MSc Control and Robotics
Academic year 2023 - 2024

SOROMO Lab1 : Numerical Methods

Raffaele Pumpo
Emanuele Buzzurro
Under the supervision of Andrea Gotelli

Contents

1 Root finding method

Root-finding methods play a crucial role in solving complex mathematical problems associated with the behavior of soft robotic systems. These methods are algorithms designed to determine the zeros or roots of a function, particularly in the context of soft robotics, where intricate systems are described by a set of parameters governing the system's state and properties.

The foundation of root-finding methods relies on three essential elements: the set of parameters, the residual vector, and the Jacobian matrix. The set of parameters, denoted as n, represents the variables characterizing the soft robotic system. These variables encompass the system's state and properties, such as generalized coordinates, providing a comprehensive description of the system's configuration.

The residual vector is a critical component containing all the constraints that the soft robotic system must satisfy. Constraints are typically expressed as equations that enforce specific properties, with the residual vector's value reaching zero when these properties are met. Therefore, achieving a residual vector containing all zeros indicates the satisfaction of the imposed constraints.

There are multiple algorithms for implementing the root finding, and the one used in this laboratory is the Newton-Raphson method, which is an iterative numerical technique used to find the root of a real-valued function. The basic idea behind the method is to iteratively refine an initial guess for the root by using the tangent line of the function at that point. Here is a brief overview of the Newton-Raphson iterative process:

1. **Initialization:** Start with an initial guess x_0 for the root.
2. **Iteration**:
   - At each iteration k, calculate the function value $f(x_k)$ and its derivative $f'(x_k)$ at the current guess x_k. We can also define the Jacobian matrix (J), which serves as a mapping of the influence of the parameters on the residual vector. Mathematically, it is defined as:

$$J = \frac{\partial R(x)}{\partial \vec{x_i}} \ \in \ R^n \text{for all (i) in the range [1, n]}$$

It is clear that:

$$J = f'(x_k) \text{ and } R = f(x_k)$$

- Update the guess for the root using the formula:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

  Indeed, the Jacobian matrix becomes instrumental in updating the set of parameters iteratively, as described by the equation:

$$x_{\text{update}} = -J^{-1}R(x)$$

  The interpretation of this equation is pivotal: mapping the residual vector through the inverse Jacobian matrix yields the parameters that increase the residual vector (direct relation). Inverting the sign provides the parameters that decrease the residual vector.

3. **Termination:** Repeat the iteration until a stopping criterion is met. This criterion is often based on the magnitude of $f(x_k)$ becoming sufficiently close to zero.

The formula $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ is derived from linear approximation, using the tangent line at the current guess to estimate the root. The method is efficient when the initial guess is reasonably close to the actual root and when the function has continuous derivatives.

The Newton-Raphson method can converge rapidly for well-behaved functions, but it may diverge or fail to converge for certain cases or when the initial guess is far from the true root. It may also find different roots, with respect to the initial guess.

The computation of the Jacobian matrix can be approached through two methods:

1. **Analytical Formulation:** If an analytical formulation is available, the Jacobian matrix can be computed analytically by applying the appropriate mathematical expression to the residual vector.
2. **Numerical Computation:** Alternatively, the Jacobian matrix can be computed numerically by applying specific formulations to calculate it, that will be explained right after.

The root_finding function aims to find the root of a given function f using different methods. The methods include an analytical method, a numerical forward method, and a numerical central method. The choice of method is controlled by the parameter method.

The code employs an iterative process to approximate the root of the function.

The iteration continues until the norm of the residual vector (R) becomes sufficiently small or a maximum number of iterations (1000 in this case) is reached.

Therefore, a predefined threshold, typically denoted as ϵ, is utilized to determine when the norm is satisfactorily small, with a common choice being $\|R\| < \epsilon$. In the context of this lab,the choice for ϵ is 1×10^{-6}.

1.1 Analytical computation of the Jacobian

The analytical method for root finding is a powerful approach utilized when the function and its derivative are known and can be expressed in closed-form analytical expressions. In the provided MATLAB code, the analytical method can be activated by selecting: method == 1.

In this method, the Jacobian matrix, which represents the partial derivatives of the residual vector with respect to the parameters, is calculated directly from the known derivatives of the function. The calculated Jacobian matrix is then applied in the root-finding iteration to update the parameter values. After that, the update is performed using the formula:

$$x_{\text{update}} = -\text{pinv}(J) \cdot R$$

where pinv is the pseudo-inverse function. The analytical method is advantageous when the mathematical expressions for both the function and its derivative are available. It allows for precise calculation of the Jacobian matrix, potentially leading to faster convergence in the root-finding process. This method is particularly useful in scenarios where an analytical solution is feasible and efficient.

1.2 Numerical computation of the Jacobian

In contrast to analytical methods, numerical methods for root finding are employed when the mathematical expressions for the function or its derivatives are not available in closed form. These methods rely on iterative procedures to approximate the root of a function by calculating numerical derivatives or finite differences. In the provided MATLAB code, the numerical methods include the numerical forward method (method == 2) and the numerical central method (method == 3).

