

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
1.5 Long-time existence	20
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
3.1 Convergence of onestep methods	45
3.2 Motivation of stability	48

3.3	Zero-stability and convergence	50
3.4	Absolute stability	51
3.5	Plotting regions of absolute stability	60
3.6	Heuristic implications of regions of absolute stability	63
3.7	Relative stability regions	66
3.8	Systems of stiff ordinary differential equations	68
3.9	L-stability and $A(\alpha)$ -stability	74
4	Numerical solution of diffusion problems	75
4.1	Notation	75
4.2	Method of lines discretization	78
4.3	The Crank-Nicolson method	78
4.4	Local truncation errors	79
4.5	Stability and convergence for methods for the heat equation	80
4.6	Complexity considerations	81
4.7	Using the methods	83
5	von Neumann stability analysis: The heat equation	87
5.1	The (periodic) Fourier transform	87
5.2	The discretization of the Cauchy problem on the line	88
5.3	von Neumann analysis in multiple space dimensions	92
6	Introduction to wave propagation: The advection equation	93
6.1	Periodic boundary conditions	94
6.2	Stability of the Lax-Friedrichs method	96
A	Functions of matrices	97
B	A proof of some of Dahlquist's theorem	101
B.1	A proof in the scalar case	101
C	The discrete Fourier transform	107
C.1	Approximation properties of the DFT	108
Bibliography		111

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.

```
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.

```
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 .+ scalar

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
```

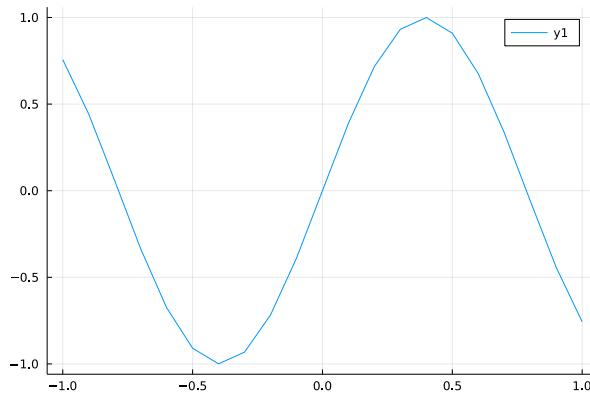
0.5 • Plotting

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.

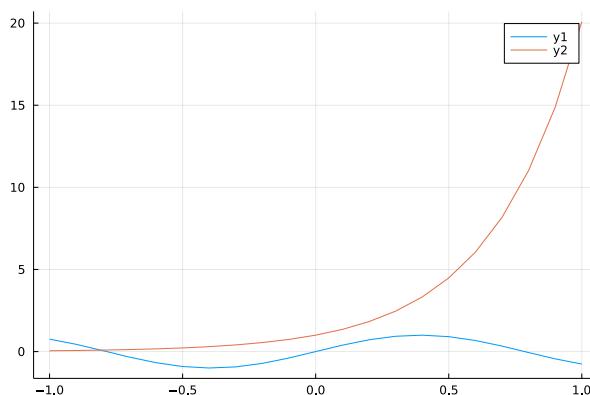
```
x = -1:0.1:1
f = x -> sin.(4x)
p = plot(x,f(x))
```



```
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

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

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 form¹

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

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

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

Define

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

Then we have

$$\begin{aligned} u'_1(t) &= u_2(t), \\ u'_2(t) &= u_3(t), \\ u'_3(t) &= v'''(t) = -v'(t)v(t) = -u_1(t)u_2(t). \end{aligned}$$

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

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 $g : \mathbb{R}^n \rightarrow \mathbb{R}$ 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 A below. We want to generalize functions

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

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.

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

Three important properties of the matrix exponential are

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

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

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

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

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

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

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

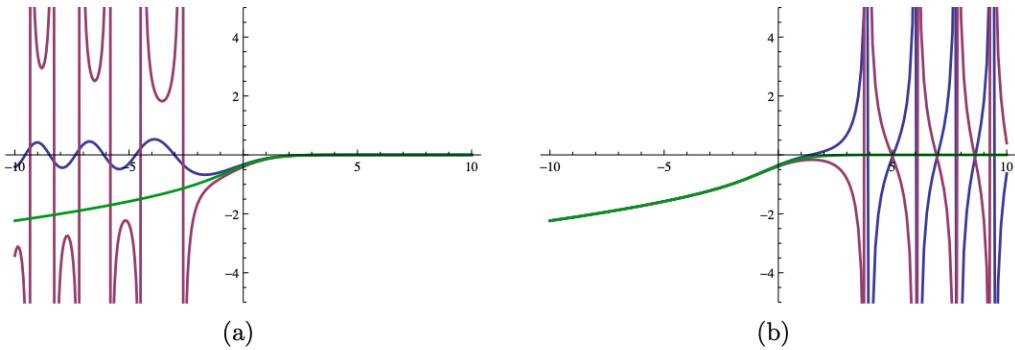


Figure 1.1: Solutions of the Painlevé II ODE with nearby initial conditions.

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 $f : \mathcal{D} \rightarrow \mathbb{R}^n$ 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-norm²:

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

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

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

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

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

²See Appendix in [LeV07] for more detail.

for

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

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:

$$\frac{du}{dt} = u^2 \Rightarrow \int \frac{du}{u^2} = \int dt.$$

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

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

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

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

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

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

So, we are guaranteed to have a solution for

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

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

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

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

$$\begin{cases} u'_1(t) = f(u_1(t), t), & u_1(0) \text{ given}, \\ u'_2(t) = f(u_2(t), t), & u_2(0) \text{ given}. \end{cases}$$

We now want to see how the difference evolves by computing

$$\begin{aligned} \frac{d}{dt} \|u_1(t) - u_2(t)\|_2^2 &= \frac{d}{dt} (u_1(t) - u_2(t))^T (u_1(t) - u_2(t)) \\ &= (u'_1(t) - u'_2(t))^T (u_1(t) - u_2(t)) + (u_1(t) - u_2(t))^T (u'_1(t) - u'_2(t)) \\ &= 2(u'_1(t) - u'_2(t))^T (u_1(t) - u_2(t)). \end{aligned}$$

Therefore

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

This is a differential inequality and typically, they are difficult to analyze. But this 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 $v(t) = e^{2Lt}$. Another simple observation is that $\frac{\|u_1(t) - u_2(t)\|_2^2}{v(t)} \geq 0$. Now differentiate this quantity

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

So, this is a decreasing function and we find that

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

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.

