Exploration of Time Integration Schemes for Ordinary Differential Equations with Application to Equations of Motion

Correa, Alan Jason

Abstract

Differential Equations commonly appear in most science and engineering domains. Although analytical solutions to these problems may exist, they are often complicated and can only be determined for fairly simple problems. Furthermore, to model real world engineering problems with sufficient accuracy, differential equations from multiple scientific domains are coupled and solved simultaneously. In such cases, it is almost impossible to analytically solve these differential equations (e.g. Navier-Stokes Equations). However, using numerical techniques, approximate solutions that satisfy the differential equations along with the prescribed boundary and initial conditions could be found. Thus, in the recent past, due to the rise and widespread availability of powerful computing resources, numerical approximations to differential equations have become the preferred way forward.

Numerical time-integration schemes are commonly used for solving initial value problems. Most engineering solver codes already implement these schemes in an efficient manner. However, the aim of this document is to explore and understand time-integration schemes and their algorithms that are already available in literature. Furthermore, a reference implementation of these algorithms in a programming language is sought after. Finally, an attempt is made to apply these algorithms to solve the equations of motion for mechanical systems.

Contents

1. Introduction **1**
1.1. Initial Value Problems 1
1.2. Numerical Time Integration Schemes 1
1.2.1. Explicit Euler Method 1
1.2.2. Implicit Euler Method 2
1.2.3. Comparison of Euler Time Integration Schemes 6
1.3. Solution to Example Problems 7

2. Solution Techniques for Equation of Motion **8**
2.1. Equation of Motion 8
2.2. State-Space Representation 9
2.3. Scalar Solution Schemes 10
2.3.1. Explicit Euler Method 10
2.3.2. Semi-Implicit Euler Method 10
2.4. Vectorized Solution Schemes 11
2.4.1. Explicit Euler Method (Vectorized) 11
2.4.2. Implicit Euler - Iterative Convergence Method 11
2.4.3. Implicit Euler - Matrix Inversion Method 12
2.5. Example Problems - Derivation of Equation of Motion 13

A. Python Code Examples **14**
A.1. Numerical Time Integration Schemes 14
A.1.1. Explicit Euler Method 14
A.1.2. Implicit Euler Method - Analytical Reformulation 14
A.1.3. Implicit Euler Method - Matrix Inversion 15
A.1.4. Implicit Euler Method - Fixed Point Iteration 16

1. Introduction

1.1. Initial Value Problems

Initial value problems are nothing but differential equations that require finding solutions, given the initial conditions at time t_0.

$$\dot{y} = f(t, y) \; with \; y_0 = y(t_0) \tag{1.1}$$

The problem as described in eq. (1.1) is an *ordinary differential equation* of the first order. One can determine the solution analytically using eq. (1.2).

$$y(t) = y_0 + \int_{t_0}^{t} f(\tau, y(\tau))\, d\tau \tag{1.2}$$

In this case, the integral in eq. (1.2) needs to be determined in order to find the solution for time interval of interest. Though it may seem easy, it is often a herculean task to evaluate this integral, particularly when $f(\tau, y(\tau))$ is a non-linear function.

Furthermore, initial value problems with *higher order* differential equations can be transformed into a system of differential equations and expressed in a vectorized form as shown below.

$$\dot{Y} = F(t, \mathbf{A}, Y, B) \; with \; Y_0 = Y(t_0)$$

$$\text{where, } Y = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{pmatrix}, \; \dot{Y} = \begin{pmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \\ \vdots \\ \dot{y}_n(t) \end{pmatrix}, \; Y_0 = \begin{pmatrix} y_1(t_0) \\ y_2(t_0) \\ \vdots \\ y_n(t_0) \end{pmatrix} \tag{1.3}$$

A solution to the formulations as described in eq. (1.3), where $\mathbf{A}$ is the System Matrix and B is the Excitation Vector, using a procedure like in eq. (1.2) is complicated often not possible. Hence, it is sought to approximate them numerically using methods that will be discussed hereafter.

1.2. Numerical Time Integration Schemes

1.2.1. Explicit Euler Method

The Explicit Euler time integration scheme was the first method developed to solve initial value problems. In this scheme, the time derivative $\dot{y}(t)$ in eq. (1.1) is calculated using the *forward difference method.* Therefore, as we march forward in time, the new values of the solution are computed using only known values of the solution at the previous time step [see FS01, pg. 97].

In short, the Explicit Euler time integration scheme can be summarized as

$$y_{n+1} = y_n + hf(t_n, y_n) \qquad n = 0, 1, \dots \tag{1.4}$$

where, $y_0 = y(t_0)$ is the initial condition and $h = t_{n+1} - t_n$ is the time increment interval.

