Fixed-Step ODE Solvers

Tamas Kis | tamas.a.kis@outlook.com

Tamas Kis
https://tamaskis.github.io

LICENSE

Contents

List of Algorithms

PART I

Basics

1 Initial Value Problems

1.1 Definition

There are many classes of problems involving ordinary differential equations (ODEs) and many different ways to solve them numerically. However, colloquially, "ODE solvers" are tools that can (numerically) solve initial value problems (IVPs). In these types of problems, we are given a differential equation along with an initial condition. In the single-variable case,

$$\frac{dy}{dt} = f(t, y), \quad x(t_0) = x_0$$

Our goal is to solve for $y(t)$. The above equation indicates that the rate of change is y is dependent on (a) the instantaneous value of y and (b) the independent variable, t[1].

The previous equation represented a 1st-order IVP. However, ODE solvers can are also design to be able to solve higher-order IVPs. For example, we can express a 2nd-order IVP as

$$\frac{d^2y}{dt^2} = f\left(t, x, \frac{dy}{dt}\right), \quad y(t_0) = y_0, \quad \left.\frac{dy}{dt}\right|_{t=t_0} = y_0'$$

For a 1st-order IVP, we know f and y_0 and wish to find $y(t)$. For a 2nd-order IVP, we know f, y_0, and y_0', and wish to find $y(t)$. In some (extremely rare) cases, it is possible to solve these problems explicitly. However, even then, it is often easier to just use a numerical method. Once again, it is *vital* to note that these numerical methods are applicable to vector valued IVP's as well. If we consider a vector-valued variable $\mathbf{y} = (y_1, ..., y_n)^T$, then the 1st and 2nd-order IVP's become

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}), \quad \mathbf{y}(t_0) = \mathbf{y}_0$$

$$\frac{d^2\mathbf{y}}{dt^2} = f\left(t, \mathbf{y}, \frac{d\mathbf{y}}{dt}\right), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad \left.\frac{d\mathbf{y}}{dt}\right|_{t=t_0} = \mathbf{y}_0'$$

[1] The independent variable is denoted as t because in most cases ODE solvers are used in situations where time is the independent variable.

1.2 Converting Higher-Order IVPs to First-Order IVPs: The Scalar Case

Consider the general form of a 2nd-order IVP. Noting that the second derivative of a variable can be a function of the independent variable, the variable itself, and the first derivative of the variable, we write (in the general case)

$$\frac{d^2z}{dt^2} = g\left(t, z, \frac{dz}{dt}\right), \qquad \left.\frac{dz}{dt}\right|_{t=t_0} = z_0', \qquad z(t_0) = z_0 \tag{1.1}$$

Our goal is to solve such IVPs using an ODE solver. *However, ODE solvers are only suitable for solving 1st-order IVPs*. The solution to this issue is to define a **vector-valued differential equation** using a vector, $\mathbf{y}$, to represent the dependent variable. The general form of such a vector-valued differential equation is

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{1.2}$$

At first glance, going from Eq. (1.1) to Eq. (1.2) seems like an impossible undertaking. However, the conversion can be accomplished in three steps, in a rather algorithmic fashion. Thus, we first introduce Algorithm 1 to define the procedure for an m^{th}-order IVP, and then use this algorithm to show how to convert Eq. (1.1) to Eq. (1.2).

Algorithm 1:

Converting a higher-order IVP to a first-order IVP (scalar dependent variable).

Given:

- $\dfrac{d^m z}{dt^m} = g\left(t, z, \dfrac{dz}{dt}, \dfrac{d^2z}{dt^2}, ..., \dfrac{d^{m-1}z}{dt^{m-1}}\right)$ - m^{th}-order ODE
- $z_0, z_0', z_0'', ..., z_0^{(m-1)}$ - initial conditions

Procedure:

1. Define the vector $\mathbf{y}$ as the original dependent variable z and its derivatives up to the $(m-1)^{\text{th}}$ derivative.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} z \\ dz/dt \\ d^2z/dt^2 \\ \vdots \\ d^{m-1}z/dt^{m-1} \end{bmatrix}$$

2. Find the derivative of $\mathbf{y}$.

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} dz/dt \\ d^2z/dt^2 \\ d^3z/dt^3 \\ \vdots \\ d^mz/dt^m \end{bmatrix}$$

3. Note that the original differential equation was defined as $d^mz/dt^2 = g(...)$.

Thus, we can substitute the function g into the last component of $d\mathbf{y}/dt$.

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} dz/dt \\ d^2z/dt^2 \\ d^3z/dt^3 \\ \vdots \\ g\left(t, z, \dfrac{dz}{dt}, \dfrac{d^2z}{dt^2}, ..., \dfrac{d^{m-1}z}{dt^{m-1}}\right) \end{bmatrix}$$

4. From step 1, we can see that $z = y_1$, $dz/dt = y_2$, $d^2z/dt^2 = y_3$, etc. Make these substitutions into the RHS of the equation from step 3. This essentially gives us $\mathbf{f}(t, \mathbf{y})$.

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ \vdots \\ g\left(t, y_1, y_2, y_3, ..., y_n\right) \end{bmatrix}$$

5. At this point, we have the function $\mathbf{f}(t, \mathbf{y})$ defining the vector-valued differential equation. Now, we just need to define its initial conditions. Once again exploiting the fact that $z = y_1$, $dz/dt = y_2$, $d^2z/dt^2 = y_3$, etc., we can write

$$\mathbf{y}_0 = \begin{bmatrix} y_{1,0} \\ y_{2,0} \\ y_{3,0} \\ \vdots \\ y_{n,0} \end{bmatrix} = \begin{bmatrix} z_0 \\ z_0' \\ z_0'' \\ \vdots \\ z_0^{(m-1)} \end{bmatrix}$$

Return:

- $\mathbf{f}(t, \mathbf{y})$ - multivariate, vector-valued function ($\mathbf{f} : \mathbb{R}^m \to \mathbb{R}^m$) defining the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$
- $\mathbf{y}_0 \in \mathbb{R}^m$ - initial condition

Let's return to the 2nd-order case and apply Algorithm 1 to write it as a 1st-order IVP. Since z is a scalar variable, we need $\mathbf{y} \in \mathbb{R}^2$; this is because one component of $\mathbf{y}$ will store z, while the other will store its derivative with respect to t.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} z \\ dz/dt \end{bmatrix}$$

Differentiating $\mathbf{y}$,

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} dz/dt \\ d^2z/dt^2 \end{bmatrix}$$

Noting that $d^2z/dt^2 = g(t, z, dz/dt)$,

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} dz/dt \\ g(t, z, dz/dt) \end{bmatrix}$$

From our definition of $\mathbf{y}$, we can see that $z = y_1$ and $dz/dt = y_2$. Making these substitutions,

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} y_2 \\ g(t, y_1, y_2) \end{bmatrix}$$

Thus, we have

$$\mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} y_2 \\ g(t, y_1, y_2) \end{bmatrix}$$

Finally, defining $\mathbf{y}_0$ using the initial conditions for z and dz/dt,

$$\mathbf{y}_0 = \begin{bmatrix} z_0 \\ z_0' \end{bmatrix} \tag{1.3}$$

For additional understanding, consider Example 1.2.1, which also demonstrates implementation of a vector-valued differential equation in MATLAB.

Example 1.2.1: Converting a 3rd-order IVP to a 1st-order IVP.

Consider the following 3rd-order IVP:

$$\frac{d^3b}{dx^3} = 25x^2b^3\frac{d^2b}{dx^2} + xb, \qquad b(x_0) = 5, \qquad \left.\frac{db}{dx}\right|_{x=x_0} = 0, \qquad \left.\frac{d^2b^2}{dx}\right|_{x=x_0} = -10$$

Write this IVP as a 1st-order IVP and implement the resulting vector-valued differential equation in MATLAB.

■ ***SOLUTION***

First, let's define the vector variable:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b \\ db/dx \\ d^2b/dx^2 \end{bmatrix}$$

Differentiating,

$$\frac{d\mathbf{y}}{dx} = \begin{bmatrix} db/dx \\ d^2b/dx^2 \\ d^3b/dx^3 \end{bmatrix}$$

Substituting the original ODE into the last element of $d\mathbf{y}/dx$,

$$\frac{d\mathbf{y}}{dx} = \begin{bmatrix} db/dx \\ d^2b/dx^2 \\ 25x^2b^3(d^2b/dx^2) + xb \end{bmatrix}$$

Noting that $y_1 = b$, $y_2 = db/dx$, and $y_3 = d^2b/dx^2$, we get

$$\frac{d\mathbf{y}}{dx} = \begin{bmatrix} y_2 \\ y_3 \\ 25x^2y_1^3y_3 + xy_1 \end{bmatrix}$$

Since, by convention, we write $d\mathbf{y}/dx = \mathbf{f}(x, \mathbf{y})$, we get

$$\boxed{\mathbf{f}(x, \mathbf{y}) = \begin{bmatrix} y_2 \\ y_3 \\ 25x^2y_1^3y_3 + xy_1 \end{bmatrix}}$$

Assembling the initial conditions into the vector $\mathbf{y}_0$ (and recalling that $y_1 = b$, $y_2 = db/dx$, and $y_3 = d^2b/dx^2$),

$$\boxed{\mathbf{y}_0 = \begin{bmatrix} 5 \\ 0 \\ -10 \end{bmatrix}}$$

To define the vector-valued differential equation and the corresponding initial condition in MATLAB,

```
% vector-valued differential equation
f = @(x,y) [y(2);y(3);
            25*x^2*y(1)^3*y(3)+x*y(1)];

% initial condition
y0 = [ 5;
       0;
      -10];
```

1.3 Converting Higher-Order IVPs to First-Order IVPs: The Vector Case

Higher order IVPs are not restricted to scalar variables; in fact, most of the time, the IVPs we deal with our higher-order IVPs where the dependent variable is a vector. The general form of such an IVP is shown in Eq. (1.4) below, where we assume that $\mathbf{z} \in \mathbb{R}^p$.

$$\frac{d^m\mathbf{z}}{dt^m} = \mathbf{g}\left(t, \mathbf{z}, \frac{d\mathbf{z}}{dt}, ..., \frac{d^{m-1}\mathbf{z}}{dt^{m-1}}\right), \qquad \mathbf{z}(t_0) = \mathbf{z}_0, \qquad \left.\frac{d\mathbf{z}}{dt}\right|_{t=t_0} = \mathbf{z}_0', \qquad ..., \qquad \left.\frac{d^{m-1}\mathbf{z}}{dt^{m-1}}\right|_{t=t_0} = \mathbf{z}_0^{(m-1)} \tag{1.4}$$

To convert Eq. (1.4) to a 1st-order IVP, we introduce Algorithm 2, which is essentially identical to Algorithm 1, just extended to the case of a vector dependent variable.

Algorithm 2:

Converting a higher-order IVP to a first-order IVP (vector dependent variable).

Given:

- $\dfrac{d^m\mathbf{z}}{dt^m} = g\left(t, \mathbf{z}, \dfrac{d\mathbf{z}}{dt}, \dfrac{d^2\mathbf{z}}{dt^2}, ..., \dfrac{d^{m-1}\mathbf{z}}{dt^{m-1}}\right)$ - m^{th}-order ODE
- $\mathbf{z}_0, \mathbf{z}_0', \mathbf{z}_0'', ..., \mathbf{z}_0^{(m-1)} \in \mathbb{R}^p$ - initial conditions

Procedure:

1. Define the vector $\mathbf{y}$ as the original dependent variable $\mathbf{z}$ and its derivatives up to the $(m-1)^{\text{th}}$ derivative.

$$\mathbf{y} = \begin{bmatrix} \mathbf{y}_{1:p} \\ \mathbf{y}_{(n+1):2p} \\ \mathbf{y}_{(2n+1):3p} \\ \vdots \\ \mathbf{y}_{((m-1)n+1):mp} \end{bmatrix} = \begin{bmatrix} \mathbf{z} \\ d\mathbf{z}/dt \\ d^2\mathbf{z}/dt^2 \\ \vdots \\ d^{n-1}\mathbf{z}/dt^{n-1} \end{bmatrix}$$

In an expanded form, we write

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_p \\ \hline y_{n+1} \\ \vdots \\ y_{2p} \\ \hline y_{2n+1} \\ \vdots \\ y_{3p} \\ \hline \vdots \\ \hline y_{(m-1)n+1} \\ \vdots \\ y_{mp} \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_p \\ \hline dz_1/dt \\ \vdots \\ dz_p/dt \\ \hline d^2z_1/dt^2 \\ \vdots \\ d^2z_p/dt^2 \\ \hline \vdots \\ \hline d^m z_1/dt^m \\ \vdots \\ d^m z_p/dt^m \end{bmatrix}$$

2. Find the derivative of $\mathbf{y}$.

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} d\mathbf{z}/dt \\ d^2\mathbf{z}/dt^2 \\ d^3\mathbf{z}/dt^3 \\ \vdots \\ d^m\mathbf{z}/dt^m \end{bmatrix} = \begin{bmatrix} dz_1/dt \\ \vdots \\ dz_p/dt \\ \hline d^2z_1/dt^2 \\ \vdots \\ d^2z_p/dt^2 \\ \hline d^3z_1/dt^3 \\ \vdots \\ d^3z_p/dt^3 \\ \hline \vdots \\ \hline d^{m-1}z_1/dt^{m-1} \\ \vdots \\ d^{m-1}z_p/dt^{m-1} \end{bmatrix}$$

3. Note that the original differential equation was defined as $d^m\mathbf{z}/dt^m = \mathbf{g}(...)$.

Thus, we can substitute the function $\mathbf{g}$ into the last component of $d\mathbf{y}/dt$.

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} dz_1/dt \\ \vdots \\ dz_p/dt \\ \hline d^2z_1/dt^2 \\ \vdots \\ d^2z_p/dt^2 \\ \hline d^3z_1/dt^3 \\ \vdots \\ d^3z_p/dt^3 \\ \hline \vdots \\ \hline \mathbf{g}\left(t, \mathbf{z}, \dfrac{d\mathbf{z}}{dt}, \dfrac{d^2\mathbf{z}}{dt^2}, ..., \dfrac{d^{m-1}\mathbf{z}}{dt^{m-1}}\right) \end{bmatrix}$$

4. From step 1, we can see that $\mathbf{z} = (y_1, ..., y_p)^T$, $d\mathbf{z}/dt = (y_{n+1}, ..., y_{2p})^T$, $d^2\mathbf{z}/dt^2 = (y_{2n+1}, ..., y_{3p})^T$, etc. Make these substitutions into the RHS of the equation from step 3. This essentially gives us $\mathbf{f}(t, \mathbf{y})$.

