;

#elif PARAMETER == 3 // Search chaos for Omega1

  ofstream file_chaos;
  file_chaos.open("Omega1_100.txt");
  file_chaos << setiosflags(ios::scientific);
  file_chaos << setiosflags(ios::scientific);
  step = omega_end / N_point;
  param = 2;

```

```
#elif PARAMETER == 4 // Search chaos for Omega2

  ofstream file_chaos;
  file_chaos.open("Omega2_100.txt");
  file_chaos << setiosflags(ios::scientific);
  file_chaos << setiosflags(ios::scientific);
  step = omega_end / N_point;
  param = 3;

#endif

  // Looping over parameter values
  for(int i = 0; i <= N_point; i++){


    //Set the initial condition
    for(int z = 0; z < neq; z++){
      if(z != param) IC[z] = 0.;
      else IC[z] = step * i;
    }


    // Looping for the number of Lyapunov exponents
    for(int j = 0; j < n_iter; j++){

      // Set initial condition with deviation
      for(int k = 0; k < neq; k++) Y[k] = IC[k] + Deviation[k];

      // Compute pendulum positions
      x1 = L1 * sin(IC[0]);
      y1 = - L1 * cos(IC[0]);
      x2 = x1 + L2 * sin(IC[1]);
      y2 = y1 - L2 * cos(IC[1]);

      // Position for deviated coordinates
      x1_a = L1 * sin(Y[0]);
      y1_a = - L1 * cos(Y[0]);
      x2_a = x1_a + L2 * sin(Y[1]);
      y2_a = y1_a - L2 * cos(Y[1]);

      // Initial distance between the two trajectories for both the masses
      d0_1 += sqrt((x1_a - x1) * (x1_a - x1) + (y1_a - y1) * (y1_a - y1));
      d0_2 += sqrt((x2_a - x2) * (x2_a - x2) + (y2_a - y2) * (y2_a - y2));


      // Integrate the deviated system
      for(int k = 0; k < end; k++){
        t += h;
        RK4Step(t, Y, h, &dYdt, neq);
      }

      t = 0.; // Reset time

      // Compute new pendulum deviated positions
      x1_a = L1 * sin(Y[0]);
      y1_a = - L1 * cos(Y[0]);
      x2_a = x1_a + L2 * sin(Y[1]);
      y2_a = y1_a - L2 * cos(Y[1]);

      // Reset initial condition to the non deviated ones
      for(int k = 0; k < neq; k++) Y[k] = IC[k];


      // Integrate the non deviated the system
      for(int k = 0; k < end; k++){
        t += h;
        RK4Step(t, Y, h, &dYdt, neq);
      }

```

```
      // Compute new pendulum non deviated positions
      x1 = L1 * sin(Y[0]);
      y1 = - L1 * cos(Y[0]);
      x2 = x1 + L2 * sin(Y[1]);
      y2 = y1 - L2 * cos(Y[1]);

      t = 0.; // Reset time

      // Compute the distance at the end of the integration and sum
      d1 += sqrt((x1_a - x1) * (x1_a - x1) + (y1_a - y1) * (y1_a - y1));
      d2 += sqrt((x2_a - x2) * (x2_a - x2) + (y2_a - y2) * (y2_a - y2));

      // Set the final trajectory values as the new initial conditions
      for(int k = 0; k < neq; k++) IC[k] = Y[k];
    }

    // Compute the Lyapunov exponents
    Ly1 = log2(fabs(d1 / d0_1)) / t_end;
    Ly2 = log2(fabs(d2 / d0_2)) / t_end;

    // Write to file
    file_chaos << (step * i) * 180 / M_PI << " " << Ly1 << " " << Ly2 << endl;

    // Reset distance variables
    d0_1 = 0.;
    d0_2 = 0.;
    d1 = 0.;
    d2 = 0.;

    // Clear output and print the progression of the process
    system("clear");
    cout << i << endl;
  }

  // Close file
  file_chaos.close();

#endif


#if SESSION == 4

  // Opening files to store flip times
  ofstream file_flip1, file_flip2;
  file_flip1.open("Flip1.txt");
  file_flip2.open("Flip2.txt");

  // Variables to track if each pendulum has flipped
  bool flipped1 = false;
  bool flipped2 = false;

  // Setting step size and total number of iterations
  h = 0.01;
  end = 1.e5;

  // Initializing velocities
  Y[2] = 0.;
  Y[3] = 0.;

  // Setting parameters for angle computation
  int data_points = 720; // Number of data points
  int deg_lim = data_points / 2; // Range of computation
  double div = 180. / double(deg_lim); // Increment in angle value

  double theta_old1, theta_old2; // Variables to store previous angles

  // Looping through angle combinations
  for(int i = -deg_lim; i <= deg_lim; i++){
    for(int j = -deg_lim; j <= deg_lim; j ++){
```