1.2.1 Forward

The forward difference method is a numerical technique used to approximate the derivative of a function at a given point. Specifically, it approximates the first derivative using information from the current point and a nearby point ahead in the direction of the independent variable.

The forward difference formula for the first derivative is given by:

$$J \approx \frac{R(x+h) - R(x)}{h}$$

where:
R is the residual value computed at a point x and a bit after $x + h$ with h is a small step size.

1.2.2 Central

The central difference method is another numerical technique for approximating derivatives, and it uses information from points on both sides of the target point. It provides a more accurate estimation by considering points symmetrically around the target point.
The central difference formula for the first derivative is given by:

$$J \approx \frac{R(x+h) - R(x-h)}{2h}$$

where:
R is the residual value computed at a point $x + h$ and before $x - h$ with h is a small step size.

2 Functions

Since the Jacobian matrix represents the partial derivatives of the residual vector with respect to the parameters. For the given function:

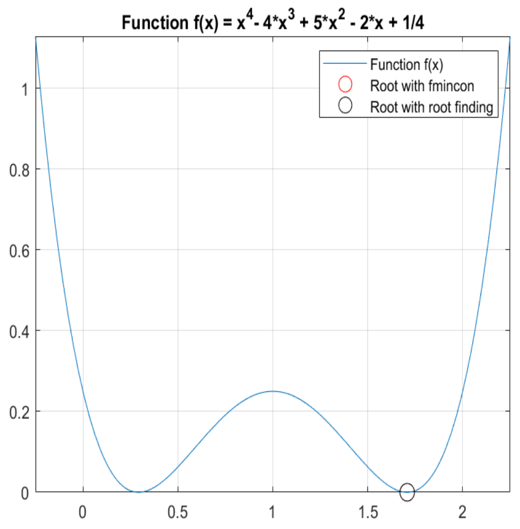
$$f(x) = x^4 - 4x^3 + 5x^2 - 2x + \frac{1}{4}$$

the analytical expression for its derivative, which constitutes the Jacobian matrix, is computed. The Jacobian matrix (J) is defined as a function of the parameter x in the script:

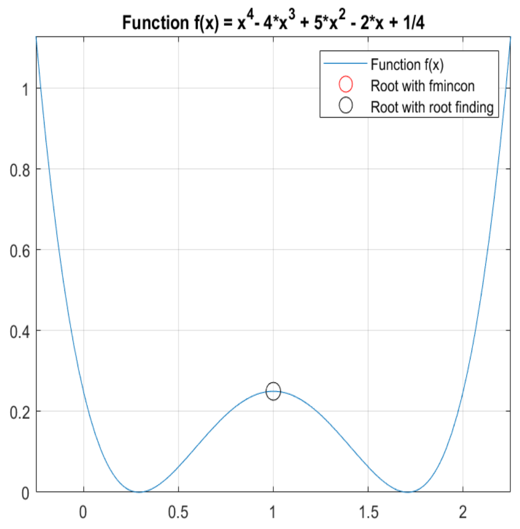
$$J(x) = 4x^3 - 12x^2 + 10x - 2$$

This expression corresponds to the derivative of the given function with respect to x.

Below, we can find the first plot, obtained with the analytical method:



(a) Starting point $x_0 = 6$



(b) Starting point $x_0 = 1$

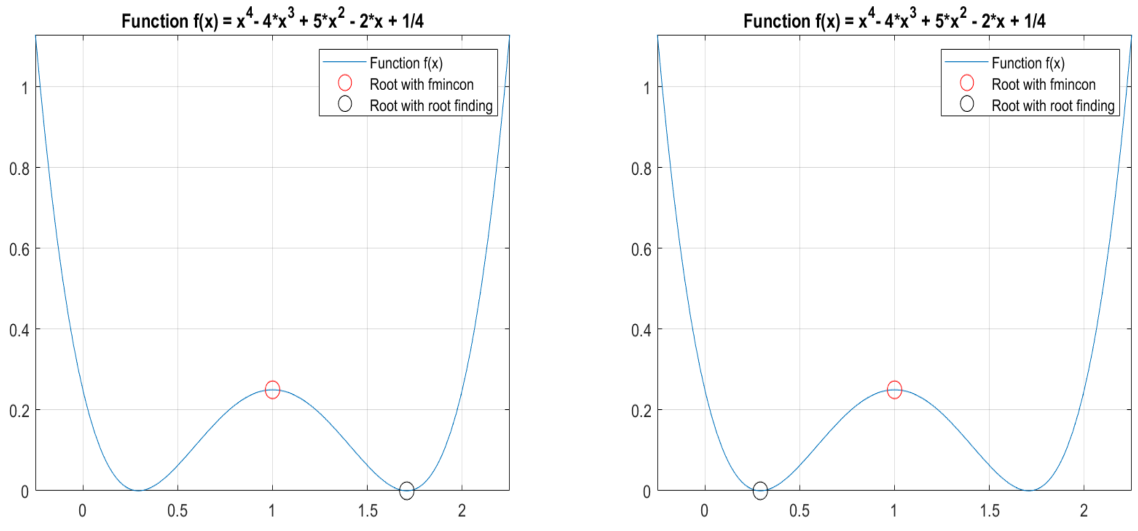
Figure 1: Analitycal Jacobian computation

It has been tested for two different starting points, and it is clear that in the left case (x_0 = 6) the root-finding technique, is able to correctly detect the root, which corresponds to the point where the function intersects the x-axis and it also coincide with the root founded by the $fmincon$ algorithm.

