A Crash Course in Numerical Methods for Initial Value Problems

How to make a computer do horrible math for you

Ben Goldman

May 2, 2023

Abstract

Ordinary differential equations are fundamental to nearly all applications of mathematics, especially physics, in which one must work backwards from a known function of derivatives of an unknown function. However, many ordinary differential equations that appear in applications lack a closed-form solution. Scientists employ numerical methods to approximately solve these problems, enabling the computational simulation of nearly any physical system. Here I present the most frequently-used numerical tools for ordinary differential equations and analyze their properties. This paper assumes a background in differential calculus, through Taylor's Theorem.

1 Motivation: why we want numerical methods for ODE's

1.1 Formulating the problem

Most problems in physics are presented in an ordinary differential equation (ODE), in which a function relating the derivatives of an unknown function is known. Formally stated, an ODE looks like this:

Definition 1.1. *Given a function $f : U \subset \mathbb{R}^{n+1} \to \mathbb{R}$, an **ordinary differential equation (ODE) of degree** n is an equation of the form*

$$0 = f\big(t, y, y', \cdots, y^{(n)}\big)$$

where y is some C^n function $y : \mathbb{R} \to \mathbb{R}$.

Example 1.1. *The equation $y'' + 2y - e^{\tan^{-1}(\sqrt{y'''})} = \sin(1/y + \cos(t)) - \ln\big(y^t\big)$ is a third-order ODE that corresponds to the equation*

$$f(t, y, y', y'', y''') = y'' + 2y - e^{tan^{-1}(\sqrt{(y''')})} - \sin\left(\frac{1}{y} + \cos(t)\right) + \ln\big(y^t\big) = 0$$

Given such a problem, we can define the solution as:

Definition 1.2. *A C^n function $u : (a, b) \to \mathbb{R}$ is a **solution** to the ODE $f(t, y', \cdots, y^{(n)}) = 0$ if:*

$$f\Big(t, u(t), u'(t), \cdots, u^{(n)}(t)\Big) = 0$$

Example 1.2. *The function $u(t) = e^{-ct}$ is a solution to the first-order ODE $y' + cy = 0$ because $u'(t) - u(t) = \frac{\mathrm{d}}{\mathrm{d}t}\big(e^{-ct}\big) - ce^{-ct} = -ce^{-ct} + ce^{-ct} = 0$.*

Such a problem can be generalized to vector valued functions $y : \mathbb{R} \to \mathbb{R}^m$, where

$$y(t) = \begin{pmatrix} y_1(t) \\ \vdots \\ y_m(t) \end{pmatrix}.$$

Definition 1.3. *Given a function $f : U \subset \mathbb{R} \times (\mathbb{R}^m)^n \to \mathbb{R}^m$ a **system of** m **differential equations of order** n is the equation*

$$f(t, y, y', \cdots, y^{(n)}) = \mathbf{0}$$

where $u : I \subset \mathbb{R} \to \mathbb{R}^m$ is a solution if

$$f\Big(t, u(t), u'(t), \cdots, u^{(n)}(t)\Big) = \mathbf{0}$$

Example 1.3. *The system*

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 10(y-x) \\ x(28-z)-y \\ xy-\frac{8}{3}z \end{pmatrix} \qquad \begin{pmatrix} x(0) \\ y(0) \\ z(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 28 \end{pmatrix}$$

is an initial value problem over a simple system of ordinary differential equations that has a fun and disturbing solution.

However, there is no guarantee that an ODE admits a unique solution, or a solution at all. However, in well-behaved cases, by specifying the value of the solution at a single point, we can guarantee a unique solution, (see Theorem 3.1).

Definition 1.4. *An* ***Initial Value Problem (IVP)*** *is a first order ordinary differential equation with* $U = [a, b] \times \mathbb{R}^n$ *and* $f : U \to \mathbb{R}^n$

$$y'(t) = f(t, y(t))$$

when for some $t_0 \in [a, b]$ *we have*

$$y(t_0) = y_0$$

Now that we know what an ODE looks like, we'll want to look for ways of solving them. First, I'll present a few examples of problems in which ODE's arise to motivate the use of numerical methods in their solution.

1.2 Motivating example: pendulums

Say we have an (ideal) pendulum: a mass suspended by a massless rigid bar subject to a gravitational force. We want to find the evolution of the angle of the mass given an initial angle and velocity. For simplicity's sake, assume the mass to be 1 kg, the string's length to be 1 m and the strength of gravity to be $1\,\mathrm{m/s^2}$. The mass has an angle from the vertical $\theta(t)$, and begins with an initial angle $\theta_0 \in [-\pi, \pi]$. Then we release the mass and it swings back and forth. It can be shown that the mass obeys the following equation of motion:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\sin\theta \qquad \theta(0) = \theta_0, \theta'(0) = 0$$

We can phrase this as the 2^{nd} order ODE

$$\theta'' + \sin\theta = 0$$

which can be reduced to the 1^{st} order system

$$\begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix} = \begin{pmatrix} y(t) \\ -\sin x(t) \end{pmatrix}$$

Due to the nonlinear behavior of the $\sin\theta$ term, this equation is incredibly difficult to solve analytically. For situations in which $\theta_0 \approx 0$ we can use the first order taylor expansion $\sin\theta \approx \theta$ to produce an approximate solution to this equation:

$$\theta(t) \approx \theta_0 \cos(t)$$

ODE's where analytical solutions are unwieldy, if not imposible, dominate science. In these cases, the best way to determine the behavior of the solution is through numerical calculations. We trade not knowing the exact symbolic form of the solution for at least knowing its value over a range of points. Therefore, the ability to produce numerical results for the solution of an initial value problem has immense utility.

2 How computers do math

Obviously you don't want to do all those computations by hand—so we turn to computers. Unfortunately, as we will see, computers are always a little bit wrong. Therefore, a good computer program should be *convergent*: we should be able to keep the error under control. While computers are fast at math, solving an ODE requires a lot of calculations. Therefore, we also want our algorithms to run *quickly.* Finally, computers have limited processing and storage capacity. Therefore, a good algorithm runs *efficiently.* Let's address these requirements.