1.5 • Long-time existence

In this section, we consider two related questions:

- Under what conditions do we know that a solution exists for all $t > t_0$?
- How can solutions fail to exist at a finite time?

We answer both of these questions, in a restricted sense, in tandem. Suppose that $f(u)$ has continuous partial derivatives, i.e., $D_u f(u)$ is continuous for all $u \in \mathbb{R}^n$. Now suppose we know the following: For a bounded set $\mathcal{F} \subset \mathbb{R}^n$, if $u(t_0) = \eta \in \mathcal{F}$ then $u(t) \in \mathcal{F}$ for all times at which the solution exists. Suppose that³ $\mathcal{F} \subset B_{R-\delta}(0)$, for some $\delta > 0$. We know that, by continuity, $\|f(u)\| \leq C_1$ for all $u \in B_R(0)$, and $\|D_u f(u)\| \leq C_2$ for all $u \in B_R(0)$. So in applying Theorem 1.8, t_1 can be taken arbitrarily largest and $\gamma := a/S$ is independent of η . So, provided $\eta \in \mathcal{F}$, the solution exists until $t_0 + \gamma$. Repeating this argument, the solution exists until $t_0 + n\gamma$ for every n , i.e., for all time.

This, first, gives a sufficient condition for long-time existence. Secondly, it demonstrates that for smooth, globally-defined functions f , the solution has to head off to infinity, in some way, for the solution to fail to exist at some point in time.

³Here $B_r(x)$ denotes the ball of radius $r > 0$ centered at x .

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

$$t_0 < t_1 < \dots < t_n < \dots.$$

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

$$\begin{aligned} \frac{U^{n+1} - U^n}{k} &= f(U_n) \\ \boxed{U^{n+1}} &= U^n + kf(U^n). \end{aligned}$$

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

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

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

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 $g : \mathbb{R}^n \rightarrow \mathbb{R}^n$. 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

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

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

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

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 $g : \Omega \rightarrow \mathbb{R}^n$ 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

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

This gives

$$0 = w^T g(x^*) = w^T g(x_0) + w^T D_x g(x_0)(x^* - x_0) + \frac{w^T}{2} H_x g(\zeta)(x^* - x_0, x^* - x_0),$$

$$\zeta = \xi x^* + (1 - \xi)x_0,$$

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

$$w^T g(x_0) + w^T D_x g(x_0)(x^* - x_0) = w^T g(x_0) + w^T D_x g(x_0)(x_1 - x_0) + w^T D_x g(x_0)(x^* - x_1)$$

$$= w^T D_x g(x_0)(x^* - x_1).$$

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

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

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 Newton'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`.

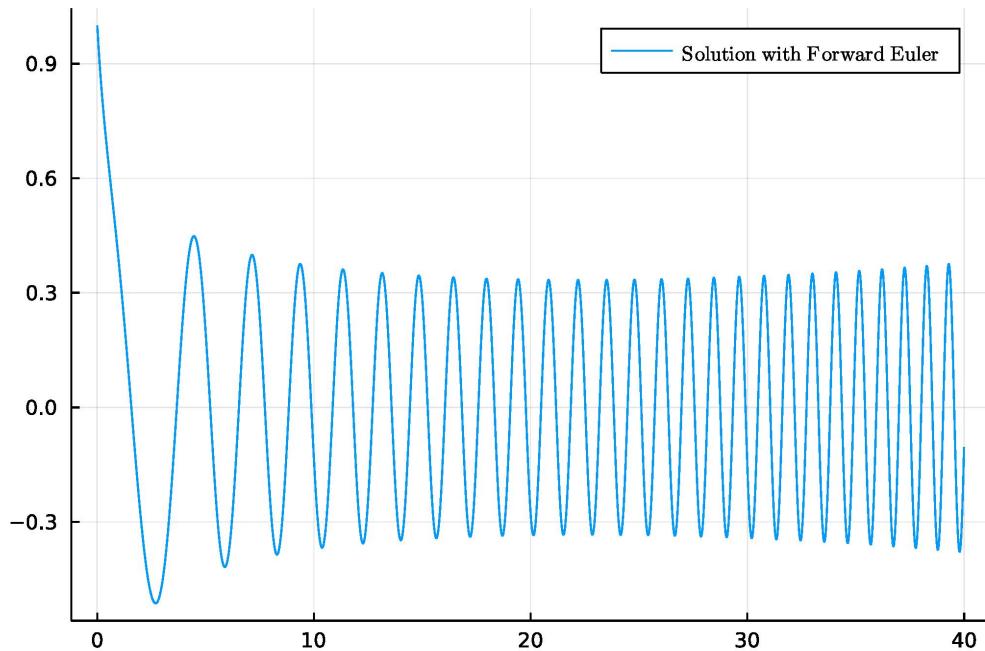
```
T = 40.# Final time.
k = 0.001 # Step size
```

The implementation of forward Euler is straightforward.

```
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 ]