While for the particular case, with starting point of $x_0 = 1$, since this methods is based directly on evaluating the derivative of the function in a specific point, it already recognizes that the computed value is zero, bringing it to the wrong conclusion, mistaking the root with the saddle point of the function.

Here it is clearly visible the importance of the starting point, for this specific algorithm, that can lead to different solution, and also to wrong ones.

The successive plots shows how, the two different methodologies of the numerical computation approach, performs in similar scenarios:



(a) Forward computation with $x_0 = 1$ (b) Central computation with $x_0 = 1$

Figure 2: Numerical Jacobian Computation

The forward difference formula for the derivative used in the numerical forward method is:

$$J = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

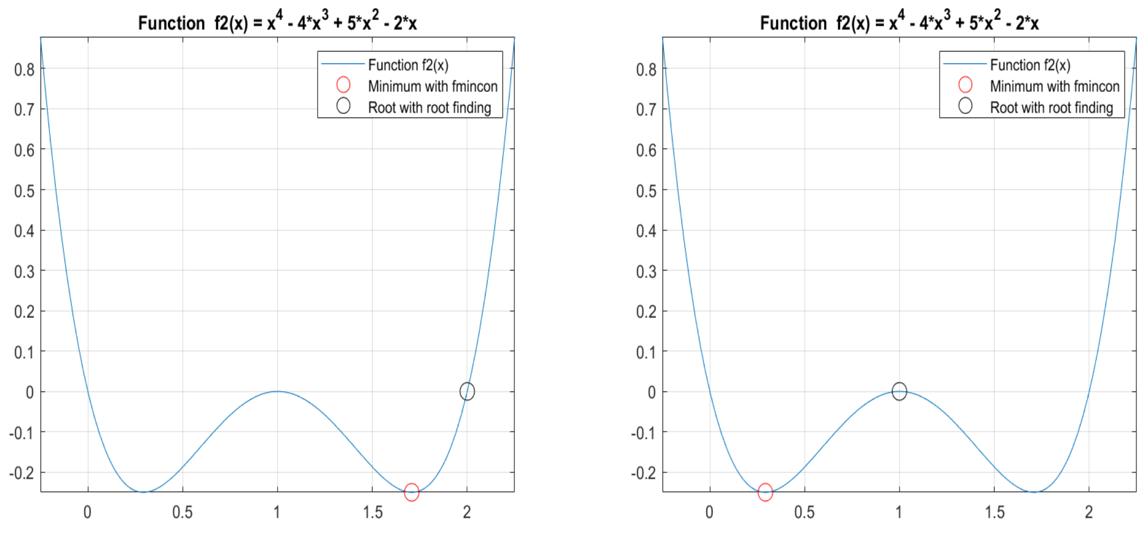
Here, Δx is a small step size. When $x_0 = 1$, the calculation for the forward method involves evaluating the function at $x_0 + \Delta x$ and x_0. Since the function is monotonically decreasing in the neighborhood of $x_0 = 1$, the forward difference is negative, indicating that the function is decreasing. This negative slope suggests that moving in the positive direction from $x_0 = 1$ will lead towards a root.

The central difference formula for the derivative used in the numerical central method is:

$$J = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}$$

In this case, the central method evaluates the function at $x_0 + \Delta x$ and $x_0 - \Delta x$. The function is non-monotonic, and the slope may be negative in one direction and positive in the other. When $x_0 = 1$, the central difference may capture the change in slope around x_0 and indicate the presence of a root in the opposite direction.

The last experiment, has been done using the same function, but translated along the y axis of 1/4 in the negative direction, as shown below:



(a) Starting point $x_0 = 6$ (b) Starting point $x_0 = 0.7$

Figure 3: Minimum and root in f2(x)

Here, it is observable that with the root finding technique, we are always able to find a root, that always depends on the starting point; while the fmincon looks for the min of the function, which in this case does not correspond with its root.
We can observe in both cases, in fact the method finds the roots closest to the starting point and not the minimum finds through $fmincon$. The second starting point ($x_0 = 0.7$) has a negative value , smaller value of the residual, but the norm of it is greater and the algorithm goes to the root ($x = 1$) and not the minimum.

2.1 Difference and Comparison

- Analytical
  - Smallest computation time as the derivatives are directly expressed.
  - May encounter challenges with certain points due to a lack of information about the neighborhood, as we have seen in the first problem with saddle points.
- Forward Numerical
  - Middle ground between analytical and central methods in terms of time and accuracy.
  - Offers a balance between computational efficiency and accuracy.

