Advanced Topics in Numerical Analysis

Thomas Trogdon
University of Washington
trogdon@uw.edu

Contents

Preface **v**

0 Julia basics **1**
- 0.1 Introduction . . . 1
- 0.2 Installing packages . . . 3
- 0.3 Loading packages . . . 3
- 0.4 Loops and conditionals . . . 3
- 0.5 Plotting . . . 4

I Numerical linear algebra **7**

II Approximation theory **9**

III Numerical solution of evolution problems **11**

1 Review of the theory of ordinary differential equations **13**
- 1.1 The initial-value problem for systems of ordinary differential equations 13
- 1.2 The matrix exponential . . . 14
- 1.3 A cautionary tale in ODE theory . . . 15
- 1.4 ODE existence and uniqueness theory . . . 16

2 Numerical methods for ordinary differential equations **21**
- 2.1 The Euler methods . . . 21
- 2.2 Newton's method . . . 22
- 2.3 A nonlinear ODE with the Euler methods . . . 24
- 2.4 The cautionary tale IVP . . . 26
- 2.5 Truncation errors . . . 27
- 2.6 Onestep methods . . . 30
- 2.7 Linear multistep methods . . . 33
- 2.8 A nonlinear test problem . . . 35
- 2.9 The value of higher-order methods . . . 42

3 Consistency, stability and convergence **45**

Bibliography **47**

Preface

Part I & II of these notes are just a thought at this point. Part III of these notes are for AMATH 586 taught from [LeV07] using `Julia`.

Throughout this text the results that are deemed the most important in the sense that they are critical for the main theoretical development of the subject highlighted by being boxed.

Chapter 0

Julia basics

0.1 ▪ Introduction

Julia is a scripting language like Matlab or Python. The main difference is that, by default, Julia uses just-in-time (JIT) compilation. Julia, like Python, and unlike Matlab, uses data types. In my opinion, Julia is more in tune with mathematicians' needs. Julia has data types like `SymTridiagonal` for a symmetric tridiagonal matrix. So, when you are then using backslash \ (yes, Julia has backslash, just like Matlab), you can be assured you are using the methods that are tuned for a symmetric tridiagonal matrix.

The syntax for Julia is very similar to Matlab and Python. There are some important differences. By default, Julia does not copy array when it is a function input:

> In Julia, all arguments to functions are passed by reference. Some technical computing languages pass arrays by value, and this is convenient in many cases. In Julia, modifications made to input arrays within a function will be visible in the parent function. The entire Julia array library ensures that inputs are not modified by library functions. User code, if it needs to exhibit similar behaviour, should take care to create a copy of inputs that it may modify.

This saves significant memory but it can easily cause unexpected behavior. Let us define a function to see how this goes.

```julia
function test_fun!(A)
    A[1,1] = 2*A[1,1]
    return A
end
```

Next, we define an array and apply the function to the array.

```julia
A = [1 2 3; 4 5 6] # Integer array
test_fun(A)
```

Last, we revisit the matrix `A`:

```
A # A has changed
```

```
2×3 Matrix{Int64}:
 2  2  3
 4  5  6
```

This is something that will never happen MATLAB.

But this, is not the end of the story. If you operate on the matrix as a whole, its value will not change

```
A = [1 2 3; 4 5 6] # Integer array
function test_fun2(A)
    A = 2*A
    return A
end
test_fun2(A)
```

Then we check `A`:

```
A # A has not changed
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

Next, if you “slice" the matrix then you will change the value, even if you get the whole matrix.

```
A = [1 2 3; 4 5 6] # Integer array
function test_fun3!(A)
    A[:,:] = 2*A[:,:]
    return A
end
test_fun3(A)
```

Then we again check `A`:

```
A # A has not changed
```

```
2×3 Matrix{Int64}:
 2   4   6
 8  10  12
```

Note that we use the `!` character in accordance with JULIA convention: functions that end in `!` modify one or more of their inputs.

MATLAB has different vectorized versions of arithmetic operations such as `.*`, `./`. JULIA has the same for functions like `abs(x)`. If `x` is a vector then you should call `abs.(x)`. Similarly, MATLAB will allow you to add a scalar to a vector with no change of syntax. JULIA will throw an error.

```
x = randn(10);
x + 1.0
```

```
ERROR: MethodError: no method matching +(::Vector{Float64}, ::Float64)
For element-wise addition, use broadcasting with dot syntax: array .+ scal
```

Instead, one needs to use `.+`:

```
x = randn(10);
x .+ 1.0
```

Something that is particularly helpful for reading complex code is that Julia allows the use of Unicode characters, and Greek letter in particular.

```
α = 1.
```

To get this, type `\alpha` then hit the tab key.

Julia is also very particular about types. For example, Matlab would have no issue with `zeros(10.0,10.0)` and would create a 10 × 10 matrix. Julia will throw an error. One should call `zeros(10,10)` instead.

0.2 ▪ Installing packages

There are a couple of ways to install new packages in Julia.The first is to execute

```
using Pkg
Pkg.add("NewPackage.jl")
```

to install `NewPackage.jl` The second method is to, when in the Julia terminal, press the `]` key to enter the package manager. Then just type

```
add NewPackage
```

Packages that you will want to install are `IJulia.jl`, `Plots.jl`, `FFTW.jl`. The most important native package to load is `LinearAlgebra.jl`.

0.3 ▪ Loading packages

To load `NewPackage.jl` simply enter

```
using NewPackage
```

0.4 ▪ Loops and conditionals

Loops in Julia take on aspects of both Matlab and Python. The most basic for loop is

```
sum = 0
for i = 1:10
    sum += i
end
```

Note that JULIA has scopes to its loops. In a clean JULIA instance, the following throws an error.

```
for i = 1:10
   h = i
end
h
```

```
ERROR: UndefVarError: h not defined
```

This is because the first instance of `h` is inside the loop so `h` only exists in that context. On the other hand, if `h` is used outside the loop first, then no error is encountered

```
h = 0
for i = 1:10
   h = i
end
h
```

```
10
```

JULIA allows you to loop through an array as well.

```
d = randn(100);
sum = 0;
for i in d
    sum += i
end
sum /= 100
```

```
0.12998325979929964
```

If statements are nearly the same as in MATLAB.

```
first = 0
for i in randn(1000)
    first += 1
    if i > 1
        break
    end
end
first
```

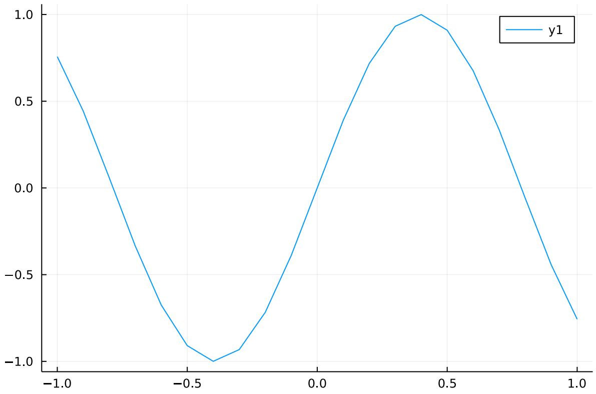
0.5 ▪ Plotting

The basic plotting functionality for JULIA is included in the `Plots.jl` package.

```
using Plots
```

The MATLAB `linspace` command can be easily replaced with commands like `-1:0.1:1` in most cases. And it is often nice to save a plot as a variable so that it can be saved later.

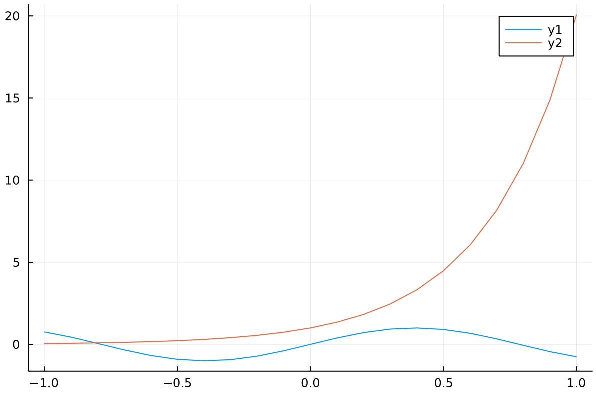
```
x = -1:0.1:1
f = x -> sin.(4x)
p = plot(x,f(x))
```



```
savefig(p,"sine.pdf")
```

If you wish to plot another function on the same axes, you should use `plot!` which will modify the given plot.