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} y_{n+1} \\ \vdots \\ y_{2p} \\ \hline y_{2n+1} \\ \vdots \\ y_{3p} \\ \hline y_{3n+1} \\ \vdots \\ y_{4p} \\ \hline \vdots \\ \hline \mathbf{g}\left(t, \begin{bmatrix} y_1 \\ \vdots \\ y_p \end{bmatrix}, \begin{bmatrix} y_{n+1} \\ \vdots \\ y_{2p} \end{bmatrix}, \begin{bmatrix} y_{2n+1} \\ \vdots \\ y_{3p} \end{bmatrix}, ..., \begin{bmatrix} y_{(m-1)n+1} \\ \vdots \\ y_{mp} \end{bmatrix}\right) \end{bmatrix}$$

5. At this point, we have the function $\mathbf{f}(t, \mathbf{y})$ defining the vector-valued differential equation. Now, we just need to define its initial conditions.

$$\mathbf{y}_0 = \begin{bmatrix} \mathbf{z}_0 \\ \mathbf{z}_0' \\ \mathbf{z}_0'' \\ \vdots \\ \mathbf{z}_0^{(m-1)} \end{bmatrix}$$

Return:

- $\mathbf{f}(t, \mathbf{y})$ - multivariate, vector-valued function ($\mathbf{f} : \mathbb{R}^{mp} \to \mathbb{R}^{mp}$) defining the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$
- $\mathbf{y}_0 \in \mathbb{R}^{mp}$ - initial condition

Example 1.3.1: Converting a 2nd-order vector IVP to a 1st-order IVP

Consider the following 2nd-order IVP:

$$\frac{d^2\mathbf{x}}{dt^2} = (x_1 + x_2)\frac{d\mathbf{x}}{dt} + \mathbf{x}\frac{dx_1}{dt}, \qquad \mathbf{x}_0 = \begin{bmatrix}0\\0\end{bmatrix}, \qquad \mathbf{x}_0' = \begin{bmatrix}1\\1\end{bmatrix}$$

where $\mathbf{x} = (x_1, x_2)^T$. Write this IVP as a 1st-order IVP and implement the resulting differential equation in MATLAB.

■ ***SOLUTION***

First, let's define our new vector variable:

$$\mathbf{y} = \begin{bmatrix}\mathbf{y}_{1:2}\\ \mathbf{y}_{3:4}\end{bmatrix} = \begin{bmatrix}\mathbf{x}\\ d\mathbf{x}/dt\end{bmatrix}$$

In an expanded form,

$$\mathbf{y} = \begin{bmatrix}y_1\\y_2\\y_3\\y_4\end{bmatrix} = \begin{bmatrix}x_1\\x_2\\dx_1/dt\\dx_2/dt\end{bmatrix}$$

Differentiating,

$$\frac{d\mathbf{y}}{dx} = \begin{bmatrix}dx_1/dt\\dx_2/dt\\d^2x_1/dt^2\\d^2x_2/dt^2\end{bmatrix}$$

Expanding the original ODE,

$$\begin{cases}\dfrac{dx_1^2}{dt^2} = (x_1 + x_2)\dfrac{dx_1}{dt} + x_1\dfrac{dx_1}{dt}\\[2ex] \dfrac{dx_2^2}{dt^2} = (x_1 + x_2)\dfrac{dx_2}{dt} + x_2\dfrac{dx_1}{dt}\end{cases}$$

Replacing the last two element of $d\mathbf{y}/dx$ with the above equation,

$$\frac{d\mathbf{y}}{dx} = \begin{bmatrix}dx_1/dt\\dx_2/dt\\(x_1 + x_2)\dfrac{dx_1}{dt} + x_1\dfrac{dx_1}{dt}\\[2ex](x_1 + x_2)\dfrac{dx_2}{dt} + x_2\dfrac{dx_1}{dt}\end{bmatrix}$$

Noting that $y_1 = x_1$, $y_2 = x_2$, and $y_3 = dx_1/dt$, and $y_4 = dx_2/dt$, we get

$$\frac{d\mathbf{y}}{dx} = \begin{bmatrix}y_3\\y_4\\(y_1 + y_2)y_3 + y_1y_3\\(y_1 + y_2)y_4 + y_2y_3\end{bmatrix}$$

Since, by convention, we write $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$, we get

$$\boxed{\mathbf{f}(t, \mathbf{y}) = \begin{bmatrix}y_3\\y_4\\(y_1 + y_2)y_3 + y_1y_3\\(y_1 + y_2)y_4 + y_2y_3\end{bmatrix}}$$

Assembling the initial conditions into the vector $\mathbf{y}_0$ (and recalling that $\mathbf{y}_{1:2} = \mathbf{x}$ and $\mathbf{y}_{3:4} = d\mathbf{x}/dt$,

$$\boxed{\mathbf{y}_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}}$$

To define the vector-valued differential equation and the corresponding initial condition in MATLAB,

```
% vector-valued differential equation
f = @(t,y) [y(3);
            y(4);
            (y(1)+y(2))*y(3)+y(1)*y(3);
            (y(1)+y(2))*y(4)+y(2)*y(3)];

% initial condition
y0 = [0;
      0;
      1;
      1];
```

2

Defining Systems Mathematically and Computationally

2.1 State Vectors and State Space

In physics and engineering, most differential equations we deal with are second-order or higher. For example, consider the generic mass-spring-damper system shown in Fig. 2.1. If we assume that x is defined such that $x = 0$ when the

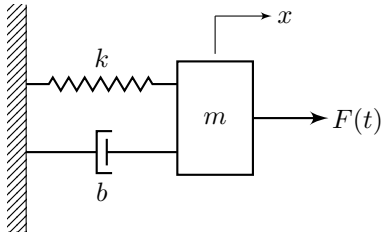


Figure 2.1: Mass-spring-damper system.

spring is at equilibrium, then (ignoring gravity) we can draw the free body diagram shown in Fig. 2.2. From Newton's

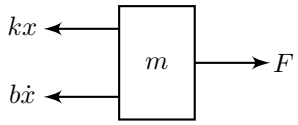


Figure 2.2: Free body diagram of mass.

second law,

$$F - kx - b\dot{x} = m\ddot{x}$$

Solving for $\ddot{x}$,

$$\ddot{x} = -\left(\frac{b}{m}\right)\dot{x} - \left(\frac{k}{m}\right)x + \left(\frac{1}{m}\right)F$$

If we know the initial conditions $x(t_0) = x_0$ and $\dot{x}(t_0) = \dot{x}_0$, then we have the following 2[nd]-order IVP:

$$\ddot{x} = -\left(\frac{b}{m}\right)\dot{x} - \left(\frac{k}{m}\right)x + \left(\frac{1}{m}\right)F, \qquad x(t_0) = x_0, \qquad \dot{x}(t_0) = \dot{x}_0 \tag{2.1}$$

Since ODE solvers are only suitable for solving 1st-order IVPs (as discussed previously), we need to convert the 2nd-order IVP given by Eq. (2.1) to a 1st-order IVP. In the context of engineering, we do this by defining a **state vector**, which fully describes the state of the physical system. Essentially, the vector $\mathbf{y}$ in Section 1.3 served as the state vector. There are two commonly used conventions for denoting a state vector as outlined in Convention 1 below.

Convention 1: State vector nomenclature.

- $\mathbf{y}$ – used when defining/discussing ODE solvers, as in mathematics, "y" is typically used to denote the dependent variable
- $\mathbf{x}$ – used in engineering contexts to denote the state of a physical system

The components of the state vector are referred to as **state variables**. A state variable is just a variable that helps describe some component of the mathematical state of a physical system. Thus, the state vector is essentially the set of state variables that fully describes the system.

But what counts as "fully describing" the system? Intuitively, we want the state variables (and thereby the state vector) to describe enough about the system for us to determine its future behavior *in the absence of external forces acting on the system*[1]. By "future behavior", we mean its position (i.e. trajectory) and its velocity. In the case of unforced motion in the mass-spring-damper system above, we only need the mass's position and velocity at one instant to completely determine its position and velocity at any time in the future [10, 11]. Therefore, our state variables in this case are

$$\text{state variables} = x,\ \dot{x}$$

Our state vector (i.e. collection of state variables) is then

$$\mathbf{y} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \tag{2.2}$$

Note that in this case, we violate Convention 1 and denote our state vector as $\mathbf{y}$. We do this because when we go to input the math into a computer, it will be easier to do if we don't have so many x's floating around.

Now that we've decided on our state-vector, we want to rewrite the 2nd-order IVP given by Eq. (2.1) as a 1st-order IVP in terms of the state vector. Mathematically[2], the goal is to write

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{2.3}$$

To do this, we can use an identical procedure to that taken in Section 1.2. If $\mathbf{y} = (x, \dot{x})^T$ is our state vector, then the derivative of the state vector is $\dot{\mathbf{y}} = (\dot{x}, \ddot{x})^T$. Therefore, we can write the following system of differential equations,

---

[1] While the force, F, certainly does affect how the mass will move in the *future*, it does not affect how the mass is moving *in this instant* (i.e. it affects the acceleration instantaneously, but the position and velocity cannot change instantaneously). As an example, we can consider two cars who, at some instant, are exactly side by side traveling at the exact same speed. In one of the cars, the driver is pushing the gas pedal, while in the other car, the driver has slammed on the brakes. The fact that the drivers are applying different forces to their car has no bearing on the fact that at the instant considered, the state of the cars are the same; they are both at the same position, going at the same speed. The forces that the drivers are applying certainly do, however, affect how the car will be moving in the *following* instant.

[2] Since this system is linear, we could also write it as

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

where we would find that

$$\overbrace{\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix}}^{\dot{\mathbf{x}}} = \overbrace{\begin{bmatrix} 0 & 1 \\ -k/m & -b/m \end{bmatrix}}^{\mathbf{A}} \overbrace{\begin{bmatrix} x \\ \dot{x} \end{bmatrix}}^{\mathbf{x}} + \overbrace{\begin{bmatrix} 1 \\ m \end{bmatrix}}^{\mathbf{B}} \overbrace{[f]}^{\mathbf{u}}$$

In the general case, $\mathbf{u}$ is typically a function of $\mathbf{x}$ and t, so we could define the function $\mathbf{f}(t, \mathbf{x})$ as

$$\mathbf{f}(t, \mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}(t, \mathbf{x})$$

in order to simulate the system with an ODE solver.

together with their initial conditions:

$$\begin{cases} \dot{x} = \dot{x} \\ \ddot{x} = -\left(\frac{b}{m}\right)\dot{x} - \left(\frac{k}{m}\right)x + \left(\frac{1}{m}\right)F \end{cases}, \qquad \begin{cases} x(t_0) = x_0 \\ \dot{x}(t_0) = \dot{x}_0 \end{cases}$$

In matrix form, these can be written as

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ -\left(\frac{b}{m}\right)\dot{x} - \left(\frac{k}{m}\right)x + \left(\frac{1}{m}\right)F \end{bmatrix}, \qquad \begin{bmatrix} x(t_0) \\ \dot{x}(t_0) \end{bmatrix} = \begin{bmatrix} x_0 \\ \dot{x}_0 \end{bmatrix} \tag{2.4}$$

Since we know $\mathbf{y} = (x, \dot{x})^T$ and $\dot{\mathbf{y}} = (\dot{x}, \ddot{x})^T$, we can simply compare Eq. (2.4) to Eq. (2.3) to find that

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} \dot{x} \\ -\left(\frac{b}{m}\right)\dot{x} - \left(\frac{k}{m}\right)x + \left(\frac{1}{m}\right)F \end{bmatrix}, \qquad \mathbf{y}_0 = \begin{bmatrix} x_0 \\ \dot{x}_0 \end{bmatrix}$$

Another source of confusion at this point can be the fact that in the equation above, we wrote the left hand side as $\mathbf{f}(t, \mathbf{y})$, but the right hand side is not explicitly a function of $\mathbf{y}$; rather, it is a function of the *components* (i.e. x and $\dot{x}$) of $\mathbf{y}$. If we wanted to be overly-rigorous, we *could* distinguish the two as

$$\mathbf{f}(t, x, \dot{x}) = \begin{bmatrix} \dot{x} \\ -\left(\frac{b}{m}\right)\dot{x} - \left(\frac{k}{m}\right)x + \left(\frac{1}{m}\right)F \end{bmatrix}, \qquad \mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} \mathbf{y}^T \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ -\left(\frac{b}{m}\right)\mathbf{y}^T \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \left(\frac{k}{m}\right)\mathbf{y}^T \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \left(\frac{1}{m}\right)F \end{bmatrix}$$

However, making this distinction is unnecessary in both a mathematical sense and a practical sense; when programming these equations, it is trivial to access an element of a vector – we do *not* need to use dot products to do this.

This brings us to the topic of actually programming these functions. As with many concepts, there can be a substantial disconnect between writing these equations down on paper and actually implementing them in a computer. Consequently, the best way to explain how to do it is to show by example. There are two main approaches we use, which we outline in Convention 2. Both conventions are used in Example 2.1.1.

Convention 2: Programming a differential equation.

There are **two** main approaches to programming a differential equation:

1. **Using a change of variables.**
   This is manageable for simple systems (this is the approach we took previously in Examples 1.2.1 and 1.3.1) We use this method under "SOLUTION" in the example below.

2. **"Unpacking the state vector", performing calculations, and "assembling the state vector derivative"**.
   Results in slightly longer code, but *much* easier/more intuitive to read; this is the method we use most often, especially for more complex systems. We use this method under "ALTERNATE SOLUTION" in the example below.

Example 2.1.1: Implementing the mass spring damper IVP in MATLAB.

Recall the 2nd-order IVP (Eq. (2.1)) governing the mass-spring-damper system shown in Fig. 2.1.

$$\ddot{x} = -\left(\frac{b}{m}\right)\dot{x} - \left(\frac{k}{m}\right)x + \left(\frac{1}{m}\right)F, \qquad x(t_0) = x_0, \qquad \dot{x}(t_0) = \dot{x}_0 \tag{(2.1)}$$

For this example, assume the following values:

$$b = 5\ (\mathrm{N}\cdot\mathrm{s})/\mathrm{m}, \qquad k = 1\ \mathrm{N}/\mathrm{m}, \qquad m = 2\ \mathrm{kg}, \qquad x_0 = 1\ \mathrm{m}, \qquad \dot{x}_0 = 0\ \mathrm{m}/\mathrm{s}$$

Additionally, let $F(t)$ be given by

$$F(t) = \cos \pi t$$

Implement a vector-valued differential equation $\mathbf{f}(t, \mathbf{y})$ and initial condition $\mathbf{y}_0$ (as shown below) in MATLAB that describes the evolution of this system.

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

■ *SOLUTION*

We can begin by defining known quantities and the forcing function.

```
% parameters
b = 5;          % damping constant [N.s/m]
k = 1;          % spring constant [N/m]
m = 2;          % mass [kg]
x0 = 1;         % initial position [m]
xdot0 = 0;        % initial velocity [m/s]

% forcing function
F = @(t) cos(pi*t);
```

From our work before this example, we alreaydot found that

$$\mathbf{y} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}, \qquad \mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} \dot{x} \\ -\left(\frac{b}{m}\right)\dot{x} - \left(\frac{k}{m}\right)x + \left(\frac{1}{m}\right)F \end{bmatrix}, \qquad \mathbf{y}_0 = \begin{bmatrix} x_0 \\ \dot{x}_0 \end{bmatrix} \tag{2.5}$$

Writing the initial condition in MATLAB,

```
% initial condition
y0 = [x0;
      xdot0];
```

Now, we must bridge the gap between the mathematical definition of $\mathbf{y}$ and how the computer "sees" $\mathbf{y}$. In the computer, $\mathbf{y}$ is just a 2×1 array that has the indices 1 and 2. Therefore, what the computer sees is

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{2.6}$$

If we compare Eqs. (2.5) (which is the mathematical definition of the state vector) and (2.6) (which is the state vector used by the computer), we can see that

$$x = y_1, \qquad \dot{x} = y_2$$

This essentially represents a change of variables; we now use y_1 to denote x and y_2 to denote $\dot{x}$. Performing this change of variables in Eq. (2.5),

$$\mathbf{f}(t,\mathbf{y}) = \begin{bmatrix} y_2 \\ -\left(\frac{b}{m}\right) y_2 - \left(\frac{k}{m}\right) y_1 + \left(\frac{1}{m}\right) F \end{bmatrix}$$

In MATLAB, we can write

```
% differential equation
f = @(t,y) [y(2);
            -(b/m)*y(2)-(k/m)*y(1)+(1/m)*F(t)];
```

■ *ALTERNATE SOLUTION*

While the solution above is adequate for simple functions, it is more cumbersome to use for complicated functions. Additionally, it makes the code less "readable" – all the state variables are written as elements of the state vector, so it is difficult to see what they physically represent. A better solution, especially when dealing with more complex systems, is to "unpack" the state vector, perform all calculations, and then assemble the state vector derivative. Again, this is best explained by just showing the solution in MATLAB.