```

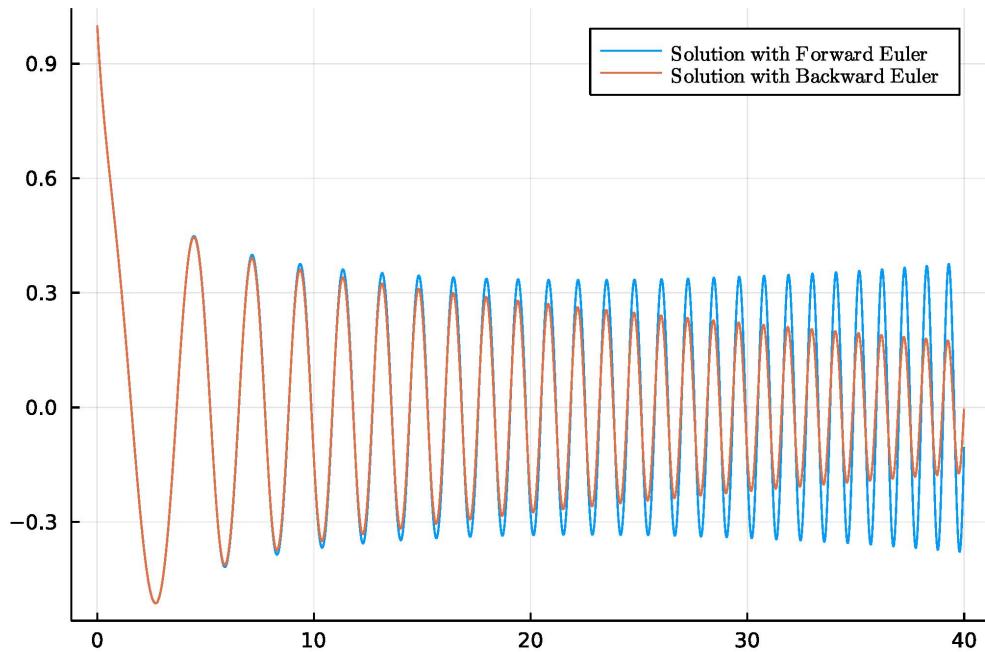
Then we simply run the iteration.

```

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.

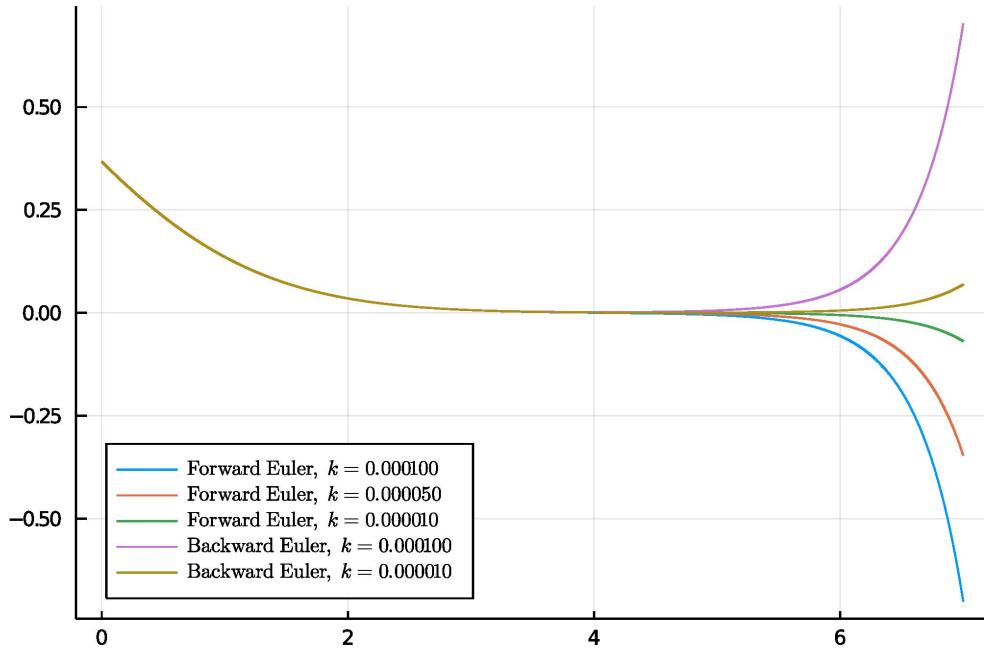


2.4 • The cautionary tale IVP

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?

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

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

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

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 $u'(t_n) = f(u(t_n))$

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

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

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

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

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

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

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

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 $\tau_{\text{Trap}}^n = \left(\frac{1}{6} - \frac{1}{4}\right)u'''(t)k^2 + O(k^3)$.

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*

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

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

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

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 $F : \underbrace{(\mathbb{R}^n \times \mathbb{R}^n \times \dots \times \mathbb{R}^n)}_{p+1 \text{ times}} \rightarrow \mathbb{R}^n$.

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

$$\tau_{\text{BDF}}^n = -\frac{2k^2}{6}u'''(t+k) + O(k^3).$$

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 $u(t)$:

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

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

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

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

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

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 $D_u f(u, t)$. 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

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

or

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

We might fear that because we have used 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 it is often convenient to also write out the non-autonomous versions,

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

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

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

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

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

with

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

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

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

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

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}
1	b_1	b_2	\cdots	b_r

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)

	0				
	$\frac{1}{2}$	$\frac{1}{2}$			
	$\frac{1}{2}$	0	$\frac{1}{2}$		
	$\frac{1}{2}$	0	0	1	
1					
1	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$	

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

c_1	a_{11}				
c_2	a_{21}	a_{22}			
:	:	:	..		
c_r	a_{r1}	a_{r2}	...	a_{rr}	
1	b_1	b_2	...	b_r	

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

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

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 $j < r - 1$,

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

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

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

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

We can do some heuristic parameter counting to conjecture the order of accuracy we might expect in these methods. For Adams-Basforth methods we need $\beta_r = 0$ which leaves us with $\beta_0, \dots, \beta_{r-1}$ — 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

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

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

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

is a solution where $\operatorname{cn}(x, k)$ is the Jacobi elliptic cosine function <https://dlmf.nist.gov/22> [OLBC10]. Some notations use $\operatorname{cn}(x, m)$ 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

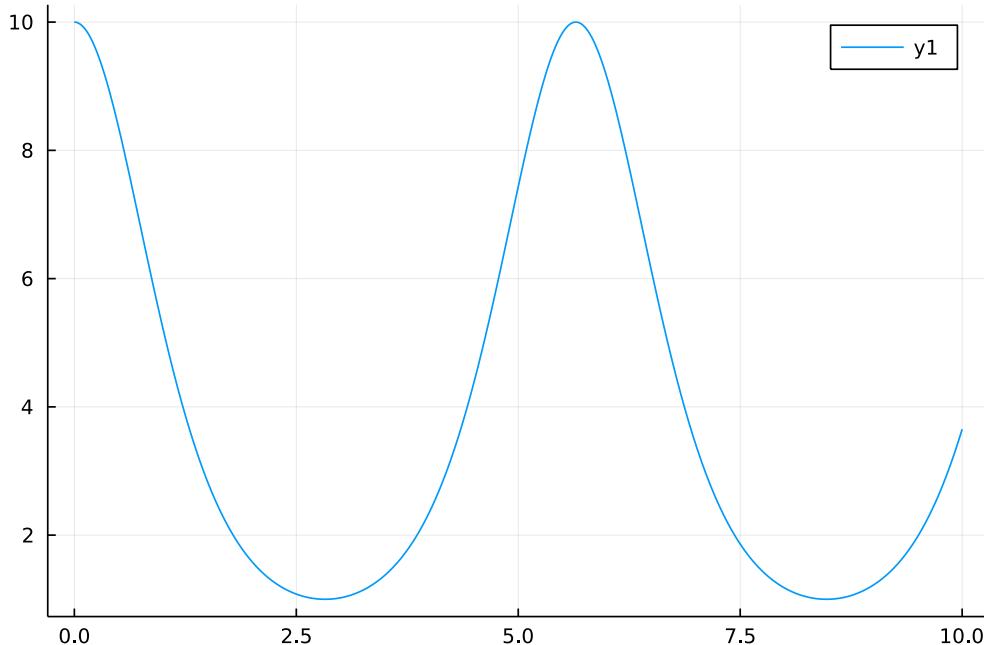
$$D_u f(u) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -u_2 & c - u_1 & 0 \end{bmatrix}.$$

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))
```