2.1 Floating-point numbers and round-off error

Computers represent all numbers with limited precision. The dominant method for encoding the reals is through floating-point representation. To a high level of abstraction, computers represent numbers the same way as humans do when using scientific notation. They write a number as a number times a base raised to an exponent. However, like humans using paper, these numbers can only fit in a finite set of storage. Therefore, we define the set of numbers representable on a computer as the following:

Definition 2.1. *Given a computer with p-bit precision, the set of* ***floating-point numbers*** *is the set of numbers precicely representable on such a computer, given by the set*

$$F_p = \left\{ M \times 2^E | M \in \mathbb{Z}, E \in \mathbb{Z}, {}^{|M|<2^p}_{|E|<2^p} \right\}$$

(This is not precicely how numerical representation is handled on circut boards, but it will work fine for our purposes.)

For any finite p, there will exist many reals with no floating-point representation. The floating-points are a subset of the rationals, so irrational numbers have no exact representation. There are plenty of rationals with no decimal representation either: e.g. $\frac{1}{3} \notin F_p$ for any p.

So when a computer with a precision p encounters a real number not in the set F_p, it chooses the nearest floating-point and changes the real number to that floating-point, producing a round-off error. Similarly, certain numbers that are too large or too small are not exactly representable: for a given p, the numbers $\pm(2^p + a) \times 2^E$ and $M \times 2^{\pm(p+a)}$ have no floating-point representation for any integer $a > 0$. Therefore, we will define the round-off error for a particular number as

Definition 2.2. *The* ***round-off error*** *for a number a on a p-bit machine is given by*

$$\mathrm{Error_p}(a) = \mathrm{glb}\{|a - n| \mid n \in F_p\}$$

Also, notice that while computers cannot truly store the value of an irrational number, there still exist plenty of ways of handling them. For example, we could devise a perfectly finite algorithm for calculating the value of e to an arbitrary level of precision based on the Taylor series for e^x:

```
function e_n:
  let e_i = 0
  while i < n:
    set e = e + 1/(i!)
  output e_i
```

The Taylor Series definition of e guarantees that we can get an arbitrarily precice answer for the value of e by choosing a sufficiently large n. However, notice that higher values of n will require more time to computer e_n. Therefore, we care about how accurate a particular algorithm is, given a finite n.

2.2 Error, convergence and stability

Most of the algorithms we will soon explore take in a parameter h, where as $h \to 0$ the value of the algorithm's output $A(h)$ approaches the true solution A. We can use the concept of $\mathcal{O}(F(h))$ notation to understand the rate of this convergence.

Definition 2.3. *The* ***error*** *of a function $A(h)$ with $\lim_{h\to 0} A(h) = A$ is given by $E(h) = |A - A(h)|$. We say $E(h) = \mathcal{O}(F(h))$ where $F(h)$ has $\lim_{h\to 0} F(h) = 0$ there exists an $H > 0$ and $K > 0$ such that for all $0 < h < H$ we have $E(h) \leq K|F(h)|$.*

Convention is to choose some $F(h)$ taking the form $F(h) = h^k$. A few examples will make this notion easier to work with. However, we've already discussed a general class of examples in class: the k^{th} derivative approximation to a function has error $\mathcal{O}\left(\|h\|^k\right)$.

Example 2.1. *Given a twice-differentiable function $f : \mathbb{R} \to \mathbb{R}$, we have*

$$\left| f(x+h) - f(x) - hf'(x) - \frac{h^2}{2} f''(x) \right| < \varepsilon |h|^2.$$

Take $A(h) = f(x+h) - f(x) - hf'(x) - h^2/2f''(x)$. For all h, we have $a|A(h) - 0|\varepsilon h^2$, so $A(h)$ is of order $\mathcal{O}(h^2)$.

The smaller $F(h)$ is for small h, the better. Therefore, when we decrease the parameter h in the input to an algorithm, the output should approach the "actual result" proportionally to how $F(h)$ approaches 0. However, having an algorithm that narrows down to a solution quickly usually isn't good enough to call a numerical result a good solution. What we need is the notion of stability.

Given some numerical simulation of a system, we should hope that the model be stable: that when perturbed, the error of any iterated computation should remain under control. Given a numerical algorithm that has an initial error E_0 after some number of iterations, and an error E_n, n iterations later, we say the error grows linearly if there exists some C such that for all n $E_n \leq |nCE_0|$, and exponential if there exists a C such that $E_n \leq |C^n E_0|$. We call algorithms with linear error growth stable, and ones with exponential error growth unstable.

Stability is a critical requirement for a numerical method because we want the ability to restrict the error bound after some number of iterations E_n to some upper limit. For linearly growing error, this means limiting the initial error E_0 to some quantity to the desired ratio E_n/Cn (through more accurate measurements, higher computational precision, etc). In contrast, for an unstable algorithm, this demands restricting the initial error to E_n/C^n. For any usefully large value of n, this is infeasable.

3 Existence and uniqueness of solutions to an IVP

You hand me an ordinary differential equation of order 1, ie. a function $f : U \subset \mathbb{R}^2 \to \mathbb{R}^n$ such that for some function $y : \mathbb{R} \to \mathbb{R}^n$, we have $f(t, y(t)) = y'(t)$. This becomes an initial value problem if you also hand me some value of $y(t_0) = y_0$. The Picard-Lindelhöf theorem tells us when there exists a unique solution u to this problem. First, a definition:

Definition 3.1. *A function $f : U \subset \mathbb{R}^n \to \mathbb{R}^m$ is Lipschitz continuous on U if there exists a $K \geq 0$ such that for all $\mathbf{x}_1, \mathbf{x}_2 \in U$, we have*

$$\|f(\mathbf{x}_1) - f(\mathbf{x}_2)\| \leq K\|\mathbf{x}_1 - \mathbf{x}_2\|$$

Theorem 3.1 (Picard-Lindelhöf Theorem)**.** *Let $U \subset \mathbb{R} \times \mathbb{R}^m$ be a closed box with $(t_0, y_0) \in U$, and $f : U \to \mathbb{R}^m$ is a Lipschitz continuous function. Then, there exists a unique solution to the initial value problem*

$$f(t, y(t)) = y'(t) \qquad y(t_0) = y_0$$

Proof. Omitted. See Coddington (1955). □

However, just because an initial value problem permits a unique solution, doesn't mean that it is appropriate to apply a numerical algorithm to find such a solution. There exist many initial value problems for which all numerical solutions are unstable. By our definition of stability, such a solution quickly becomes useless. Therefore, we must place a second condition on the application of numerical methods to an initial value problem: that perturbing a problem does not change its solution by too much.