- Central Numerical
  - Slower in terms of computation time.
  - Provides the most accurate results.

  Application:
  -Analytical method is efficient when function values are directly available, it is fastly computed, but may lead to errors.

  -Forward difference is often used when function values are available only in the forward direction or when computational efficiency is crucial, it is a good trade-off between efficiency and accuracy.

  -Central Difference is preferred when accuracy is paramount. Suitable when function values are available on both sides of the target point.

3 Numerical Integrations

In the field of robotics, numerical integration is indispensable for simulating and understanding the behavior of robotic systems. This tool facilitates the exploration of system dynamics, enabling to predict the outcomes of various control strategies and designs.
Having the mathematical model of a system, often is necessary a numerical integration.
This process involves discretizing time and numerically solving the set of ODEs that describe the system's evolution over time. The results of time integration provide valuable information about the dynamic behavior, stability, and response of the system under various conditions.
In the context of this laboratory, the focus narrows down to two primary methods of numerical integration: explicit and implicit time integrations.

3.1 Explicit method

Explicit numerical integration method is a technique used to solve Ordinary Differential Equations (ODEs) by computing the future state variables based on the current state.
In this method given only the initial configuration and the ODE, is computed the evolution of the system for each time instant given an interval of time analysed.
The pipeline for this method is the following:

- Discretize time into small steps, given the initial time and the final one.
- Define the initial configuration of the system.
- Compute the next state for each time instant using a specific method.

The method chosen for computing the next state is the Runge-Kutta method through *ode45*. The general form of a Runge-Kutta method involves evaluating the derivative of the solution at several points within each time step. The idea is to use a weighted average of these derivatives to estimate the change in the solution over the time step.
For a single ordinary differential equation $dy/dt = f(t, y)$, the method involves the following steps:

- Calculate the slope at the initial point.
- Predict intermediate values using the initial slope and compute for each one a slope.
- Combine these slopes to obtain the next state.

3.2 Implicit method

The implicit integration is different from the explicit method because it consists in defining, not directly explicit equation ODE, but in writing an implicit equation called *residual*, that should be equal to 0, and try to find iteratively the values that satisfy the equation.
Doing this we consider not only the current state but also the future one through a prevision. We can consider the following pipeline:

- **Prediction**: at every time step, an estimation of the current state is attempted based on the state at the previous time step. This estimation might not be accurate.
- **Root Finder-Correction**: to correct the inaccuracies in the estimated state, a root-finding method, such as Newton-Raphson, is employed.
- **Repeat**: once a solution is found, the process is repeated for the next time step. The currently available information is used to estimate the state for the next time step.

In summary, the implicit time integration method combines numerical integration with a root-finding algorithm to iteratively estimate the system's dynamic state, correcting for inaccuracies at each time step.

The advantages and drawbacks of the methods are analysed successively, once applied to some systems.

4 Spring Pendulum

In this part of the lab a physical example has been analysed: Spring Pendulum.
It is simply a revolute joint and a prismatic joint with a spring, and we can visualize it in the figure below:

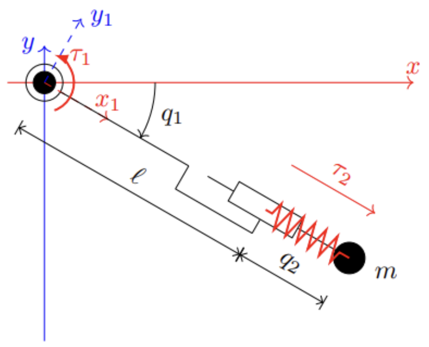


Figure 4: Representation of the Spring Pendulum

The spring pendulum system is described by the following expressions for kinetic (E) and potential (U) energies:

$$E = \frac{1}{2}m\dot{q}_2^2 + \frac{\dot{q}_1^2}{2}\left(\frac{1}{(l+q_2)^2}\right)$$

$$U = mg(l+q_2)\sin(q_1) + \frac{1}{2}kq_2^2$$

Here, E and U represent the kinetic and potential energies of the system, respectively. The system variables include q_1 (revolute joint angle), q_2 (prismatic joint translation), τ_1 (torque of the revolute joint), and τ_2 (force at the prismatic joint). The parameters m, g, k, and l denote the mass at the end of the prismatic joint, gravity constant, spring constant, and the fixed distance between the revolute and prismatic joints, respectively.

In the context of the simulation, positive values for q_1 and τ_1 signify counterclockwise rotation, while positive values for q_2 and τ_2 indicate motion along the x_1 direction. For numerical simulations, the following parameters are considered: $l = 1\,\mathrm{m}$, $m = 1\,\mathrm{kg}$, $k = 10\,\mathrm{N}\cdot\mathrm{m}$, $g = 9.81\,\mathrm{kg}\cdot\mathrm{m/s}^2$.