```
g = x -> exp.(3x)
plot!(x,g(x))
```



More detail for options that can be passed into `plot` can be found here: `https://docs.juliaplots.org/latest/attributes/`.

Part I

Numerical linear algebra

Part II

Approximation theory

Part III

Numerical solution of evolution problems

Chapter 1

Review of the theory of ordinary differential equations

1.1 ▪ The initial-value problem for systems of ordinary differential equations

Suppose $(u, t) \mapsto f(u, t)$ is a function that maps

$$\mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n.$$

In other words, $u \in \mathbb{R}^n$ and $t \in \mathbb{R}$. We think of u as the state of the system and t is a time variable. The initial-value problem (IVP) then takes the form[1]

$$\begin{cases} u'(t) = f(u(t), t), \quad t > t_0, \\ u(t) \in \mathbb{R}^n, \\ u(t_0) = \boldsymbol{\eta} \in \mathbb{R}^n. \end{cases} \tag{1.1}$$

To be precise, we look to solve this problem on some time interval $[t_0, t_1]$ and enforce that $u(t)$ should be, at a minimum, continuous on this interval, and continuously differentiable on (t_0, t_1).

Example 1.1. Many systems that may not initially look to be of this form can be transformed so that they are. Consider

$$\begin{cases} v'''(t) = -v'(t)v(t), \quad t > 0, \\ v(t) \in \mathbb{R}, \\ v(0) = \eta_1, \\ v'(0) = \eta_2, \\ v''(0) = \eta_3. \end{cases}$$

Define

$$u_1(t) = v(t), \quad u_2(t) = v'(t), \quad u_3(t) = v''(t).$$

Then we have

$$\begin{aligned} u_1'(t) &= u_2(t), \\ u_2'(t) &= u_3(t), \\ u_3'(t) &= v'''(t) = -v'(t)v(t) = -u_1(t)u_2(t). \end{aligned}$$

---

[1] Here we use the notation $u'(t) = \frac{\mathrm{d}}{\mathrm{d}t}u(t)$.

Assemble the vector

$$u(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \end{bmatrix}.$$

Then

$$u'(t) = \begin{bmatrix} u_2(t) \\ u_3(t) \\ -u_1(t)u_2(t) \end{bmatrix} = f(u(t), t).$$

In the previous example, we see that $f(u, t)$ actually has no dependence on t.

Definition 1.2. *If $f(u, t) = g(u)$ for some function $g : \mathbb{R}^n \to \mathbb{R}$ then the IVP* (1.1) *is said to be* autonomous.

It is also worth noting that non-autonomous systems can be made autonomous at the cost of increasing the dimension of the solution.

Example 1.3. Consider

$$v''(t) = tv(t), \quad t \geq 0.$$

Define

$$u_1(t) = v(t), \quad u_2(t) = v'(t), \quad u_3(t) = t.$$

Then assemble the solution vector u as in the previous example to find

$$u'(t) = \begin{bmatrix} u_2(t) \\ u_3(t)u_1(t) \\ 1 \end{bmatrix} = f(u(t), t).$$

For numerical purposes, this can be convenient. For analytical purposes, this can turn out to be terribly ill-advised because now it looks as if the differential equation is nonlinear!

1.2 ▪ The matrix exponential

A good reference for what follows in [LeV07, Appendix D], see also Appendix TBD below (see posted handwritten notes for now). We want to generalize functions

$$f : \Omega \to \mathbb{C},$$

where $\Omega \subset C$. Appendix TBD discusses how to do this in some generality.

Example 1.4.

$$f(z) = z^k \longrightarrow f(A) = A^k.$$

Example 1.5.

$$f(z) = \mathrm{e}^z = \sum_{n=0}^{\infty} \frac{z^n}{n!} \longrightarrow f(A) = \mathrm{e}^A := \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

Three important properties of the matrix exponential are

1. $\frac{\mathrm{d}}{\mathrm{d}t} \mathrm{e}^{tA} = A \,\mathrm{e}^{tA}$,
2. $\mathrm{e}^{sA}\,\mathrm{e}^{tA} = \mathrm{e}^{(s+t)A}$ (semi-group property), and
3. $\mathrm{e}^{0A} = I$.

We now use the matrix exponential to solve the IVP

$$\begin{cases} u'(t) = Au(t) + f(t), & t > t_0, \\ u(t) \in \mathbb{R}^n, \\ u(t_0) = \eta \in \mathbb{R}^n. \end{cases}$$

The main calculation we make here is that

$$\mathrm{e}^{tA} \frac{\mathrm{d}}{\mathrm{d}t} \left(\mathrm{e}^{-tA}\, u(t) \right) = u'(t) - Au(t),$$

where one uses properties (2) & (3) above. Thus by the fundamental theorem of calculus,

$$\begin{aligned} \int_{t_0}^{t} \frac{\mathrm{d}}{\mathrm{d}s} \left(\mathrm{e}^{-sA}\, u(s) \right) \mathrm{d}s &= \int_{t_0}^{t} f(s)\mathrm{d}s, \\ \mathrm{e}^{-tA}\, u(t) - \mathrm{e}^{-t_0}\, \eta &= \int_{t_0}^{t} f(s)\mathrm{d}s, \\ u(t) = \mathrm{e}^{(t-t_0)}\, \eta &+ \int_{t_0}^{t} \mathrm{e}^{(t-s)A} f(s)\mathrm{d}s. \end{aligned}$$

This last equation, the solution of the IVP, is called *Duhamel's formula.* It is important in the theory of ODEs and in their computation.

1.3 ▪ A cautionary tale in ODE theory

The previous calculation, the derivation of Duhamel's formula shows that linear ODEs have solutions for all time. The same is not true of nonlinear ODEs. Consider the Painlevé II differential equation

$$\begin{cases} u''(t) = tu(t) + 2u(t)^3, \\ u(0) = u_1 \in \mathbb{R}, \\ u'(0) = u_2 \in \mathbb{R}. \end{cases}$$

There exists a solution, for a specific choice of u_1, u_2 that is an infinitely differentiable function on all of $\mathbb{R}$. It has the asymptotics

$$u(t) = \mathrm{Ai}(t)(1 + o(1)), \quad t \to \infty,$$

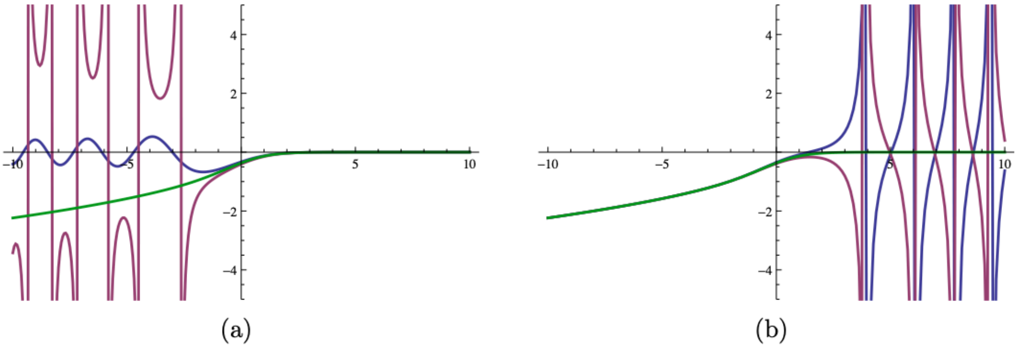


Figure 1.1: Solutions of the Painlevé II ODE with nearby initial conditions.

where Ai denotes the classical *Airy function* [OLBC10]. A generic perturbation of u_1, u_2 away from this specific choice will lead to a solution that has a pole on the real axis — the solution of the ODE fails to exist at time. In Figure 1.1 you can see solutions of this ODE with nearby initial conditions. Radically different behavior is observed for small perturbations. This is not an issue with a numerical approximation, this issue is due to the fact that the problem at hand is very difficult to solve.

1.4 ▪ ODE existence and uniqueness theory

We now discuss the theoretical underpinnings of ODE theory, at least in some detail. If you wish to read more, see [CLT55]. We recall the 2-norm for $u \in \mathbb{C}^n$

$$\|u\|_2^2 = u^* u.$$

The following definition can be made with for any norm $\|\cdot\|$ on $\mathbb{R}^n$

Definition 1.6. *A function $f : \mathcal{D} \to \mathbb{R}^n$ is said to be Lipschitz in u over the domain*

$$\mathcal{D} = \mathcal{D}(a, t_0, t_1) = \{(u,t) \in \mathbb{R}^n \times \mathbb{R} : \|u - \eta\| \leq a,\ t_0 \leq t \leq t_1\},$$

if

$$\|f(u,t) - f(u',t)\| \leq L\|u - u'\|,$$

for all $(u,t), (u',t) \in \mathcal{D}$.

One can discuss this concept in any norm, but we will stick with the 2-norm for concreteness.

Suppose $A \in \mathbb{R}^{n\times n}$ is a matrix. Recall that the operator norm, induced by a given norm $\|\cdot\|$, is defined by

$$\|A\| := \max_{x\neq 0} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \frac{\|Ax\|}{\|x\|}.$$

Proposition 1.7. *Suppose f is differentiable and the Jacobian matrix of f with respect* *to u bounded in (operator) matrix 2-norm*[2]*:*

$$\max_{(u,t)\in\mathcal{D}} \|D_u f(u,t)\|_2 = L < \infty,$$

then f is Lipschitz with constant L in the 2-norm.

Proof. Recall that differentiability of a function of multiple variables is typically written as

$$f(u,t) = f(u',t) + D_u f(u',t)(u-u') + o(\|u-u'\|), \quad u \to u'.$$

For a function of one variable (i.e., $u \in \mathbb{R}$), we could apply the mean-value theorem to conclude that

$$f(u,t) = f(u',t) + \partial_u f(c,t)(u-u'),$$

for some c between u and u'. And then if $u, u' \in \mathcal{D}$ then $c \in \mathcal{D}$ the conclusion follows. The problem is that the mean-value theorem does not apply to vector-valued functions. So, we need to turn our vector-valued function into a scalar valued one: $u \mapsto w^T f(u,t)$ is scalar valued for any vector $w \in \mathbb{R}^n$. One more thing needs to be done: We need to understand the notion of "between" in this context. So, consider

$$F(s) = w^T f(us + u'(1-s), t), \quad s \in [0,1].$$

The chain rule implies

$$F'(s) = w^T D_u f(us + u'(1-s), t)(u-u').$$

The mean value theorem in this context implies the existence of c such that

$$F(1) = F(0) + F'(c) \Rightarrow w^T f(u,t) = w^T f(u',t) + w^T D_u f(uc + u'(1-c), t)(u-u').$$

Then, we choose w to be a unit vector that points in the direction of $f(u,t) - f(u',t)$ giving

$$\begin{aligned} w^T f(u,t) - w^T f(u',t) = \|f(u,t) - f(u',t)\|_2 &\leq |w^T D_u f(uc + u'(1-c), t)(u-u')| \\ &\leq \|w\|_2 \|D_u f(uc + u'(1-c), t)(u-u')\|_2 \leq L\|u-u'\|_2. \end{aligned}$$

So, if f is continuously differentiable on $\mathcal{D}$ then it is Lipschitz, making this condition fairly easy to check in practice.

Theorem 1.8. *Suppose f is Lipschitz continuous in u with constant L over $\mathcal{D}$. Suppose further that $f(u,t)$ is continuous on $\mathcal{D}$. Then there is a unique solution to*

$$\begin{cases} u'(t) = f(u(t),t), & t > t_0, \\ u(t_0) = \eta, \end{cases}$$

---

[2] See Appendix in [LeV07] for more detail.

for

$$t_0 < t \le \min\{t_1, t_0 + a/S\}, \quad S = \max_{(x,t)\in\mathcal{D}} |f(u,t)|.$$

Example 1.9. Consider

$$\begin{cases} u'(t) = u(t)^2, & t > 0,\\ u(0) = 1. \end{cases}$$

This first-order ODE is solvable by separating variables:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = u^2 \Rightarrow \int \frac{\mathrm{d}u}{u^2} = \int \mathrm{d}t.$$

From this we find that

$$-\frac{1}{u} = t + C \Rightarrow C = -1.$$

Solving for u, we find

$$u(t) = \frac{1}{1-t}.$$

The solution blows up at $t = 1$! This does not contradict the above theorem because of how it accounts for S. From time $t = t_0$ we would solve

$$\begin{cases} u'(t) = u(t)^2, & 0 \le t_0 < t < 1,\\ u(t_0) = \frac{1}{1-t_0}. \end{cases}$$

It is then clear that $S = \left[\frac{1}{1-t_0} + a\right]^2$ then

$$t_0 + a/S \le t_0 + \frac{a}{\left[\frac{1}{1-t_0} + a\right]^2} < t_0 + (1-t_0)^2 \le 1.$$

The theorem gives us a smaller and smaller existence window as we approach the singularity and the window never includes $t = 1$.

Example 1.10. Consider

$$\begin{cases} u'(t) = Au(t), & t > 0,\\ u(0) = \eta. \end{cases}$$

For $f(u,t) = Au$, we have that $D_u f(u,t) = A$ and therefore the Lipschitz constant $L = \|A\|_2$. Then

$$S = \max_{|u-\eta|\le a} \|Au\|_2 \le \|A\|_2(\|\eta\|_2 + a).$$

So, we are guaranteed to have a solution for

$$0 < t \le \frac{a}{\|A\|_2(\|\eta\|_2 + a)} \le a/S.$$

This might seem to indicate that the solution will be valid over smaller and smaller time intervals if η is larger. We know this is not true because the solution is

$$u(t) = \mathrm{e}^{tA}\,\eta, \quad t > 0,$$

which is valid for all t.

Exercise 1.11. *Suppose the ODE system*

$$\begin{cases} u'(t) = f(u(t),t), & t > t_0, \\ u(t) = \eta, \end{cases}$$

is known to have a solution over the interval $t_0 < t < t_0 + \Delta t$ *for a fixed* Δt *that is independent of both* t_0 *and* η*. Show the solution exists for all time.*

1.4.1 ▪ The importance of the Lipschitz constant

Note that the Lipschitz constant does not appear in these calculations. The proof of Theorem 1.8 does require a finite Lipschitz constant but then the result is optimized in such a way that the constant does not appear in the final formula. But note that how large f can be on $\mathcal{D}$ can be bounded using η, a and the Lipschitz constant. Nevertheless, the Lipschitz constant does tell us something important about solutions. Consider two solutions of the same ODE

$$\begin{cases} u_1'(t) = f(u_1(t),t), \quad u_1(0) \text{ given}, u_2'(t) = f(u_2(t),t), \quad u_2(0) \text{ given}. \end{cases}$$

We now want to see how the difference evolves by computing

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\|u_1(t) - u_2(t)\|_2^2 &= \frac{\mathrm{d}}{\mathrm{d}t}(u_1(t) - u_2(t))^T(u_1(t) - u_2(t)) \\ &= (u_1'(t) - u_2'(t))^T(u_1(t) - u_2(t)) + (u_1(t) - u_2(t))^T(u_1'(t) - u_2'(t)) \\ &= 2(u_1'(t) - u_2'(t))^T(u_1(t) - u_2(t)). \end{aligned}$$

Therefore

$$\frac{\mathrm{d}}{\mathrm{d}t}\|u_1(t) - u_2(t)\|_2^2 \le 2\|f(u_1(t),t) - f(u_2(t),t)\|_2\|u_1(t) - u_2(t)\|_2 \le 2L\|u_1(t) - u_2(t)\|_2^2.$$

This is a differential inequality and typically, they are difficult to analyze. But his is reasonable and we find a simple ODE to compare things too. Consider

$$\begin{cases} v'(t) = 2Lv(t), \\ v(0) = 1. \end{cases}$$

Then, of course $v(t) = \mathrm{e}^{2Lt}$. Another simple observation is that $\frac{\|u_1(t)-u_2(t)\|_2^2}{v(t)} \ge 0$. Now differentiate this quantity

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\frac{\|u_1(t) - u_2(t)\|_2^2}{v(t)} &= \frac{-v(t)\|u_1(t) - u_2(t)\|_2^2 + v(t)\frac{\mathrm{d}}{\mathrm{d}t}\|u_1(t) - u_2(t)\|_2^2}{v(t)^2} \\ &\le \frac{-2Lv(t)\|u_1(t) - u_2(t)\|_2^2 + 2L\|u_1(t) - u_2(t)\|_2^2}{v(t)^2} = 0. \end{aligned}$$

So, this is a decreasing function and we find that

$$\frac{\|u_1(t)-u_2(t)\|_2^2}{v(t)} \le \frac{\|u_1(0)-u_2(0)\|_2^2}{v(0)} \Rightarrow \|u_1(t)-u_2(t)\| \le \mathrm{e}^{Lt}\,\|u_1(0)-u_2(0)\|.$$

This is a form of what is known as *Gronwall's inequality* and it the maximum rate of deviation of two solutions — and uniqueness.

Example 1.12. Consider the two ODEs for $t > 0$

$$\begin{aligned} u(t) &= u(t), \\ v(t) &= -v(t). \end{aligned}$$

Solutions of these two problems behave very differently but the above inequality will give the same estimate for both.

Chapter 2

Numerical methods for ordinary differential equations

2.1 ▪ The Euler methods

To describe numerical methods for ODEs, we start with some notation. When approximating the solution of

$$\begin{aligned} u'(t) &= f(u(t)), \\ u(t_0) &= \eta, \end{aligned}$$

we use a sequence $U^0, U^1, \dots, U^n, \dots$ such that

$$U^0 = \eta, \quad U^n \approx U(t_n),$$

for a sequence of times

$$t_0 < t_1 < \cdots < t_n < \cdots .$$

The simplest method is called the *forward Euler* method and it is derived by replacing the derivative $u'(t_n)$ with its forward difference approximation

$$f(u(t_n)) = u'(t_n) \approx \frac{u(t_{n+1}) - u(t_n)}{t_{n+1} - t_n} \approx \frac{U^{n+1} - U^n}{t_{n+1} - t_n}.$$

For almost all of our discussion we will use

$$t_n = t_0 + nk, \quad k > 0,$$

and k will be called the *time step*. Thus, we arrive at

$$\frac{U^{n+1} - U^n}{k} = f(U_n)$$

$$\boxed{U^{n+1} = U^n + kf(U^n).}$$

This is the forward Euler method. This method is called *explicit* because U^{n+1} is given by an explicit formula in terms of the value at the previous time step.

If we replace the forward difference with a backward difference we obtain the *backward* Euler method:

$$f(u(t_{n+1})) = u'(t_{n+1}) \approx \frac{u(t_{n+1}) - u(t_n)}{t_{n+1} - t_n} \approx \frac{U^{n+1} - U^n}{t_{n+1} - t_n}.$$

Thus, we arrive at

$$\frac{U^{n+1}-U^n}{k} = f(U_{n+1})$$
$$\boxed{U^{n+1} = U^n + kf(U^{n+1}).}$$

This is the backward Euler method and this is an *implicit* method because this represents a formula that still needs to be solved for U^{n+1}. We now pause to discuss one of the most popular methods for doing just this.

2.2 ▪ Newton's method

Consider $g : \mathbb{R}^n \to \mathbb{R}^n$. We want to find a value x^* such that $g(x^*) = 0$, i.e., find a root. Supposing first that g is continuously differentiable, we have

$$g(x^*) = g(x) + D_x g(x)(x^* - x) + o(\|x^* - x\|), \quad x \to x^*.$$

Supposing that $g(x^*) = 0$ we solve for x^*, neglecting the lower order terms:

$$\begin{aligned} g(x) + D_x g(x)(x^* - x) &\approx 0, \\ x^* &\approx x - [D_x g(x)]^{-1} g(x). \end{aligned}$$

So, this gives a new guess for x^* based on our old guess x. And the Newton's method takes the form

$$\begin{aligned} x_0 &= \text{given}, \\ x_{n+1} = x_n - [D_x g(x_n)]^{-1} g(x_n), &\quad n = 0, 1, 2, \ldots. \end{aligned}$$

A key question is when does one stop? The simplest condition is that if a tolerance ϵ is specified then the iteration is run until

$$\|x_{n+1} - x_n\| = \| [D_x g(x_n)]^{-1} g(x_n)\| < \epsilon.$$

This is an *absolute error* condition. In some situations, it may make sense to use a *relative error* stopping condition.

Theorem 2.1. *Suppose $g : \Omega \to \mathbb{R}^n$ is twice continuously differentiable on an open set $\Omega \subset \mathbb{R}$. Suppose that $g(x^*) = 0$ for $x \in \Omega$ and that $D_x g(x^*)$ is non-singular. Then Newton's method converges if x_0 is sufficiently close to x^*. Furthermore, there exists a constant $c > 0$ such that*

$$\|x_{n+1} - x^*\| \leq c\|x_n - x^*\|^2.$$

Proof. We consider one step of Newton's method using our trick to still apply the mean-value theorem. Define

$$G(s) = w^T g(sx^* + (1-s)x_0).$$

Using Taylor's theorem

$$G(1) = G(0) + G'(0) + \frac{G''(\xi)}{2}, \text{ for some } \xi \in (0,1).$$

This gives

$$\begin{aligned} 0 = w^T g(x^*) &= w^T g(x_0) + w^T D_x g(x_0)(x^* - x_0) + \frac{w^T}{2} H_x g(\zeta)(x^* - x_0, x^* - x_0), \\ \zeta &= \xi x^* + (1-\xi)x_0, \end{aligned}$$

where $H_x g$ is the Hessian of g. Now, recall that x_1 is then chosen such that $g(x_0) + D_x g(x_0)(x_1 - x_0) = 0$ and this implies

$$\begin{aligned} w^T g(x_0) + w^T D_x g(x_0)(x^* - x_0) &= w^T g(x_0) + w^T D_x g(x_0)(x_1 - x_0) + w^T D_x g(x_0)(x^* - x_1) \\ &= w^T D_x g(x_0)(x^* - x_1). \end{aligned}$$

We are left with the relation

$$w^T D_x g(x_0)(x_1 - x_*) = \frac{w^T}{2} H_x g(\zeta)(x^* - x_0, x^* - x_0).$$

A useful estimate is that for any invertible matrix $\|x\| = \|A^{-1}Ax\| \leq \|A^{-1}\|\|Ax\|$. So we choose w to be the unit vector in the direction of $D_x g(x_0)(x_1 - x_*)$ (supposing it is non-zero, if it is zero, we have converged). And this gives

$$\frac{\|x_1 - x^*\|}{\|A^{-1}\|} \leq \frac{1}{2}\|H_x g(\zeta)(x^* - x_0, x^* - x_0)\|.$$

Now, there exists $c(\zeta)$ such that $\|H_x g(\zeta)(x^* - x_0, x^* - x_0)\| \leq c(\zeta)\|x^* - x_0\|^2$ so that

$$\|x_1 - x^*\| \leq \frac{\|A^{-1}\|c(\zeta)}{2}\|x^* - x_0\|^2.$$

Suppose that x_0 is in the ball $B_\epsilon(x^*) := \{x \in \mathbb{R}^n : \|x - x^*\| < \epsilon\}$ and that $\sup_{\zeta \in B_\epsilon(x^*)} c(\zeta) = L < \infty$. Then, by possibly shrinking ϵ, we find that

$$\frac{\|A^{-1}\|c(\zeta)}{2}\|x^* - x_0\| < 1/2.$$

This implies the theorem.

What follows is a numerical implementation of Newton's method. Note the exclamation point — the initial guess `x` is modified by the function.