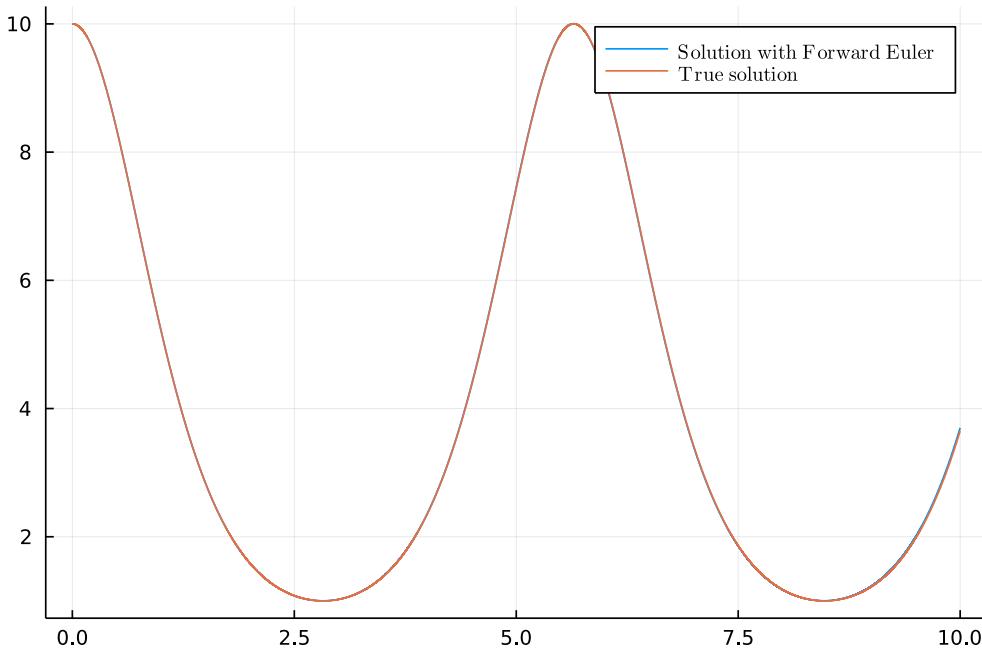


Now, we need to set up all the variables and functions so that we can apply various numerical methods.

```
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 = [\beta_3, 0., -1.0/6*(\beta_3-\beta_1)*(\beta_3-\beta_2)]
```

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

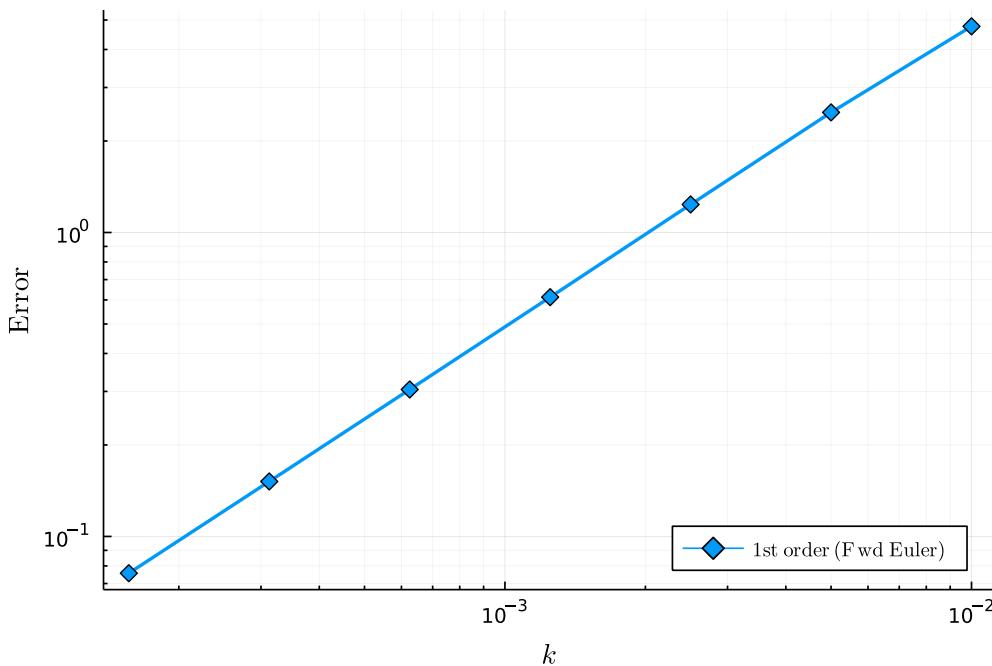
```
Number of time steps = 2000
Number of time steps = 4000
Number of time steps = 8000
Number of time steps = 16000
Number of time steps = 32000
Number of time steps = 64000
```