There are multiple things to note, however, when defining the differential equation:

1. We first define a local function `f_extra`, which must be located at the very end of the script.
2. To be able to later use `f_extra` in an ODE solver, we assign it the function handle `f`.
3. We named the local function `f_extra` because it has some *extra* input parameters beyond what `f` will have (`f` can only have the input parameters `t` and `y` due to how ODE solvers are defined). See Section 2.3.1 for more information on passing extra parameters.

```
% parameters
b = 5;          % damping constant [N.s/m]
k = 1;          % spring constant [N/m]
m = 2;          % mass [kg]
x0 = 1;         % initial position [m]
xdot0 = 0;      % initial velocity [m/s]

% forcing function
F = @(t) cos(pi*t);

% initial condition
y0 = [x0;
      xdot0];

% assigns function handle to differential equation
f = @(t,y) f_extra(t,y,b,k,m,F);

% defines differential equation
function ydot = f_extra(t,y,b,k,m,F)

    % unpacks state vector
    x = y(1);
    xdot = y(2);

    % preallocates state vector derivative
```

```
        ydot = zeros(size(y));

        % assembles state vector derivative
        ydot(1) = xdot;
        ydot(2) = -(b/m)*xdot-(k/m)*x+(1/m)*F(t);

    end
```

2.2 Coupled ODEs

Consider the coupled mass-spring-damper system shown in Fig. 2.3. Using Newton's second law, one can derive the

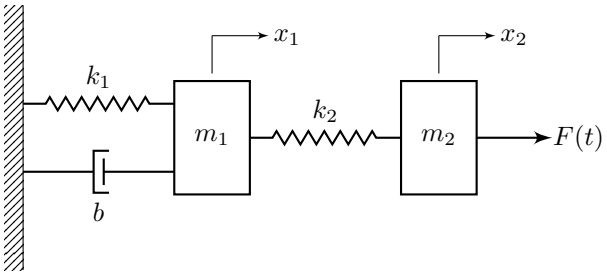


Figure 2.3: Coupled mass-spring-damper system.

following equations of motion for the two blocks:

$$\ddot{x}_1 = -\left(\frac{k_1 + k_2}{m_1}\right) x_1 + \left(\frac{k_2}{m_1}\right) x_2 - \left(\frac{b}{m_1}\right) \dot{x}_1$$
$$\ddot{x}_2 = \left(\frac{k_2}{m_2}\right) x_1 - \left(\frac{k_2}{m_2}\right) x_2 + F(t)$$

$\ddot{x}_1$ and $\ddot{x}_2$ are both 2$^{\text{nd}}$-order ODEs, and they are **coupled** to one another (x_1 and x_2 appear in both equations). Since they are coupled, we cannot solve one without solving the other simultaneously. In the context of ODE solvers, we will still need to represent the system using a single 1$^{\text{st}}$-order vector-valued ODE. To do this, we can simply define the state vector for the overall system as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \dot{x}_1 \\ x_2 \\ \dot{x}_2 \end{bmatrix}$$

Then the time derivative of the state vector is

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \ddot{x}_1 \\ \dot{x}_2 \\ \ddot{x}_2 \end{bmatrix}$$

Since $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x})$, we have

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}) = \begin{bmatrix} \dot{x}_1 \\ -\left(\frac{k_1 + k_2}{m_1}\right) x_1 + \left(\frac{k_2}{m_1}\right) x_2 - \left(\frac{b}{m_1}\right) \dot{x}_1 \\ \dot{x}_2 \\ \left(\frac{k_2}{m_2}\right) x_1 - \left(\frac{k_2}{m_2}\right) x_2 + F(t) \end{bmatrix}$$

2.3 Extra Parameters

2.3.1 Passing Extra Parameters

Consider the function $f(x, y) = -a(x - b)^2 - c(y - d)^2$ where x and y are independent variables, and a, b, c, and d are constants. In MATLAB, we can program this in two ways:

1. as an **anonymous** function
2. as a **MATLAB** function

As an anomymous function, we can program $f(x, y)$ as

```
f = @(x,y) -a*(x-b)^2-c*(y-d)^2;
```

In the above function definition, `a`, `b`, `c`, and `d` must already have been initialization. Whenever the function `f(x,y)` is called, it well than use the values of `a`, `b`, `c`, and `d` at the time `f` was defined. Consider the snippet of code below:

```
% function definition
a = 5; b = 5; c = 5; d = 5;
f = @(x,y) -a*(x-b)^2-c*(y-d)^2;

% call function, update "a", then call function again
f(2,2)
a = 20;
f(2,2)
```

Both calls of `f(2,2)` will evaluate to the same result because `f(2,2)` will still be using `a = 5` both times. Now, say that we want to update the values of the constants in `f(x,y)`. We *could* do the following:

```
% original function definition
a = 5; b = 5; c = 5; d = 5;
f = @(x,y) -a*(x-b)^2-c*(y-d)^2;

% update values of constants
a = 10; b = 10; c = 10; d = 10;
f = @(x,y) -a*(x-b)^2-c*(y-d)^2;
```

An alternate way to do this is

```
% define function where constants can vary as well
f_extra = @(x,y,a,b,c,d) -a*(x-b)^2-c*(y-d)^2;

% define original function by assigning function handle to f_extra
f = @(x,y) f_extra(x,y,5,5,5,5);

% update values of constants
f = @(x,y) f_extra(x,y,10,10,10,10)
```

The code above demonstrates the concept of **passing extra parameters** [7]. Typically, our functions will have constants in them who's values are stored in variables. However, for ODE solvers in particular, we can only pass in

functions of the form `f(t,y)`. Therefore, if we want to be able to easily vary the parameter values later, we first define a function `f_extra` (where the "`extra`" denotes that "extra" parameters are being passed), and then assign it a new **function handle**.

In many cases, we have really complicated functions that could be calculated in steps across 10+ lines of code. In these cases, the functions must be defined as *MATLAB* functions. For the same equation as considered above, we could write

```
function f = f_extra(x,y,a,b,c,d)
    f = -a*(x-b)^2-c*(y-d)^2;
end
```

We can assign function handles to MATLAB functions as well, and essentially treat them like anonymous functions thereafter. Assigning `f_extra` a function handle just like before (and with the same parameter values),

```
a = 10; b = 10; c = 10; d = 10;
f = @(x,y) f_extra(x,y,a,b,c,d);
```

2.3.2 Recovering Extra Parameters

From the block diagrams in Section 3.4, we can see that ODE solvers only output (a) the time vector and (b) a matrix storing the solution at each time in the time vector. However, oftentimes, the functions we pass to the ODE solvers will be very complicated functions within which many other quantities are calculated. As an example, for rocket simulations, the function defining the differential equation will be calculating the force due to drag at every time step. The drag force is a function of both the rocket's position (drag depends on air density, air density decreases with altitude) and velocity. However, there is no way for the ODE solver to directly output the drag force.

The workaround is to define the differential equation so that it returns more than just the state vector derivative. An example of how we typically define functions representing differential equations is

```
f = @(t,y) f_extra(t,y,a);
```

where

```
function f = f_extra(t,y,a);
    x = 2*a-5;
    f = x^2;
end
```

However, consider the case where we want to recover a variable `x` that is calculated *inside of* `f_extra`. Then we can instead define `f_extra` as

```
function [f,x] = f_extra(t,y,a);
    x = 2*a-5;
    f = x^2;
end
```

We can still assign a function handle to `f_extra`, just like before. When we do so, only the *first* output parameter (`f`) of `f_extra` will be returned.

At this point, we have all the ingredients we need in order to be able to recover extra parameters. The overall process is best explained using an example – see Example 2.3.1 below.

Example 2.3.1: Recovering extra parameters from an ODE solution.

Solve the following IVP on the interval $[0, 10]$:

$$\frac{d\mathbf{y}}{dt} = x\mathbf{y} \quad \text{where} \quad x = 2a - 5t, \quad a = 5, \quad \mathbf{y}_0 = (1, 1)^T$$

Then, plot $x(t)$ over this interval.

■ ***SOLUTION***

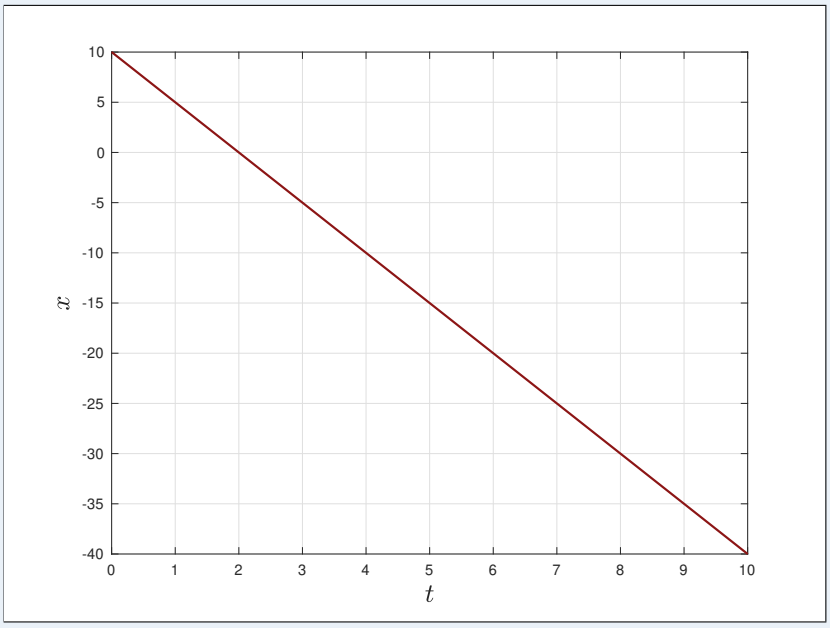
```
% assign function handle and pass extra parameter
f = @(t,y) f_extra(t,y,5);

% solve ODE
y0 = [1;1];
t0 = 0;
tf = 10;
[t,y] = ode45(f,[t0,tf],y0);

% recover "x" at every time step
x = zeros(size(t));
for i = 1:length(t)
    [~,x(i)] = f(t(i),y(i,:)');
end

% plot x vs. t
figure;
plot(t,x,'linewidth',1.5,'color',[0.5490,0.0824,0.0824]);
grid on;
xlabel('$t$','Interpreter','latex','FontSize',18);
ylabel('$x$','Interpreter','latex','FontSize',18);

% MATLAB function must be declared at end
function [f,x] = f_extra(t,y,a)
    x = 2*a-5*t;
    f = x*y;
end
```



3 ODE Solvers

3.1 The General Solution

Consider the initial value problem

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

Solving for $\mathbf{y}(t)$,

$$\mathbf{y}(t) - \underbrace{\mathbf{y}(t_0)}_{\mathbf{y}_0} = \int_{t_0}^{t} \mathbf{f}(t, \mathbf{y})\, dt$$

$$\boxed{\mathbf{y}(t) = \mathbf{y}_0 + \int_{t_0}^{t} \mathbf{f}(t, \mathbf{y})\, dt} \tag{3.1}$$

3.2 Approximation Through Discretization

In many cases, we cannot analytically evaluate the integral in Eq. (3.1) (hence our need for ODE solvers). The solution is to approximate the result by discretizing time, sample $\mathbf{f}(t, \mathbf{y})$ at those discrete points in time, and then assume that $\mathbf{f}(t, \mathbf{y})$ behaves a certain way between the two sample times. A visual representation of discretizing a scalar ODE in time is shown in Fig. 3.1. An example of a behavior we could assume between consecutive sample times is that $f(t, y) = f(t_n, y_n)\ \forall\ t \in [t_n, t_{n+1})$.

Using ODE solvers, we find the solution, $\mathbf{y}(t)$, at these discrete points in time, which we call **sample times**. The corresponding **time vector** storing the sample times, $\mathbf{t} \in \mathbb{R}^{N+1}$, is defined as

$$\boxed{\mathbf{t} = \begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ \vdots \\ t_N \end{bmatrix} = \begin{bmatrix} t_0 \\ t_0 + h \\ t_0 + 2h \\ \vdots \\ t_0 + (N-1)h \\ t_N \end{bmatrix}} \tag{3.2}$$

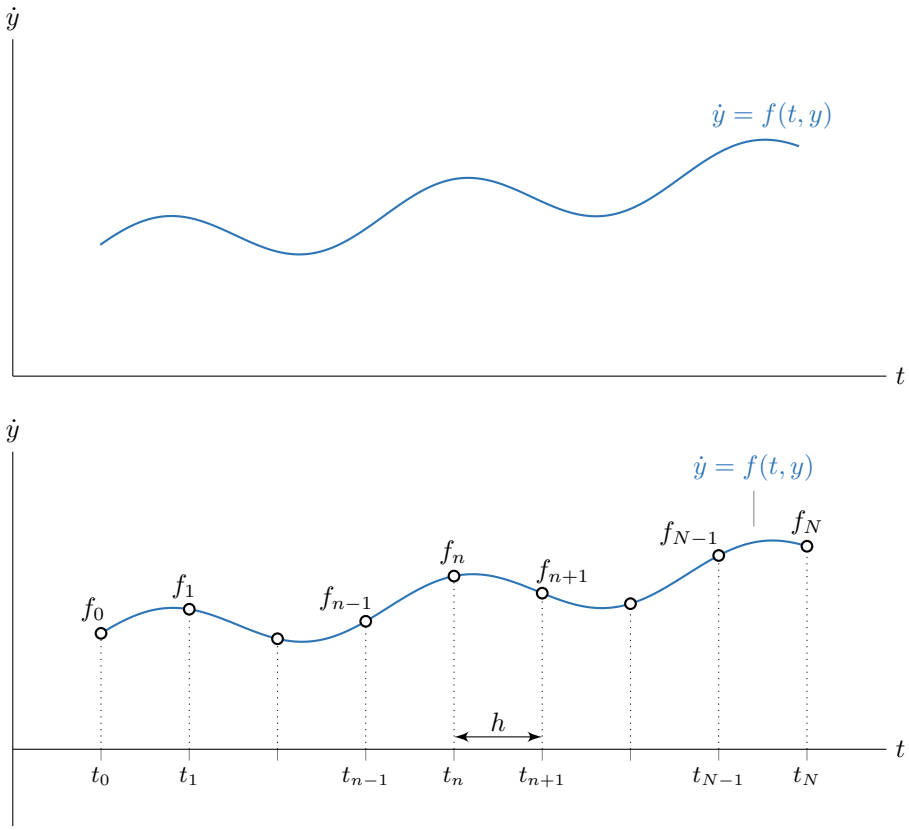


Figure 3.1: Discretization of an ODE in time.