```
function Newton!(x,g,Dg; tol = 1e-13, nmax = 100)
        for j = 1:nmax
            step = Dg(x)\g(x)
            x[1:end] -= step
            if maximum(abs.(step)) < tol
                break
            end
            if j == nmax
                println("Newton's method did not terminate")
            end
        end
        x
end
```

One also needs to choose the tolerance for stopping Newtons's method. As a rule, one needs it to be less than the truncation error (see Section 2.5 below). And to be safe, if the truncation error is $O(k^\alpha)$, setting the tolerance to be $O(k^{\alpha+1})$ should suffice in most instances.

2.3 ▪ A nonlinear ODE with the Euler methods

Consider

$$\begin{cases} v''(t) = -tv(t) + 2v^3(t), \\ v(0) = 1, \\ v'(0) = -1. \end{cases}$$

We are going to solve this by both forward and backward Euler methods. We first turn it into an autonomous system:

$$\begin{aligned} &u_1'(t) = v'(t) = u_2(t), \\ &u_2'(t) = v''(t) = -u_3(t)u_1(t) + 2u_1^3(t), \\ &u_3'(t) = 1. \end{aligned}$$

So,

$$f(u) = \begin{bmatrix} u_2 \\ -u_3u_1 + 2u_1^3 \\ 1 \end{bmatrix}.$$

First, define `f`.