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\sim order\sim(Fwd\sim 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] = u_0
    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

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

So, we look for a zero of

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

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] = u_0
    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

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

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)

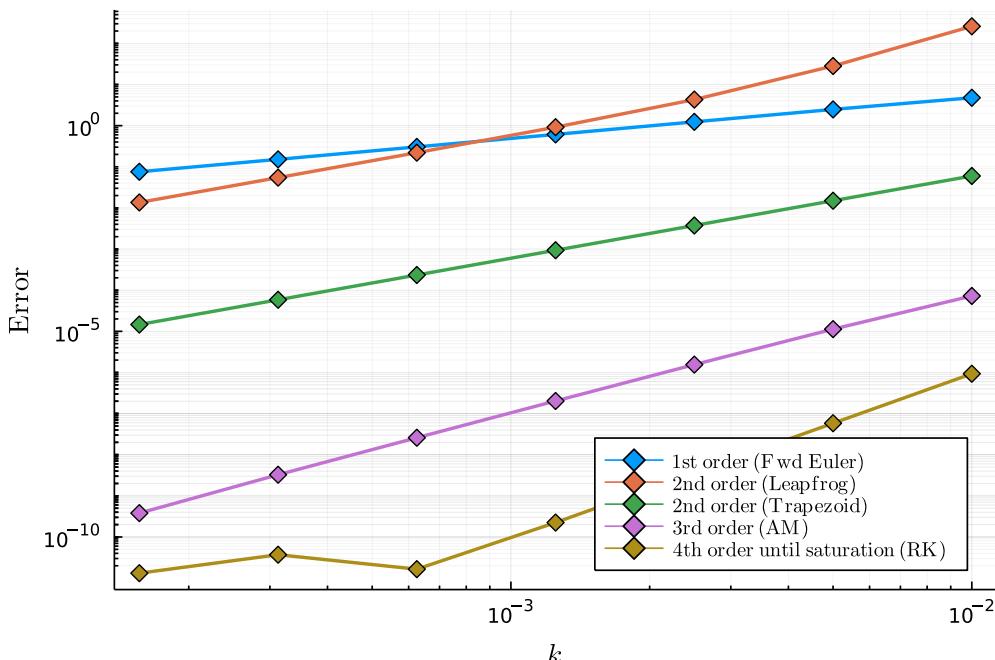
```

```

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] = u_0
    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 r th-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 r th 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 r th-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 r th 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 $\epsilon' = \frac{\epsilon T C}{E k}$. 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.

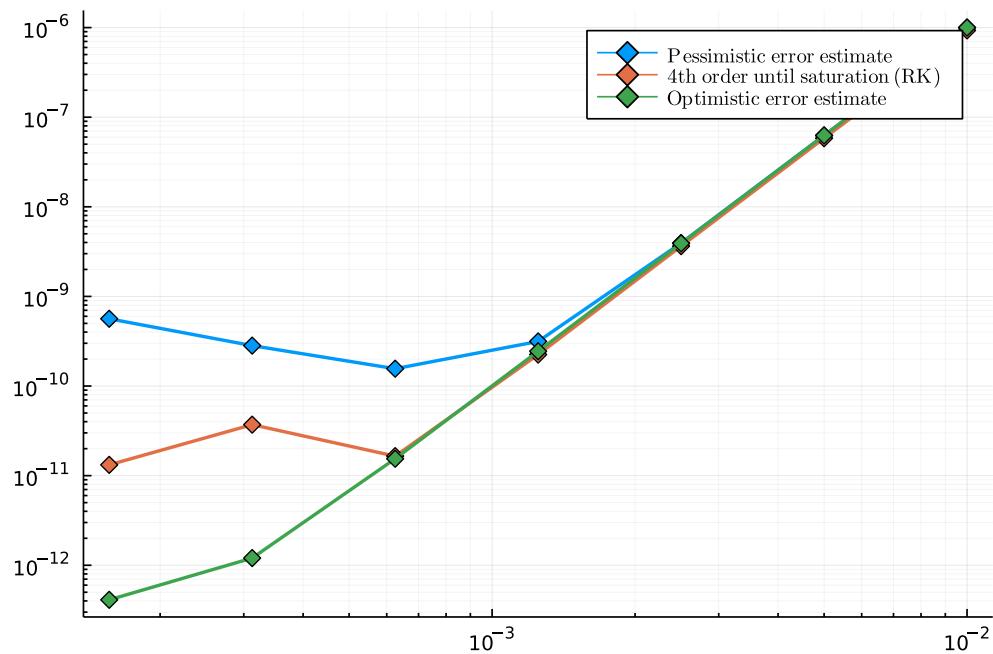
Order	Optimal step size	Optimal error
1	1.4832×10^{-8}	2.9666×10^{-8}
2	6.0368×10^{-6}	1.0933×10^{-10}
3	0.00012179	7.2257×10^{-12}
4	0.00073873	1.4890×10^{-12}
5	0.0024570	5.3724×10^{-13}

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 pessimistic and one is too optimistic.

Chapter 3

Consistency, stability and convergence

3.1 • Convergence of onestep methods

We begin this section with the notion of what it means for a numerical method to converge. Recall that the iterates of our methods U^0, U^1, \dots are actually functions of k , $U^0(k), U^1(k)$.