The algorithm for procedural implementation of the Explicit Euler time integration scheme is as shown below:

Algorithm 1: Explicit Euler Time Integration

```
Data: $h, n, f(t, y), y_0, t_0$
Result: $y_{exp} = \{y_0, y_1, \dots, y_n\}$ at $t = \{t_0, t_1, \dots, t_n\}$
1 Initialize arrays $y_{exp} = \{y_0\}$ and $t = \{t_0\}$;
2 $y_i \leftarrow y_0$ and $t_i \leftarrow t_0$;
3 for $i \leftarrow 0$ to $n$ do
4     $y_{i+1} = y_i + hf(t_i, y_i)$;
5     $t_{i+1} = t_i + h$;
6     Append $y_{exp} \leftarrow y_{i+1}$; Append $t \leftarrow t_{i+1}$;
7 end
8 return $y_{exp}, t$
```

Although this method is fast and can implemented numerically with ease, it is only *conditionally stable* [see FS01]. An implementation of the Explicit Euler time integration scheme in python can be found in Appendix A.

1.2.2. Implicit Euler Method

The Implicit Euler time integration scheme is used to alleviate the stability issues of the Explicit Euler Method. In this scheme, the time derivative $\dot{y}(t)$ in eq. (1.1) is calculated using the *backward difference method.* Therefore, as we march forward in time, the new values of the solution cannot be computed explicitly using known values of the solution at the previous time step [see FS01, pg. 105].

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \qquad n = 0, 1, \dots \tag{1.5}$$

where, $y_0 = y(t_0)$ is the initial condition and $h = t_{n+1} - t_n$ is the time increment interval.

It is evident from eq. (1.5) that calculating the solution at the next time step is not straigt forward. However, using the a few mathematical tools that are explined further, the Implicit Euler time integration scheme can be numerically solved.

Analytical Reformulation

Given eq. (1.1), the derivative of the function at every time step can be calculated. Thus, for the time t_{n+1} the we have

$$\dot{y}_{n+1} = f(t_{n+1}, y_{n+1}) \tag{1.6}$$

Furthermore, eq. (1.5) can be rewritten as

$$y_{n+1} = y_n + h\dot{y}_{n+1} \tag{1.7}$$

Now by combining eq. (1.6) and eq. (1.7), the Implicit Euler time integration scheme can be formulated using only the values of solution y_n at previous time step. [see And15]

$$y_{n+1} = y_n + f(h, t_{n+1}, y_n) \tag{1.8}$$

As an example of this method, the Implicit Euler time integration scheme for eq. (1.9) can be formulated as shown in eq. (1.10) below.

$$\dot{y} = \lambda(y - \sin(t)) + \cos(t) \tag{1.9}$$

$$\begin{gathered} y_{n+1} = y_n + \frac{h}{1 - h\lambda} f(t_{n+1}, y_n) \\ \text{for } n = 0, 1, \ldots \end{gathered} \tag{1.10}$$

Using the formulation in eq. (1.8), numerical implementation with time marching is now straightforward. The algorithm for this implementation is as follows:

Algorithm 2: Implicit Euler Time Integration - using Analytical Reformulation

```
  Data: $h, n, f(h,t,y), y_0, t_0$
  Result: $y_{imp} = \{y_0, y_1, \ldots, y_n\}$ at $t = \{t_0, t_1, \ldots, t_n\}$
1 Initialize arrays $y_{imp} = \{y_0\}$ and $t = \{t_0\}$;
2 $y_i \leftarrow y_0$ and $t_i \leftarrow t_0$;
3 for $i \leftarrow 0$ to $n$ do
4     $t_{i+1} = t_i + h$;
5     $y_{i+1} = y_i + f(h, t_{i+1}, y_i)$;
6     Append $y_{exp} \leftarrow y_{i+1}$; Append $t \leftarrow t_{i+1}$;
7 end
8 return $y_{imp}, t$
```

In addition to an easy numerical implementation, this scheme is also *uncondtionally stable.* However, the function $f(h, t, y)$ needs to be found out manually for each problem, thus, limiting its applicability to simple problems and predominantly scalar valued functions $f(t, y)$.

Matrix Inversion

Using eq. (1.9) as an example, eq. (1.7) can be evaluated for $n = 0, 1, 2 \ldots$ as

$$\begin{aligned} y_1 &= y_0 + h[\lambda(y_1 - \sin(t_1)) + \cos(t_1)] \\ y_2 &= y_1 + h[\lambda(y_2 - \sin(t_2)) + \cos(t_2)] \\ &\vdots \\ y_n &= y_{n-1} + h[\lambda(y_n - \sin(t_n)) + \cos(t_n)] \end{aligned} \tag{1.11}$$

The linear set of equations in eq. (1.11) can be rearranged into eq. (1.12) and then written in matrix form as shown in eq. (1.13). After inverting the matrix $\mathbf{A}$, we can solve for $\mathbf{y}$ using eq. (1.14)

$$\begin{aligned}(1-h\lambda)y_1 &= h[\lambda(-\sin(t_1)) + \cos(t_1)] + y_0 \\ (1-h\lambda)y_2 - y_1 &= h[\lambda(-\sin(t_2)) + \cos(t_2)] \\ &\vdots \\ (1-h\lambda)y_n - y_{n-1} &= h[\lambda(-\sin(t_n)) + \cos(t_n)]\end{aligned} \tag{1.12}$$

$$\mathbf{Ay} = \mathbf{b} \tag{1.13}$$

$$\text{where, } \mathbf{A} = \begin{bmatrix} (1-h\lambda) & 0 & \dots & 0 & 0 \\ -1 & (1-h\lambda) & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & (1-h\lambda) & 0 \\ 0 & 0 & \dots & -1 & (1-h\lambda) \end{bmatrix}$$