Definition 3.2. *Given an initial value problem for some function $f : [t_0, t_f] \times \mathbb{R}^n \to \mathbb{R}^n$*

$$y'(t) = f(t, y(t)) \qquad y(t_0) = \mathbf{y}_0$$

*a **perturbation** of this problem is the initial value problem*

$$y'(t) = f(t, y(t)) \qquad y(t_0) = \mathbf{y}_0 + \delta_0,$$

where δ_0 is some constant.

Given that all numerical algorithms will produce round-off error, and in unstable algorithms, such errors grow uncontrollably, we should require that solutions to perturbations of some initial value problem always remain constrained. If this is true, then an initial value problem will yield stable solutions. We will call the combination of this condition with Lipschitz continuity "well-posedness".

Definition 3.3. *Given a function $f : [t_0, t_f] \times \mathbb{R}^n \to \mathbb{R}^n$, the initial value problem*

$$y' = f(t, y(t)) \qquad y(t_0) = \mathbf{y}_0$$

is well-posed if

- *it has a unique has a unique solution,*

- *there exists a bound b and a constant k where for all ε with $0 < \varepsilon < b$, then for all continuous functions $\delta : [t_0, t_f] \to \mathbb{R}^n$ and for all $t \in [t_0, t_f]$, whenever $\|\delta(t)\| < \varepsilon$ and $|\delta_0| < \varepsilon$, the solution to the perturbation*
$$y' = f(t, y_\delta(t)) + \delta(t) \qquad y_\delta(t_0) = \mathbf{y}_0 + \delta_0$$
has
$$\|y(t) - y_\delta(t)\| < k\varepsilon.$$

It can be shown that whenever a function $f : [t_0, t_f] \times \mathbb{R}^n \to \mathbb{R}^n$ is Lipschitz continuous, then it defines a well-posed IVP. Unfortunately, we won't be proving that here either.

4 One-step methods

The idea of a numerical solution to an initial value problem is to produce a discretized approximation, i.e. given a finite list $T \in \mathbb{R}$ (keeping in mind that realistically each element of $\mathbb{R}$ will get downgraded to a floating-point rational), the solution will be another finite set $\{y(t) \mid t \in T\}$. To generate this set, specify a resolution, n and a domain $[a, b]$. Now, define the spacing constant $h := (a - b)/n$ and set $t_i = a + nh$.

Definition 4.1. *Given a well-posed initial value problem*
$$y'(t) = f(t, y(t)) \qquad y(a) = y_0$$
*with a solution $u : [a, b] \to \mathbb{R}^n$, then w_i is a **numerical solution** if $w_0 = y_0$ and $w_i \approx u(t_i)$ for all $t_i \in [a, b]$. Of course, the definition of approximately equal to will vary from problem to problem.*

A simple class of numerical methods for IVP's is the set of explicit one-step methods. For this class, the value of the approximation at each point is determined entirely by that at the previous point. This is done by taking a step of size h in some direction determined by a function ϕ. If $y' = f(t, y)$ is an ODE, and w_i is a sequence, then each w_i is given by
$$w_{i+1} = w_i + h\phi(h, t_i, w_i)$$
with $w_0 = y_0 = y(a)$. We call such a rule for approximating the solution to an ODE a one-step method.

A basic first try to generate a sequence w_i is to step in the direction of y', i.e. use a linear approximation at each time t_i. Notice how this follows from Taylor's theorem, where, if y is C^2, we have
$$y(t + h) = y(t) + hy'(t) + \mathcal{O}(h^2)$$
Dropping the $\mathcal{O}(h^2)$ terms produces Euler's Method:

Algorithm 4.1 (Euler's Method)**.** *Given $U = [a, b] \times \mathbb{R}^n$ with $f : U \to \mathbb{R}^n$ and the well-posed initial value problem*
$$y'(t) = f(t, y(t)) \qquad y(a) = y_0$$
An Euler's method approximation $w_i \approx y(t_i)$ is given by the process
$$w_i = \begin{cases} y_0 & i = 0 \\ w_{i-1} + hf(t_i, w_{i-1}) & i > 0 \end{cases}$$

How good is Euler's method? More specifically, if the function $y : [a, b] \to \mathbb{R}^n$ satisfies the differential equation $y' = f(t, y)$, while the sequence w_i is the Euler's method approximation with $w_i \approx y(t_i)$ for all $t_i \in [a, b]$, then by how much does $y(t_{i+1})$ differ from w_{i+1}? Most sources phrase this as, "By how much does the solution y_{i+1} fail to satisfy the Euler's method formula?". We call this quantity *truncation error*. For simplicity of notation, for each i, set the sequence $y_i = y(t_i)$

Definition 4.2. *Given a one-step difference method $w_i = h\phi(t_{i-1}, w_{i-1})$ for some well-posed initial value problem, define the **local truncation error** as*
$$\tau_{i+1}(h) = \frac{y(t_{i+1}) - y(t_i)}{h} - \phi(t_i, y(t_i))$$
where $y(t)$ is an exact solution to the initial value problem. (Conventional notation tends to assume τ is a function of h so one may simply write τ_i.)

Proposition 4.1. *Euler's method has local truncation error $\mathcal{O}(h)$.*

Proof. For Euler's method, we have $\phi = f$. Therefore, the local truncation error is given by

$$\tau_{i+1} = \frac{y_{i+1} - y_i}{h} - f(t_i, y_i).$$

Assuming y is C^2, Taylor's theorem gives us $y_{i+1} = y_i + hy_i' + \mathcal{O}\big(h^2\big)$. Substituting, we now have

$$\tau_{i+1} = \frac{y_i + hy_i' + \mathcal{O}\big(h^2\big) - y_i}{h} - f(t_i - y_i)$$

Since $y'(t) = f(t, y)$ by definition, this means

$$\tau_{i+1} = \frac{y_i + hf(t_i, y_i) + \mathcal{O}\big(h^2\big) - y_i}{h} - f(t_i - y_i)$$

Which simplifies to

$$\tau_{i+1} = \mathcal{O}(h).$$

□

A reasonable expectation for good numerical methods is that one can make the truncation error arbitrarily small by decreasing the stepsize (assuming sufficient computational precision). We call this consistency.