```
f = u -> [u[2], -u[3]*u[1]+2*u[1]^3, 1.0] # use commas to get a vector in Julia
```

Then, we choose a final time `T` and a time step `k`.

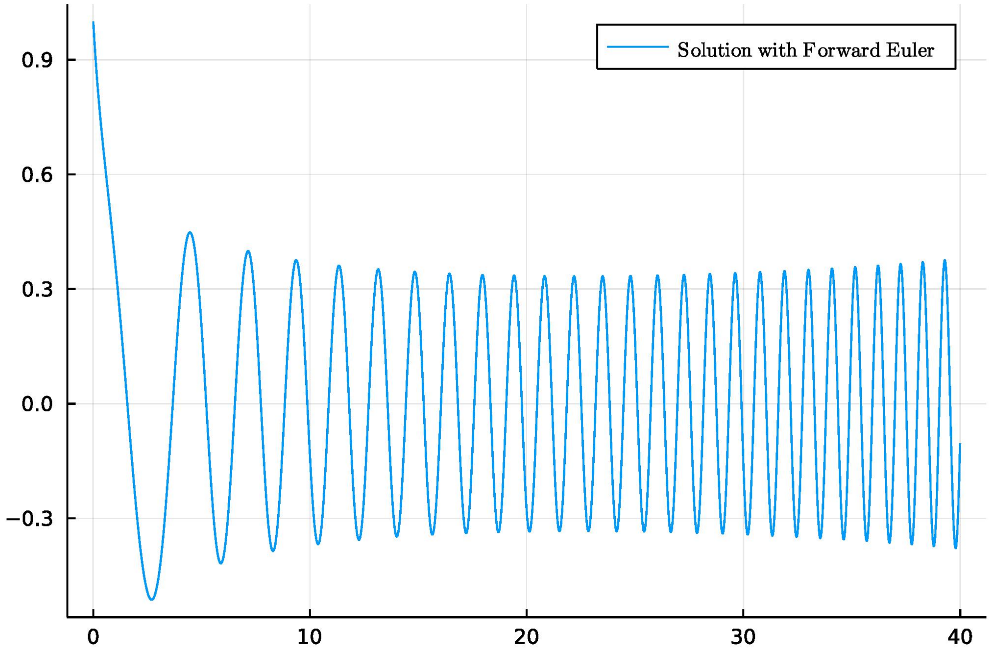
```
T = 40.# Final time.
k = 0.001 # Step size
```

The implementation of forward Euler is straighforward.

```
n = convert(Int64,T/k)# Number of time steps, converted to Int64
U = zeros(3,n+1) # To save the solution values
U[:,1] = [1.,-1.,0.]
for i = 2:n+1
    U[:,i] = U[:,i-1] + k*f(U[:,i-1])
end
```

The result is then plotted.

```
using Plots, LaTeXStrings # Import plotting functionality, and LaTeX
p = plot(U[3,:],U[1,:],label=L"\mathrm{Solution~with~Forward~Euler}")
```



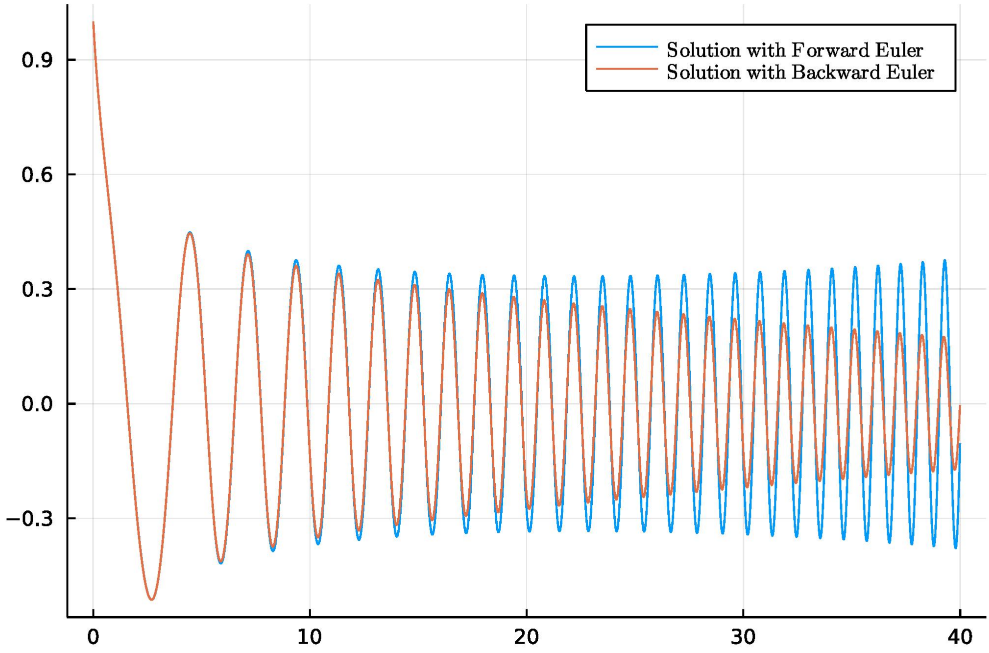
The implementation of backward Euler is more involved, but not too much because we have `Newton!` defined already. Set up the function `g` that we will find the roots of:

```
g = (U,Un) -> U - Un - k*f(U)
Dg = (U) -> [1. -k 0.0;
    k*U[3]-6*k*U[1]^2 1 k*U[1];
    0.0 0.0 1.0 ]
```

Then we simply run the iteration.