4.1 Implicit

The provided code implements an implicit time integration method to simulate the intricate dynamics of a spring pendulum system. At its core, the algorithm relies on the Implicit

Dynamic Model (IDM) to determine the efforts (τ) acting on the system, aiming to reconcile these with the desired efforts (τ_d). This reconciliation process involves an iterative approach, relying on the Newton-Raphson method, to consistently find the correct set of parameters that satisfy the system dynamics ($\tau = \tau_d$).

The heart of the algorithm lies in the prediction and correction routines, as depicted in the following Equations:

$$\begin{aligned} q_{n+1} &= q_n + h\dot{q}_n + (1/2 - \beta)dt^2\ddot{q}_n, \\ \dot{q}_{n+1} &= \dot{q}_n + (1 - \gamma)dt\ddot{q}_n, \\ \ddot{q}_{n+1} &= 0. \end{aligned} \tag{1}$$

The prediction routine employs information from the previous time step to estimate the state for the next time step, and it lays the groundwork for the subsequent nonlinear solver iterations.

The nonlinear solver, encapsulated in the `implicit_residual` function, is where the Jacobian mapping is applied to update the parameters (*qupdate*) in accordance with Equation:

$$qupdate = -J^{-1}R. \tag{2}$$

This update is instrumental in computing the next set of [q, q', q"], labeled as $k + 1$, and serves as the bridge between the nonlinear solver and the correction routine.

The correction routine adjusts the state based on the parameters obtained from the nonlinear solver. In the provided code, this is accomplished in the `root_finding` function, where the iterative process ensures that the system dynamics are consistently satisfied throughout the simulation.

Moreover, the visualizations included in the code, such as the plotting of joint values and the dynamic representation of the spring's position, offer a tangible connection to the evolving system behavior over time. These visualizations provide insights into how the implicit time integration method maintains stability and accuracy in capturing the complex dynamics of the spring pendulum system.

4.2 Explicit

The provided code implements an explicit time integration method to simulate the dynamics of a spring pendulum system. Explicit time integration involves integrating the system's Ordinary Differential Equations (ODEs) over time, where both the first and second derivatives are considered. The state $y(t)$ is defined in terms of generalized coordinates and their derivatives, and the integration is performed explicitly on $y(t)$.

The algorithm employs an ODE solver, such as the ode45 function in MATLAB, to perform explicit time integration. The DDM (Dynamic Differential Model) function, defined by Equations:

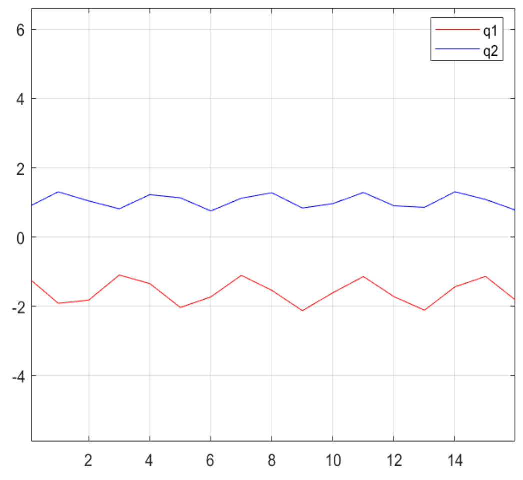
$$\ddot{q} = M^{-1}(q)(\tau_d - C(q, \dot{q}) - K(q))$$

is passed to the solver. The DDM function decomposes the state y into q and $\dot{q}$, computes the second derivatives $\ddot{q}$ using the DDM, and composes $\dot{y}$ from $\dot{q}$ and $\ddot{q}$.

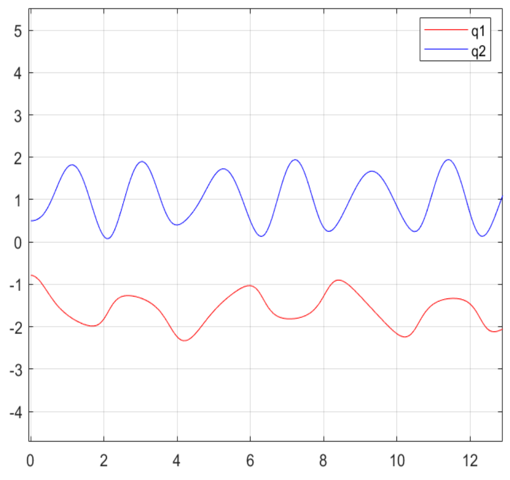
$$\begin{aligned} \dot{y}(t) &= \begin{bmatrix} \dot{q}(t) \\ \ddot{q}(t) \end{bmatrix}, \\ \text{where} \quad \dot{q}(t) &= \dot{q}(t), \\ \ddot{q}(t) &= \text{Compute from DDM.} \end{aligned} \tag{3}$$

4.3 Comparison

In the pursuit of numerical simulations for dynamic systems, the choice between implicit and explicit time integration methods plays a crucial role. In this section, we delve into a comparative analysis of these methods based on their execution times and the accuracy of joint variable predictions. By exploring the behavior of joint variables at different time step (DT) values, which are shown in the figures below:

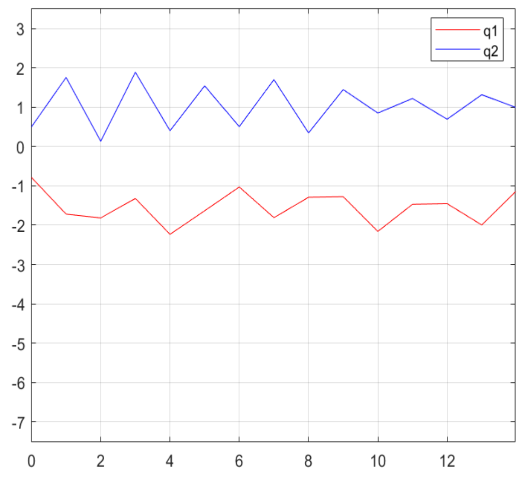


(a) Joint Variables for DT = 1

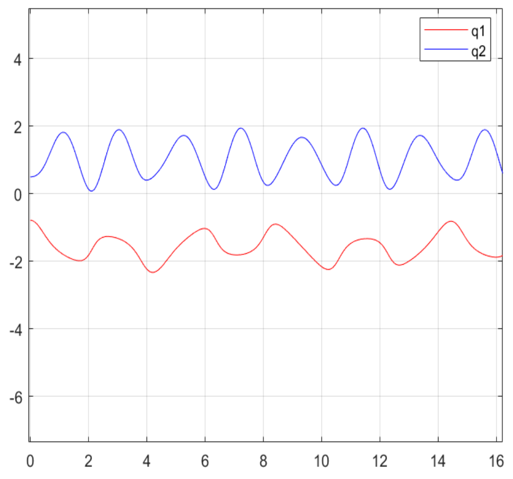


(b) Joint Variables for DT = 0.01

Figure 5: Implicit time integration



(a) Joint Variables for DT = 1



(b) Joint Variables for DT = 0.01

Figure 6: Explicit time integration

Where it is observable that both implicit and explicit methods yield consistent results, portraying similar system dynamics in joint variable plots. An important role is played by the step size variable, indeed, smaller time steps allow for a more detailed and accurate representation of the system's behavior. The smoother plots at smaller DT values indicate a more refined simulation capturing finer details of the dynamics.

However, it's essential to consider the trade-off between accuracy and computational cost. Smaller time steps increase the computational load, as seen in execution time results shown right after.

Another way to compare the two methodologies, since both seems to correctly reach the solution, can be based on their time of execution, which have been provided in the table below:

Step size	**Implicit**	**Explicit**
1	0.2586 seconds	0.24964 seconds
1e-2	98.0278 seconds	0.79022 seconds

Table 1: Time of execution of the algorithms for different step size for 1000s of simulation

The table shows that, for the given system and conditions, the implicit method consistently takes more time than the explicit method for the same step size. The difference in execution times becomes more pronounced as the step size decreases. This aligns with the typical behavior of implicit methods, which often involve solving nonlinear equations and require iterative processes.

Moreover, as expected, smaller step sizes generally result in longer execution times for both implicit and explicit methods. This is because a smaller step size requires more iterations, leading to increased computational time.

In the end, taking into account that results may vary, with respect to the machine and conditions, it might be beneficial to explore a wider range of step sizes and analyze the trends in execution times to identify an optimal step size for both methods.

5 Cosserat rod

In this part of the lab another physical example has been analysed: *Cosserat rod*.
The Cosserat rod, also known as the Kirchhoff rod or elastic rod, is a mathematical model used in the field of continuum mechanics to describe the behavior of slender, flexible structures like rods, fibers, or filaments. It provides a comprehensive framework for analyzing the complex behavior of slender structures, taking into account both translational and rotational effects.
Following, the representation:

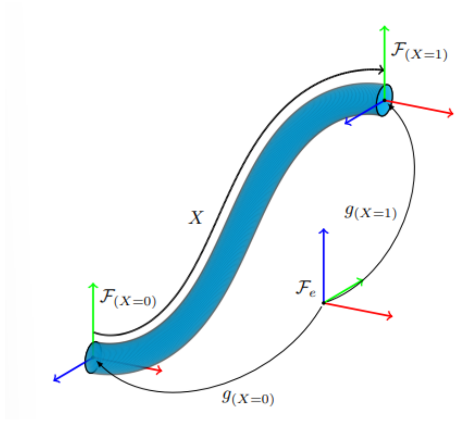


Figure 7: Representation of the Cosserat rod

We know that the Dynamic model can be written as the following formula:

$$K_{\epsilon\epsilon}q + D_{\epsilon\epsilon}\dot{q} - Q_a(X=0) = 0$$

with:

- $K_{\epsilon\epsilon}q$: the elastic response of the system to deformations.
- $D_{\epsilon\epsilon}\dot{q}$: the damping forces that resist the motion of the system.
- $Q_a(X=0)$: forces acting on the system at the base of the rod.