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} h[\lambda(-\sin(t_1)) + \cos(t_1)] + y_0 \\ h[\lambda(-\sin(t_2)) + \cos(t_2)] \\ \vdots \\ h[\lambda(-\sin(t_{n-1})) + \cos(t_{n-1})] \\ h[\lambda(-\sin(t_n)) + \cos(t_n)] \end{pmatrix}$$

$$\mathbf{y} = \mathbf{A^{-1}b} \tag{1.14}$$

An algorithm for this implementation is as follows:

Algorithm 3: Implicit Euler Time Integration - using Matrix Inversion

```
Data: $h, n, y_0, t_0$
Result: $y_{imp} = \{y_0, y_1, \dots, y_n\}$ at $t = \{t_0, t_1, \dots, t_n\}$
1 Initialize arrays $y_{imp} = \{y_0\}$ and $t = \{t_0\}$;
2 Initialize vector $\mathbf{b}$ and matrix $\mathbf{A}$;
3 Setup vector $\mathbf{b}$;
          // Append $t \leftarrow t_i$ for every row;
4 Setup matrix $\mathbf{A}$;
5 Calculate $\mathbf{A^{-1}}$;
6 Calculate $\mathbf{y_{sol}} \leftarrow \mathbf{A^{-1}b}$;
7 Append $y_{imp} \leftarrow \mathbf{y_{sol}}$;
8 return $y_{imp}, t$
```

Although $\mathbf{A}$ is a sparse matrix, the availability of sophisticated matrix inversion methods for sparse matrices makes the solution of this formulation easily possible. An additional benefit is the elimination of forward time marching. Thus, the solution for the entire time domain is calculated at once. It is, however, important to note that when dealing with initial value problems of higher-order, the size of $\mathbf{A}$ would increase quickly and could lead to problems during matrix inversion.

Fixed Point Iteration

A fixed point iteration can be applied to solve eq. (1.5). [See FS01, pg. 105]. Thus, we have the following reformulation for the Implicit Euler time integration scheme

$$\begin{gathered} y_{n+1}^{(j+1)} = y_n + hf(t_{n+1}, y_{n+1}^{(j)}) \\ j = 0, 1, \dots \end{gathered} \tag{1.15}$$

In this way, for every step forward in time, we perform additional iterations to converge to a solution. In order to find an initial guess $y_{n+1}^{(0)}$, the Explicit Euler scheme can be used as given in eq. (1.16). This formulation, thus, comes under the the class of Predictor-Corrector schemes for solving initial value problems.

$$y_{n+1}^{(0)} = y_n + hf(t_n, y_n) \tag{1.16}$$

The algorithm for procedural implementation of the Implicit Euler time integration scheme with Fixed Point Iteration is as shown below:

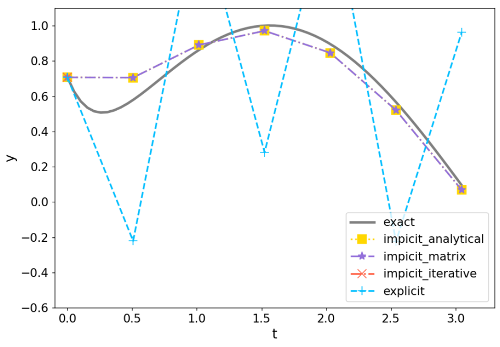
Algorithm 4: Implicit Euler Time Integration - using Fixed Point Iteration

```
Data: $h, n, f(t, y), y_0, t_0, iter_{max}$
Result: $y_{imp} = \{y_0, y_1, \dots, y_n\}$ at $t = \{t_0, t_1, \dots, t_n\}$
1  Initialize arrays $y_{imp} = \{y_0\}$ and $t = \{t_0\}$;
2  Set $convergence \leftarrow True$;
3  $y_i \leftarrow y_0$ and $t_i \leftarrow t_0$;
4  for $i \leftarrow 0$ to $n$ do
5      if $convergence$ then
6          Initialize $err \leftarrow 1$ and $j \leftarrow 0$;
7          $y_{i+1}^{j} = y_i + hf(t_i, y_i)$;
8          $t_{i+1} = t_i + h$;
9          while $convergence$ do
10             $y_{i+1}^{j+1} = y_i + hf(t_{i+1}, y_{i+1}^{j})$;
11             $err = abs(y_{i+1}^{j+1} - y_{i+1}^{j})$;
12             $j \leftarrow j + 1$;
13             if $err < tol$ then // Solution Converged
14                 $y_{i+1} = y_{i+1}^{j+1}$;
15                 Append $y_{imp} \leftarrow y_{i+1}$;
16                 Append $t \leftarrow t_{i+1}$;
17                 break;
18             end
19             if $j > iter_{max}$ then // Solution Diverged
20                 $convergence \leftarrow False$
21             end
22         end
23     end
24 end
25 return $y_{imp}, t$
```

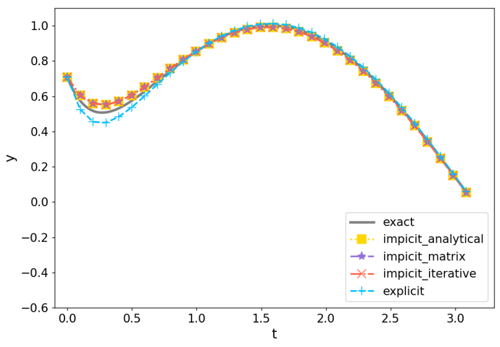
It is important to note that, although this is an Implicit Euler time integration scheme, using fixed point iteration renders this scheme to be *conditionally stable.*

Again, python code implementations for all the Implicit Euler time integration schemes discussed above can be found in Appendix A.