3.3 A Brute Force Solution Method

One of the primary applications of ODE solvers is trajectory simulation. As a first introduction to the numerical solution to a trajectory, undergraduate courses often start with the fact that (a) velocity, $\mathbf{v}$, is the derivative of position, $\mathbf{r}$, and (b) acceleration, $\mathbf{a}$, is the derivative of velocity.

$$\mathbf{v} = \frac{d\mathbf{r}}{dt}, \qquad \mathbf{a} = \frac{d\mathbf{v}}{dt}$$

If we know the initial position and velocity ($\mathbf{r}_0 = \mathbf{r}(t_0)$ and $\mathbf{v}_0 = \mathbf{v}(t_0)$, respectively) at the initial time t_0, and if we know the acceleration as a function of time, position, and acceleration (i.e. $\mathbf{a} = \mathbf{a}(t, \mathbf{r}, \mathbf{v})$), then to find the position and velocity at any time t, we must evaluate the following expressions:

$$\mathbf{v}(t) = \mathbf{v}(t_0) + \int_{t_0}^{t} \mathbf{a}(t, \mathbf{r}, \mathbf{v})\, dt$$

$$\mathbf{r}(t) = \mathbf{r}(t_0) + \int_{t_0}^{t} \mathbf{v}(t, \mathbf{r}, \mathbf{v})\, dt$$

In simple cases, such as when $\mathbf{a}$ is a constant or is an integrable function of t, then we can evaluate these integrals analytically. However, in most cases, $\mathbf{a}$ is a very complicated function, and it is impossible or very difficult to solve these integrals analyically.

A first approach to solving such a problem numerically is discretizing time into very small increments (i.e. $t_{n+1} = t_n + \Delta t$, as illustrated in Figure 3.1), and assuming that the velocity and acceleration remain constant over those small time intervals. Mathematically, if we know the position, $\mathbf{r}_n = \mathbf{r}(t_n)$, velocity, $\mathbf{v}_n = \mathbf{v}(t_n)$, and acceleration,

$\mathbf{a}_n = \mathbf{a}(t_n)$, at the n^{th} sample time, t_n, then at the next (i.e. $(n+1)^{\text{th}}$) sample time, t_{n+1}, we have

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \int_{t_n}^{t_{n+1}} \mathbf{a}_n \, dt$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \int_{t_n}^{t_{n+1}} \mathbf{v}_n \, dt$$

Since we have assumed that $\mathbf{a}_n$ and $\mathbf{v}_n$ are constant over a short interval of time Δt, and since $\Delta t = t_{n+1} - t_n$,

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n \Delta t$$
$$\mathbf{r}_{n+1} = \mathbf{r}_n + \mathbf{v}_n \Delta t$$

For physical systems, we typically have an expression of the acceleration as $\mathbf{a} = \mathbf{a}(t, \mathbf{r}, \mathbf{v})$ from Newton's second law (i.e. it is our equation of motion/dynamics model). Thus, we calculate $\mathbf{a}_n$ using

$$\mathbf{a}_n = \mathbf{a}(t_n, \mathbf{r}_n, \mathbf{v}_n)$$

See Example 3.3.1 below for an example of this approach to solving a simple problem involving Newton's laws. We refer to this approach as a **brute force** approach for two main reasons:

1. It uses the simplest and least accurate approximation available (this is actually the Euler method; see Section 5.2).
2. It is problem-specific; that is, as you will see in the code in the example, for every single problem, you'd have to manually edit some aspects of the "solver".

Example 3.3.1: Brute force solution to the motion of a car.

Consider a car of mass $m = 1500$ kg, cross-sectional area $A = 2.5$ m^2, and drag coefficient $C_D = 0.25$ that starts from rest at $x = 0$ m. A constant force of $F_{\text{eng}} = 4500$ N is supplied by the engine to move the car. Find the velocity and acceleration of the car, as well as the drag force acting on the car, when the car is at a position of $x = 300$ m. Assume a constant air density of $\rho = 1.2$ kg/m^3.

■ ***SOLUTION***

Applying Newton's Second Law to find the acceleration yields

$$a = \frac{F_{\text{eng}} - F_D}{m}$$

where the drag force, F_D, is given by

$$F_D = \frac{1}{2} C_D A \rho v^2$$

Implementing a "brute force" approach in MATLAB to numerically solve for the resulting motion,

```
% car parameters
m = 1500;        % mass [kg]
A = 2.5;         % cross-sectional area [m^2]
CD = 0.25;       % drag coefficient [-]
F_eng = 4500;    % applied force [N]

% IVP parameters
x0 = 0;          % initial position [m]
v0 = 0;          % initial velocity [m/s]
xf = 300;        % final position [m]
```

```
% physical parameters
rho = 1.2;      % air density [kg/m^3]

% computational parameters
dt = 0.1;       % time step [s]

% preallocate arrays
t = zeros(10000,1);
x = zeros(size(t));
v = zeros(size(t));
a = zeros(size(t));

% solves initial value problem
n = 1;
while x(n) < xf

    % calculates drag force [N]
    FD = 0.5*CD*A*rho*v(n)^2;

    % calculates acceleration [m/s^2]
    a(n) = (F_eng-FD)/m;

    % integrates acceleration to find velocity [m/s]
    v(n+1) = v(n)+a(n)*dt;

    % integrates velocity to find position [m]
    x(n+1) = x(n)+v(n)*dt;

    % increments loop index
    n = n+1;

end

% trims arrays
t = t(1:(n-1));
x = x(1:(n-1));
v = v(1:(n-1));
a = a(1:(n-1));

% velocity and acceleration of car when position is x = 300 m
v300 = v(end);
a300 = a(end);

% prints results
fprintf("    velocity = %.3f m/s\n",v300)
fprintf("acceleration = %.3f m/s^2\n",a300)
fprintf("  drag force = %.3f N\n\n",FD)
```

The code above produces the following results:

```
    velocity = 40.853 m/s
acceleration = 2.583 m/s^2
  drag force = 625.858 N
```

3.4 General Form of an ODE Solver

An **ODE solver** (where "ODE" stands for "ordinary differential equation") is a computational function that solves *initial value problems* defined using ordinary differential equations[1]. ODE solvers act as a black box; we pass them a very complicated, vector-valued differential equation, an initial condition, an initial time, some termination condition (more on this later), and a step size, and it magically returns the solution over some interval of time. This basic functionality is depicted as a block diagram in Fig. 3.2.

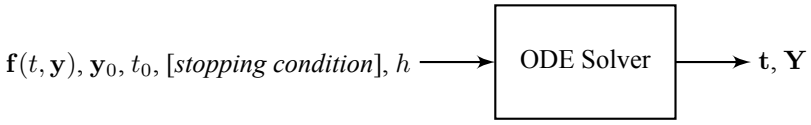


Figure 3.2: ODE solver block diagram.

The input and output parameters of an ODE solver are defined in more detail in Tables 3.1 and 3.2

Table 3.1: Inputs to an ODE solver.

Variable	Name	Description
$\mathbf{f}(t, \mathbf{y}) \in \mathbb{R}^{N+1}$	ODE	multivariate, vector-valued function ($\mathbf{f} : \mathbb{R} \times \mathbb{R}^p \to \mathbb{R}^p$) defining the ordinary differential equation $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$
$\mathbf{y} \in \mathbb{R}^p$	initial condition	initial condition $\mathbf{y}_0 = \mathbf{y}(t_0)$
[termination condition]	termination condition	condition to terminate the ODE solver; see Sections 3.5 and 3.6
$t_0 \in \mathbb{R}$	initial time	initial time
$h \in \mathbb{R}$	step size	step size between sample times ($t_{n+1} = t_n + h$)

Table 3.2: Outputs of an ODE solver.

Variable	Name	Description
$\mathbf{t} \in \mathbb{R}^{N+1}$	time vector	stores sequence of sample times (see Section 3.2)
$\mathbf{Y} \in \mathbb{R}^{(N+1)\times p}$	solution matrix	Stores the solution $\mathbf{y}(t)$ at discrete points in time. Specifically, the n^{th} *row* of $\mathbf{Y}$ stores $\mathbf{y}_n^T$, where $\mathbf{y}_n = \mathbf{y}(t_n) \in \mathbb{R}^p$ is the solution of the IVP at time t_n (where time t_n is stored in the $(n+1)^{\text{th}}$ element of $\mathbf{t}$).

[1] They do *not* solve boundary value problems; in this regard, the name "ODE solver" is somewhat of a misnomer, because ODE solvers do not solve all types of problems involving ordinary differential equations

Writing out the components of $\mathbf{t}$ and $\mathbf{Y}$,

$$\boxed{\mathbf{t} = \begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ \vdots \\ t_N \end{bmatrix}, \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{y}_0^T \\ \mathbf{y}_1^T \\ \mathbf{y}_2^T \\ \vdots \\ \mathbf{y}_N^T \end{bmatrix}} \tag{3.3}$$

Figure 3.2 depicts the block diagram of a generic ODE solver. However, there are two basic implementations of ODE solvers, discussed in the next two sections, that differ in how the termination condition

1. time detection – the IVP is solved until some specified final time (see Section 3.5)
2. event detection – the IVP is solved until some event occurs (see Section 3.6)

3.5 Time Detection Implementation

Most ODE solvers use what we refer to as the **time detection implementation**. In such an implementation, we are given an initial time t_0 and wish to solve the IVP until some final time t_f. The reason we refer to this as the time detection implementation is that the algorithm will continue solving the IVP until it *detects* that it has reached a *time*, t_f. A block diagram of an ODE solver using the time detection implementation is shown in Fig. 3.3.

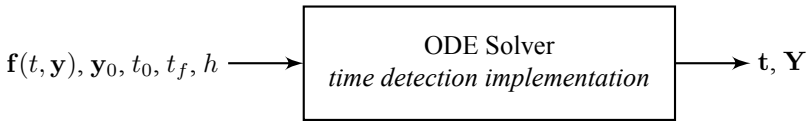


Figure 3.3: ODE solver block diagram (time detection implementation).

Recall from Eq. (3.3) that the ODE solver should produce a time vector $\mathbf{t} \in \mathbb{R}^{N+1}$, where the first element of $\mathbf{t}$ is equal to t_0. In this case, for the time detection implementation, we also want the last element of $\mathbf{t}$ to be equal to the specified final time, t_f (i.e. $t_N = t_f$). However, we also specify the **step size**, h, which is defined[2] as

$$\boxed{h = t_{n+1} - t_n} \tag{3.4}$$

This presents an issue because $t_f - t_0$ may not be an exact multiple of h. If we want to solve the ODE until t_f, we need to choose the number of sample times $N+1$ such that the time at the last sample time is *greater than or equal to* t_f. To do this, we simply use the ceiling function (i.e. rounding up).

$$\boxed{N = \left\lceil \frac{t_f - t_0}{h} \right\rceil} \tag{3.5}$$

Thus, the last element of $\mathbf{t}$ is

$$\boxed{t_N = t_0 + Nh} \tag{3.6}$$

The issue now is that $t_N \neq t_f$ (and consequently $\mathbf{y}_N \neq \mathbf{y}(t_f)$). To solve this issue, in the last step of an ODE solver algorithm, we use linear interpolation to estimate $\mathbf{y}_f = \mathbf{y}(t_f)$.

$$\boxed{\mathbf{y}_f = \mathbf{y}_{N-1} + \left[\frac{\mathbf{y}_N - \mathbf{y}_{N-1}}{t_N - t_{N-1}} \right] (t_f - t_{N-1})} \tag{3.7}$$

[2] Essentially, the step size is the time step, Δt, that we used in Example 3.3.1.

Finally, we replace the last row of $\mathbf{Y}$ with $\mathbf{y}_f^T$.

$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}_0^T \\ \mathbf{y}_1^T \\ \mathbf{y}_2^T \\ \vdots \\ \mathbf{y}_N^T \end{bmatrix} \xrightarrow[\text{obtained via linear interpolation}]{\text{replacing last row with the } \mathbf{y}_f^T} \mathbf{Y} = \begin{bmatrix} \mathbf{y}_0^T \\ \mathbf{y}_1^T \\ \mathbf{y}_2^T \\ \vdots \\ \mathbf{y}_f^T \end{bmatrix}$$

Algorithm 3 below shows the basic time detection implementation of an ODE solver.

Algorithm 3:
Time detection implementation of an ODE solver for single-step methods.

Given:

- $\mathbf{f}(t, \mathbf{y})$ - multivariate, vector-valued function ($\mathbf{f} : \mathbb{R} \times \mathbb{R}^p \to \mathbb{R}^p$) defining the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$
- $t_0 \in \mathbb{R}$ - initial time
- $t_f \in \mathbb{R}$ - final time
- $\mathbf{y}_0 \in \mathbb{R}^p$ - initial condition
- $h \in \mathbb{R}$ - step size

Procedure:

1. Make the step size negative if $t_0 > t_f$.

   if $t_0 > t_f$
   $\quad h = -h$
   end

2. Determine the number of subintervals between sample times.

   $$N = \left\lceil \frac{t_f - t_0}{h} \right\rceil - 1$$

3. Calculate the last element of the time vector.

   $$t_N = t_0 + Nh$$

4. Define the time vector $\mathbf{t} \in \mathbb{R}^{N+1}$ with first element t_0, last element t_N, and all intermediate elements equally spaced by h.

   $$\mathbf{t} = \begin{bmatrix} t_0 \\ t_0 + h \\ t_0 + 2h \\ \vdots \\ t_0 + (N-1)h \\ t_N \end{bmatrix}$$

5. Preallocate the transpose of the solution matrix $\mathbf{Y} \in \mathbb{R}^{(N+1)\times p}$.

   $$\mathbf{Y}^T = \mathbf{0}_{p\times(N+1)}$$

6. Store the initial condition in the solution matrix.

   $$(\mathbf{Y}^T)_{1:p,1} = \mathbf{y}_0$$

7. Propagate the state vector (i.e. find $\mathbf{y}_{n+1}$ using some approximation technique) and store it in the appropriate column of $\mathbf{Y}^T$ (i.e. solve the IVP).

   for $n = 1$ **to** $N - 1$
   $\quad \mathbf{y}_{n+1} = \cdots$
   end

8. Estimate $\mathbf{y}_f = \mathbf{y}(t_f)$ using linear interpolation.

$$\mathbf{y}_f = \mathbf{y}_{N-1} + \left[\frac{\mathbf{y}_N - \mathbf{y}_{N-1}}{t_N - t_{N-1}}\right](t_f - t_{N-1})$$

9. Replace the last column of $\mathbf{Y}^T$ with $\mathbf{y}_f$ and the last element of $\mathbf{t}$ with t_f.

Return:

- $\mathbf{t} \in \mathbb{R}^{N+1}$ - time vector
- $\mathbf{Y} \in \mathbb{R}^{(N+1)\times p}$ - solution matrix (the n^{th} row of $\mathbf{Y}$ stores $\mathbf{y}_n^T$, where $\mathbf{y}_n \in \mathbb{R}^p$ is the solution at $t = t_n$)

In the actual implementation of this algorithm, we use a lowercase variable, $\mathbf{y}$, to represent the solution matrix, $\mathbf{Y}$. This is in part due to convention but also because $\mathbf{y}$ stores the approximation to $\mathbf{y}(t)$. While it is important to distinguish between $\mathbf{y}$ and $\mathbf{Y}$ in a mathematical setting (where these methods are defined using mathematical notation), in a computational setting it makes sense to just denote $\mathbf{Y}$ as $\mathbf{y}$ because it represents the approximation to $\mathbf{y}(t)$.