```
342
343       // Setting initial angles
344       Y[0] = (double(i) * div) * M_PI / 180.;
345       Y[1] = (double(j) * div) * M_PI / 180.;
346       Y[2] = 0.;
347       Y[3] = 0.;
348
349       // Integration loop
350       for(int k = 0; k < end; k++){
351         RK4Step(t, Y, h, &dYdt, neq);
352
353         // Checking if pendulum 1 has flipped
354         if((flipped1 == false && t > 1.)
355           && ((sin(Y[0]) * sin(theta_old1) < 0.
356           && Y[0] != theta_old1
357           && cos(Y[0]) < -1. + 1.e-7))){
358
359           file_flip1 << (double(i) * div) << "\t" << (double(j) * div)
360                      << "\t" << t << endl; // Writing flip time
361
362           flipped1 = true;
363         }
364
365         if(flipped1 == false) theta_old1 = Y[0]; // Updating previous angle
366
367         // Checking if pendulum 2 has flipped
368         if((flipped2 == false && t > 1.)
369           && ((sin(Y[1]) * sin(theta_old2) < 0.
370           && Y[1] != theta_old2
371           && cos(Y[1]) < -1. + 1.e-7))){
372
373           file_flip2 << (double(i) * div) << "\t" << (double(j) * div)
374                      << "\t" << t << endl; // Writing flip time
375
376           flipped2 = true;
377         }
378
379         // Updating previous angle
380         if(flipped2 == false) theta_old2 = Y[1];
381
382         // Exiting loop if both pendulums have flipped
383         if(flipped1 == true && flipped2 == true) break;
384
385         t += h;
386       }
387
388       // Writing -1 if pendulum did not flip within the given time
389       if(flipped1 == false) {
390         file_flip1 << (double(i) * div) << "   "
391                    << (double(j) * div) << "   " << -1 << endl;
392       }
393
394       if(flipped2 == false){
395         file_flip2 << (double(i) * div) << "   "
396                    << (double(j) * div) << "   " << -1 << endl;
397       }
398
399       t = 0.; // Resetting time
400       flipped1 = false; // Resetting flip flag for pendulum 1
401       flipped2 = false; // Resetting flip flag for pendulum 2
402     }
403
404     system("clear");
405     cout << int(i * div) << endl; // Displaying progress
406
407     file_flip1 << endl;
408     file_flip2 << endl;
409   }
410
```

```
411   // Closing files
412   file_flip1.close();
413   file_flip2.close();
414 
415 #endif
416 
417   return 0;
418 }
419 
420 
421 void dYdt(double t, double *Y, double *R){
422 
423   double theta1 = Y[0];
424   double theta2 = Y[1];
425   double omega1 = Y[2];
426   double omega2 = Y[3];
427 
428 
429   double g = G_ACCg;
430   double den = 2. * M1 + M2 - M2 * cos(2. * theta1 - 2. * theta2);
431 
432   R[0] = omega1;
433   R[1] = omega2;
434   R[2] = (-g * (2. * M1 + M2) * sin(theta1) - M2 * g * sin(theta1 - 2. * theta2) - 2. *
        sin(theta1 - theta2) * M2 * (omega2 * omega2 * L2 + omega1 * omega1 * L1 * cos(theta1
        - theta2))) / (L1 * den);
435   R[3] = (2. * sin(theta1 - theta2) * (omega1 * omega1 * L1 * (M1 + M2) + g * (M1 + M2) *
        cos(theta1) + omega2 * omega2 * L2 * M2 * cos(theta1 - theta2))) / (L2 * den);
436 
437 }
438 
439 
440 
441 void RK4Step(double t, double *Y, double h, void (*RHSFunc)(double, double *, double *),
      int neq){
442 
443   double Y1[neq], k1[neq], k2[neq], k3[neq], k4[neq];
444 
445   RHSFunc(t, Y, k1);
446 
447   for(int i = 0; i < neq; i++) Y1[i] = Y[i] + 0.5 * h * k1[i];
448 
449   RHSFunc(t + 0.5 * h, Y1, k2);
450 
451   for(int i = 0; i < neq; i++) Y1[i] = Y[i] + 0.5 * h * k2[i];
452 
453   RHSFunc(t + 0.5 * h, Y1, k3);
454 
455   for(int i = 0; i < neq; i++) Y1[i] = Y[i] + h * k3[i];
456 
457   RHSFunc(t + h, Y1, k4);
458 
459   for(int i = 0; i < neq; i++) Y[i] += (h / 6.) * (k1[i] + 2. * k2[i] + 2. * k3[i] + k4[i
        ]);
460 
461 
462 }
463 
464 double Energy(double theta1, double theta2, double omega1, double omega2){
465 
466   return .5 * M1 * L1 * L1 * omega1 * omega1 + .5 * M2 * (L1 * L1 * omega1 * omega1
467       + L2 * L2 * omega2 * omega2 + 2 * L1 * L2 * omega1 * omega2 * cos(theta1 - theta2))
468       - (M1 + M2) * G_ACCg * L1 * cos (theta1) - M2 * G_ACCg * L2 * cos (theta2);
469 }
```