Definition 3.1. A sequence $U^0(k), U^1(k), U^2(k), \dots, U^n(k), \dots$ is said to converge to the solution of

$$\begin{cases} u'(t) = f(u(t)), \\ u(0) = \eta, \end{cases}$$

if for any $0 \leq T \leq \mathcal{T}$

$$\lim_{\substack{k \rightarrow 0 \\ Nk=T}} U^N(k) = u(T).$$

Using this we can define what it means for a method to converge.

Definition 3.2. An r -step method

$$U^{n+r} = F(f, k, U^n, U^{n+1}, \dots, U^{n+r}),$$

for

$$\begin{cases} u'(t) = f(u(t)), \\ u'(0) = \eta, \end{cases}$$

is said to converge if it produces a convergent sequence up to time $\mathcal{T} = \mathcal{T}_{f,\eta}$ for a wide class \mathcal{F} of functions f , and starting values $(U^0(k), U^1(k), \dots, U^{r-1}(k)) \in \mathcal{U}_{f,\eta}$.

So, the method converges only within a class. And typically, one will take

$$\begin{aligned}\mathcal{F} &= \{f : f \text{ is } u \text{ Lipschitz, continuous in } t\}, \\ \mathcal{T}_{f,\eta} &= \{t : u(t) \text{ exists and is unique}\}, \\ \mathcal{U}_{f,\eta} &= \{(U^0(k), U^1(k), \dots, U^{r-1}(k)) : \lim_{k \rightarrow 0} U^j(k) = \eta, \quad 0 \leq j \leq r-1\}.\end{aligned}$$

3.1.1 • Convergence of forward and backward Euler for $u' = \lambda u$

Before we discuss any generalities, we will apply the simplest methods to the simplest problem. Consider

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

In the text, you can find the following arguments applied to $u'(t) = \lambda u(t) + g(t)$. We write out the method and the LTE

$$U^{n+1} = U^n + kf(U^n) \Leftrightarrow u(t_{n+1}) = u(t_n) + kf(u(t_n)) + k\tau_{\text{FE}}^n.$$

Subtract, and define the error $E^n = U^n - u(t_n)$ which satisfies

$$\begin{aligned}E^{n+1} &= E^n + k[f(U^n) - f(u(t_n))] - k\tau^n \\ &= (1 + kn)E^n - k\tau^n\end{aligned}$$

Then we recall that

$$\tau_{\text{FE}}^n = \frac{k}{2}u''(t_n) + O(k^2).$$

We can go back and apply Taylor's theorem with the remainder to conclude that

$$\tau_{\text{FE}}^n = \frac{k}{2}u''(\xi_n),$$

for ξ_n between t_n and t_{n+1} . And this can then be bounded using $\|u''\|_\infty := \max_{0 \leq t \leq T} |u''(t)|$:

$$|\tau_{\text{FE}}^n| \leq \|\tau_{\text{FE}}^n\|_\infty \leq \frac{k}{2}\|u''\|_\infty.$$

So, we find an inequality that is satisfied by $|E^n|$:

$$|E^{n+1}| \leq |1 + k\lambda||E^n| + k\|\tau_{\text{FE}}^n\|_\infty.$$

A simplifying estimate is $|1 + k\lambda| \leq e^{k|\lambda|}$, giving

$$|E^{n+1}| \leq e^{k|\lambda|} |E^n| + k\|\tau_{\text{FE}}^n\|_\infty.$$

Let us now see how this bound is iterated:

$$\begin{aligned}|E^1| &\leq e^{k|\lambda|} |E^0| + k\|\tau_{\text{FE}}^0\|_\infty, \\ |E^2| &\leq e^{k|\lambda|} |E^1| + k\|\tau_{\text{FE}}^1\|_\infty \\ &\leq e^{2k|\lambda|} |E^0| + e^{k|\lambda|} k\|\tau_{\text{FE}}^0\|_\infty + k\|\tau_{\text{FE}}^1\|_\infty, \\ |E^3| &\leq e^{k|\lambda|} |E^2| + k\|\tau_{\text{FE}}^2\|_\infty \\ &\leq e^{3k|\lambda|} |E^0| + e^{2k|\lambda|} k\|\tau_{\text{FE}}^0\|_\infty + e^{k|\lambda|} k\|\tau_{\text{FE}}^1\|_\infty + k\|\tau_{\text{FE}}^2\|_\infty, \\ &\vdots \\ |E^n| &\leq e^{nk|\lambda|} |E^0| + \sum_{j=0}^{n-1} e^{jk|\lambda|} k\|\tau_{\text{FE}}^j\|_\infty.\end{aligned}$$

A simple estimate is to bound

$$\sum_{j=0}^{n-1} e^{jk|\lambda|} \leq n e^{nk|\lambda|},$$

which gives

$$|E^n| \leq e^{nk|\lambda|} [|E^0| + nk\|\tau_{\text{FE}}^n\|_\infty].$$

For $n = N$, $Nk = T$, this just gives

$$|E^N| \leq e^{T|\lambda|} [|E^0| + T\|\tau_{\text{FE}}^n\|_\infty].$$

And this establishes convergence, supposing that $|E^0| = 0$ (or at least that it tends to zero). With a judicious use of norms instead of the absolute value, and with one use of f being Lipschitz, this argument extends convergence of forward Euler to a large class of ODEs (Note: We use that u'' is bounded, and this requires more than just f being Lipschitz).