Additionally, the indexing appears to be off by 1 in the MATLAB implementation of the linear interpolation for $\mathbf{y}_f$ versus what is shown in Algorithm 3. However, the indexing in the implementation is correct; the reason the indexing is different is because MATLAB uses 1-based indexing, so for example, the N^{th} sample time is stored in the $(N+1)^{\text{th}}$ element of $\mathbf{t}$ when using 1-based indexing.

3.6 Event Detection Implementation

To motivate the need for an event detection implementation, consider a satellite slowly deorbiting due to atmospheric drag. We want to keep solving for its motion until it impacts the Earth. In the time detection implementation, we specified a final time until which we wished to keep solving the IVP. However, in this case, we don't know ahead of time when the satellite will impact the Earth. We could make some guesses for t_f, but either we underestimate it and the satellite never impacts the Earth before the solver terminates, or we overestimate it and waste time and computational effort calculating motion that (a) we don't need to and (b) doesn't make sense (we would essentially be simulating the trajectory of the satellite as if the Earth isn't there).

An **event detection** implementation of an ODE solver terminates the solver at the occurrence of an *event* rather than at a certain time. Instead of passing a final time t_f to the ODE solver, we pass it a **condition function** $C(t, \mathbf{y})$.

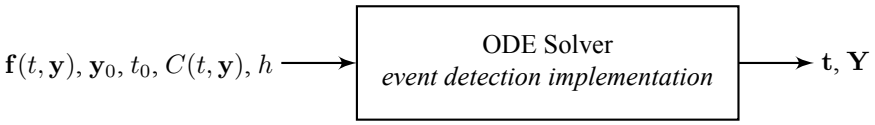


Figure 3.4: ODE solver block diagram (event detection implementation).

Based on the time, t, and/or the current value of the state vector, $\mathbf{y}$, the condition function $C(t, \mathbf{y})$ determines whether the solver should continue its solution process or if the solver should be terminated. Therefore, the condition function is evaluated at each and every iteration of the ODE solver. Below, we first introduce a more mathematical definition of a condition function, to keep in line with our practice thus far of introducing everything from a mathematical

standpoint.

The condition function is defined in a piecewise manner, where it will evaluate to some value while the ODE solver should continue integrating. However, at some point, the condition function will switch to a different value, signaling to the ODE solver that it should be terminated. Specifically, the condition function returns 1 while the condition is satisfied, and 0 after it is no longer satisfied.

Convention 3: Mathematical definition of the condition function.

A condition function $C(t, \mathbf{y})$ (in a mathematical setting) is defined such that it returns 1 *until* the event (which defines the point at which the solver should be terminated) occurs, after which it will return 0.

$$C(t, \mathbf{y}) = \begin{cases} 1, & \text{event has } \textit{not} \text{ occurred; continue integration} \\ 0, & \text{event has occurred; terminate integration} \end{cases}$$

Thus, while $C(t, \mathbf{y}) = 1$, the solver should continue integrating to find the solution.

In a computational setting, we have the condition function `C(t,y)`, which returns `true` if integration should continue and `false` if it should be terminated.

Convention 4: Computational definition of the condition function.

A condition function `C(t,y)` (in a computational setting) is defined such that it returns `true` *until* the event (which defines the point at which the solver should be terminated) occurs, after which it will return `false`.

Thus, while `C(t,y) = true`, the solver should continue integrating to find the solution.

In MATLAB, the integer `1` can be used interchangeably with the logical `true`. Similarly, the integer `0` can be used interchangeably with the logical `false`.

Example 3.6.1: Defining a condition function.

Implement a condition function $C(t, \mathbf{y})$ in MATLAB that can be used to terminate an ODE solver when y_2 is greater than 300 *and* y_5 is negative (these two conditions should be satisfied simultaneously for the solver to terminate).

■ ***SOLUTION***

We can write the condition function in a single line as an anonymous function.

```
C = @(t,y) ~((y(2) > 300) && (y(5) < 0));
```

■ ***ALTERNATE SOLUTION***

Alternatively, we can use a MATLAB function with an if/else structure, and then assign a function handle to the MATLAB function. While in this case it is (a) not as efficient and (b) unnecessary to do this, for more complicated condition functions this may be necessary.

```
% assign function handle
C = @(t,y) condition(t,y);

% MATLAB functions must be at end of script
function C = condition(t,y)
    if (y(2) > 300) && (y(5) < 0)
        C = false;
    else
        C = true;
    end
end
```

■ *ALTERNATE SOLUTION*

There exists yet another alternate solution, which is identical to the previous solution, but uses `1` instead of `true` and `0` instead of false `false` (since these values are interchangeable in MATLAB).

```
% assign function handle
C = @(t,y) condition(t,y);

% MATLAB functions must be at end of script
function C = condition(t,y)
    if (y(2) > 300) && (y(5) < 0)
        C = 0;
    else
        C = 1;
    end
end
```

Note that in this example, we never used the time variable. However, in some cases we will need to, and by convention, we must still include it in our function definition.

Algorithm 4 below shows the basic event detection implementation of an ODE solver.

Algorithm 4:
Event detection implementation of an ODE solver for single-step methods.

Given:

- $\mathbf{f}(t, \mathbf{y})$ - multivariate, vector-valued function ($\mathbf{f} : \mathbb{R} \times \mathbb{R}^p \rightarrow \mathbb{R}^p$) defining the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$
- $t_0 \in \mathbb{R}$ - initial time
- $C(t, \mathbf{y})$ - condition function ($C : \mathbb{R} \times \mathbb{R}^p \rightarrow \mathbb{B}$)
- $\mathbf{y}_p \in \mathbb{R}^n$ - initial condition
- $h \in \mathbb{R}$ - step size

Procedure:

1. Preallocate the time vector, $\mathbf{t} \in \mathbb{R}^{10000}$, and the transpose of the solution matrix, where $\mathbf{Y} \in \mathbb{R}^{10000 \times p}$.

$$\mathbf{t} = \mathbf{0}_{10000 \times 1}$$
$$\mathbf{Y}^T = \mathbf{0}_{p \times 10000}$$

2. Store the initial time, t_0 in the first element of $\mathbf{t}$, and the initial condition, $\mathbf{y}_0$, in the first column of $\mathbf{Y}^T$.
3. Solve the IVP.

$$n = 1$$

while $C(t_n, \mathbf{y}_n)$

(a) Double the size of (i.e. preallocate more memory to) $\mathbf{t}$ and $\mathbf{Y}^T$ if these arrays have run out of room.

(b) Propagate the state vector (i.e. find $\mathbf{y}_{n+1}$ using some approximation technique) and store it in the appropriate column of $\mathbf{Y}^T$.

$$\mathbf{y}_{n+1} = \cdots$$

(c) Increment time and the loop index.

$$t_{n+1} = t_n + h$$
$$n = n + 1$$

end

Return:

- $\mathbf{t} \in \mathbb{R}^{N+1}$ - time vector
- $\mathbf{Y} \in \mathbb{R}^{(N+1) \times p}$ - solution matrix (the n^{th} row of $\mathbf{Y}$ stores $\mathbf{y}_n^T$, where $\mathbf{y}_n \in \mathbb{R}^p$ is the solution at $t = t_n$)

3.7 ODE Solvers for Multistep Methods

A common class of method for solving ODEs are **multistep methods** that utilize the solutions at previous sample times to approximate the solution at the next sample time. More specifically, a multistep method of order m will evaluate $\mathbf{f}(t, \mathbf{y})$ at m sample times:

1. $\mathbf{f}(t_n, \mathbf{y}_n)$
2. $\mathbf{f}(t_{n-1}, \mathbf{y}_{n-1})$
3. $\mathbf{f}(t_{n-2}, \mathbf{y}_{n-2})$

$\vdots$

m. $\mathbf{f}(t_{n-m+1}, \mathbf{y}_{n-m+1})$

Typically, function evaluations are the most expensive computation in an ODE solver. To avoid repeating function

evaluations unnecessarily, we define the matrix $\mathbf{F}_m \in \mathbb{R}^{p\times m}$ as[3]

$$\mathbf{F}_m = \left[\begin{array}{c|c|c|c|c} | & | & | & & | \\ \mathbf{f}(t_n,\mathbf{y}_n) & \mathbf{f}(t_{n-1},\mathbf{y}_{n-1}) & \mathbf{f}(t_{n-2},\mathbf{y}_{n-2}) & \cdots & \mathbf{f}(t_{n-m+1},\mathbf{y}_{n-m+1}) \\ | & | & | & & | \end{array}\right] \tag{3.8}$$

Since we need the solutions at m sample times to populate $\mathbf{F}_m$, we need to use a single-step solver for the first m iterations of the multistep solver (i.e. multistep methods are *not* self-starting) [6, pp. 133-134]. Therefore, for multistep methods, we need to udpate Algorithms 3 and 4 accordingly.

Algorithm 5:
Time detection implementation of an ODE solver for multistep methods.

Given:

- $\mathbf{f}(t,\mathbf{y})$ - multivariate, vector-valued function ($\mathbf{f}:\mathbb{R}\times\mathbb{R}^p\to\mathbb{R}^p$) defining the ODE $d\mathbf{y}/dt = \mathbf{f}(t,\mathbf{y})$
- $t_0 \in \mathbb{R}$ - initial time
- $t_f \in \mathbb{R}$ - final time
- $\mathbf{y}_0 \in \mathbb{R}^p$ - initial condition
- $h \in \mathbb{R}$ - step size

Procedure:

1. Define the order, m, of the multistep method.
2. Make the step size negative if $t_0 > t_f$.

   if $t_0 > t_f$
   $\quad h = -h$
   end

3. Determine the number of subintervals between sample times.

   $$N = \left\lceil \frac{t_f - t_0}{h} \right\rceil - 1$$

4. Calculate the last element of the time vector.

   $$t_N = t_0 + Nh$$

5. Define the time vector $\mathbf{t} \in \mathbb{R}^{N+1}$ with first element t_0, last element t_N, and all intermediate elements equally spaced by h.

   $$\mathbf{t} = \begin{bmatrix} t_0 \\ t_0 + h \\ t_0 + 2h \\ \vdots \\ t_0 + (N-1)h \\ t_N \end{bmatrix}$$

6. Preallocate the transpose of the solution matrix $\mathbf{Y}$, where $\mathbf{Y} \in \mathbb{R}^{(N+1)\times p}$.

   $$\mathbf{Y}^T = \mathbf{0}_{p\times(N+1)}$$

[3] Similar to how we use `y` in MATLAB to represent $\mathbf{Y}$ (see Section 3.5), we use `fm` in MATLAB to represent $\mathbf{F}_m$.

7. Store the initial condition in the solution matrix.

$$(\mathbf{Y}^T)_{1:p,1} = \mathbf{y}_0$$

8. Preallocate the matrix $\mathbf{F}_m \in \mathbb{R}^{p \times m}$ to store the previous m function evaluations.
9. Propagate the state vector using the Runge-Kutta fourth-order method (`RK4`, see Section 5.5) for the first m sample times, and store the results in the appropriate columns of $\mathbf{Y}^T$.

for $n = 0$ **to** $m - 1$

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}\left(t_n, \mathbf{y}_n\right) \\
\mathbf{k}_2 &= \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_1}{2}\right) \\
\mathbf{k}_3 &= \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_2}{2}\right) \\
\mathbf{k}_4 &= \mathbf{f}\left(t_n + h, \mathbf{y}_n + h\mathbf{k}_3\right) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right)
\end{aligned}$$

end

10. Store function evaluations for first m sample times in $\mathbf{F}_m$.

for $n = 1$ **to** m

$$(\mathbf{F}_m)_{1:p,n} = \mathbf{f}(t_n, \mathbf{y}_n)$$

end

11. Continue solving the IVP using a multistep method and store the solutions in the appropriate columns of $\mathbf{Y}^T$.

for $n = m$ **to** $N - 1$

(a) Update $\mathbf{F}_m$ by shifting all the columns to the left (in the process discarding the first/oldest function evaluation in the first column), and add the function evaluation of the most recent solution to the last column.

$$\mathbf{F}_m = \left[\begin{array}{c|c} & | \\ (\mathbf{F}_m)_{1:p,2:m} & \mathbf{f}(t_n, \mathbf{y}_n) \\ & | \end{array}\right]$$

(b) Propagate state vector using a multistep method.

$$\mathbf{y}_{n+1} = \cdots$$

end

12. Estimate $\mathbf{y}_f = \mathbf{y}(t_f)$ using linear interpolation.

$$\mathbf{y}_f = \mathbf{y}_{N-1} + \left[\frac{\mathbf{y}_N - \mathbf{y}_{N-1}}{t_N - t_{N-1}}\right](t_f - t_{N-1})$$

13. Replace the last column of $\mathbf{Y}^T$ with $\mathbf{y}_f$ and the last element of $\mathbf{t}$ with t_f.

Return:

- $\mathbf{t} \in \mathbb{R}^{N+1}$ - time vector
- $\mathbf{Y} \in \mathbb{R}^{(N+1)\times p}$ - solution matrix (the n^{th} row of $\mathbf{Y}$ stores $\mathbf{y}_n^T$, where $\mathbf{y}_n \in \mathbb{R}^p$ is the solution at $t = t_n$)

Algorithm 6:

Event detection implementation of an ODE solver for multistep methods.

Given:

- $\mathbf{f}(t, \mathbf{y})$ - multivariate, vector-valued function ($\mathbf{f} : \mathbb{R} \times \mathbb{R}^p \to \mathbb{R}^p$) defining the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$
- $t_0 \in \mathbb{R}$ - initial time
- $C(t, \mathbf{y})$ - condition function ($C : \mathbb{R} \times \mathbb{R}^p \to \mathbb{B}$)
- $\mathbf{y}_p \in \mathbb{R}^n$ - initial condition
- $h \in \mathbb{R}$ - step size

Procedure:

1. Define the order, m, of the multistep method.
2. Preallocate the time vector, $\mathbf{t} \in \mathbb{R}^{10000}$, and the transpose of the solution matrix, where $\mathbf{Y} \in \mathbb{R}^{10000\times p}$.

$$\mathbf{t} = \mathbf{0}_{10000\times 1}$$
$$\mathbf{Y}^T = \mathbf{0}_{p\times 10000}$$

3. Populate the first m elements of the time vector.

$$\mathbf{t}_{1:m} = \begin{bmatrix} t_0 \\ t_0 + h \\ t_0 + 2h \\ \vdots \\ t_0 + (m-2)h \\ t_0 + (m-1)h \end{bmatrix}$$

4. Store initial condition in the solution matrix (i.e. in the first column of $\mathbf{Y}^T$).

$$(\mathbf{Y}^T)_{1:p,1} = \mathbf{y}_0$$

5. Preallocate the matrix $\mathbf{F}_m \in \mathbb{R}^{p\times m}$ to store the previous m function evaluations.
6. Propagate the state vector using the Runge-Kutta fourth-order method (`RK4`, see Section 5.5) for the first m sample times, and store the results in the appropriate columns of $\mathbf{Y}^T$.

for $n = 0$ **to** $m - 1$

$$\mathbf{k}_1 = \mathbf{f}\left(t_n, \mathbf{y}_n\right)$$
$$\mathbf{k}_2 = \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_1}{2}\right)$$
$$\mathbf{k}_3 = \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_2}{2}\right)$$
$$\mathbf{k}_4 = \mathbf{f}\left(t_n + h, \mathbf{y}_n + h\mathbf{k}_3\right)$$
$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right)$$

end

7. Store function evaluations for first m sample times in $\mathbf{F}_m$.

for $n = 1$ **to** m

$$(\mathbf{F}_m)_{1:p,n} = \mathbf{f}(t_n, \mathbf{y}_n)$$

end

8. Continue solving the IVP using a multistep method and store the solutions in the appropriate columns of $\mathbf{Y}^T$.