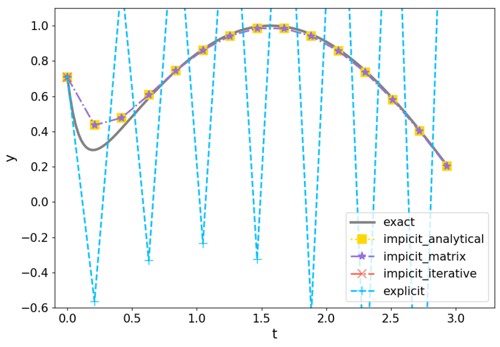
1.2.3. Comparison of Euler Time Integration Schemes



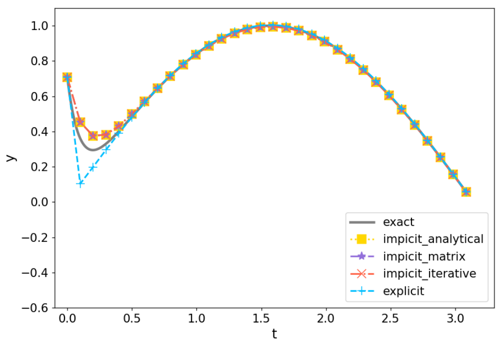
(a) $\lambda = -4.0, h = 0.5$



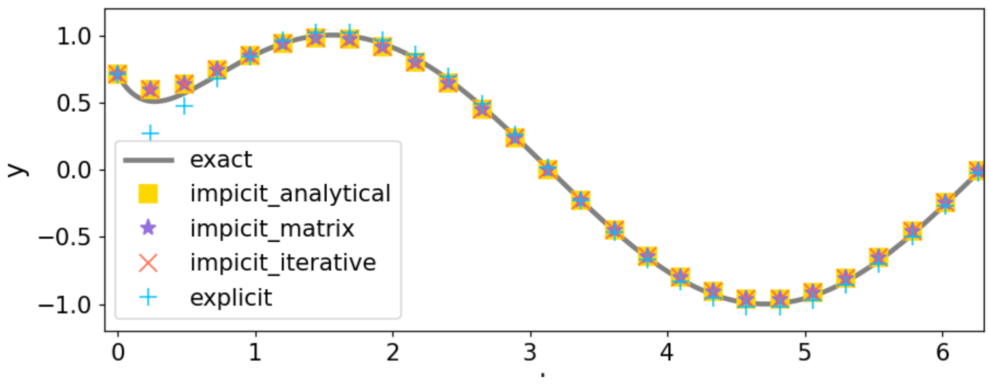
(b) $\lambda = -4.0, h = 0.1$



(c) $\lambda = -10.0, h = 0.2$



(d) $\lambda = -10.0, h = 0.1$



(e) $\lambda = -4, h = 0.24$

Figure 1.1.: Solution to $\dot{y} = \lambda(y - \sin(t)) + \cos(t),\ y(0) = 1/\sqrt{2}$

A graphical representation showing solutions to eq. (1.9) with initial condition $y(0) = 1/\sqrt{2}$, solved with all the above stated time integration schemes with varying parameters λ and h can be found in fig. 1.1. It can be observed that all the three Implicit Euler time integration schemes lead to the same solution. The analytical solution for this initial value problem is given below

$$y = (y_0 - \sin t_0) \exp^{\lambda(t-t_0)} + \sin t \tag{1.17}$$

1.3. Solution to Example Problems

// TODO

Solution for $\dot{u} + \gamma u = f$

//TODO

Solution for $\beta \ddot{u} + \alpha \dot{u} + \lambda u = f$ - (Only graphs) Detailed solution will be discussed in next chapter due to similarity with equation of motion

2. Solution Techniques for Equation of Motion

2.1. Equation of Motion

The Equation of Motion is one of the most frequently solved equations in engineering and science. It is an outcome of Newton's second law of motion and expresses the equality of internal and external forces for a system.

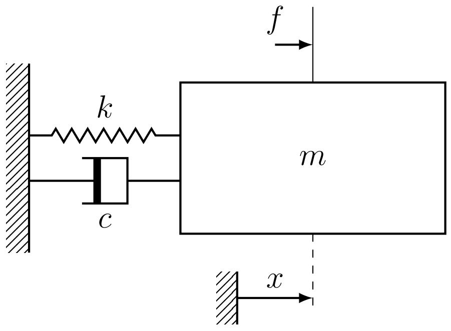


Figure 2.1.: Simple Oscillatory System

Given a simple oscillatory system as shown in fig. 2.1 with a spring having stiffness k, a damping element having damping coefficient c and a body having mass m and x as its generalized location co-ordinate, the equation of motion could be written as

$$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = f(t) \tag{2.1}$$

In eq. (2.1), the term $m\ddot{x}(t)$ represents the inertia force, $c\dot{x}(t)$ represents the force due to damping element, $kx(t)$ represents the spring force. Together, these terms represent the internal forces and the term $f(t)$ represents the external excitation force on the simple oscillatory system.