```
n = convert(Int64,T/k) # Number of time steps, converted to Int64
U = zeros(3,n+1) # To save the solution values
U[:,1] = [1.,-1.,0.]
max_iter = 10
for i = 2:n+1
    Unew = U[:,i-1] |> copy
    Newton!(Unew,u -> g(u,U[:,i-1]), Dg)
    U[:,i] = Unew
end
```

The one nuance here is that our `Newton!` function takes, as its second and third arguments, functions of one variable. As we have set it up, `g` is a function of two variables. So, we pass in a function that is really `g` with one argument specified.

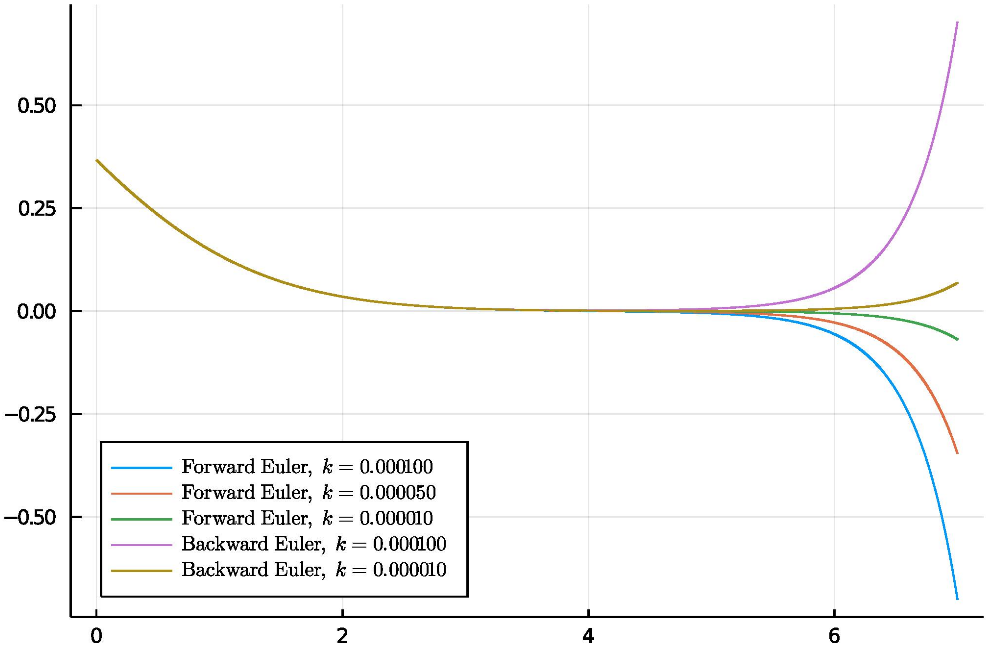


2.4 ▪ The cautionary tale IVP

Consider the following IVP

$$\begin{cases} u''(t) = tu(t) + 2u^3(t) \\ u(0) = 0.3670615515480782 \\ u'(0) = -0.2953721054475503 \end{cases}$$

Using the numerical methods (forward and backward Euler) from the previous section, we find the following approximate solutions.



It becomes clear that, while reducing the time step helps, the two numerical methods produce qualitatively different approximations.

2.5 ▪ Truncation errors

Before we begin the error analysis for the Euler methods, we consider an improvement upon them. We have seen that forward Euler tends to overestimate solutions and backward Euler tends to underestimate. them. What about their average?

$$\begin{aligned}
&U^{n+1} = U^n + kf(U^n),\\
&U^{n+1} = U^n + kf(U^{n+1}),\\
&\Downarrow\\
&\boxed{U^{n+1} = U^n + \frac{k}{2}\left[f(U^n) + f(U^{n+1})\right].}
\end{aligned}$$

This is called the *trapezoidal rule* because if applied to $u'(t) = g(t)$, it gives the trapezoidal rule approximation to the integral of g.

To perform the error analysis for a method, first make sure it is written so that there is no factor of k in front of f. For forward Euler

$$\frac{U^{n+1} - U^n}{k} - f(U^n) = 0.$$

Then “undo" the approximation

$$\tau^n_{\mathrm{FE}} = \frac{u(t_{n+1}) - u(t_n)}{k} - f(u(t_n)).$$

Thus τ^n is not zero. We use Taylor expansions, supposing $u(t)$ is as smooth as we want to analyze the order of τ^n. For example,

$$u(t_{n+1}) = u(t_n + k) = u(t_n) + ku'(t_n) + \frac{k^2}{2}u''(t_n) + O(k^3).$$

Then using that $u'(t_n) = f(u(t_n))$

$$\tau^n_{\text{FE}} = u'(t_n) + \frac{k}{2}u''(t_n) + O(k^2) - u'(t_n) = O(k).$$

Usually, we drop the n from t_n since this analysis does not depend on it. For backward Euler

$$\tau^n_{\text{BE}} = \frac{u(t+k) - u(t)}{k} - f(u(t+k)).$$

We can either Taylor expand around t or $t+k$. Using $t+k$ will result in one less term:

$$\begin{aligned}\tau^n_{\text{BE}} &= \frac{u(t+k) - u(t+k) + u'(t+k)k - \frac{k^2}{2}u''(t+k) + O(k^3)}{k} - u'(t+k)\\ &= -\frac{k}{2}u''(t+k) + O(k^2) = O(k).\end{aligned}$$

To analyze the trapezoidal method, we can see that $u''(t+k) = u''(t) + O(k)$ and averaging, we find

$$\tau^n_{\text{Trap}} = \frac{\tau^n_{\text{FE}} + \tau^n_{\text{BE}}}{2} = O(k^2).$$

In practice, you'll want to compute the coefficient of the leading term explicitly to make sure it does not also vanish, and ensure you have determined the true order of the truncation error.

Exercise 2.2. *Show that* $\tau^n_{\mathit{Trap}} = \left(\frac{1}{6} - \frac{1}{4}\right)u'''(t)k^2 + O(k^3)$.

We say that the Euler methods are first-order accurate and the trapezoidal method is second-order accurate. Having a method that is at least first-order accurate is a sense of *consistency*. We will soon introduce a notion of *stability* that will combine with consistency to ensure convergence.

Remark 2.3. *Some authors define* $k\tau^n$ *to be the truncation error. For us* $k\tau^n$ *will be referred to as the one-step error and it is, of course, always one order better than the truncation error.*

Another way to get a higher-order method is to replace the first-order accurate forward and backward differences with a centered difference,

$$u'(t_n) \approx \frac{u(t_{n+1}) - u(t_{n-1})}{2k}.$$

This gives the *leapfrog method*

$$\boxed{U^{n+1} = U^{n-1} + 2kf(U^n).}$$

Note that to advance the method to the next time step, one needs the previous two values. This is called *multistep method* whereas the Euler and trapezoidal methods are *onestep methods*. To analyze the truncation error here, expand about t

$$\begin{aligned}\tau_{\mathrm{LF}}^n &= \frac{u(t+k)-u(t-k)}{2k} - u'(t)\\ &= \frac{u(t)+ku'(t)+\frac{k^2}{2}u''(t)+\frac{k^3}{6}u'''(t)-u(t)+ku'(t)-\frac{k^2}{2}u''(t)+\frac{k^3}{6}u'''(t)+O(k^4)}{2k} - u'(t)\\ &= \frac{k^2}{6}u'''(t)+O(k^3).\end{aligned}$$

We then have the general defintion.

Definition 2.4. *A p-step multistep method is of the form*

$$U^{n+1} = F(U^{n+1}, U^n, U^{n-1}, \dots, U^{n-p+1}).$$

for some function $F : \underbrace{(\mathbb{R}^n \times \mathbb{R}^n \times \cdots \times \mathbb{R}^n)}_{p+1 \text{ times}} \to \mathbb{R}^n$.

The IVP is specified by giving $\eta = U^0$. For the leapfrog method, how does one find U^1?

The Starting Problem: For a multistep method, how does one compute the initial values required to begin using the method?

We will see that it suffices to use, if one wants, a method that has a local truncation error that is one order larger than the multistep method under consideration. So, we could pair leapfrog with forward Euler to obtain

$$\begin{aligned}U^1 &= U^0 + kf(U^0),\\ U^{n+1} &= U^{n-1} + 2kf(U^n), \quad n \geq 1.\end{aligned}$$

Now, let's consider an implicit multistep method. We first recall the backward differentiation formula (BDF)

$$\frac{3u(t+k)-4u(t)+u(t-k)}{2k} \approx u'(t+k).$$

And before we use this in a numerical method, we should review the analysis of this formula. We have

$$\begin{aligned}u(t) &= u(t+k) - ku'(t+k) + \frac{k^2}{2}u''(t+k) - \frac{k^3}{6}u'''(t+k) + O(k^4),\\ u(t-k) &= u(t+k) - 2ku'(t+k) + \frac{4k^2}{2}u''(t+k) - \frac{8k^3}{6}u'''(t+k) + O(k^4).\end{aligned}$$

From this, we find

$$\begin{aligned}3u(t+k)-4u(t)+u(t-k) &= 4ku'(t+k) - \frac{4k^2}{2}u''(t+k) + \frac{4k^3}{6}u'''(t+k)\\ &\quad - 2ku'(t+k) + \frac{4k^2}{2}u''(t+k) - \frac{8k^3}{6}u'''(t+k) + O(k^4)\\ &= 2ku'(t+k) - \frac{4k^3}{6}u'''(t+k) + O(k^4).\end{aligned}$$

This then implies that for the BDF method

$$\boxed{\frac{3U^{n+1}-4U^n+U^{n-1}}{2k}=f(U^{n+1}),}$$

we have

$$\tau^n_{\mathrm{BDF}}=-\frac{2k^2}{6}u'''(t+k)+O(k^3).$$

Method	Order	Steps	Implicit/Explicit
Forward Euler	1st	1	Explicit
Backward Euler	1st	1	Implicit
Trapezoidal	2nd	1	Implicit
Leapfrog	2nd	2	Explicit
BDF	2nd	2	Implicit

2.6 ▪ Onestep methods

Of the methods in the previous section, the forward Euler method is clearly the easiest to implement because it is

- onestep (no “starting problem"), and
- explicit (no rootfinding).

So, it is natural to ask if there exists higher-order methods with this same ease of implementation. And, of course, there does and there will also have to be a tradeoff of some sort.

2.6.1 ▪ Taylor series methods

One way to derive some onestep explicit methods is to user the Taylor expansion of $u(t)$:

$$\begin{aligned} u(t+k) &= u(t)+ku'(t)+\frac{k^2}{2}u''(t)+\cdots, \\ U^{n+1} &\approx U^n+kf(U^n,t)+\frac{k^2}{2}u''(t). \end{aligned}$$

Suppose $u(t)\in\mathbb{R}$. Since we do not know $u''(t)$, we need to close the system by considering

$$u''(t)=\frac{\mathrm{d}}{\mathrm{d}t}u'(t)=\frac{\mathrm{d}}{\mathrm{d}t}f(u(t),t)=f_u(u(t),t)u'(t)+f_t(u(t),t).$$

This then gives the second-order accurate *Taylor series method*

$$\boxed{U^{n+1}=U^n+kf(U^n,t_n)+\frac{k^2}{2}\left[f_u(U^n,t_n)f(U^n,t_n)+f_t(U^n,t_n)\right].}$$

This may be quite useful if $f(u,t)$ is simple. Note that if $u(t)\in\mathbb{R}^n$ then we need to replace $f_u(u,t)$ with the Jacobian $D_uf(u,t)$. This is not prohibitive, in a sense, because if we were using an implicit method we would want the Jacobian. But, we do not want to push this to third order because we will then need to use the Hessian (tensors!).

2.6.2 ▪ Runge–Kutta methods

Recall that in the use of leapfrog we had the starting problem

$$\frac{U^2 - U^0}{2k} = f(U^1),$$

and since we do not know U^1 we fail to start the method. But let us reduce k by a factor of 2:

$$\frac{U^1 - U^0}{k} = f(U^{1/2}), \quad U^{1/2} \approx u(t + k/2).$$

How do we get $U^{1/2}$? Well, we can simply take forward Euler with half a time step

$$U^{1/2} = U^0 + \frac{k}{2} f(U^0).$$

This gives the onestep, *multistage* Runge–Kutta (RK) method

$$\begin{aligned} U^{1/2} &= U^0 + \frac{k}{2} f(U^0), \\ U^1 &= U^0 + k f(U^{1/2}), \end{aligned}$$

or

$$\boxed{\begin{aligned} U^* &= U^n + \frac{k}{2} f(U^n), \\ U^{n+1} &= U^n + k f(U^*). \end{aligned}}$$