$n = 1$

while $C(t_n, \mathbf{y}_n)$

(a) Double the size of (i.e. preallocate more memory to) $\mathbf{t}$ and $\mathbf{Y}^T$ if these arrays have run out of room.

(b) Update $\mathbf{F}_m$ by shifting all the columns to the left (in the process discarding the first/oldest function evaluation in the first column), and add the function evaluation of the most recent solution to the last column.

$$\mathbf{F}_m = \left[\begin{array}{c|c} & | \\ (\mathbf{F}_m)_{1:p,2:m} & \mathbf{f}(t_n, \mathbf{y}_n) \\ & | \end{array}\right]$$

(c) Propagate state vector using a multistep method.

$$\mathbf{y}_{n+1} = \cdots$$

(d) Increment time and the loop index.

$$t_{n+1} = t_n + h$$
$$n = n + 1$$

end

Return:

- $\mathbf{t} \in \mathbb{R}^{N+1}$ - time vector
- $\mathbf{Y} \in \mathbb{R}^{(N+1)\times p}$ - solution matrix (the n^{th} row of $\mathbf{Y}$ stores $\mathbf{y}_n^T$, where $\mathbf{y}_n \in \mathbb{R}^p$ is the solution at $t = t_n$)

3.8 One-Step Propagation

In Sections 3.4–3.5, we discussed solving ODE's over some time span $[t_0, t_f]$. However, in some cases, it is useful to just return the solution, $\mathbf{y}$, at the very next sample time (i.e. $\mathbf{y}_{k+1}$ at time t_{k+1}), given the solution at the current sample time (i.e. $\mathbf{y}_k$ at time t_k). We refer to this as **one-step propagation** (not to be confused with the single-step methods detailed in Chapter 5), since the state vector (i.e. the solution) is being propagated forward "one step" in time. The block diagram representing one-step propagation is shown in Fig. 3.5.

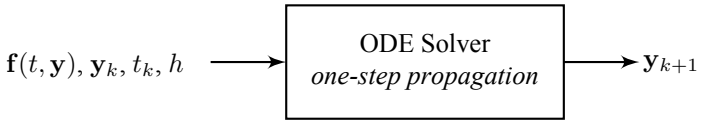


Figure 3.5: ODE solver block diagram (one-step propagation).

Since we don't have to deal with termination conditions, loops, time vectors, etc., the procedure for one-step propagation is much simpler than for the ODE solver implementations introduced in the previous two sections. In fact, the calculation itself is only a single step, defined using the single-step methods outlined in Chapter 5. Algorithm 7 is included below, primarily to illustrate the input/output behavior of a one-step propagation ODE solver.

Algorithm 7:
One-step propagation

Given:
- $\mathbf{f}(t, \mathbf{y})$ - multivariate, vector-valued function ($\mathbf{f} : \mathbb{R} \times \mathbb{R}^p \to \mathbb{R}^p$) defining the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$
- $t_k \in \mathbb{R}$ - current sample time
- $\mathbf{y}_k \in \mathbb{R}^p$ - state vector (i.e. solution) at current sample time
- $h \in \mathbb{R}$ - step size

Procedure:
Propagate state vector to next sample time using one of the single-step methods from Sections 5.2–5.5.

Return:
- $\mathbf{y}_{k+1} \in \mathbb{R}^{(N+1)\times p}$ - state vector (i.e. solution) at the next sample time (i.e. $t_{k+1} = t_k + h$)

3.9 0- vs. 1-Based Indexing

The time vector, $\mathbf{t}$, is defined using 0-based indexing (i.e. its first element has index 0 and stores t_0). Similarly, the solution matrix, $\mathbf{Y}$, is also defined using 0-based indexing (i.e. its first row has index 0 and stores $\mathbf{y}_0^T$). Thus, Algorithms 3–6 are all written using 0-based indexing. However, MATLAB uses 1-based indexing. This causes the implementation of these algorithms in MATLAB to appear to have off-by-one indexing errors. However, the indices in the MATLAB implementation are shifted by one to account for this difference between 0-based and 1-based indexing.

4

Matrix-Valued Differential Equations

4.1 Definition

Until now, we have considered vector-valued ODE's of the form

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y})$$

where $\mathbf{y} \in \mathbb{R}^p$ is a *vector* and where $\mathbf{f} : \mathbb{R} \times \mathbb{R}^p \to \mathbb{R}^p$. However, differential equations of the form

$$\frac{d\mathbf{M}}{dt} = \mathbf{F}(t, \mathbf{M})$$

are also common, where $\mathbf{M} \in \mathbb{R}^{p \times q}$ is a matrix and where $\mathbf{F} : \mathbb{R} \times \mathbb{R}^{p \times q} \to \mathbb{R}^{p \times q}$ is a **matrix-valued ODE**.

For vector-valued ODE's, we referred to $\mathbf{y}$ as the state vector. In the context of matrix-valued ODE's, we refer to $\mathbf{M}$ as the **state matrix**[1].

4.2 Transforming a Matrix-Valued ODE into a Vector-Valued ODE

Consider writing the matrix $\mathbf{M} \in \mathbb{R}^{p \times q}$ in terms of its column vectors:

$$\boxed{\mathbf{M} = \begin{bmatrix} | & & | \\ \mathbf{m}_1 & \cdots & \mathbf{m}_q \\ | & & | \end{bmatrix}} \tag{4.1}$$

We know that

$$\frac{d\mathbf{M}}{dt} = \mathbf{F}(t, \mathbf{M})$$

$$\begin{bmatrix} | & & | \\ \frac{d\mathbf{m}_1}{dt} & \cdots & \frac{d\mathbf{m}_q}{dt} \\ | & & | \end{bmatrix} = \mathbf{F}\left(t, \begin{bmatrix} | & & | \\ \mathbf{m}_1 & \cdots & \mathbf{m}_q \\ | & & | \end{bmatrix}\right)$$

[1] Not to be confused with the matrix $\mathbf{A}$ in the dynamics equation $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$, which is also often called the state matrix (in the context of linear systems and control theory).

where $\mathbf{F} : \mathbb{R} \times \mathbb{R}^{p\times q} \to \mathbb{R}^{p\times q}$.

Our goal is to write the matrix-valued ODE, $\mathbf{F}$, as a vector-valued ODE, $\mathbf{f}$, where

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y})$$

Let's imagine stacking all the column vectors of $\mathbf{M}$ on top of each other. Then our goal becomes to write a function of the form

$$\boxed{\begin{bmatrix} d\mathbf{m}_1/dt \\ \vdots \\ d\mathbf{m}_q/dt \end{bmatrix} = \mathbf{f}\left(t, \begin{bmatrix} \mathbf{m}_1 \\ \vdots \\ \mathbf{m}_q \end{bmatrix}\right)} \tag{4.2}$$

Since the matrix $\mathbf{M}$ has pq entries, stacking all the column vectors of $\mathbf{M}$ on top of one another will produce a vector of length pq. Thus, we have $\mathbf{f} : \mathbb{R} \times \mathbb{R}^{pq} \to \mathbb{R}^{pq}$.

In an ODE solver setting, our vector-valued function $\mathbf{f}$ should have the inputs t and $\mathbf{y} \in \mathbb{R}^{pq}$, where

$$\boxed{\mathbf{y} = \begin{bmatrix} \mathbf{m}_1 \\ \vdots \\ \mathbf{m}_q \end{bmatrix}} \tag{4.3}$$

It follows that essentially, given $\mathbf{y} \in \mathbb{R}^{pq}$, we construct $\mathbf{M} \in \mathbb{R}^{p\times q}$, evaluate the matrix-valued ODE $\mathbf{F}$ to find $d\mathbf{M}/dt \in \mathbb{R}^{p\times q}$, and then construct $d\mathbf{y}/dt \in \mathbb{R}^{pq}$ from $d\mathbf{M}/dt$. We formalize this procedure for defining a vector-valued ODE $\mathbf{f}(t, \mathbf{y})$ in terms of a matrix-valued ODE $\mathbf{F}(t, \mathbf{M})$ as Algorithm 8.

Algorithm 8: `odefun_mat2vec`

Transforms a matrix-valued ODE into a vector-valued ODE.

Given:

- $\mathbf{F}(t, \mathbf{M})$ - multivariate, matrix-valued function ($\mathbf{f} : \mathbb{R} \times \mathbb{R}^{p\times q} \to \mathbb{R}^{p\times q}$) defining the ODE $d\mathbf{M}/dt = \mathbf{F}(t, \mathbf{M})$
- $t \in \mathbb{R}$ - current time
- $\mathbf{y} \in \mathbb{R}^{pq}$ - current state vector, where $\mathbf{y}$ is a vector formed by stacking the column vectors of $\mathbf{M}$ on top of one another
- $p \in \mathbb{R}$ - (OPTIONAL) number of rows of $\mathbf{M}$

Procedure:

1. Determine the product pq, given that $\mathbf{y} \in \mathbb{R}^{pq}$.
2. If p is not input, assume that $\mathbf{M}$ is square (i.e. $q = p$, so $\mathbf{M} \in \mathbb{R}^{p^2}$), and calculate p accordingly.

   if (p is not input)
   $\quad p = \sqrt{pq}$
   end

3. Determine q.

   $$q = \frac{pq}{p}$$

4. Obtain $\mathbf{M} \in \mathbb{R}^{p\times q}$ from $\mathbf{y} \in \mathbb{R}^{pq}$.

$$\mathbf{y} = \begin{bmatrix} \mathbf{m}_1 \\ \vdots \\ \mathbf{m}_q \end{bmatrix} \quad \rightarrow \quad \mathbf{M} = \begin{bmatrix} | & & | \\ \mathbf{m}_1 & \cdots & \mathbf{m}_q \\ | & & | \end{bmatrix}$$

   This can be done in MATLAB using the `reshape` function:

$$\mathbf{M} = \texttt{reshape}(\mathbf{y}, [p, q])$$

5. Evaluate the matrix-valued ODE.

$$\frac{d\mathbf{M}}{dt} = \mathbf{F}(t, \mathbf{M})$$

6. Obtain $d\mathbf{y}/dt \in \mathbb{R}^{pq}$ from $d\mathbf{M}/dt \in \mathbb{R}^{p\times q}$.

$$\frac{d\mathbf{M}}{dt} = \begin{bmatrix} | & & | \\ \dfrac{d\mathbf{m}_1}{dt} & \cdots & \dfrac{d\mathbf{m}_q}{dt} \\ | & & | \end{bmatrix} \quad \rightarrow \quad \frac{d\mathbf{y}}{dt} = \begin{bmatrix} d\mathbf{m}_1/dt \\ \vdots \\ d\mathbf{m}_q/dt \end{bmatrix}$$

   This can be done in MATLAB as

$$\frac{d\mathbf{y}}{dt} = \frac{d\mathbf{M}}{dt}\texttt{(:)}$$

Return:

- $\dfrac{d\mathbf{y}}{dt} \in \mathbb{R}^{pq}$ - state vector derivative

Similarly, given an IVP involving a matrix-valued ODE, the associated initial condition will also be a matrix:

$$\mathbf{M}(t_0) = \mathbf{M}_0 \in \mathbb{R}^{p\times q}$$

Algorithm 8 essentially defines $\mathbf{f}(t, \mathbf{y})$ as a calculation procedure. To obtain the associated vector initial condition $\mathbf{y}_0$ for use in an ODE solver, we can simply note that

$$\mathbf{M}_0 = \begin{bmatrix} | & & | \\ \mathbf{m}_{1,0} & \cdots & \mathbf{m}_{q,0} \\ | & & | \end{bmatrix} \quad \rightarrow \quad \mathbf{y}_0 = \begin{bmatrix} \mathbf{m}_{1,0} \\ \vdots \\ \mathbf{m}_{q,0} \end{bmatrix}$$

Defining this step as Algorithm 9,

Algorithm 9: `odeIC_mat2vec`

Transforms a matrix initial condition into a vector initial condition.

Given:

- $\mathbf{M}_0 \in \mathbb{R}^{p\times q}$ - initial condition for matrix-valued ODE

Procedure: Obtain $\mathbf{y}_0 \in \mathbb{R}^{pq}$ from $\mathbf{M} \in \mathbb{R}^{p\times q}$.

$$\mathbf{M}_0 = \begin{bmatrix} | & & | \\ \mathbf{m}_{1,0} & \cdots & \mathbf{m}_{q,0} \\ | & & | \end{bmatrix} \quad \rightarrow \quad \mathbf{y}_0 = \begin{bmatrix} \mathbf{m}_{1,0} \\ \vdots \\ \mathbf{m}_{q,0} \end{bmatrix}$$

This can be done in MATLAB as

$\mathbf{y}_0 = \mathbf{M}_0$`(:)`

Return:

- $\mathbf{y}_0 \in \mathbb{R}^{pq}$ - initial condition for corresponding vector-valued ODE

4.3 Obtaining the Matrix Results

`odefun_mat2vec` and `odeIC_mat2vec` (Algorithms 8 and 9, respectively) from the previous section essentially help us define the IVP

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(t_0) = \mathbf{y}_0$$

Running an ODE solver on this IVP will give us the solution matrix $\mathbf{Y} \in \mathbb{R}^{(N+1)\times pq}$. The n^{th} *row* of $\mathbf{Y}$ stores $\mathbf{y}_n^T$, where $\mathbf{y}_n = \mathbf{y}(t_n) \in \mathbb{R}^{pq}$ is the solution of the IVP at time t_n. We want the solution array $\mathbf{M}_{\text{sol}} \in \mathbb{R}^{p\times q\times (N+1)}$, where the n^{th} *layer* of $\mathbf{M}_{\text{sol}}$ stores the solution, $\mathbf{M}(t_n)$, at time t_n to the ODE

$$\frac{d\mathbf{M}}{dt} = \mathbf{F}(t, \mathbf{M}), \quad \mathbf{M}(t_0) = \mathbf{M}_0$$

To obtain $\mathbf{M}_{\text{sol}}$, we can use Algorithm 10.

Algorithm 10: `odesol_vec2mat`

Transforms the solution matrix for a vector-valued ODE into the solution array for the corresponding matrix-valued ODE.