Furthermore, for an oscillatory system with many bodies as shown in fig. 2.2, a system of equations can be developed and the vectorized equation of motion can be expressed as below
TODO mention non linar form and then linearized form where C = P + G and D = N + K

$$\mathbf{M}\ddot{X}(t) + \mathbf{C}\dot{X}(t) + \mathbf{K}X(t) = F(t)$$

$$\text{where, } X = \begin{pmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{pmatrix}, \dot{X} = \begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \vdots \\ \dot{x}_n(t) \end{pmatrix}, \text{and } \ddot{X} = \begin{pmatrix} \ddot{x}_1(t) \\ \ddot{x}_2(t) \\ \vdots \\ \ddot{x}_n(t) \end{pmatrix} \tag{2.2}$$

In eq. (2.2), $X(t)$ is the vector of generalized location co-ordinates for the system, $\mathbf{M}$ is the Inertia Matrix, $\mathbf{C}\dot{X}(t)$ represents forces due to velocity dependent damping and gyroscopic effects, $\mathbf{K}$ is the Stiffness Matrix and $F(t)$ is the vector of external excitation forces.

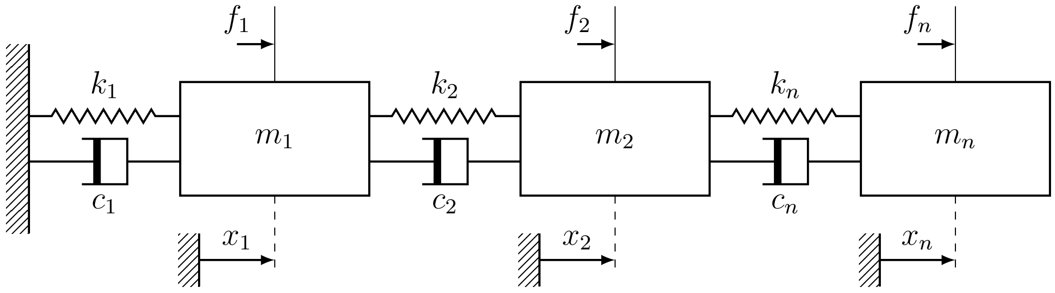


Figure 2.2.: Oscillatory System with Many Bodies

The eq. (2.2) could also represent the equation of motion for a Finite Element Model. In such a formulation, the vector $X(t)$ would then represent the nodal displacements.

Inorder to solve eq. (2.1) using the Time Integration schemes that were discussed in the previous chapter, it would be helpful to reformulate higher-order differential equation into a set of first-order differential equations. Doing so would eventually lead to the State-Space representation.

2.2. State-Space Representation

Let us define two functions $u(t)$ and $v(t)$ as shown below. On substituting eq. (2.3) in eq. (2.1) we get eq. (2.4). Further, rearanging eq. (2.4) and rewriting eq. (2.3b) we get a system of first-order ordinary differential equations in eq. (2.5) and eq. (2.6). Representing these together in a vector formulation gives the *non-linear* form of the State-Space representation.

$$u(t) = x(t) \tag{2.3a}$$

$$v(t) = \dot{x}(t) = \dot{u}(t) \tag{2.3b}$$

$$m\dot{v}(t) = -cv(t) - ku(t) + f(t) \tag{2.4}$$

$$\dot{v}(t) = -\frac{c}{m}v(t) - \frac{k}{m}u(t) + f(t) \tag{2.5}$$

$$\dot{u}(t) = v(t) \tag{2.6}$$

$$\begin{pmatrix} \dot{v}(t) \\ \dot{u}(t) \end{pmatrix} = \begin{pmatrix} -\frac{c}{m}v(t) - \frac{k}{m}u(t) + f(t) \\ v(t) \end{pmatrix} \tag{2.7}$$

In the case of the simple oscillatory system considered, eq. (2.7) turns out to be linear and can be expressed as shown in eq. (2.8).

$$\dot{Y}(t) = \mathbf{A}Y(t) + B(t) = F(t, \mathbf{A}, Y, B) \tag{2.8}$$

where, $\dot{Y}(t) = \begin{pmatrix} \dot{v}(t) \\ \dot{u}(t) \end{pmatrix}$, $Y(t) = \begin{pmatrix} v(t) \\ u(t) \end{pmatrix}$, $\mathbf{A} = \begin{bmatrix} -\frac{c}{m} & -\frac{k}{m} \\ 1 & 0 \end{bmatrix}$ and $B(t) = \begin{pmatrix} f(t) \\ 0 \end{pmatrix}$

Here, eq. (2.8) is called the *linearized* State-Space representation and $Y(t)$ is the *state vector* for the given system of first-order ordinary differential equations. Also, notice that eq. (2.8) is of the form of eq. (1.3) as briefly described in section 1.1. This method can be generalized to any higher-order ordinary differential equation. For eg. one would need define $w(t) = \ddot{x}(t)$ and proceed with the steps as described in eq. (2.3) to eq. (2.7) to finally arrive at the form in eq. (2.8).