We might fear that because we have use two first-order methods in the derivation of this that we will be left with just a first-order method. But this is not the case! See [LeV07, Section 5.7] for a demonstration that this is indeed second-order accurate. With these multistage methods is is often convenient to also write out the non-autonomous versions,

$$\boxed{\begin{aligned} U^* &= U^n + \frac{k}{2} f(U^n, t_n), \\ U^{n+1} &= U^n + k f(U^*, t_n + k/2). \end{aligned}}$$

One of the most useful methods in existence is the fourth-order RK method (explicit!):

$$\boxed{\begin{aligned} Y_1 &= U^n, \\ Y_2 &= U^n + \frac{k}{2} f(Y_1, t_n), \\ Y_3 &= U^n + \frac{k}{2} f(Y_2, t_n + k/2), \\ Y_4 &= U^n + k f(Y_3, t_n + k/2), \\ U^{n+1} &= U^n + \frac{k}{6} \left[f(Y_1, t_n) + 2 f(Y_2, t_n + k/2) + 2 f(Y_3, t_n + k/2) + f(Y_4, t_n + k) \right]. \end{aligned}}$$

Note that if the global error is $O(k^4)$ for this method and we choose $k = 10^{-4}$ then we might hope to get a global error that is on the order of machine precision.

Lastly, with such a method, we need to ask about what the tradeoffs are. For systems of ODEs, $u(t) \in \mathbb{R}^n$, when n is small there is hardly any drawback to using this method.

But when n is very large, the evaluation of $f(u,t)$ make take significant computational effort and this function requires the evaluation of f four times (it might look like 7 on first glance, but this can be reduced to 4).

We now turn to generalities concerning Runge-Kutta methods.

Definition 2.5. *A general r-stage Runge–Kutta (RK) method is*

$$\boxed{\begin{aligned} Y_1 &= U^n + k\sum_{j=1}^{r} a_{1j} f(Y_j, t+c_jk), \\ Y_2 &= U^n + k\sum_{j=1}^{r} a_{2j} f(Y_j, t+c_jk), \\ &\vdots \\ Y_r &= U^n + k\sum_{j=1}^{r} a_{rj} f(Y_j, t+c_jk), \end{aligned}} \tag{2.1}$$

with

$$\boxed{U^{n+1} = U^n + k\sum_{j=1}^{r} b_j f(Y_j, t+c_jk).}$$

Remark 2.6. *Note that if $u(t) \in \mathbb{R}^n$ the a general r-stage RK method represents a system of rn nonlinear equations.*

In order for the method to be at least first-order accurate, i.e., consistent, one needs

$$\sum_{j=1}^{r} b_j = 1.$$

For convenience, we impose

$$c_p = \sum_{j=1}^{r} a_{pj}, \quad p = 1, 2, \ldots, r.$$

The coefficients that define an RK method are conveniently represented in a *Butcher tableau*,

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1r} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2r} \\ \vdots & \vdots & \vdots & & \vdots \\ c_r & a_{r1} & a_{r2} & \cdots & a_{rr} \\ \hline 1 & b_1 & b_2 & \cdots & b_r \end{array}.$$

If all the a_{pj}'s on and above the diagonal are zero then the method is fully explicit. For the fourth-order method (2.1)

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline 1 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}.$$

An important class of RK methods are the so-called diagonally implicit Runge–Kutta (DIRK) methods. These all have Butcher tableaus that are of the form

$$\begin{array}{c|cccc} c_1 & a_{11} & & & \\ c_2 & a_{21} & a_{22} & & \\ \vdots & \vdots & \vdots & \ddots & \\ c_r & a_{r1} & a_{r2} & \cdots & a_{rr} \\ \hline 1 & b_1 & b_2 & \cdots & b_r \end{array}.$$

These involve solving r systems of n nonlinear equations as opposed to one system of rn nonlinear equations. The general inversion of the Jacobian in the case of DIRK methods requires $O(rn^3)$ FLOPs as opposed to $O(r^3n^3)$ FLOPs for a general RK method — potentially significant savings.

Remark 2.7. *The general analysis of RK methods is difficult due to the nonlinear relationships that the parameters must satisfy.*

2.7 ▪ Linear multistep methods

We have already encountered multistep methods (leapfrog and BDF) but now we put them within a general framework.

Definition 2.8. *An r-step linear multistep method (LMM) is given by*

$$\boxed{\sum_{j=0}^{r} \alpha_j U^{n+j} = k \sum_{j=0}^{r} \beta_j f(U^{n+j}),} \tag{2.2}$$

with $\alpha_r = 1$.

Note that we only treat the autonomous case because, unlike the RK methods, it is clear that $f(U^{n_j})$ should simply be replaced by $f(U^{n+j}, t_{n_j})$. It should also be noted that if $\beta_r = 0$ then the method is explicit.

Two subsets of LMMs are:

- Adams methods: $\alpha_{r-1} = -1, \alpha_j = 0$ for $j < r - 1$,

$$\boxed{U^{n+r} = U^{n+r-1} + k \sum_{j=0}^{r} \beta_j f(U^{n+j}).}$$

  - Explicit Adams methods are called *Adams-Bashforth methods.*

  - Implicit Adams methods are called *Adams-Moulton methods.*

- Nyström methods: $\alpha_{r-1} = 0, \alpha_{r-2} = -1$, $\alpha_j = 0$ for $j < r-2$,

$$\boxed{U^{n+r} = U^{n+r-2} + k\sum_{j=0}^{r} \beta_j f(U^{n+j}).}$$

We can do some heuristic parameter counting to conjecture the order of accuracy we might expect in these methods. For Adams-Bashforth methods we need $\beta_r = 0$ which leaves us with $\beta_0, \ldots, \beta_{r-1}$ — r free parameters. We might expect that each one of these parameters can be used to eliminate a term given $O(k^r)$ LTE. With Adams-Moulton methods, we might think that the extra parameter β_r allows us to eliminate one more term, giving $O(k^{r+1})$ LTE. This is indeed correct and can be established in general.

We want to analyze the LTE of a LMMs, so we consider

$$\begin{aligned} 0 &= k^{-1}\sum_{j=0}^{r} \alpha_j U^{n+r} - \sum_{j=0}^{r} \beta_j f(U^{n+j}), \\ \tau_{\text{LMM}}^{n} &= k^{-1}\sum_{j=0}^{r} \alpha_j u(t+kr) - \sum_{j=0}^{r} \beta_j u'(t+kr). \end{aligned}$$

So, we need general expansions

$$\begin{aligned} u(t+jk) &= \sum_{\ell=0}^{m} u^{(\ell)}(t)\frac{(jk)^\ell}{\ell!} + O(k^{m+1}), \\ u'(t+jk) &= \sum_{\ell=0}^{m-1} u^{(\ell+1)}(t)\frac{(jk)^\ell}{\ell!} + O(k^{m}). \end{aligned}$$

This gives

$$\tau_{\text{LMM}}^{n} = \sum_{j=0}^{r} \frac{\alpha_j}{k} \sum_{\ell=0}^{m} u^{(\ell)}(t)\frac{(jk)^\ell}{\ell!} - \sum_{j=0}^{r} \beta_j \sum_{\ell=0}^{m-1} u^{(\ell+1)}(t)\frac{(jk)^\ell}{\ell!} + O(k^m).$$

We then interchange the order of summation to collect powers of k,

$$\begin{aligned} \tau_{\text{LMM}}^{n} &= \sum_{j=0}^{r} \frac{\alpha_j}{k} + \sum_{j=0}^{r} \frac{\alpha_j}{k} \sum_{\ell=1}^{m} u^{(\ell)}(t)\frac{(jk)^\ell}{\ell!} - \sum_{j=0}^{r} \beta_j \sum_{\ell=0}^{m-1} u^{(\ell+1)}(t)\frac{(jk)^\ell}{\ell!} + O(k^m), \\ &= \sum_{j=0}^{r} \frac{\alpha_j}{k} + \sum_{j=0}^{r} \alpha_j \sum_{\ell=0}^{m-1} u^{(\ell+1)}(t) j\frac{(jk)^\ell}{(\ell+1)!} - \sum_{j=0}^{r} \beta_j \sum_{\ell=0}^{m-1} u^{(\ell+1)}(t)\frac{(jk)^\ell}{\ell!} + O(k^m) \\ &= \sum_{j=0}^{r} \frac{\alpha_j}{k} + \sum_{\ell=0}^{m+1} k^\ell u^{(\ell+1)}(t) \left[\sum_{j=0}^{r} j^\ell \left(\frac{j a_j}{(\ell+1)!} - \frac{\beta_j}{\ell!}\right)\right] + O(k^m). \end{aligned}$$

To get first-order accuracy, we need

$$\sum_{j=0}^{r} \alpha_j = 0, \quad \sum_{j=0}^{r} (j\alpha_j - \beta_j) = 0.$$

To generate higher-order methods, solve

$$0 = \sum_{j=0}^{r} j^{\ell} \left(\frac{j a_j}{(\ell+1)!} - \frac{\beta_j}{\ell!} \right), \quad \ell = 1, 2, \ldots .$$

2.8 ▪ A nonlinear test problem

Nontrivial problems where a solution is known explicitly, preferably depending on a number of parameters, are of great interest to the numerical analyst. This gives concrete test problems to benchmark numerical methods. Here we consider

$$v'''(t) + v'(t)v(t) - \frac{\beta_1 + \beta_2 + \beta_3}{3} v'(t) = 0,$$

where $\beta_1 < \beta_2 < \beta_3$. It follows that

$$v(t) = \beta_2 + (\beta_3 - \beta_2)\mathrm{cn}^2\left(\sqrt{\frac{\beta_3 - \beta_1}{12}} t, \sqrt{\frac{\beta_3 - \beta_2}{\beta_3 - \beta_1}} \right)$$

is a solution where $\mathrm{cn}(x, k)$ is the Jacobi elliptic cosine function `https://dlmf.nist.gov/22` [OLBC10]. Some notations use $\mathrm{cn}(x, m)$ where $m = k^2$ (see `https://en.wikipedia.org/wiki/Jacobi_elliptic_functions`). The second argument of the cn function is called the elliptic modulus. The corresponding initial conditions are

$$\begin{aligned} v(0) &= \beta_3, \\ v'(0) &= 0, \\ v''(0) &= -\frac{(\beta_3 - \beta_1)(\beta_3 - \beta_2)}{6}. \end{aligned}$$

As always, we turn it into a system,

$$\begin{aligned} u_1'(t) &= v'(t) = u_2(t), \\ u_2'(t) &= v''(t) = u_3(t), \\ u_3''(t) &= \frac{\beta_1 + \beta_2 + \beta_3}{3} u_2(t) - u_2(t)u_1(t). \end{aligned}$$

So, set $c = \frac{\beta_1+\beta_2+\beta_3}{3}$

$$f(u) = \begin{bmatrix} u_2 \\ u_3 \\ u_2(c - u_1) \end{bmatrix}.$$

Because it will come back again, we have

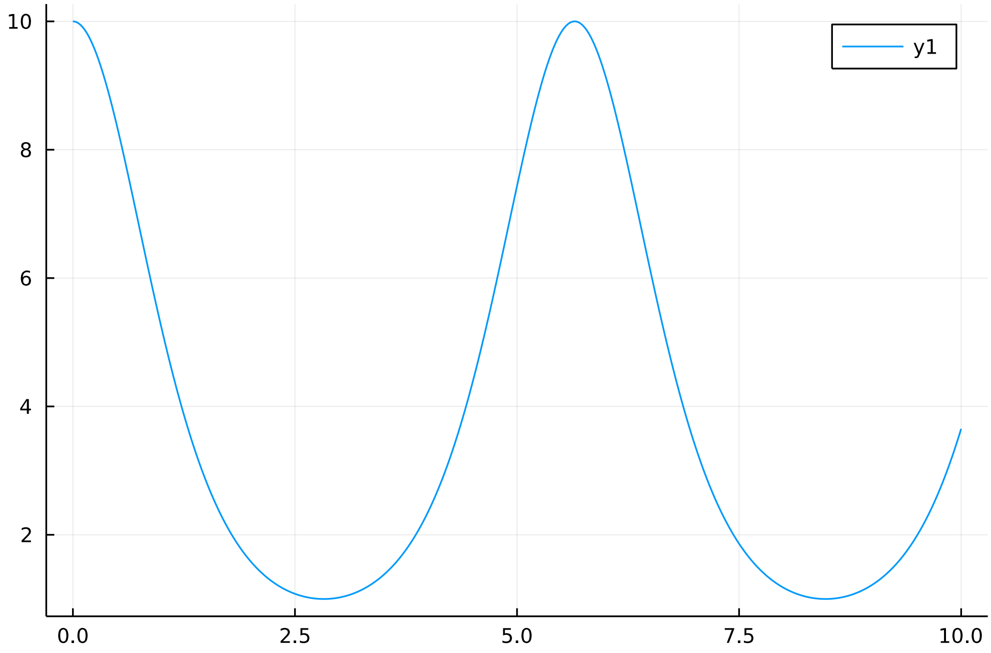
$$D_u f(u) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -u_2 & c - u_1 & 0 \end{bmatrix}.$$

To set up parameters and define the function that gives the solution