Definition 4.3. *A numerical method with local truncation error $\tau_i(h)$ is* ***consistent*** *if for all i, we have*

$$\lim_{h \to 0} \tau_i(h) = 0.$$

Proposition 4.2. *Euler's method is consistent.*

Proof. The $h \to 0$ limit of the local truncation error for Euler's method is

$$\lim_{h \to 0} \tau_{i+1} = \lim_{h \to 0} \frac{y_{i+1} - y_i}{h} - f(t_i, y_i).$$

Since $f(t, y) = y'$ is the derivative of y, this simplifies to

$$\lim_{h \to 0} \tau_{i+1} = f(t_i, y_i) - f(t_i, y_i) = 0.$$

□

A stronger condition is also desirable: we want to be able to bring the error after an arbitrary time $t_i \in [a, b]$ arbitrarily small by choosing a sufficiently small stepsize. We call the total error propagated after some time the **global error**, equal to $|y_i - w_i|$ where y satisfies a given well-posed initial value problem and w_i is the numerical solution. We call this condition convergence. A numerical solution is convergent when for all $t_i \in [a, b]$, as $h \to 0$, we have $|y_i - w_i| \to 0$. Rather than prove that Euler's method is convergent, let's do one better, and find when all explicit one-step methods are consistent and convergent:

Definition 4.4. *A one-step method bearing the solution w_i is* ***convergent*** *if for any well-posed initial value problem $y'(t) = f(t, y(t))$ with $y(t_0) = y_0$, we have*

$$\lim_{\substack{w_0 \to y_0 \\ h \to 0}} |w_i - y_i| = 0.$$

Proposition 4.3. *An explicit one-step method given by $w_{i+1} = w_i + h\phi(h, t_i, w_i)$ where $y' = f(t, y)$ is consistent whenever $\lim_{h \to 0} \phi(h, t, y) = f(t, y)$.*

Proof. Assume that for the above explicit one-step method, we have

$$\lim_{h \to 0} \phi(h, t, y) = f(t, y)$$

Given this rule, the truncation error would be

$$\tau_{i+1} = \frac{y_{i+1} - y_i}{h} - \phi(h, t_i, y_i)$$

who's small-h limit is equal to

$$\lim_{h \to 0} \tau_{i+1} = \lim_{h \to 0} \left[\frac{y_{i+1} - y_i}{h} - \phi(h, t_i, y_i)\right].$$

Since $\lim_{h\to 0} \frac{y_{i+1}-y_i}{h}$ is equal to $y'(t_i)$, which equals $f(t, y_i)$, we have

$$\lim_{h\to 0} \tau_{i+1} = f(t_i, y_i) - \lim_{h\to 0} \phi(h, t_i, y_i),$$

which by assumption, is equivalent to

$$\lim_{h\to 0} \tau_{i+1} = f(t_i, y_i) - f(t_i, y_i) = 0. \qquad \square$$

We must also place one extra constraint on numerical solutions, stability. This definition will follow clearly from the one provided in the previous chapter:

Definition 4.5. *A well-posed initial value problem is* ***stable*** *if for any perturbation of this problem $\tilde{y}' = f(t, y)$ with $\tilde{y}(a) = \tilde{y}_0 = y_0 + \delta_0$, there exists an $h > 0$ and a constant K such that the solutions produced from the original and perturbed IVP's, w_i and $\tilde{w}_i$ have*

$$|w_i - \tilde{w}_i| \leq K|y_0 - \tilde{y}_0|$$

for all $t_i \in [a, b]$.

It can be shown that all consistent one-step methods for which $\phi(t, h, y)$ is Lipschitz continuous are stable.

Theorem 4.1. *All stable and consistent one-step methods have a global error bounded by*

$$\max_{t_i\in[a,b]} |w_i - y_i| \leq \tau \frac{e^{L(b-a)} - 1}{L}$$

wherein L is the Lipschitz constant for $\phi(t_i, h, w_i)$, and the truncation error is bounded by $|\tau_i| \leq \tau$.

Proof. Here, we will assume that the initial value is input exactly, i.e. $w_0 = y_0$. For a finite difference $|w_0 - y_0$, the analysis is similar. The numerical approximation using a one-step method is given by

$$w_{i+1} = w_i + h\phi(t_i, h, w_i)$$

while the real solution satisfies

$$y_{i+1} = y_i + h\phi(t_i, h, y_i) + h\tau_i$$

by the definition of local truncation error. If we define $d_i = y_i - w_i$, then we have

$$\begin{aligned} d_{i+1} &= y_{i-1} - w_{i-1} \\ &= y_i + h\phi(t_i, h, y_i) + h\tau_{i+1} - w_i - h\phi(t_i, h, w_i) \\ &= d_i + h[\phi(t_i, h, y_i) - \phi(t_i, h, w_i)] + h\tau_i. \end{aligned}$$

By the assumption that ϕ is Lipschitz with constant L and our bound on τ_i, we have

$$d_{i+1} \leq d_i + hL|d_i| + \tau$$

adding absolute values and applying the triangle inequality gets us

$$|d_{i+1}| \leq |d_i + hL|d_i| + \tau| \leq |d_i| + hL|d_i| + \tau$$

(remember that $h, L > 0$). This simplifies to

$$|d_{i+1}| \leq |d_i|(1 + hL) + \tau.$$

It can be shown by induction that given a recurrence relation $x_{i+1} \leq (a + 1)x_i + b$, we have

$$x_{i+1} \leq e^{ia} x_0 + b\frac{e^{ia} - 1}{a}.$$

Applying this fact to the bounds on d_{i+1}, we have

$$|d_{i+1}| \leq e^{(i+1)hL}|d_0| + \tau\frac{e^{(i+1)hL} - 1}{L}.$$

Since $|d_0| = |w_0 - y_0| = 0$, this is equivalent to

$$|d_{i+1}| = \tau\frac{e^{(i+1)hL} - 1}{L}.$$

Selecting the maximum d_{i+1} where $1 \leq i \leq n$ and $h(n + 1) = b - a$ with $t_i \in [a, b] = a + hi$, we have

$$\max_{t_i\in[a,b]} |d_i| = \tau\frac{e^{(b-a)L} - 1}{L}. \qquad \square$$

Corollary 4.1. *A one-step method $w_{i+1} = w_i + h\phi(t_i, h, w_i)$ that is stable, consistent, and has a bounded truncation error, is convergent.*

Proof. By theorem 4.1, since there exists a τ with $|\tau_i| \leq \tau$ and the one-step method is stable, we have

$$\max_{t_i \in [a,b]} |d_{i+1}| = \tau \frac{e^{(b-a)L} - 1}{L}$$

taking the limit of the global error as $h \to 0$, we see that

$$\lim_{h \to 0} \max_{t_i \in [a,b]} |d_{i+1}| = \lim_{h \to 0} \tau(h) \frac{e^{(b-a)L} - 1}{L}$$