Inorder to proceed with the solution of eq. (2.1), Initial Conditions need to be provided. Since eq. (2.1) is a second-order differential equation, two initial values are needed. In the case of the simple oscillatory system considered, these would be the initial position $x_0 = x(t_0)$ and initial velocity $v_0 = v(t_0)$. For the vectorized solution schemes, the Initial Condition is prescribed in terms of the state vector at time t_0 as shown below.

$$Y_0 = Y(t_0) = \begin{pmatrix} v(t_0) \\ u(t_0) \end{pmatrix} = \begin{pmatrix} v_0 \\ u_0 \end{pmatrix} \tag{2.9}$$

2.3. Scalar Solution Schemes

In this section the focus would be to use eq. (2.5) and eq. (2.6) to find a solution to the Equation of Motion by applying the Time Integration schemes as discussed earlier. Although they are simple to implement numerically, scalar solution schemes would lead to lagging of the approximate solution behind the exact solution. This is because while marching forward in time, eq. (2.5) and eq. (2.6) are not simultateously solved.

2.3.1. Explicit Euler Method

On application of the Explicit Euler scheme as described in eq. (1.4) to eq. (2.5) and eq. (2.6), the following time discretization is obtained for the position u and velocity v of the simple oscillatory system. As it can be seen, marching forward in time is straightforward and only uses values computed in the previous time step.

$$v_{n+1} = v_n + h\,(-\frac{c}{m}v_n - \frac{k}{m}u_n + f_n) \tag{2.10}$$

$$u_{n+1} = u_n + h\,(v_n) \tag{2.11}$$

Due to discretization using the Explicit Euler scheme in eq. (2.11), for the computation of displacement at next time step u_{n+1}, the value of velocity from the previous time step v_n is used, eventhough a newer value v_{n+1} is available after computing eq. (2.10).

//TODO: Add Plot for Solution from Python Code

2.3.2. Semi-Implicit Euler Method

As an improvement to the Explicit Euler scheme, if the newly computed value of velocity at the next time step v_{n+1} is used for the computation of position at the next time step u_{n+1} in eq. (2.13), the appoximate solution would be improved and the lag of the approximated displacement in comparison to the exact solution would be reduced.

$$v_{n+1} = v_n + h\ (-\frac{c}{m} v_n - \frac{k}{m} u_n + f_n) \tag{2.12}$$

$$u_{n+1} = u_n + h\ (v_{n+1}) \tag{2.13}$$

It can be seen that eq. (2.13) is the result of application of the Implicit Euler scheme as described in eq. (1.5) to eq. (2.6), whereas eq. (2.12) is identical to eq. (2.10), a result of application of Explicit Euler scheme to eq. (2.5). Hence, this scheme is called Semi-Implicit Euler Method.

//TODO: Add Plot for Solution from Python Code

2.4. Vectorized Solution Schemes

In this section, solution of the Equation of Motion is sought by using Matrix-Vector methods. Although, their implementation is not as straight-forward as the scalar solution schemes, the phase lag between the approximate solution and the exact solution is eliminated. This results from the simultaneous computation of velocity and position at the next time step by combining them in the *state-vector* as shown in eq. (2.8) and restated below.

$$Y(t) = \begin{pmatrix} v(t) \\ u(t) \end{pmatrix} = \begin{pmatrix} \dot{x}(t) \\ x(t) \end{pmatrix} \tag{2.14}$$

2.4.1. Explicit Euler Method (Vectorized)

The vectorized Explicit Euler scheme is obtained by plainly combining eq. (2.10) and eq. (2.11) into eq. (2.15). Hence, although this is a vectorized solution scheme, the approximate solution for displacement $u(t)$ still lags behind the exact solution. The intial condition Y_0 for time stepping is as described in eq. (2.9).

$$Y_{n+1} = \ Y_n \ + \ h\ F(t_n, \mathbf{A}, Y_n, B_n) \tag{2.15}$$

$$\text{where, } F(t_n, \mathbf{A}, Y_n, B_n) = \begin{pmatrix} -\frac{c}{m} v_n - \frac{k}{m} u_n + f_n \\ v_n \end{pmatrix}$$

$$\text{and } n = 0, 1, 2, \ldots$$

2.4.2. Implicit Euler - Iterative Convergence Method

Building upon Fixed Point Iteration scheme as explained in section 1.2.2 and extending eq. (1.15) and eq. (1.16) to the State-Space representation given by eq. (2.8), the following Implicit Euler scheme is obtained. Here, eq. (2.16) is the Predictor equation for every new time step and eq. (2.17) is the Corrector equation that is used for convergence of the approximate solution for the current time step. Although this is an Implicit Euler scheme, numerical approximation errors could lead to divergence when a large time step is used for computations.