```
using Elliptic.Jacobi
β₁  = 0.
β₂  = 1.
β₃  = 10.
c = (β₁ + β₂ + β₃)/3
t = 0.:.01:10
v = t -> β₂ + (β₃ - β₂)*cn(sqrt((β₃-β₁)/12)*t, (β₃-β₂)/(β₃-β₁) )^2
```

Since the function `v` will throw an error if it is called with its argument being a vector, we use the extremely convenient `map` function.

```
plot(t, map(v,t))
```

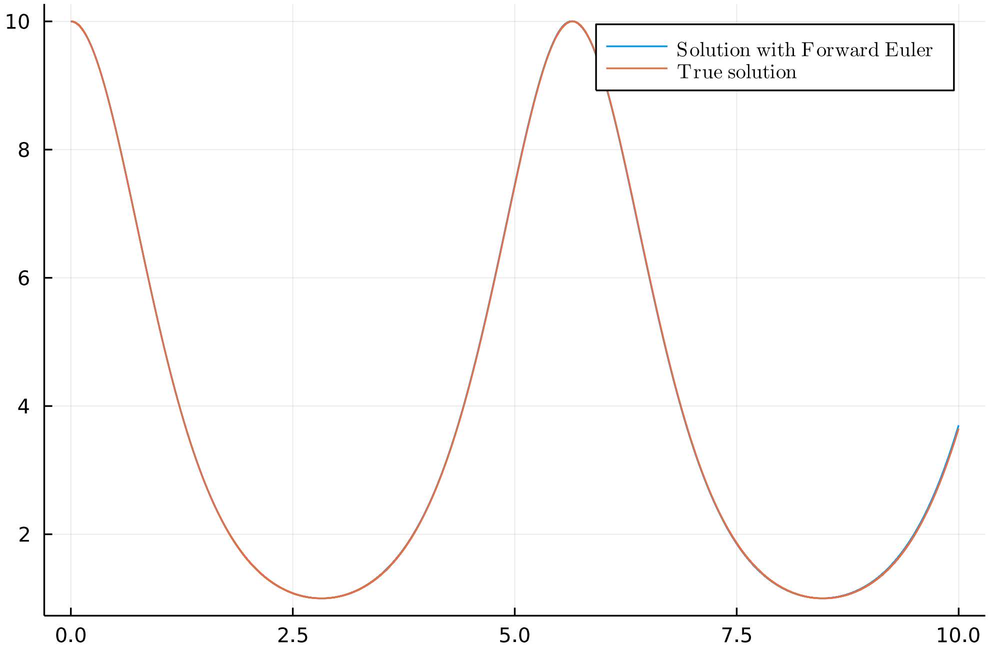


Now, we need to set up all the variables and functions so that we can apply various numerical methods.

```
f = u -> [u[2], u[3], u[2]*(c - u[1])]
Df = u -> [0. 1. 0.; 0. 0. 1.; -u[2] c-u[1] 0.]
u₀ = [β₃,0.,-1.0/6*(β₃-β₁)*(β₃-β₂)]
```

We start with forward Euler.

```
# Forward Euler
n = convert(Int64,ceil(T/k))# Number of time steps, converted to Int64
U = zeros(3,n+1) # To save the solution values
U[:,1] = u₀
t = zeros(n+1) # To save times
t[1] = 0.
for i = 2:n+1
    U[:,i] = U[:,i-1] + k*f(U[:,i-1])
    t[i] = t[i-1] + k
end
p = plot(t,U[1,:],label=L"\mathrm{Solution~with~Forward~Euler}")
plot!(t,map(v,t),label=L"\mathrm{True~solution}")
```



To the eye, one can see that the two curves deviate by $t = 10$. This is implying that forward Euler with this time step is not doing a good job.

An empirical approach to error analysis

We have determined the order of the LTE for all the methods we will consider in this section. The question remains to convince ourselves via a robust numerical experiment that the global error is on the same order. The following procedure is how we will do this in general. Here is the empirical error analysis for forward Euler.

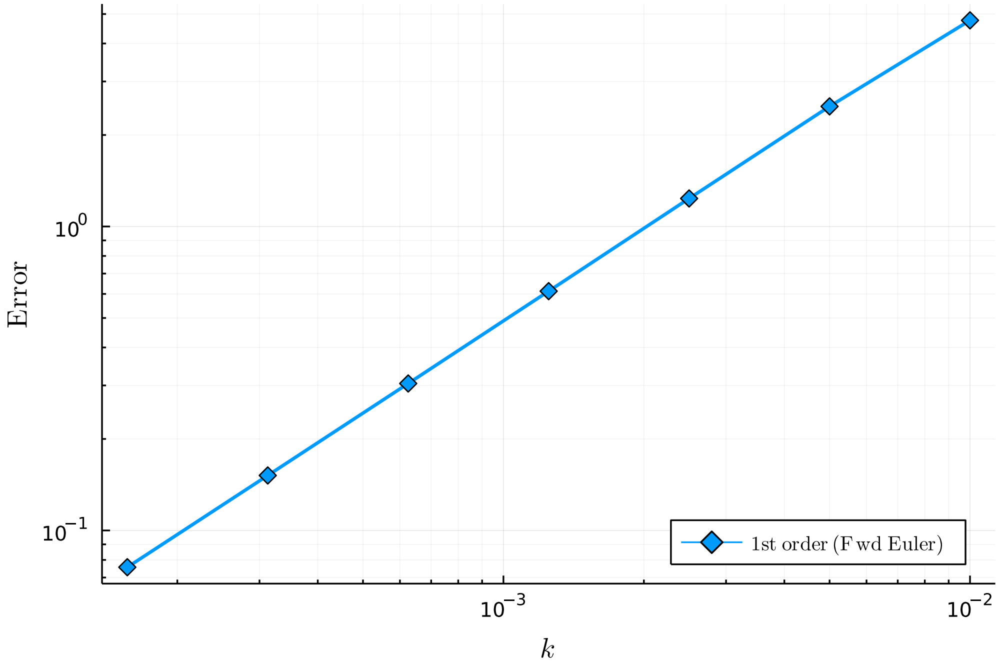
```
T = 10. # Final time.
k = .02
p = 7
data = zeros(p)
ks = zeros(p)
for i = 1:p
    k = k/2
    n = convert(Int64,ceil(T/k))
    println("Number of time steps = ", n)
    U = zeros(3,n+1) # To save the solution values
    U[:,1] = u₀
    t = zeros(n+1,1)
    t[1] = 0.
    for i = 2:n+1
        U[:,i] = U[:,i-1] + k*f(U[:,i-1])
        t[i] = t[i-1] + k
    end
    data[i] = abs(U[1,end] - v(t[end]))
    ks[i] = k
end
data_fe = data
```

```
Number of time steps = 1000
Number of time steps = 2000
Number of time steps = 4000
Number of time steps = 8000
Number of time steps = 16000
Number of time steps = 32000
Number of time steps = 64000

7-element Vector{Float64}:
 4.765943405224732
 2.4835157036567233
 1.2365055907962028
 0.6127307338668069
 0.3044443673615964
 0.1516739069309181
 0.07569136627506579
```

The vector that results is the error. We see that it approximately decreases by $1/2$ each time the time step is reduced by $1/2$. This is first-order accuracy. The relationship is maybe clearer when plotted on a log-log plot.

```
plot(ks,data,lw=2,ms=5,marker=:d, minorgrid = true, xaxis=(L"k",:log),
    yaxis= (L"\mathrm{Error}",:log),label=L"\mathrm{1st~order~(Fwd~Euler)}",
    legend = :bottomright)
```



We then produce the same kind of data for handful of other numerical methods.

Leapfrog

This is a two-step method so we start it with forward Euler.

```
T = 10. # Final time.
k = .02
p = 7
data = zeros(p)
ks = zeros(p)
for i = 1:p
    k = k/2
    n = convert(Int64,ceil(T/k))
    println("Number of time steps = ", n)
    U = zeros(3,n+1) # To save the solution values
    U[:,1] = u₀
    t = zeros(n+1,1)
    t[1] = 0.
    U[:,2] = U[:,1] + k*f(U[:,1])  # Begin the method using
    t[2] = t[1] + k                # forward Euler
    for i = 3:n+1
        U[:,i] = U[:,i-2] + (2*k)*f(U[:,i-1]) #Leapfrog
        t[i] = t[i-1] + k
    end
    data[i] = abs(U[1,end] - v(t[end]))
    ks[i] = k
end
data_leap = data
```

Trapezoid

At each time step we seek U^{n+1} which solves

$$U^{n+1} - U^n = \frac{k}{2}\left(f(U^{n+1}) + f(U^n)\right).$$

So, we look for a zero of

$$g(u, U^n) = u - U^n - \frac{k}{2}\left(f(u) + f(U^n)\right).$$

The Jacobian is given by

$$D_u g(u) = I - \frac{k}{2} D_u f(u).$$

In MATLAB and PYTHON the identity matrix is constructed by `eye(n)`. JULIA handles the identity matrix in a different way. When you perform `eye(n) + A` in MATLAB, it has to add (possibly zero) to every entry of `A`, $O(n^2)$ complexity. Julia does this by just adding to the diagonal using the `I` object, $O(n)$ complexity. You first have to import the `LinearAlgebra` package to use this.

```
using LinearAlgebra
I
```

```
UniformScaling{Bool}
true*I
```

```
Matrix{Float64}(I,2,2) # If you REALLY need to construct the identity matrix
```

```
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

```
A = randn(3,3)
A + I # The "size" of I is inferred
```

Constructing the Jacobian for g is now easier.

```
g = (u,Un) -> u - Un - (k/2)*(f(u)+f(Un))
Dg = u -> I - (k/2)*Df(u)
```

```
T = 10 # Final time.
k = 0.02
p = 7
data = zeros(p)
ks = zeros(p)
for i = 1:p
    k = k/2
    n = convert(Int64,ceil(T/k))
    println("Number of time steps = ", n)
    U = zeros(3,n+1) # To save the solution values
    U[:,1] = u₀
    t = zeros(n+1,1)
    t[1] = 0.
    max_iter = 10
    for i = 2:n+1
        t[i] = t[i-1] + k
        Unew = U[:,i-1] |> copy
        Newton!(Unew,u -> g(u,U[:,i-1]), Dg; tol = k^3/10)
        U[:,i] = Unew
    end
    data[i] = abs(U[1,end] - v(t[end]))
    ks[i] = k
end
data_trap = data
```

Two-step Adams–Moulton (AM) method

This method is given by

$$U^{n+2} = U^{n+1} + \frac{k}{12}\left(-f\left(U^{n}\right) + 8f\left(U^{n+1}\right) + 5f\left(U^{n+2}\right)\right).$$

Since this method is third order, we cannot start with Forward Euler as one step gives an error contribution of $O(k^2)$ which is, of course, much larger than the overall error of $O(k^3)$ that we expect. But we can start off with a second-order method. Let's choose the 2-stage second order Runge-Kutta method.