(reintroducing the $\tau(h)$ notation to show this formula's dependence on h. This simplifies to

$$\lim_{h \to 0} \max_{t_i \in [a,b]} |d_{i+1}| = \frac{e^{(b-a)L} - 1}{L} \lim_{h \to 0} \tau = 0. \qquad \square$$

So Euler's method is pretty good: it's stable, consistent, and convergent. But it's only $\mathcal{O}(h)$. We can do better. For problems in which y'' is large, a linear approximation can be quite inaccurate. Therefore, it might be smart to use a higher-order Taylor approximation. However, this method is rarely used because evaluating the derivative of a function can be computationally expensive, making higher-order Taylor methods generally inefficient. A better plan would be to use extra evaluations of f to gain more information about the behavior of a solution at intermediate steps. One of the most popular classes of one-step numerical IVP solvers that uses this method is the Runge-Kutta algorithms. An R-step Runge-Kutta algorithm uses R evaluations of $f(t, y)$ at carefully selected points to gain a higher degree of accuracy, at the expense of increased computation time. Here is the general formula for an R-stage Runge-Kutta method:

Algorithm 4.2 (General R-Step Runge-Kutta Method)**.**

$$\begin{aligned}
w_{i+1} &= w_i + h\phi(t_i, w_i) \\
\phi(t_i, w_i) &= \sum_{r=1}^{R} c_r k_r \\
k_r &= f\left(t_i + h\sum_{s=1}^{r-1} a_{rs}, y + h\sum_{s=1}^{r-1} a_{rs} k_s\right) \qquad 2 \leq r \leq R \\
k_1 &= f(t_i, w_i)
\end{aligned}$$

where the values of c_r and a_{rs} can be chosen based on one's whims. Euler's method is a special case of 1^{st}-order Runge-Kutta methods. Runge-Kutta methods are based on a recursive process for estimating w_{i+1}. The algorithm measures $f(t_i, w_i)$, and then takes an Euler-style trial step along the resulting vector. It then samples f at this intermediate point, and then re-takes the step along the updated vector. After repeating R times, the algorithm takes a weighted average of all the generated values of f, and performs a final step h distance in that direction.

The most frequently used Runge-Kutta method is the "classical" fourth-order Runge-Kutta method. It looks like this:

Algorithm 4.3 (Fourth-order classical Runge-Kutta method)**.**

$$\begin{aligned}
k_1 &= f(t_i, y_i) \\
k_2 &= f(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1) \\
k_3 &= f(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1) \\
k_4 &= f(t_i + h, w_i + hk_3) \\
w_{i+1} &= y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}$$

This method has local truncation error $\mathcal{O}(h^4)$. Proving that would be arduous and uninformative, so you'll have to trust me. In general, Runge-Kutta methods of order $\mathcal{O}(h^p)$ use p evaluations of f for $p \leq 4$ and $p + 1$ for $p > 4$. The reason for this limitation is horrible casework involving covering all possible configurations of the coefficients c_r and a_{rs}. See Butcher (1964).

Example 4.1 (Apply the classical 4th-order Runge-Kutta method to the pendulum system, with an initial angle of $\theta_0 = 0.9\pi$ for $0 \leq t \leq 40$ using 10000 iterations.).

However, Runge-Kutta methods are inherently wasteful: after each new iteration, the extra function evaluations are deleted, only to be basically recalculated after the next step. Therefore, we should look now for methods that reuse past function evaluations. This boost in efficiency will come at the cost of stranger convergence behavior, however.

5 Multistep Methods

Definition 5.1. *Given an IVP $y' = f(x, y); y_0$, a linear m-step method is the algorithm that produces a numerical solution w_i, where each w_i is given by*

$$\sum_{l=0}^{m} a_l w_{i+l} = h \sum_{l=0}^{m} b_l f(t_{i+l}, w_{i+l})$$

*where a_l and b_l are well-chosen constants and $a_m = 1$. When $b_m = 0$, the final value y_{k+l} appears only on the left side, and the method is thus called "*explicit*".*

This is a very general formula, and as expected, it only forms a serviceable numerical method for certain values of a_l and b_l. First, let's analyze the local truncation error:

Definition 5.2. *Given the linear multistep method*

$$\sum_{l=0}^{m} a_l w_{i+l} = h \sum_{l=0}^{m} b_l f(t_{i+l}, w_{i+l}),$$

for some $t \in [a, b]$, $h \geq 0$, and a solution y to the initial value problem, the local truncation error is

$$\tau = \frac{1}{h} \sum_{l=0}^{m} a_l y(t + lh) - \sum_{l=0}^{m} b_l f(t + lh, y(t + lh)).$$

Remark 5.1. *(I haven't been able to find a "proof" of why this is the local truncation error, but following the thought that τ represents how much y fails to satisfy the linear m-step method, then this definition follows from a substitution of $y(t_i + lh)$ for w_{i+l} in the definition of the multistep method with an error term of $h\tau$),*

$$\sum_{l=0}^{m} a_l y(t + hl) = h \sum_{l=0}^{m} b_l f(t + hl, y(t + hl)) + h\tau.$$

Now, we can use the truncation error to put restrictions on the values of a_l and b_l that we want our algorithms to have.

Theorem 5.1. *A linear multistep method has local truncation error $\mathcal{O}(h^p)$ for $p \geq 1$ if and only if the following holds,*

$$\sum_{l=0}^{m} a_l = 0 \quad \text{and} \quad \sum_{l=0}^{m} l^j = j \sum_{l=0}^{m} l^{j-1} b_l \quad \text{for} \quad j \in \{1, \cdots, p\}.$$

Proof. First, write the local truncation error for some t, y, h:

$$\tau = \frac{1}{h} \sum_{l=0}^{m} a_l y(t + lh) - \sum_{l=0}^{m} b_l f(t + lh, y(t + lh))$$

Since $y'(t) = f(t, y)$, this is equivalent to

$$\tau = \frac{1}{h} \sum_{l=0}^{m} a_l y(t + lh) - \sum_{l=0}^{m} b_l y'(t + lh).$$