Given:

- $\mathbf{Y} \in \mathbb{R}^{(N+1)\times pq}$ - solution matrix (the n^{th} row of $\mathbf{Y}$ stores $\mathbf{y}_n^T$, where $\mathbf{y}_n \in \mathbb{R}^{pq}$ is the solution at $t = t_n$)
- $p \in \mathbb{R}$ - (OPTIONAL) number of rows of $\mathbf{M}$

Procedure:

1. Determine the product pq and the scalar N, given that $\mathbf{Y} \in \mathbb{R}^{(N+1)\times pq}$.
2. If p is not input, assume that $\mathbf{M}$ is square (i.e. $q = p$ so $\mathbf{M} \in \mathbb{R}^{p^2}$), and calculate p accordingly.

if (p is not input)
  $p = \sqrt{pq}$
end

3. Determine q.

$$q = \frac{pq}{p}$$

4. Preallocate the solution array $\mathbf{M}_{\text{sol}} \in \mathbb{R}^{p \times q \times (N+1)}$.

$$\mathbf{M}_{\text{sol}} = \mathbf{0}_{p \times q \times (N+1)}$$

5. Populate the solution array.

for $n = 1$ **to** $N + 1$
$\quad (\mathbf{M}_{\text{sol}})_{1:p,1:q,n} = \texttt{reshape}(\mathbf{Y}_{n,1:pq}, [p, q])$
end

Return:

- $\mathbf{M}_{\text{sol}} \in \mathbb{R}^{p \times q \times (N+1)}$ - solution array (n^{th} *layer* of $\mathbf{M}_{\text{sol}}$ stores the solution, $\mathbf{M}(t_n) = \mathbf{M}_n$, at time t_n)

4.4 One-Step Propagation for Matrix-Valued ODEs

The documentation for one-step propagation in the *ODE Solver Toolbox* is written specifically for the case of vector-valued ODEs. However, the exact same functions can be used for matrix-valued ODEs, since the functions make no assumptions regarding the dimensions of the inputs.

PART II

Fixed-Step ODE Solvers

5

Explicit Runga-Kutta (Single-Step) Methods

5.1 General Form of the Explicit Runge-Kutta Method

In general, an s-stage **Runge-Kutta method** is defined as

$$\boxed{\mathbf{y}_{n+1} = \mathbf{y}_n + h\sum_{i=1}^{s} b_i \mathbf{k}_i} \tag{5.1}$$

For *explicit* Runge-Kutta methods,

$$\boxed{\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(t_n, \mathbf{y}_i) \\
\mathbf{k}_2 &= \mathbf{f}(t_n + c_2 h, \mathbf{y}_i + h(a_{21}\mathbf{k}_1)) \\
\mathbf{k}_3 &= \mathbf{f}(t_n + c_3 h, \mathbf{y}_i + h(a_{31}\mathbf{k}_1 + a_{32}\mathbf{k}_2)) \\
&\vdots \\
\mathbf{k}_i &= \mathbf{f}\left(t_n + c_i h, \mathbf{y}_n + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j\right) \\
&\vdots \\
\mathbf{k}_s &= \mathbf{f}\left(t_n + c_s h, \mathbf{y}_n + h\sum_{j=1}^{s-1} a_{ij}\mathbf{k}_j\right)
\end{aligned}} \tag{5.2}$$

The order, m, of a Runge-Kutta method is not necessarily equal to the number of stages.

$$p \neq s \qquad \text{(in general)}$$

However, for the explicit Runge-Kutta methods summarized in this document, we will in fact have that

$$p = s \qquad \text{(for the RK methods summarized in this document)}$$

Note that

$$\mathbf{y}_n, \mathbf{y}_{n+1}, \mathbf{k}_i \in \mathbb{R}^p$$
$$h, a_{ij}, b_i, c_i \in \mathbb{R}$$

For a given method, the coefficients a_{ij}, b_i, and c_i are often defined using a **Butcher tableau**:

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array}$$

We refer to Eq. (5.1) as the **propagation equation**, since it propagates the state vector $\mathbf{y}$ from one sample time to the next (i.e. from time t_n to time t_{n+1}). The `tableau2eqns`[1] function in the *ODE Solver Toolbox* can be used to generate the propagation equation, as well as all the $\mathbf{k}$ vector coefficients. The input to this function is a matrix $\mathbf{T} \in \mathbb{R}^{(s+1)\times(s+1)}$ storing the Butcher tableau.

$$\mathbf{T} = \left[\begin{array}{c|cccc} 0 & 0 & 0 & \cdots & 0 \\ c_2 & a_{21} & 0 & \cdots & 0 \\ c_3 & a_{31} & a_{32} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & 0 \\ \hline 0 & b_1 & b_2 & \cdots & b_s \end{array}\right]$$

`tableau2eqns` first extracts the **Runge-Kutta matrix** $\mathbf{A} \in \mathbb{R}^{s\times s}$, the **weight vector** $\mathbf{b} \in \mathbb{R}^{1\times s}$, and the **node vector** $\mathbf{c} \in \mathbb{R}^s$ [5, 8].

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1s} \\ \vdots & \ddots & \vdots \\ a_{s1} & \cdots & a_{ss} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ a_{21} & 0 & 0 & \cdots & 0 \\ a_{31} & a_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ a_{s1} & a_{s2} & \cdots & a_{s,s-1} & 0 \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} b_1 & \cdots & b_s \end{bmatrix}$$

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_s \end{bmatrix} = \begin{bmatrix} 0 \\ c_2 \\ \vdots \\ c_s \end{bmatrix}$$

> A list of the Butcher tableaus used to generate the equations in Sections 5.2-5.5 can be found in Section 5.6.

5.2 Euler Method

Consider the initial value problem

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

[1] https://tamaskis.github.io/ODE_Solver_Toolbox-MATLAB/tableau2eqns_doc.html

Recall from Eq. (3.1) that the solution of this IVP is

$$\mathbf{y}(t) = \mathbf{y}_0 + \int_{t_0}^{t} \mathbf{f}(t, \mathbf{y})\, dt$$

As discussed in Section 3.2, in general, it is either very difficult or mathematical impossible to evaluate the integral above in closed form. However, note that we can pick any point in time to be point t_0. Therefore, given $\mathbf{y}_n = \mathbf{y}(t_n)$ at sample time t_n, we can find $\mathbf{y}_{n+1} = \mathbf{y}(t_{n+1})$ at the next sample time as

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n_1}} \mathbf{f}(t_n, \mathbf{y})\, dt$$

The function $\mathbf{f}$ has some value $\mathbf{f}(t_n, \mathbf{y}_n)$ at time t_n. To define the Euler method (`RK1_euler`), let's assume that $\mathbf{f}$ maintains this value until the next sample time, t_{n+1} [2].

$$\mathbf{f}(t, \mathbf{y}) \approx \mathbf{f}(t_n, \mathbf{y}_n)\ \forall\ t \in [t_n, t_{n+1})$$

Since $\mathbf{f}(t_n, \mathbf{y}_n)$ is now treated as a constant over the time interval $[t_n, t_{n+1})$, we can pull it out from under the integrand.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \mathbf{f}(t, \mathbf{y}_n) \int_{t_n}^{t_{n+1}} dt = \mathbf{y}_n + \mathbf{f}(t_n, \mathbf{y}_n)(\underbrace{t_{n+1} - t_n}_{h})$$

$$\boxed{\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t_n, y_n)} \tag{5.3}$$

5.3 Runge-Kutta Second-Order Methods

Method	Abbreviation	Equations	Reference(s)
Midpoint Method	`RK2`	$\mathbf{k}_1 = \mathbf{f}\left(t_n, \mathbf{y}_n\right)$ $\mathbf{k}_2 = \mathbf{f}\left(t_n + \frac{h}{2}, y_n + \frac{h\mathbf{k}_1}{2}\right)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{k}_2$	[5, 8]
Heun's Second-Order Method	`RK2_heun`	$\mathbf{k}_1 = \mathbf{f}\left(t_n, \mathbf{y}_n\right)$ $\mathbf{k}_2 = \mathbf{f}\left(t_n + h, y_n + h\mathbf{k}_1\right)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}\left(\mathbf{k}_1 + \mathbf{k}_2\right)$	[5, 8]
Ralston's Second-Order Method	`RK2_ralston`	$\mathbf{k}_1 = \mathbf{f}\left(t_n, \mathbf{y}_n\right)$ $\mathbf{k}_2 = \mathbf{f}\left(t_n + \frac{2h}{3}, y_n + \frac{2h\mathbf{k}_1}{3}\right)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{4}\left(\mathbf{k}_1 + 3\mathbf{k}_2\right)$	[5, 8]

5.4 Runge-Kutta Third-Order Methods

Method	Abbreviation	Equations	Reference(s)
Classic (Kutta's) Third-Order Method	`RK3`	$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n)$ $\mathbf{k}_2 = \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_1}{2}\right)$ $\mathbf{k}_3 = \mathbf{f}(t_n + h, \mathbf{y}_n - h\mathbf{k}_1 + 2h\mathbf{k}_2)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 4\mathbf{k}_2 + \mathbf{k}_3)$	[3, pp. 129–131][5]
Heun's Third-Order Method	`RK3_heun`	$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n)$ $\mathbf{k}_2 = \mathbf{f}\left(t_n + \frac{h}{3}, \mathbf{y}_n + \frac{h\mathbf{k}_1}{3}\right)$ $\mathbf{k}_3 = \mathbf{f}\left(t_n + \frac{2h}{3}, \mathbf{y}_n + \frac{2h\mathbf{k}_2}{3}\right)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{4}(\mathbf{k}_1 + 3\mathbf{k}_3)$	[1, p. 287][3, pp. 129–131][5]
Ralston's Third-Order Method	`RK3_ralston`	$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n)$ $\mathbf{k}_2 = \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_1}{2}\right)$ $\mathbf{k}_3 = \mathbf{f}\left(t_n + \frac{3h}{4}, \mathbf{y}_n + \frac{3h\mathbf{k}_2}{4}\right)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{9}(2\mathbf{k}_1 + 3\mathbf{k}_2 + 4\mathbf{k}_3)$	[5]
Strong Stability Preserving Runge-Kutta Third-Order Method	`SSPRK3`	$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n)$ $\mathbf{k}_2 = \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{k}_1)$ $\mathbf{k}_3 = \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_1}{4} + \frac{h\mathbf{k}_2}{4}\right)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + \mathbf{k}_2 + 4\mathbf{k}_3)$	[5]

5.5 Runge-Kutta Fourth-Order Methods

Method	Abbreviation	Equations	Reference(s)
Classic Fourth-Order Method	`RK4`	$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n)$ $\mathbf{k}_2 = \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_1}{2}\right)$ $\mathbf{k}_3 = \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h\mathbf{k}_2}{2}\right)$ $\mathbf{k}_4 = \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{k}_3)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$	[1, p. 288][3, p. 131][5]

Ralston's Fourth-Order Method	`RK4_ralston`	$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n)$ $\mathbf{k}_2 = \mathbf{f}(t_n + 0.4h, \mathbf{y}_n + 0.4h\mathbf{k}_1)$ $\mathbf{k}_3 = \mathbf{f}(t_n + 0.45573725h, \mathbf{y}_n + 0.29697761h\mathbf{k}_1 + 0.15875964h\mathbf{k}_2)$ $\mathbf{k}_4 = \mathbf{f}(t_n + h, \mathbf{y}_n + 0.21810040h\mathbf{k}_1 - 3.05096516h\mathbf{k}_2 + 3.83286476h\mathbf{k}_3)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + h(0.17476028\mathbf{k}_1 - 0.55148066\mathbf{k}_2 + 1.20553560\mathbf{k}_3 + 0.17118478\mathbf{k}_4)$	[5]
3/8-Rule Fourth-Order Method	`RK4_38`	$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n)$ $\mathbf{k}_2 = \mathbf{f}\left(t_n + \frac{h}{3}, \mathbf{y}_n + \frac{h\mathbf{k}_1}{3}\right)$ $\mathbf{k}_3 = \mathbf{f}\left(t_n + \frac{2h}{3}, \mathbf{y}_n - \frac{h\mathbf{k}_1}{3} + h\mathbf{k}_2\right)$ $\mathbf{k}_4 = \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{k}_1 - h\mathbf{k}_2 + h\mathbf{k}_3)$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{8}(\mathbf{k}_1 + 3\mathbf{k}_2 + 3\mathbf{k}_3 + \mathbf{k}_4)$	[5]

5.6 List of Butcher Tableaus

Euler Method [5] (`RK1_euler`)	$\begin{array}{c\|c} 0 & 0 \\ \hline & 1 \end{array}$
Midpoint Method [5] (`RK2`)	$\begin{array}{c\|cc} 0 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array}$
Heun's Second-Order Method [5] (`RK2_heun`)	$\begin{array}{c\|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array}$
Ralston's Second-Order Method [5] (`RK2_ralston`)	$\begin{array}{c\|cc} 0 & 0 & 0 \\ 2/3 & 2/3 & 0 \\ \hline & 1/4 & 3/4 \end{array}$
Classic (Kutta's) Third-Order Method [3, p. 131][5] (`RK3`)	$\begin{array}{c\|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & 1/6 & 2/3 & 1/6 \end{array}$

Heun's Third-Order Method [3, p. 131][5] (`RK3_heun`)

0	0	0	0
1/3	1/3	0	0
2/3	0	2/3	0
	1/4	0	3/4

Ralston's Third-Order Method [5] (`RK3_ralston`)

0	0	0	0
1/2	1/2	0	0
3/4	0	3/4	0
	2/9	1/3	4/9

Strong Stability Preserving Runge-Kutta Third-Order Method [5] (`SSPRK3`)

0	0	0	0
1	1	0	0
1/2	1/4	1/4	0
	1/6	1/6	2/3

Classic Fourth-Order Method [3, p. 131][5] (`RK4`)

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
	1/6	1/3	1/3	1/6

Ralston's Fourth-Order Method [5] (`RK4_ralston`)

0	0	0	0	0
0.4	0.4	0	0	0
0.45573725	0.29697761	0.15875964	0	0
1	0.21810040	−3.05096516	3.83286476	0
	0.17476028	−0.55148066	1.20553560	0.17118478

3/8-Rule Fourth-Order Method [5] (`RK4_38`)

0	0	0	0	0
1/3	1/3	0	0	0
2/3	−1/3	1	0	0
1	1	−1	1	0
	1/8	3/8	3/8	1/8

6

Adams-Bashforth (Multistep Predictor) Methods

6.1 Adams-Bashforth Predictor

The general Adams-Bashforth predictor is defined as

$$\boxed{\mathbf{y}_{n+1} = \mathbf{y}_n + h\sum_{i=1}^{m} b_i \mathbf{f}(t_{n-i+1}, \mathbf{y}_{n-i+1})} \tag{6.1}$$

where m is the order of the method.