Now, we want to handle backward Euler. Recall

$$\tau_{\text{BE}}^n = -\frac{k}{2}u''(t_n) + O(k^2).$$

Again, we apply Taylor's theorem with the remainder to conclude that

$$\tau_{\text{BE}}^n = -\frac{k}{2}u''(\xi_n),$$

for ξ_n between t_n and t_{n+1} ,

$$|\tau_{\text{BE}}^n| \leq \|\tau_{\text{BE}}^n\|_\infty \leq \frac{k}{2}\|u''\|_\infty.$$

So, we find an inequality that is satisfied by $|E^n|$:

$$|E^{n+1}| \leq |1 - k\lambda|^{-1}|E^n| + k|1 - k\lambda|^{-1}\|\tau_{\text{BE}}^n\|_\infty.$$

This leads to

$$|E^n| \leq \frac{E^0}{|1 - k\lambda|^n} + k|1 - k\lambda|^{-1}\|\tau_{\text{BE}}^n\|_\infty \sum_{j=0}^{n-1} \frac{1}{|1 - k\lambda|^j}.$$

To analyze this, we need to consider

$$\sum_{j=0}^{n-1} \frac{1}{|1 - k\lambda|^j}.$$

If $|k\lambda| < 1$

$$\frac{1}{|1 - k\lambda|^2} = \frac{1}{(1 - k|\text{Re } \lambda|)^2 + (\text{Im } \lambda)^2} \leq \frac{1}{(1 - k|\text{Re } \lambda|)^2}.$$

So,

$$\sum_{j=0}^{n-1} \frac{1}{|1 - k\lambda|^j} \leq \frac{1 - (1 - k|\text{Re } \lambda|)^{-n}}{1 - (1 - k|\text{Re } \lambda|)^{-1}} = -(1 - k|\text{Re } \lambda|) \frac{1 - (1 - k|\text{Re } \lambda|)^{-n}}{k|\text{Re } \lambda|}.$$

To estimate this, consider the function

$$\frac{1 - (1-x)^{-n}}{x},$$

for $|x-1| < \epsilon$. By the mean-value theorem applied to $f(x) = (1-x)^{-n}$ we have

$$\frac{1 - (1-x)^{-n}}{x} = f'(\xi),$$

for ξ between 1 and x . Then

$$f'(x) = n(1-x)^{-n-1} \Rightarrow |f'(\xi)| \leq n(1-\epsilon)^{-n-1}.$$

Then, consider $k \rightarrow 0$, $Nk = T$

$$(1 - k \operatorname{Re} \lambda)^N = \left(1 - \frac{T \operatorname{Re} \lambda}{N}\right)^N = e^{-T \operatorname{Re} \lambda}(1 + o(1)).$$

This gives the estimate

$$|E^n| \lesssim e^{T|\operatorname{Re} \lambda|} [|E^0| + T \|\tau_{\text{FE}}^n\|_\infty].$$

Remark 3.3. *The proof of convergence for general explicit onestep methods can be deduced from the arguments in Appendix B.*

3.2 • Motivation of stability

When it comes to numerical analysis, one should always be skeptical of statements like

$$U^N \xrightarrow[Nk=T]{k \rightarrow 0} u(T).$$

Important factors that are left out in this statement are:

- What is the convergence rate?
- What are the effects of rounding errors?

As an example, consider solving the trivial ODE

$$\begin{cases} u'(t) = 0, \\ u(0) = 0, \end{cases}$$

with the 2-step method

$$U^{n+2} - 3U^{n+1} + 2U^n = -kf(U^n).$$

You should check that the LTE is $O(k)$. The iteration is then given by

$$\begin{cases} U^{n+2} = 3U^{n+1} - 2U^n, \\ U^0 = 0. \end{cases}$$

Now, suppose we failed to capture $U^1 = 0$, either by the effect of rounding errors or by some intrinsic errors in our starting method,

$$U^1 = \epsilon.$$

The recurrence can be solved explicitly, giving

$$U^n = -\epsilon + \epsilon 2^n, \quad n = 0, 1, 2, \dots$$

This diverges from $u(t) = 0$ rapidly. Suppose that $kN = T$, then

$$U^n = -\epsilon + \epsilon 2^{T/k}, \quad (3.1)$$

$$(3.2)$$

and we would need $\epsilon \ll 2^{-T/k}$ to realize convergence. You can prove a statement for this problem to the effect of ‘‘If U^1 converges sufficiently fast to U^0 as $k \rightarrow 0$, then the numerical method will converge.’’ Put the problem is that this convergence must happen at an exponential rate — not practical.

The issue here is that while the LTE is $O(k)$, this is a bad numerical method.

3.2.1 • Solving recurrences

Now we want to see how to derive (3.1). We consider a more general setting. Consider

$$\sum_{j=0}^r \alpha_j U^{n+j} = 0, \quad (3.3)$$

with U^0, U^1, \dots, U^{r-1} all specified. We make the ansatz $U^n = \zeta^n$ (here this is ζ to a power). Then

$$\sum_{j=0}^r \alpha_j \zeta^{n+j} = 0.$$

It suffices to consider

$$\rho(\zeta) := \sum_{j=0}^r \alpha_j \zeta^j = 0,$$

which is called the *characteristic polynomial* for the recurrence. Note that if r is large, $r \geq 5$, Galois theory asserts that a closed-form solution of this equation does not exist in general. The fundamental theorem of algebra asserts the existence of r roots (including multiplicities). First, suppose they $(\zeta_1, \dots, \zeta_r)$ are distinct. Set

$$U^n = \sum_{j=1}^r c_j \zeta_j^n.$$

To satisfy the initial conditions, choose c_1, \dots, c_r so that

$$\underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \\ \zeta_1 & \zeta_2 & \cdots & \zeta_r \\ \vdots & \vdots & & \vdots \\ \zeta_1^{r-1} & \zeta_2^{r-1} & \cdots & \zeta_r^{r-1} \end{bmatrix}}_Z \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_r \end{bmatrix} = \begin{bmatrix} U^0 \\ U^1 \\ \vdots \\ U^{r-1} \end{bmatrix}.$$

The matrix Z is called a Vandermonde matrix and

$$\det Z = \prod_{1 \leq i < j \leq r} (\zeta_j - \zeta_i).$$

We guarantee to be able to find (unique) values c_1, \dots, c_r . We reproduce (3.1).