Take for granted that the solution y is C^p on $[a, b]$. Therefore, we can substitute the Taylor expansion for $y(t + lh)$ and $y'(t + lh)$:

$$\tau = \frac{1}{h} \sum_{l=0}^{m} \left[a_l \sum_{j=0}^{p} \frac{(lh)^j}{j!} y^{(j)}(t) \right] - \sum_{l=0}^{m} \left[b_l \sum_{j=0}^{p-1} \frac{(lh)^j}{j!} y^{j+1}(t) \right] + \mathcal{O}(h^p)$$

and then rearrange the terms to:

$$\tau = \frac{1}{h}\left(\sum_{l=0}^{m} a_l\right) y(t) + \sum_{j=1}^{p}\left[\frac{h^{j-1}}{j!}\left(\sum_{l=0}^{m} l^j a_l\right) y^{(j)}(t)\right]$$
$$-\sum_{j=0}^{p-1}\left[\frac{h^j}{j!}\left(\sum_{l=0}^{m} l^j b_l\right) y^{(j+1)}(t)\right] + \mathcal{O}(h^p)$$

re-indexing turns this into:

$$\tau = \frac{1}{h}\left(\sum_{l=0}^{m} a_l\right) y(t) + \sum_{j=1}^{p}\left[\frac{h^{j-1}}{j!}\left(\sum_{l=0}^{m} l^j a_l\right) y^{(j)}(t)\right]$$
$$-j\sum_{j=2}^{p}\left[\frac{h^{j-1}}{j!}\left(\sum_{l=0}^{m} l^{j-1} b_l\right) y^{(j)}(t)\right] + \mathcal{O}(h^p)$$

and after factoring, we have

$$\tau = \frac{1}{h}\left(\sum_{l=0}^{m} a_l\right) y(t) + \sum_{j=1}^{p}\left[\frac{h^{j-1}}{j!}\left(\sum_{l=0}^{m} l^j a_l - j\sum_{l=0}^{m} l^{j-1} b_l\right) y^{(j)}(t)\right] + \mathcal{O}(h^p)$$

This equation gives $\tau = \mathcal{O}(h^p)$ if and only if what's inside both parentheses equals zero. □

That takes care of consistency. What about stability?

Definition 5.3. *Given the m-step linear method*

$$\sum_{l=0}^{m} a_l w_{i+l} = h\sum_{l=0}^{m} b_l f(t_{i+l}, w_{i+l})$$

*Define the method's **characteristic polynomial** to be*

$$\chi(\lambda) = \sum_{l=0}^{m} a_l \lambda^l$$

*This polynomial satisfies the **root condition** when every root λ_j has $|\lambda_j| \le 1$, and if $|\lambda_j| = 1$ then the multiplicity of λ_j is 1.*

Notice that for an m-step method, m initial points must be specified. Therefore, we shall redefine convergence with this in mind:

Definition 5.4. *A linear m-step method is **convergent** if given some well-posed initial value problem $y'(t) = f(t, y(t)), y(t_0) = y_0$ and any $m-1$ initial values $I = \{w_0, \cdots, w_{m-1}\}$ for which $\lim_{h\to 0} |y(t_0 + ih) - w_i| = 0$ for all $0 \le i \le m-1$, the following is true:*

$$\lim_{h\to 0} |y(t_i) - w_i| = 0$$

for all $t_i \in [a, b]$.

Dahlquist's Equivalence Theorem (Dahlquist, 1956) relates these two conditions.

Theorem 5.2 (Dahlquist equivalence theorem)**.** *An m-step method is convergent iff the truncation error is of order $\tau = \mathcal{O}(h^p)$ with $p \ge 1$ and its characteristic polynomial satisfies the root condition*

A widely-used explicit linear m-step method is the class of Adams-Bashforth methods. It looks like this:

Algorithm 5.1 (Adams-Bashforth Method)**.**

$$w_{i+1} = w_i + h\sum_{l=0}^{m-1} b_l f(t_{i-l}, y_{i-l}), \quad \textit{where}$$

$$b_l = \frac{1}{h}\int_{t_i}^{t_i+1}\left(\prod_{\substack{j=0\\ j\neq l}}^{m-1} \frac{s - t_{i-j}}{t_{i-l} - t_{i-j}}\right) \mathrm{d}s$$

Saying where this formula comes from would require a background in numerical interpolation methods, but a sketch is that the product term forms a polynomial that approximates $f(s, y(s))$ for $s \in [t_i, t_{i+1}]$. This polynomial can then be integrated to approximate the change in y over $[t_i, t_{i+1}]$. It turns out that m-step Adams-Bashforth methods have local truncation error of order $\tau = \mathcal{O}(h^m)$. One important consideration is that multistep methods require multiple initial conditions. However, one can simply use a one-step method to generate such points. Also, notice that because these methods use past function evaluations, these values can be re-used, providing a great speed boost.

A related class of implicit methods is the class of Adams-Moulton methods:

Algorithm 5.2 (Adams-Moulton Method)**.**

$$w_{i+1} = w_i + h \sum_{l=0}^{m} c_l f(t_{i+1-l}, w_{i+1-l}), \quad \textit{where}$$

$$c_l = \frac{1}{h} \int_{t_i}^{t_{i+1}} \left(\sum_{\substack{j=0 \\ j \neq l}}^{m} \frac{s - t_{i+1-j}}{t_{i+1-l} - t_{i+1-j}} \right) \mathrm{d}s$$

Since w_{i+1} appears on both sides of the equation, these methods can be harder to implement. However, the 2-step version possesses some useful properties, discussed below.

Algorithm 5.3 (2-step Adams-Moulton Method)**.**

$$w_{i+1} = w_i + \frac{h}{2}(f(t_{i+1}, w_{i+1}) + f(t_i, w_i))$$

6 Absolute Stability

We now have some methods that we know to be convergent, meaning that for infinitesimal step size, the error becomes infinitesimal. However, what often matters more is the behavior of the error for some finite h. I would like to know what values of h yield a bounded error. To analyze this property, consider the test equation

$$y' = \gamma y \qquad y(0) = y_0$$

for some $\gamma \in \mathbb{C}$ with $\Re(\gamma) < 0$. It can be shown that for the solution to this IVP $y(t)$, we have $\lim_{t\to\infty} y(t) = 0$. Now, we can use this test formula to define the set of step sizes for which the numerical solution also approaches zero.

Definition 6.1. *Given a numerical method with a result w_i when applied to the test IVP $y'(t) = \gamma y, y(0) = y_0$ where $t \in [0, \infty)$, define the region of absolute stability as the set:*

$$S = \left\{ h\gamma \in \mathbb{C} \mid \lim_{i\to\infty} w_i = 0 \right\}$$

It immediately follows that with the test equation, if the numerical result approaches zero, i.e. the step size is within the region of absolute stability, then the global error approaches zero, so the method is convergent. For step sizes outside the region of absolute stability, we would expect the solution to diverge, so the numerical method clearly is not convergent.