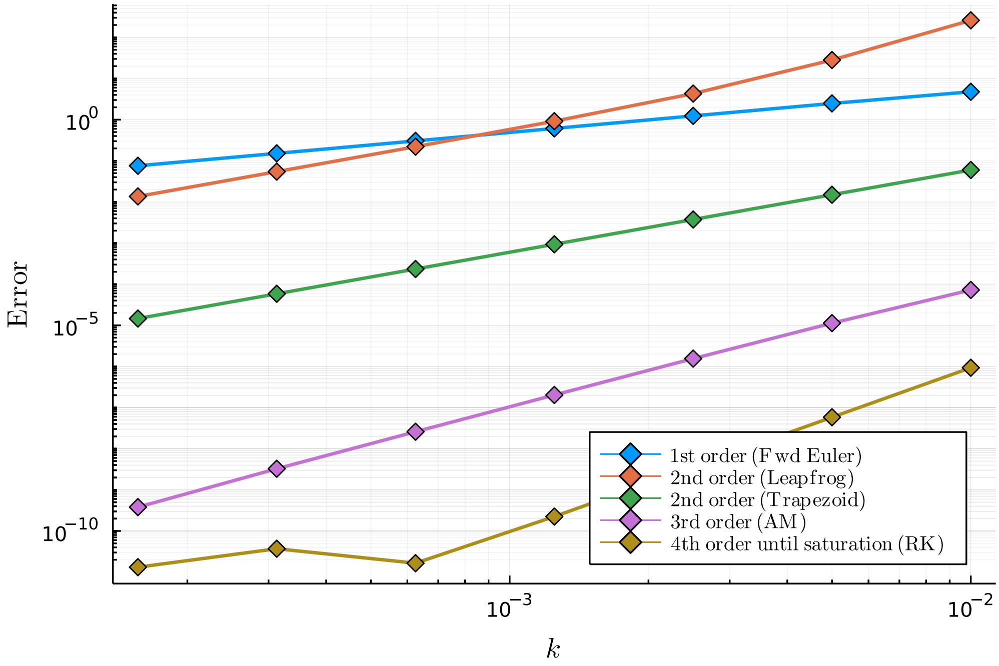
$$\begin{aligned} U^{*} &= U^{n} + \frac{k}{2}f(U^{n}), \\ U^{n+1} &= U^{n} + kf(U^{*}). \end{aligned}$$

Since this method is implicit, we need to set up our Jacobian for Newton's method.

```
g = (u,Un,Um) -> u - Un - (k/12)*(-f(Um)+8*f(Un)+5*f(u))
Dg = u -> I - (5k/12)*Df(u)
```

```
T = 10 # Final time.
k = .02
p = 7
data = zeros(p)
ks = zeros(p)
for i = 1:p
    k = k/2
    n = convert(Int64,ceil(T/k))
    println("Number of time steps = ", n)
    U = zeros(3,n+1) # To save the solution values
    U[:,1] = u₀
    t = zeros(n+1,1)
    t[1] = 0.
    max_iter = 10
    # Runge-Kutta second order here
    Us = U[:,1] + (k/2)*f(U[:,1])
    U[:,2] = U[:,1] + k*f(Us)
    t[2] = t[1] + k
    for i = 3:n+1
        t[i] = t[i-1] + k
        Unew = U[:,i-1] |> copy
        Newton!(Unew,u -> g(u,U[:,i-1]), Dg; tol = k^3/10)
        U[:,i] = Unew
    end
    data[i] = abs(U[1,end] - v(t[end]))
    ks[i] = k
end
data_am = data
```

When plotting the errors on the same axes, we find the following.



Yet another way to compare errors is to look at the error reduction ratio. This is defined by

$$\frac{\text{Error with time step } 2k}{\text{Error with time step } k}.$$

For an rth-order method we should see this be approximately 2^r.

```
methods = ["Forward Euler","Leapfrog","Trapezoid","Adams-Moulton","Runge-Kutta"];
d = Dict([(methods[1],data_fe),(methods[2],data_leap),(methods[3],data_trap),
    (methods[4],data_am),(methods[5],data_rk)]) # Use a dictionary, because we can
using Printf
@printf("%s        | %s |  %s |  %s |  %s  | %s \n",data_table[1,:]...)
for j=2:7
    @printf("%f | %0.4f        |  %0.4f   |  %0.4f    |  %0.4f
        | %0.4f \n",data_table[j,:]...)
end
```

```
k        | Forward Euler |  Leapfrog |  Trapezoid |  Adams-Moulton  | Runge-Kutta
0.005000 | 1.9190        |  9.2292   |  3.9961    |  6.4126         | 15.9713
0.002500 | 2.0085        |  6.5501   |  3.9991    |  7.2781         | 16.0036
0.001250 | 2.0180        |  4.6837   |  3.9998    |  7.6541         | 16.2192
0.000625 | 2.0126        |  4.1698   |  3.9999    |  7.8304         | 13.5952
0.000313 | 2.0072        |  4.0423   |  4.0000    |  7.9373         | 0.4452
0.000156 | 2.0038        |  4.0106   |  4.0000    |  8.5845         | 2.8099
```

2.9 ▪ The value of higher-order methods

In this section we do some “back-of-the-envelope" calculations to get a handle on what higher-order methods provide.

2.9.1 ▪ Complexity

Suppose we are working within a class of methods and the computation cost per step for the first-order method is C (i.e., the FLOPs per step). A reasonable, guess is that the rth order method will require rC FLOPs per step. LMMs are typically better than this, general RK methods are worse than this, RK methods with only a fixed number of non-zero diagonals in the Butcher tableau will satisfy this. So, fix a goal error tolerance δ and suppose the error of the rth-order method is approximately

$$Ek^r$$

for some constant $E > 0$. Solving $Ek^r = \delta$ for k, we find $k = (\delta/E)^{1/r}$. To evolve the approximate solution to a T, we have to make T/k time steps at a cost of rC, giving a total cost of

$$rCT(E/\delta)^{1/r}.$$

Setting $C = T = E = 1$, $\delta = 10^{-5}$, for example, we get the following table of computation costs.

Order	Cost
1	100000
2	632
3	139
4	71
5	50.

2.9.2 ▪ Rounding errors

Let us suppose that the runtime of an algorithm is not an issue for us. We are willing to wait as long as we need to. The issue with this is that the more FLOPs an algorithm requires, the more chances there are for rounding errors to accumulate. Let ϵ be our machine precision — a relative error of at most ϵ can occur on every arithmetic operation. We make similar assumptions as above, that a first order method requires C FLOPs and the rth order method requires rC FLOPs. Then an upper bound on the error would be of the form

$$g(k) = Ek^r + \epsilon\frac{TrC}{k} = E\left[k^r + \epsilon\frac{TrC}{Ek}\right].$$

We wish to find the minimizer k^* for g, which is the *optimal step size.* Define the new parameter $\epsilon' = \frac{\epsilon TC}{Ek}$. Then,

$$g'(k) = rEk^{r-1} - E\frac{r\epsilon'}{k^2} = 0 \Rightarrow k^* = (\epsilon')^{\frac{1}{r+1}}.$$

For an example, set $T = C = E = 1$, $\epsilon = 2.2 \times 10^{-16}$ to find the following pessimistic estimates.

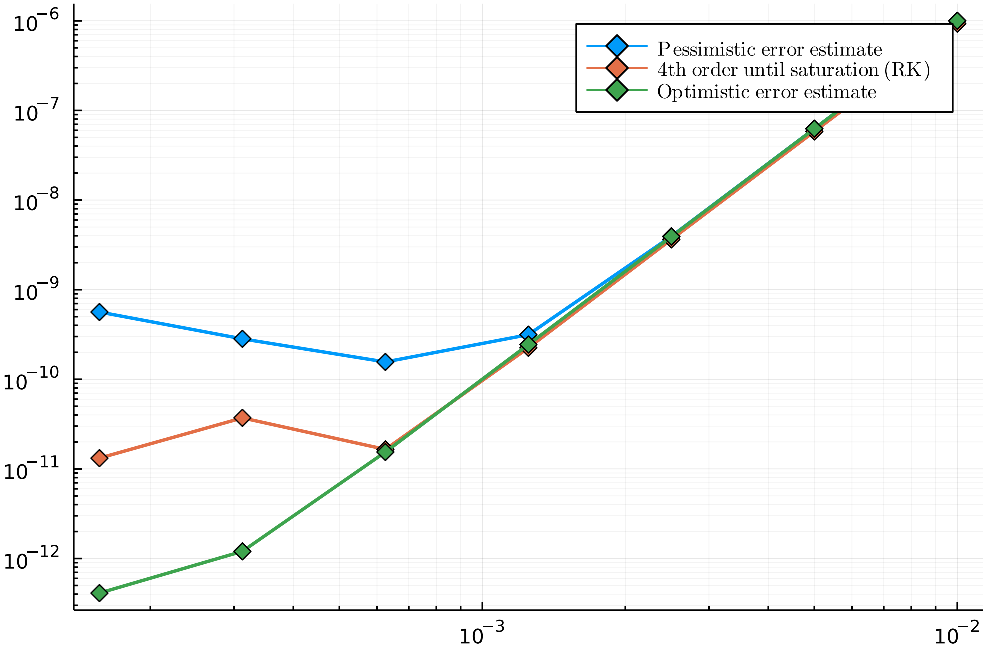
Order	Optimal step size	Optimal error
1	1.4832×10^{-8}	2.9666×10^{-8}
2	6.0368×10^{-6}	1.0933×10^{-10}
3	0.00012179	7.2257×10^{-12}
4	0.00073873	1.4890×10^{-12}
5	0.0024570	5.3724×10^{-13}

A (maybe) more realistic estimate of the error may be obtained by using

$$h(k) = E\left[k^r + \frac{\epsilon}{E}\sqrt{\frac{TrC}{k}}\right].$$

Order	Optimal step size	Optimal error
1	2.2958×10^{-11}	6.8873×10^{-11}
2	3.6004×10^{-7}	6.4814×10^{-13}
3	0.000023563	9.1582×10^{-14}
4	0.00024364	3.1712×10^{-14}
5	0.0010837	1.6438×10^{-14}

But typically, these are too optimistic. Using $E = 100, C = 10$ we use these two estimate for the RK4 method from the previous example with the Jacobi cn function.



We see that, indeed, one is too pessimisitc and one is too optimistic.

Chapter 3

Consistency, stability and convergence

Bibliography

[CLT55] E A Coddington, Norman Levinson, and T. Teichmann, *Theory of Ordinary Differential Equations*, McGraw-Hill, Inc., New York, USA, 1955.

[LeV07] R LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations*, SIAM, Philadelphia, PA, 2007.

[OLBC10] F W J Olver, D W Lozier, R F Boisvert, and C W Clark, *NIST Handbook of Mathematical Functions*, Cambridge University Press, 2010.