$$\text{Predictor: } Y_{n+1}^{(j)} = Y_n + h\, F(t_n, \mathbf{A}, Y_n, B_n) \tag{2.16}$$

$$\text{Corrector: } Y_{n+1}^{(j+1)} = Y_n + h\, F(t_{n+1}, \mathbf{A}, Y_{n+1}^{(j)}, B_{n+1}) \tag{2.17}$$

$$\text{for } n \text{ and } j = 0, 1, 2, \ldots$$

Another method for obtaining Iterative Convergence is the Newton-Raphson Method where, firstly a Newton Polynomial $G(Y_{n+1}) = 0$ is obtained by applying Implicit Euler scheme to eq. (2.8) as shown below

$$\begin{aligned} Y_{n+1} &= Y_n + \mathbf{A}hY_{n+1} + hB_{n+1} \\ (\mathbf{I} - \mathbf{A}h)&Y_{n+1} + Y_n + hB_{n+1} = 0 \end{aligned} \tag{2.18}$$

Now, the only unknown in section 2.4.2 is the state-vector at the new time step Y_{n+1}. Inorder to find the roots of this Vector Polynomial, the Newton-Raphson Method for vector fields can be utilized.

//TODO Review and expand Newton-Raphson Method for vector functions

//TODO Add Plot for Solution from Python Code

2.4.3. Implicit Euler - Matrix Inversion Method

For the Matrix Inversion Method, instead of starting with the state-space representation in eq. (2.8), the Implicit Euler scheme is applied to eq. (2.5) and eq. (2.6). As a result, the following two equations are obtained.

$$v_{n+1} = v_n + h\,(-\frac{c}{m}v_{n+1} - \frac{k}{m}u_{n+1} + f_{n+1}) \tag{2.19}$$

$$u_{n+1} = u_n + h\,(v_{n+1}) \tag{2.20}$$

The time discretization expressed in eq. (2.19) and eq. (2.20) is valid for all time steps. By substituing the values for $n = 0, 1, \ldots, n-1$ a linear system of equations eq. (2.21) to eq. (2.26) is developed by keeping known values on the right and unknown values on the left as shown below.

$$(1 + h\frac{c}{m})\, v_1 + (h\frac{k}{m})u_1 = f_1 + v_0 \tag{2.21}$$

$$(-h)\, v_1 + u_1 = u_0 \tag{2.22}$$

$$(1 + h\frac{c}{m})\, v_2 + (h\frac{k}{m})u_2 - v_1 = f_2 \tag{2.23}$$

$$(-h)\, v_2 + u_2 - u_1 = 0 \tag{2.24}$$

$$\vdots$$

$$(1 + h\frac{c}{m})\, v_n + (h\frac{k}{m})u_n - v_{n-1} = f_n \tag{2.25}$$

$$(-h)\, v_n + u_n - u_{n-1} = 0 \tag{2.26}$$

This linear system of equations could then be reformulated as a *matrix-vector* equation as in eq. (2.27). The solution vector **u** to eq. (2.27) is then obtained by eq. (2.28) which requires inverting the matrix **A**.

$$\mathbf{Au} = \mathbf{b} \tag{2.27}$$

where,

$$\mathbf{A} = \begin{bmatrix} (1+h\frac{c}{m}) & (h\frac{k}{m}) & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ (-h) & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ -1 & 0 & (1+h\frac{c}{m}) & (h\frac{k}{m}) & \dots & 0 & 0 & 0 & 0 \\ 0 & -1 & (-h) & 1 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & (1+h\frac{c}{m}) & (h\frac{k}{m}) & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & (-h) & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & -1 & 0 & (1+h\frac{c}{m}) & (h\frac{k}{m}) \\ 0 & 0 & 0 & 0 & \dots & 0 & -1 & (-h) & 1 \end{bmatrix}$$

$$\mathbf{u} = \begin{pmatrix} v_1 \\ u_1 \\ v_2 \\ u_2 \\ \vdots \\ v_{n-1} \\ u_{n-1} \\ v_n \\ u_n \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} hf_1 + v_0 \\ u_0 \\ hf_2 \\ 0 \\ \vdots \\ hf_{n-1} \\ 0 \\ hf_n \\ 0 \end{pmatrix}$$

$$\mathbf{u} = \mathbf{A^{-1}b} \tag{2.28}$$

Although, matrix inversion is necessary, using this scheme gives the solution for the complete time interval of the solution at once.

//TODO Add Plot for Solution from Python Code.