The forces Q_a can be expressed as following for a generic position in the cosserat rod:

$$M\ddot{q} + Q_v + Q_c = Q_a$$

with:

- $M\ddot{q}$: the inertial forces.
- Q_v: additional forces due to Coriolis and centrifugal effect.
- Q_c: additional applied forces dependent to the configuration.

5.1 Implicit

In this part, as previously said, an implicit equation, *Residual*, must be defined, then we should apply a Root finding technique.
In the cosserat rod we have defined it, through the function *Residualcosserat()* simply as the dynamic model of the system, as following:

$$R = K\varepsilon\varepsilon_q + D\dot{\varepsilon}\varepsilon_q - Q_a(X=0)$$

Then the Newton-Raphson is applied, and it needs the Jacobian for the correction part.
The *Jacobian* can be computed, through *Jacobiancosserat()* with the "Analytical", Forward numerical and Central numerical methods, choosing respectively the parameter "method" with value 1, 2 or 3.
The first method is also a numerical method, in fact a small variation in the value of the Dynamic model, which is the residual chosen, is applied as following:

$$J^{[i]} = K\varepsilon\varepsilon\Delta q_i + D\dot{\varepsilon}\varepsilon\Delta\dot{q}_i - \Delta Q_a^{[i]}$$

In this case the Jacobian is computed one column at time, applying a variation to the specific joint correspondent to that column, and propagating the variation in its velocity and acceleration.
Having those values, the forces acting $\Delta Q_a^{[i]}$ on the system can be computed, and finally the specific column of the Jacobian.

The forward and central methods are the usually previously explained:

$$J \approx \frac{R(x+h) - R(x)}{h} \qquad J \approx \frac{R(x+h) - R(x-h)}{2h}$$

Having the Residual and the Jacobian, the Newton-Raphson is able to predict and correct, until the root is found.

5.2 Explicit

In the explicit method, instead, we need to define a specific ordinary differential equation in such a way that the function *ode45()* can compute the different values of the state for each time instant.
The ODE function is the Direct dynamic model of the cosserat rod, that relates the accelerations to the positions and velocities of the joints, as following:

$$\ddot{q} = -pinv(M) * (Q_v + Q_c - Q_a)$$

Transformed in ordinary differential equations:

$$\begin{cases} \dot{y}_1 = y_2 \\ \dot{y}_2 = -\text{pinv}(M) \cdot (Q_v + Q_c - Q_a) \end{cases} \quad \text{with} \quad \begin{matrix} y_1 = q \\ y_2 = \dot{q} \end{matrix}$$

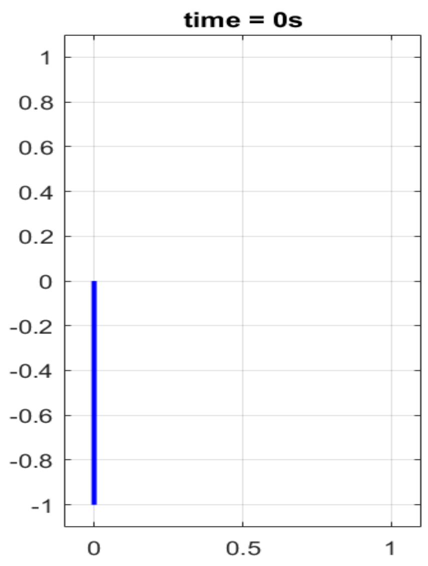
The matrices M, Q_v, Q_c, Q_a, necessary for the direct dynamic model, have been computed through the inverse dynamic model called f considering different values for positions, velocities and accelerations of the joints, as following:

- Q_a : 0 considering no forces
- Q_c : considering no velocities and accelerations but only the positions $\implies \mathrm{Q}_c = \mathrm{f(q)}$
- Q_v : considering no accelerations $\implies \mathrm{Q}_v = f(q, \dot{q}) - Q_c$
- M : Considering positions, velocities and, one at time, one component of the acceleration = 1 all the others = 0, in such a way to estimate each column of M

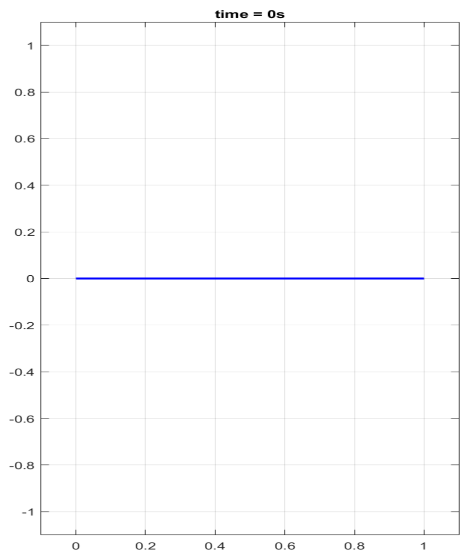
5.3 Comparison

In the lab different parameters and initial conditions have been tested for both integration method.