Example 6.1. *The region of absolute stability for Euler's method is the disk of radius* 1 *centered around $z = -1$ in the complex plane.*

The formula for Euler's method, where $f(t, y) = \gamma y$ is

$$w_i = w_{i-1} + h\gamma w_{i-1},$$

which we can rewrite as

$$w_i = w_{i-1}(1 + h\gamma),$$

which, by induction, is equivalent to

$$w_i = w_0(1 + h\gamma)^i.$$

This converges to 0 whenever $|1 + h\gamma| < 1$, so the region of absolute stability is

$$S = \{h\gamma \in \mathbb{C} \mid |1 + hy| < 1\},$$

which looks like a unit disk centered at $z = -1$.

Example 6.2. *The region of absolute stability for the 2-step Adams-Moulton method is the left half of the complex plane, all $z \in \mathbb{C}$ where $\Re(z) < 0$.*

The Adams-Moulton method for the test equation is

$$w_i = w_{i-1} + \frac{h}{2}(\gamma w_i + \gamma w_{i-1})$$

which we can re-arrange to

$$w_i(1 - \frac{1}{2}h\gamma) = w_{i-1}(1 + \frac{1}{2}h\gamma),$$

or equivalently,

$$w_i = w_{i-1}\left(\frac{1 + \frac{1}{2}h\gamma}{1 - \frac{1}{2}h\gamma}\right).$$

By induction, we have

$$w_i = w_0\left(\frac{1 + \frac{1}{2}h\gamma}{1 - \frac{1}{2}h\gamma}\right)^i.$$

This converges if and only if

$$\left|\frac{1 + \frac{1}{2}h\gamma}{1 - \frac{1}{2}h\gamma}\right| < 1$$

which is true whenever $\Re(h\gamma) < 0$. Therefore, the 2-step Adams-Moulton method's region of absolute stability is the entire left half of the plane. We call methods like this "A-stable":

Definition 6.2. *A numerical method is* ***A-Stable*** *whenever its region of absolute stability is*

$$S = \{z \in \mathbb{C} \mid \Re(z) < 0\}$$

This region is quite large. Therefore, A-stable methods are more forgiving can be used for a wider range of problems. However, A-stability is pretty hard to come by.

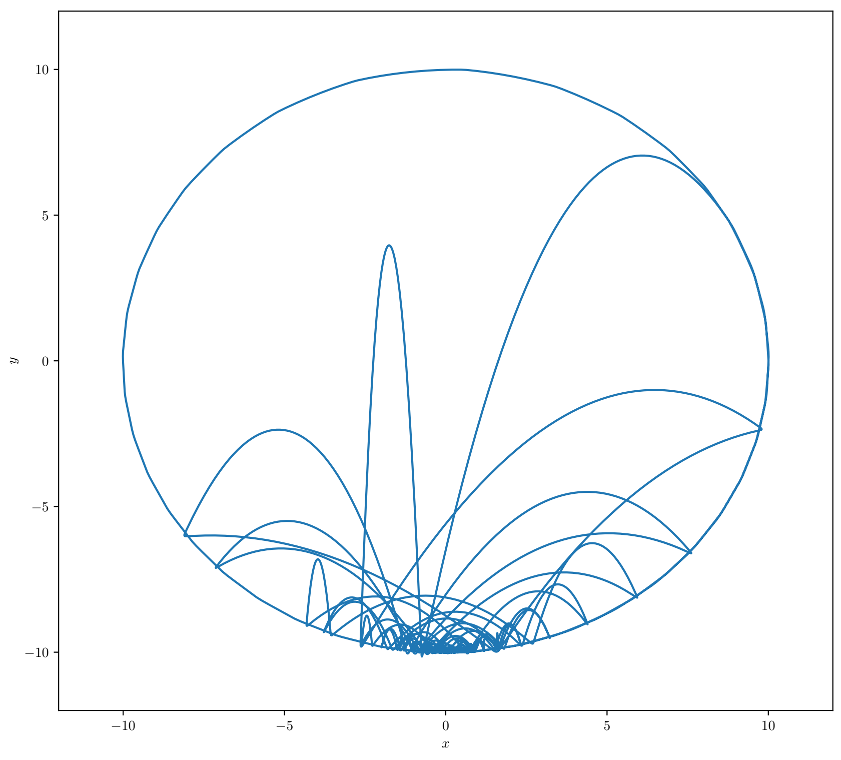
Theorem 6.1 (Dahlquist's Barrier)**.** *All A-stable linear m-step methods have $m \leq 2$.*

Proof. See Dahlquist (1956). □

7 Engineering concerns

In this paper, I've tried to provide an introduction to numerical methods with a moderately theoretical perspective. Therefore, certain considerations for actually implementing efficient numerical methods were not considered. An overview of the most important ones are presented below for the curious.

1. Let's say we have decided to apply a linear one-step method. How would we choose between, say Euler's method and Runge-Kutta. One would think that Runge-Kutta must be a better choice because it's truncation error has higher order. However, it also requires more function evaluations, and therefore will take more time to solve the same problem. Therefore, it would be a good idea to do a few test runs comparing the Runge-Kutta method with four times the step size for Euler's method.
2. All the methods presented above assume a fixed step size. However, this is not a requirement. Indeed, most numerical methods used in the real world use a variable step size. For values of t, w_i where f is relatively smooth (i.e. higher-order terms are small), the step size can be increased drastically, and then reduced again when higher-order terms become significant. This usually requires estimating the error somehow, usually using a test evaluation of an order $m + 1$ method, while the solution is then approximated using an m-order method. While variable step-size is non-trivial to implement and annoying to rigorously analyze, it can be ridiculously efficient.
3. Since each evaluation of a mathematical function introduces roundoff error, there usually will be a steep increase in global error for extremely small step-size. This critical step-size is dependent on the processor, of course.



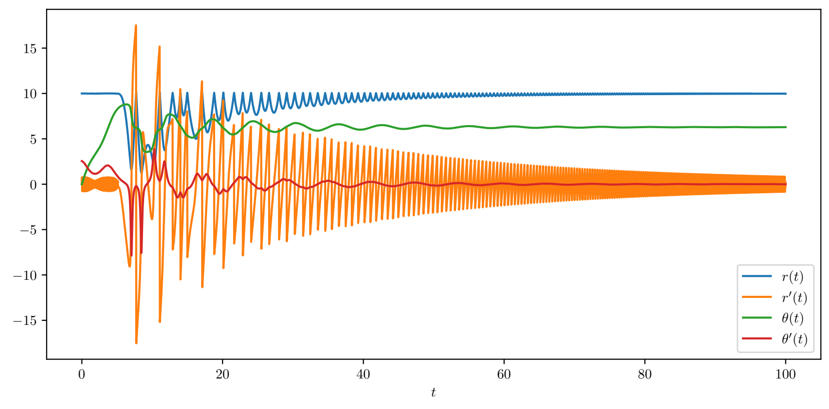


Figure 1: 4th-order Runge-Kutta approximation of the tetherball system.