2.5. Example Problems - Derivation of Equation of Motion

[See Stu95]

A. Python Code Examples

A.1. Numerical Time Integration Schemes

The algorithms discussed in section 1.2 will work for any given problem. However, for the sake of clarity, the following initial value problem will be used to describe the numerical implementation of the time integration schemes.

$$\dot{y} = \lambda(y - \sin t) + \cos t$$

A.1.1. Explicit Euler Method

```python
def explicit_euler(h, n, f, y_init, t_init):
    # Setting Initial Conditions for the Initial Value Problem
    y_0 = y_init
    t_0 = t_init

    # Initializing Solution Arrays with Initial Values
    y_explicit = np.array(y_0)
    t_explicit = np.array(t_0)

    # Initializing Parameters for Time Integration
    y_next = y_0
    t_next = t_0

    # Marching Forward in Time
    for i in range(n):

        # Calculate Solution at Next Time Step
        y_next = y_next + h * (f(y_next, t_next))

        # Increment Time Step
        t_next = t_next + h

        # Store Values for Plotting
        y_explicit = np.append(y_explicit, y_next)
        t_explicit = np.append(t_explicit, t_next)

    return y_explicit, t_explicit
```

A.1.2. Implicit Euler Method - Analytical Reformulation

```python
def f_hty(h, f, t, y):
    # Reformulated Derivative Function in h, t and y
    # Needs to be formulated for the given IVP
    return h * (1 / (1 - h*lmda) * f(y, t))
```

```python
def implicit_euler_analytical(h, n, f_hty, y_init, t_init):
    # Setting Initial Conditions for the Initial Value Problem
    y_0 = y_init
    t_0 = t_init

    # Initializing Solution Arrays
    y_implicit_1 = np.array(y_0)
    t_implicit_1 = np.array(t_0)

    # Initializing Parameters for Time Integration
    y_next = y_0
    t_next = t_0

    # Marching Forward in time
    for i in range(n):

        # Increment Time
        t_next = t_next + h

        # Calculate Solution at Next Time Step
        y_next =
         + f_hty(h, f, t_next, y_next)

        # Storing Values for Plotting
        y_implicit_1 = np.append(y_implicit_1, y_next)
        t_implicit_1 = np.append(t_implicit_1, t_next)

    return y_implicit_1, t_implicit_1
```

A.1.3. Implicit Euler Method - Matrix Inversion

```python
def implicit_euler_matrix(h, n, y_init, t_init):
    # Setting Initial Conditions for the Initial Value Problem
    y_0 = y_init
    t_0 = t_init

    # Initializing Solution Arrays with Intitial Values
    y_implicit_2 = np.array(y_0)
    t_implicit_2 = np.array(t_0)

    # Initializing Parameters for Solution
    t_next = t_0
    b = np.zeros(n)
    A = np.zeros(n*n).reshape(n, n)

    #Setting Up Vector b
    for i in range(n):
        t_next = t_next + h
        t_implicit_2 = np.append(t_implicit_2, t_next)
        b_n = -h * (lmda * math.sin(t_next) - math.cos(t_next))
        if i==0:
            b[i] = b_n + y_0
        else:
            b[i] = b_n
    # Setting Up Matrix A
    for i in range(n):
        A[i,i] = 1 - h * lmda
        if i > 0:
            A[i, i-1] = -1

    # Matrix Inversion
    A_inv = np.linalg.inv(A)

    # Calculating Solution Space
    # A * y = b -----> y = inv(A) * b
    y_sol = A_inv @ b

    # Adding Initial Value to Solution Space
    y_implicit_2 = np.append(y_implicit_2, y_sol)

    return y_implicit_2, t_implicit_2
```

A.1.4. Implicit Euler Method - Fixed Point Iteration

```
def implicit_euler_iterative(h, n, f, y_init, t_init, maxitr, tol):
  # Setting Initial Conditions for the Initial Value Problem
    y_0 = y_init
    t_0 = t_init

    # Initializing Solution Arrays
    y_implicit_3 = np.array(y_0)
    t_implicit_3 = np.array(t_0)

    # Initializing Parameters for Time Integration
    y_next = y_0
    t_next = t_0
    convergence = True

    # Marching Forward in time
    for i in range(n):
        if (convergence):
            # Initialize Error for Convergence testing
            err = 1
            j = 0

            # Predictor - Explicit Euler
            y_next_iter = y_next + h * (f(y_next, t_next))

            # Increment Time
            t_next = t_next + h

            # Iterative Convergence using Corrector
            while(convergence):
                # Store previous iteration to calculate error
                y_prev_iter = y_next_iter

                # Corrector
                y_next_iter = y_next + h * f(y_next_iter, t_next)
```

```python
                # Calculate Error
                err = abs(y_next_iter - y_prev_iter)

                # Increment Iteration Counter
                j +=1

                # When solution converges
                if(err < tol):
                    # Use converged Value for next Time Step
                    y_next = y_next_iter

                    # Store Values for Plotting
                    y_implicit_3 = np.append(y_implicit_3, y_next)
                    t_implicit_3 = np.append(t_implicit_3, t_next)
                    break

                # When solution diverges
                if(j > maxitr):
                    convergence = False

    return y_implicit_3, t_implicit_3
```

Bibliography

[FS01] Claus Führer and Achim Schroll. *Numerical Analysis: An Introduction.* 3rd. 2001.

[And15] Vivi Andasari. *Lecture 8 Handout: Euler's Methods.* `http://people.bu.edu/andasari/courses/numericalpython/Week5Lecture8/ExtraHandout_Lecture8.pdf`. Accessed: 2020–12-18. 2015.

[Stu95] Daniel S. Stutts. *Analytical Dynamics: Lagrange's Equation and its Application – A Brief Introduction.* Nov. 1995. DOI: `10.5281/zenodo.4457929`. URL: `https://doi.org/10.5281/zenodo.4457929`.