The initial conditions chosen are the quaternions which correspond to these configurations of the cosserat rod respectively:



(a) $X_0 = 0.7071068, 0, 0.7071068, 0$



(b) $X_0 = 1, 0, 0, 0$

Figure 8: Initial configurations

Following, we have reported the two initial conditions and the relative time of execution of the algorithm:

Initial Configuration	**Implicit**	**Explicit**
0.7071068, 0, 0.7071068, 0	5.8227 seconds	5.6295 seconds
1, 0, 0, 0	46.239 seconds	∞

Table 2: Time of execution of the algorithms for different initial conditions

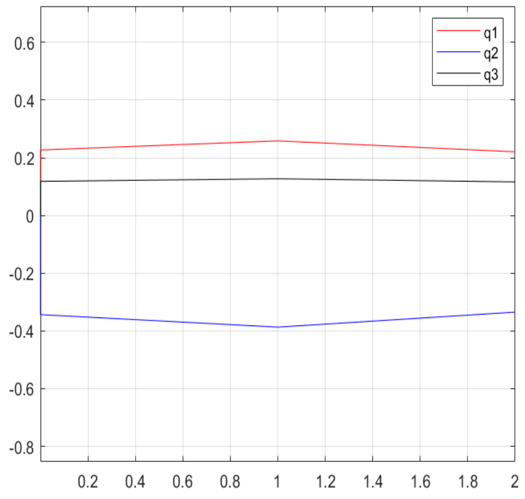
Observing the table that corresponds to a step size dt = 1e-2, we can easily see that the explicit method usually, as in the first configuration, can be considered faster and easier than the implicit method.
For this reason we can conclude that it could be better, but, by simply changing the configuration, as in the second case we obtain an infinite loop with that method.
The infinite loop that we obtain with the explicit method is due to the stiffness model of the system, that means it is highly sensitive to small variation of the state, which implies small variations in the state bring to big variations of the output.
In other words, we can see from a mathematical point of view this phenomenon, in this method in fact we obtain the evolution of the state, as previously said, from this equation:

$$\ddot{q} = -pinv(M) * (Q_v + Q_c - Q_a)$$

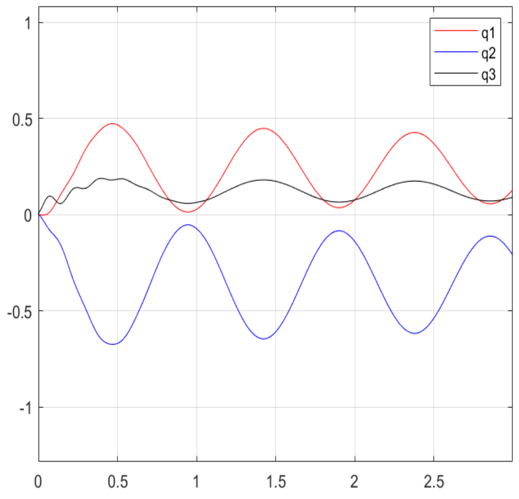
We can see easily that if the matrix M is not invertible, det(M) almost 0, we will have infinite possible values of the accelerations for any forces. We have tried in this case to plot the value of the determinant of M that is **det(M) =e-13/e-14**, then it could be considered

numerically 0.
Finally, we can conclude for the explicit method that it could be unstable and the value of the step size dt is fundamental for stability and precision, but in some cases a dt, though quite small, can't allow the stability, as in the second configurations.

For the implicit method we have tried also to change the value of the step size dt, as following:



(a) Joint values with dt = 1



(b) Joint values with dt = e-2

Figure 9: Joint values of the Cosserat rod with implicit integrator

And we have obtained the following execution time:

Initial Configuration	**dt = 1**	**dt = 1e-2**
1, 0, 0, 0	1.6964 seconds	43.054 seconds

Table 3: Time of execution of the algorithm for different step size

We can conclude that, unlike the explicit method, defining a residual, an implicit function will not cause instability for any configuration and values of the step size, this method can be considered as intrinsically stable.
The choice of the value of the step size must be done, not for stability, but as a trade-off between the execution time of the algorithm and the precision that we want to describe the dynamic behaviour of the system (Positions, velocities and accelerations).

6 Conclusion

In conclusion, the exploration of root-finding techniques, specifically the Newton-Raphson method, has provided valuable insights into finding roots efficiently. The combination of analytical and numerical algorithms, such as forward and central differences, enhances our ability to tackle diverse problems, offering a spectrum of computational efficiency and accuracy.

In the context of numerical integration, the explicit and implicit methodologies have been examined in detail, shedding light on their respective advantages and trade-offs.
The spring pendulum system exemplifies the stability and accuracy of explicit integration, while the Cosserat rod introduces the intricacies of implicit integration in the context of a more complex physical model.
Through this investigation, a comprehensive understanding of both root-finding and numerical integration techniques has been achieved, providing a solid foundation for addressing a wide range of dynamic systems in various scientific and engineering domains.