8 Results

If I haven't yet convinced you how cool numerical methods are, hopefully this will. Imagine a bouncy tetherball that is restricted to move in one plane. There is a flexible string of negligible mass with one end rigidly fixed to the origin and the other end connected to a weight. Gravity acts on the entire system, and whenever the tether is stretched to its maximum length, it bounces back with equal and opposite force. Because we can, suppose aerodynamic drag also acts on the weight. This system is governed by the following second-order system of ODE's, where $r(t)$ is the distance from the weight to the origin and $\theta(t)$ is the angle between the weight and the vertical.

$$\begin{aligned}
r''(t) &= r(t)\big(\theta'(t)\big)^2 + g\cos\theta(t) + \frac{f_s}{m} - \frac{d}{m}r'(t)\\
\theta''(t) &= -\frac{1}{r}\left(g\sin\theta + 2r\theta'(t) + \frac{rd}{m}\theta'(t)\right)\\
f_s &= \begin{cases} 0 & r(t) \le l \\ k(l - r(t)) & r(t) > l \end{cases}, \quad \text{where}\\
g &= 10\,\mathrm{m/s^2} \quad \text{is the acceleration due to gravity,}\\
m &= 1\,\mathrm{kg} \quad \text{is the mass attached to the tether,}\\
l &= 10\,\mathrm{m} \quad \text{is the length of the tether,}\\
d &= 0.1\,\mathrm{kg\,s^{-1}} \quad \text{is the strength of the aerodynamic drag, and}\\
k &= 1000\,\mathrm{N\,m^{-1}} \quad \text{is the springiness of the tether.}
\end{aligned}$$

Suppose the weight is hanging still, and someone gives it a strong counterclockwise push. Experience would lead us to expect the weight to

1. Complete a few orbits with the cable taught while it has sufficient kenetic energy,
2. leave the circular orbit as its velocity decreases, with the cable going slack,
3. fall freely until the string becomes tight again,
4. bounce erratically a few times as the cable's elasticity deflects the weight's velocity, and
5. return to a steady hang.

Any symbolic/closed-form solution that describes the weight's position as a function of time would be horrendous. (Even a pendulum simply swinging has no closed-form solution.) Therefore, we should use a numerical solver to model this situation. Applying the 4th-order Runge-Kutta Method to this system over $t \in [0, 100]$seconds with a step size $h = 0.0001$ and the initial conditions

$$r(0) = 10\,\mathrm{m} \quad r'(0) = 0\,\mathrm{m\,s^{-1}} \quad \theta(0) = 0\,\mathrm{rad}, \quad \text{and} \quad \theta'(0) = 2.55\,\mathrm{rad\,s^{-1}}$$

produces the results in figure 1. The result indeed is how we would expect a tetherball to behave.

Appendix: Python Implementation

Implementation 8.1. *Euler's Method (Algorithm 4.1)*

```
def euler(fn, y_0, h, n):
    out = np.zeros((n, len(y_0)))
    for i in tqdm(range(n)):
        y = out[i-1]
        if i == 0:
            out[0] = y_0
        else:
            t = h*i
            y = y + fn(t, y) * h
            out[i] = y
    return out
```

Implementation 8.2. *4th-Order Runge-Kutta Method (Algorithm 4.3)*

```
def runge(fn, y_0, h, n):
    out = np.zeros((n, len(y_0)))
    out[0] = y_0
    for i in tqdm(range(n)):
        y = out[i-1]
        if i == 0:
            out[0] = y_0
        else:
            t = h*i
            q1 = fn(t, y)
            q2 = fn(t + h/2, y + q1 * h/2)
            q3 = fn(t + h/2, y + q2 * h/2)
            q4 = fn(t + h/2, y + q3 * h/2)
            y = y + h/6 * (q1 + 2 * q2 + 2 * q3 + q4)
            out[i] = y
    return out
```

Implementation 8.3. *4th-order Adams-Bashforth Method*

```
def adams_b4(fn, y_0, h, n):
    out = np.zeros((n, len(y_0)))
    out[0:4] = runge(fn, y_0, h, 4)
    a0 = fn(0, out[0])
    a1 = fn(1*h, out[1])
    a2 = fn(2*h, out[2])
    a3 = fn(3*h, out[3])
    for i in range(n):
        if i < 4:
            continue
        else:
            out[i] = out[i-1] + (h/24)*(55*a3 - 59*a2 + 37*a1 - 9*a0)
            a0 = a1
            a1 = a2
            a2 = a3
            a3 = fn(i*h, out[i])
    return out
```

Implementation 8.4. *Tetherball equations of motion*

```
def tetherball(t, y):
    k = 10000
    l = 10
    m = 1
    g = 10
    d = 0.1
    r = y[0]
    r_ = y[1]
    th = y[2]
    th_ = y[3]
    fs = 0
    if r > l:
        fs = k * (l - r)

    rr = r*th_**2 + g*cos(th) + fs/m - (d/m) * r_
    thh = 1/r * (-g*sin(th) - 2*r_*th_ - (d/m) * th_ * r)
    return np.array([r_, rr, th_, thh])
```

References

Burden, R. L. (c2011). *Numerical Analysis.* Brooks/Cole, Cengage Learning, Boston, MA.

Butcher, J. C. (1964). On Runge-Kutta processes of high order. *Journal of the Australian Mathematical Society*, 4(2):179–194.

Coddington, E. A. (1955). *Theory of Ordinary Differential Equations.* McGraw-Hill, New York.

Dahlquist, G. (1956). Convergence and stability in the numerical integration of ordinary differential equations. *MATHEMATICA SCANDINAVICA*, 4:33–53.

Greenbaum, A. and Chartier, T. P. (2012). *Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms.* Princeton University Press, Princeton, N.J.

Press, W. H., editor (2007). *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, Cambridge, UK ; New York, 3rd ed edition.

Süli, E. (2010). Numerical solution of ordinary differential equations. *Mathematical Institute, University of Oxford*.