To find the vector $\mathbf{b} = (b_1, ..., b_m)^T$ storing the vectors for this predictor, we first define the matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and the vector $\mathbf{c} \in \mathbb{R}^m$ as

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 0 & 1 & 2 & \dots & m & (m-1) \\ 0 & 1 & 2^2 & \dots & m^2 & (m-1)^2 \\ 0 & 1 & 2^3 & \dots & m^3 & (m-1)^3 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 1 & 2^{m-1} & \dots & m^{m-1} & (m-1)^{m-1} \end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix} 1 \\ -\frac{1}{2} \\ \frac{1}{3} \\ -\frac{1}{4} \\ \vdots \\ \frac{(-1)^{m-1}}{m} \end{bmatrix}$$

This can be more compactly expressed as

$$\boxed{\mathbf{A} = [a_{ij}] \quad \text{where} \quad a_{ij} = (j-1)^{i-1} \quad (\text{for } i, j = 1, 2, ..., n)} \tag{6.2}$$

$$\boxed{\mathbf{c} = [c_i] \quad \text{where} \quad c_i = \frac{(-1)^{i-1}}{i} \quad (\text{for } i = 1, 2, ..., n)} \tag{6.3}$$

Next, we just need to solve the linear system $\mathbf{Ab} = \mathbf{c}$ for $\mathbf{b}$ [9].

$$\boxed{\mathbf{b} = \mathbf{A}^{-1}\mathbf{c}} \tag{6.4}$$

6.2 Adams-Bashforth Methods

Order	Abbrev.	Equations	Reference(s)
2	`AB2`	$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}(3\mathbf{f}_n - \mathbf{f}_{n-1})$	[4][6, pp. 135–136]
3	`AB3`	$\mathbf{y}_{n+1} = \mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{12}(23\mathbf{f}_n - 16\mathbf{f}_{n-1} + 5\mathbf{f}_{n-2})$	[4][6, pp. 135–136]
4	`AB4`	$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{24}(55\mathbf{f}_n - 59\mathbf{f}_{n-1} + 37\mathbf{f}_{n-2} - 9\mathbf{f}_{n-3})$	[4][6, pp. 135–136]
5	`AB5`	$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{720}(1901\mathbf{f}_n - 2774\mathbf{f}_{n-1} + 2616\mathbf{f}_{n-2} - 1274\mathbf{f}_{n-3} + 251\mathbf{f}_{n-4})$	[4][6, pp. 135–136]
6	`AB6`	$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{1440}(4277\mathbf{f}_n - 7923\mathbf{f}_{n-1} + 9982\mathbf{f}_{n-2} - 7298\mathbf{f}_{n-3} + 2877\mathbf{f}_{n-4} - 475\mathbf{f}_{n-5})$	[6, pp. 135–136]
7	`AB7`	$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{60480}(198721\mathbf{f}_n - 447288\mathbf{f}_{n-1} + 705549\mathbf{f}_{n-2} - 688256\mathbf{f}_{n-3} + 407139\mathbf{f}_{n-4} - 134472\mathbf{f}_{n-5} + 19087\mathbf{f}_{n-6})$	[6, pp. 135–136]
8	`AB8`	$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{120960}(434241\mathbf{f}_n - 1152169\mathbf{f}_{n-1} + 2183877\mathbf{f}_{n-2} - 2664477\mathbf{f}_{n-3} + 2102243\mathbf{f}_{n-4} - 1041723\mathbf{f}_{n-5} + 295767\mathbf{f}_{n-6} - 36799\mathbf{f}_{n-7})$	[6, pp. 135–136]

7

Adams-Bashforth-Moulton (Multistep Predictor-Corrector) Methods

7.1 Adams-Moulton Corrector

The general Adams-Moulton corrector is defined as

$$\boxed{\mathbf{y}_{n+1} = \mathbf{y}_n + h\sum_{i=1}^{m} b_i \mathbf{f}(t_{n-i+2}, \mathbf{y}_{n-i+2})} \tag{7.1}$$

where m is the order of the method.

To find the vector $\mathbf{b} = (b_1, ..., b_m)^T$ storing the vectors for this corrector, we first define the matrix $\mathbf{A} \in \mathbb{R}^{m\times m}$ and the vector $\mathbf{c} \in \mathbb{R}^m$ as

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 & 1 \\ -1 & 0 & 1 & 2 & \dots & (m-3) & (m-2) \\ 1 & 0 & 1 & 2^2 & \dots & (m-3)^2 & (m-2)^2 \\ \vdots & & \vdots & \vdots & & \vdots & \vdots \\ (-1)^{m-2} & 0 & 1 & 2^{m-2} & \dots & (m-3)^{m-2} & (m-2)^{m-2} \\ (-1)^{m-1} & 0 & 1 & 2^{m-1} & \dots & (m-3)^{m-1} & (m-2)^{m-1} \end{bmatrix}, \qquad \mathbf{c} = \begin{bmatrix} 1 \\ -\frac{1}{2} \\ \frac{1}{3} \\ \vdots \\ \frac{(-1)^{m-2}}{m-1} \\ \frac{(-1)^{m-1}}{m} \end{bmatrix}$$

This can be more compactly expressed as

$$\boxed{\mathbf{A} = [a_{ij}] \quad \text{where} \quad a_{ij} = (j-2)^{i-1} \quad (\text{for } i, j = 1, 2, ..., m)} \tag{7.2}$$

$$\boxed{\mathbf{c} = [c_i] \quad \text{where} \quad c_i = \frac{(-1)^{i-1}}{i} \quad (\text{for } i = 1, 2, ..., m)} \tag{7.3}$$

Next, we just need to solve the linear system $\mathbf{Ab} = \mathbf{c}$ for $\mathbf{b}$ [9].

$$\boxed{\mathbf{b} = \mathbf{A}^{-1}\mathbf{c}} \tag{7.4}$$

7.2 Predictor-Corrector (PECE) Algorithms

Let $\mathbf{f}_k$ represent the evaluation of the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$ at time t_k and state $\mathbf{y}_k$.

$$\mathbf{f}_k = \mathbf{f}(t_k, \mathbf{y}_k)$$

The Adams-Bashforth methods discussed in Chapter 6 were *explicit* methods, since the state vector at the next sample time could be calculated *explicitly* from evaluating the ODE using the state vector at previous sample times. In general, this can be expressed as

$$\mathbf{y}_{k+1} = \mathbf{g}(\mathbf{f}_k, \mathbf{f}_{k-1}, ..., \mathbf{f}_0)$$

However, the Adams-Moulton corrector introduced in Section 7.1 defines *implicit* methods, since the state vector at the next sample time is expressed as a function of $\mathbf{f}_{k+1}$, where $\mathbf{f}_{k+1} = \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})$ (i.e. $\mathbf{y}_{k+1}$ is essentially a function of itself):

$$\mathbf{y}_{k+1} = \mathbf{g}(\mathbf{f}_{k+1}, \mathbf{f}_k, \mathbf{f}_{k-1}, ..., \mathbf{f}_0)$$

Predictor-corrector algorithms combine predictor and corrector methods and are defined using the following four steps:

1. **Predict** $\mathbf{y}_{k+1}$ at time t_{k+1} using an *explicit* method of order m. Since this is a *predicted* value, we denote it using a hat as $\hat{\mathbf{y}}_{k+1}$.

   $$\hat{\mathbf{y}}_{k+1} = \mathbf{g}(\mathbf{f}_k, \mathbf{f}_{k-1}, ..., \mathbf{f}_0)$$

2. **Evaluate** the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$ at t_{k+1} and $\hat{\mathbf{y}}_{k+1}$.

   $$\hat{\mathbf{f}}_{k+1} = \mathbf{f}(t_{k+1}, \hat{\mathbf{y}}_{k+1})$$

3. **Correct** the prediction for $\mathbf{y}_{k+1}$ using an *implicit* method of order m. Since this is the corrected value (that we are assuming to be the true value, i.e. the solution), we do *not* denote it using a hat.

   $$\mathbf{y}_{k+1} = \mathbf{g}(\hat{\mathbf{f}}_{k+1}, \mathbf{f}_k, \mathbf{f}_{k-1}, ..., \mathbf{f}_0)$$

4. **Evaluate** the ODE at the corrected value for the state vector at the $(k+1)^{\text{th}}$ sample time.

   $$\mathbf{f}_{k+1} = \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})$$

Due to the four steps above (predict, evaluate, correct, evaluate), predictor-corrector algorithms are also known as **PECE** algorithms [3, pp. 112–113][6, p. 138]. In their implementation in the *ODE Solver Toolbox* (general implementation outlined in Section 3.7), the steps are slightly condensed/reordered as follows:

1. **Evaluate** the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$ at t_k and $\mathbf{y}_k$.

   $$\mathbf{f}_k = \mathbf{f}(t_k, \mathbf{y}_k)$$

2. **Predict** $\mathbf{y}_{k+1}$ at time t_{k+1} using an *explicit* method of order m. Since this is a *predicted* value, we denote it using a hat as $\hat{\mathbf{y}}_{k+1}$.

   $$\hat{\mathbf{y}}_{k+1} = \mathbf{g}(\mathbf{f}_k, \mathbf{f}_{k-1}, ..., \mathbf{f}_0)$$

3. **Correct** the prediction for $\mathbf{y}_{k+1}$ using an *implicit* method of order m. Since this is the corrected value (that we are assuming to be the true value, i.e. the solution), we do *not* denote it using a hat.

   $$\mathbf{y}_{k+1} = \mathbf{g}(\mathbf{f}(t_{k+1}, \hat{\mathbf{y}}_{k+1}), \mathbf{f}_k, \mathbf{f}_{k-1}, ..., \mathbf{f}_0)$$

7.3 Adams-Bashforth-Moulton Methods

Order	Abbrev.	Equations	Reference(s)
2	ABM2	$\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \frac{h}{2}(3\mathbf{f}_n - \mathbf{f}_{n-1})$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}[\mathbf{f}(t_{n+1}, \hat{\mathbf{y}}_{n+1}) + \mathbf{f}_n]$	[4][6, pp. 135–138]
3	ABM3	$\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \frac{h}{12}(23\mathbf{f}_n - 16\mathbf{f}_{n-1} + 5\mathbf{f}_{n-2})$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{12}[5\mathbf{f}(t_{n+1}, \hat{\mathbf{y}}_{n+1}) + 8\mathbf{f}_n - \mathbf{f}_{n-1}]$	[4][6, pp. 135–138]
4	ABM4	$\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \frac{h}{24}(55\mathbf{f}_n - 59\mathbf{f}_{n-1} + 37\mathbf{f}_{n-2} - 9\mathbf{f}_{n-3})$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{24}[9\mathbf{f}(t_{n+1}, \hat{\mathbf{y}}_{n+1}) + 19\mathbf{f}_n - 5\mathbf{f}_{n-1} + \mathbf{f}_{n-2}]$	[4][6, pp. 135–138]
5	ABM5	$\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \frac{h}{720}(1901\mathbf{f}_n - 2774\mathbf{f}_{n-1} + 2616\mathbf{f}_{n-2} - 1274\mathbf{f}_{n-3} + 251\mathbf{f}_{n-4})$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{720}[251\mathbf{f}(t_{n+1}, \hat{\mathbf{y}}_{n+1}) + 646\mathbf{f}_n - 264\mathbf{f}_{n-1} + 106\mathbf{f}_{n-2} - 19\mathbf{f}_{n-3}]$	[4][6, pp. 135–138]
6	ABM6	$\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \frac{h}{1440}(4277\mathbf{f}_n - 7923\mathbf{f}_{n-1} + 9982\mathbf{f}_{n-2} - 7298\mathbf{f}_{n-3} + 2877\mathbf{f}_{n-4} - 475\mathbf{f}_{n-5})$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{1440}[475\mathbf{f}(t_{n+1}, \hat{\mathbf{y}}_{n+1}) + 1427\mathbf{f}_n - 798\mathbf{f}_{n-1} + 482\mathbf{f}_{n-2} - 173\mathbf{f}_{n-3} + 27\mathbf{f}_{n-4}]$	[6, pp. 135–138]
7	ABM7	$\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \frac{h}{60480}(198721\mathbf{f}_n - 447288\mathbf{f}_{n-1} + 705549\mathbf{f}_{n-2} - 688256\mathbf{f}_{n-3} + 407139\mathbf{f}_{n-4} - 134472\mathbf{f}_{n-5} + 19087\mathbf{f}_{n-6})$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{60480}[19087\mathbf{f}(t_{n+1}, \hat{\mathbf{y}}_{n+1}) + 65112\mathbf{f}_n - 46461\mathbf{f}_{n-1} + 37504\mathbf{f}_{n-2} - 20211\mathbf{f}_{n-3} + 6312\mathbf{f}_{n-4} - 863\mathbf{f}_{n-5}]$	[6, pp. 135–138]
8	ABM8	$\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \frac{h}{120960}(434241\mathbf{f}_n - 1152169\mathbf{f}_{n-1} + 2183877\mathbf{f}_{n-2} - 2664477\mathbf{f}_{n-3} + 2102243\mathbf{f}_{n-4} - 1041723\mathbf{f}_{n-5} + 295767\mathbf{f}_{n-6} - 36799\mathbf{f}_{n-7})$ $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{120960}[36799\mathbf{f}(t_{n+1}, \hat{\mathbf{y}}_{n+1}) + 139849\mathbf{f}_n - 121797\mathbf{f}_{n-1} + 123133\mathbf{f}_{n-2} - 88547\mathbf{f}_{n-3} + 41499\mathbf{f}_{n-4} - 11351\mathbf{f}_{n-5} + 1375\mathbf{f}_{n-6}]$	[6, pp. 135–138]

Bibliography

[1] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. 9th. Boston, MA: Brooks/Cole, Cengage Learning, 2011.

[2] *Euler method*. Wikipedia. Accessed: December 12, 2021. URL: https://en.wikipedia.org/wiki/Euler_method.

[3] David F. Griffiths and Desmond J. Higham. *Numerical Methods for Ordinary Differential Equations: Initial Value Problems*. London, UK: Springer-Verlag, 2010.

[4] *Linear multistep method*. Wikipedia. Accessed: November 1, 2021. URL: https://en.wikipedia.org/wiki/Linear_multistep_method.

[5] *List of Runge-Kutta methods*. Wikipedia. Accessed: November 1, 2021. URL: https://en.wikipedia.org/wiki/List_of_Runge-Kutta_methods.

[6] Oliver Montenbruck and Eberhard Gill. *Satellite Orbits – Models, Methods, Applications*. 4th. Berlin Heidelberg: Springer-Verlag, 2012.

[7] *Passing Extra Parameters*. MathWorks. Accessed: June 10, 2020. URL: https://www.mathworks.com/help/optim/ug/passing-extra-parameters.html.

[8] *Runge-Kutta methods*. Wikipedia. Accessed: November 1, 2021. URL: https://en.wikipedia.org/wiki/Runge-Kutta_methods.

[9] Baba Seidu. "A Matrix System for Computing the Coefficients of the Adams Bashforth-Moulton Predictor-Corrector formulae". In: *International Journal of Computational and Applied Mathematics* 6.3 (2011), pp. 215–220. URL: https://www.researchgate.net/publication/249655976_A_Matrix_System_for_Computing_the_Coefficients_of_the_Adams_Bashforth-Moulton_Predictor-Corrector_formulae.

[10] *State variable*. Wikipedia. Accessed: December 3, 2020. URL: https://en.wikipedia.org/wiki/State_variable.

[11] *State vector (navigation)*. Wikipedia. Accessed: December 3, 2020. URL: https://en.wikipedia.org/wiki/State_vector_(navigation).