Example 3.4. Consider

$$\begin{cases} U^{n+2} - 3U^{n+1} + 2U^n = 0, & n \geq 0, \\ U^0 = 0, \\ U^1 = \epsilon. \end{cases}$$

Set $U^n = \zeta^n$ and we have the characteristic polynomial

$$\rho(\zeta) = \zeta^2 - 3\zeta + 2 = 0.$$

So,

$$\zeta_{1,2} = \frac{3 \pm \sqrt{9-8}}{2} = 1, 2.$$

Then, in solving

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ \epsilon \end{bmatrix},$$

we find that $c_2 = -c_1 = \epsilon$. From this example, we see that $\zeta_2 = 2$ gives us the problematic behavior. So, we might want to suppose that $|\zeta_j| \leq 1$ for all j .

Example 3.5. Consider the recurrence

$$\begin{cases} U^{n+2} - 2U^{n+1} + U^n = 0, & n \geq 0, \\ U^0 = 0, \\ U^1 = \epsilon. \end{cases}$$

Here, the characteristic polynomial is

$$\rho(\zeta) = \zeta^2 - 2\zeta + 1 = (\zeta - 1)^2 = 0.$$

So, we only get one root, $\zeta_1 = \zeta_2 = 1$. We have to modify the solution formula

$$\begin{aligned} U^n &= c_1 \zeta_1^n + c_2 n \zeta_1^n, & \zeta_1 = 1, \\ &= c_1 + nc_2, \\ U^n &= \epsilon n. \end{aligned}$$

This growth is not as catastrophic as it was in the previous example, but it still is enough to prevent convergence. The following gives a recurrence that will work inside an LMM.

Example 3.6. Consider

$$U^{n+3} - 2U^{n+2} + \frac{5}{4}U^{n+1} - \frac{1}{4}U^n = 0.$$

The general solution is

$$U^n = c_1 + c_2 2^{-n} + c_3 n 2^{-n}.$$

The extra factor of n provides no growth — no amplification of errors.

3.3 • Zero-stability and convergence

We begin with a definition that is of use going forward.

Definition 3.7. The roots $(\zeta_j)_{j=1}^r$ of a polynomial $p(\zeta)$ of degree r are said to satisfy the root condition if

- $|\zeta_j| \leq 1$ for $j = 1, 2, \dots, r$, and
- $|\zeta_j| < 1$ if ζ_j is a repeated root.

The notion of *zero-stability* refers to the stability of a homogeneous recurrence of the form (3.3) near the zero solution.

Definition 3.8. An r -step LMM is said to be zero-stable if the roots of the characteristic polynomial satisfy the root condition.

Let us first verify that the class of methods we have defined thus far are indeed zero-stable:

- **Onestep methods:** $\rho(\zeta) = \zeta - 1$. Then $\zeta = 1$ is a non-repeated root.
- **Adams methods:** $\rho(\zeta) = \zeta^r - \zeta^{r-1}$. We see that $\zeta = 0$ is a (repeated) root and $\zeta = 1$ is a non-repeated root.
- **Nyström methods:** $\rho(\zeta) = \zeta^r - \zeta^{r-2}$. We see that $\zeta = 0$ is a (repeated) root and $\zeta = \pm 1$ are non-repeated roots.

Any Adams or Nyström method is zero-stable. The following gives convergence [Dah56].

Theorem 3.9 (Dahlquist's theorem). For LMMs applied to $u'(t) = f(u(t))$,

$$\text{Consistency} + \text{Zero-stability} \Leftrightarrow \text{Convergence}.$$

For a proof of sufficiency, also see Appendix B.

3.4 • Absolute stability

Going forward, we will ask, at a minimum, that a method should be zero-stable. But, as we will see, a more restrictive notion of stability is required to find methods that converge for PDEs, in a reasonable sense. It is also useful in the derivation of numerical methods for ODEs, but not required for convergence, as we have seen.

Definition 3.10. Fix $a > 0$. For a given K , a numerical method

$$U^{n+r} = F(f, k, U^n, \dots, U^{n+r}),$$

for the problem

$$\begin{cases} u'(t) = f(u(t)), \\ u(0) = \eta, \end{cases}$$

is absolutely stable if there exists $C = C(a, \eta, f, k) > 0$ such that

$$|U^n| \leq C,$$

whenever $U^0, U^1, \dots, U^{r-1} \in \{u : |\eta - u| \leq a\}$.

This is a notion that is very difficult to establish for anything beyond the simplest ODE. So, before we restrict ourselves to that case, we will consider a test problem.

3.4.1 • An absolute stability test problem

Let $h : [0, \infty) \rightarrow \mathbb{R}$ be a bounded, continuously differentiable function. Then consider the IVP

$$\begin{cases} u'(t) = \lambda(u(t) - h(t)) + h'(t), \\ u(0) = \eta. \end{cases}$$

We can use Duhamel's formula to solve this problem

$$u(t) = e^{\lambda t} \eta + \int_0^t e^{\lambda(t-\tau)} [h'(\tau) - \lambda h(\tau)].$$

We then recognize

$$h'(\tau) - \lambda h(\tau) = e^{\lambda \tau} \frac{d}{d\tau} [e^{-\lambda \tau} h(\tau)].$$

And from this, we have

$$u(t) = e^{\lambda t} [\eta - h(0)] + h(t).$$

So, we see that $u(t) - h(t)$ tends to zero if $\operatorname{Re} \lambda < 0$.

We use a variety of consistent and zero-stable methods on this problem. But, of course, the statement of convergence requires $k \rightarrow 0$. So, for finite k , the numerically computed solution can have "nothing" to do with the real solution, even if k is "small".

First, set up the problem.

```

h = t -> sin(t)^2
dh = t -> 2*sin(t)*cos(t)
f = (u,t) -> λ*(u-h(t))+dh(t)
Df = u -> λ

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

```

Note that we have a new Newton method that does not modify the input. One reason to do this here is that we will be using Newton's method on a scalar-valued function and

there is no reason to try to modify the input to save memory. Another reason is that the `Newton!` function defined previously will only work for vector-valued functions because of the reference to `x[1:end]` in the fourth line.

First, just run forward Euler.

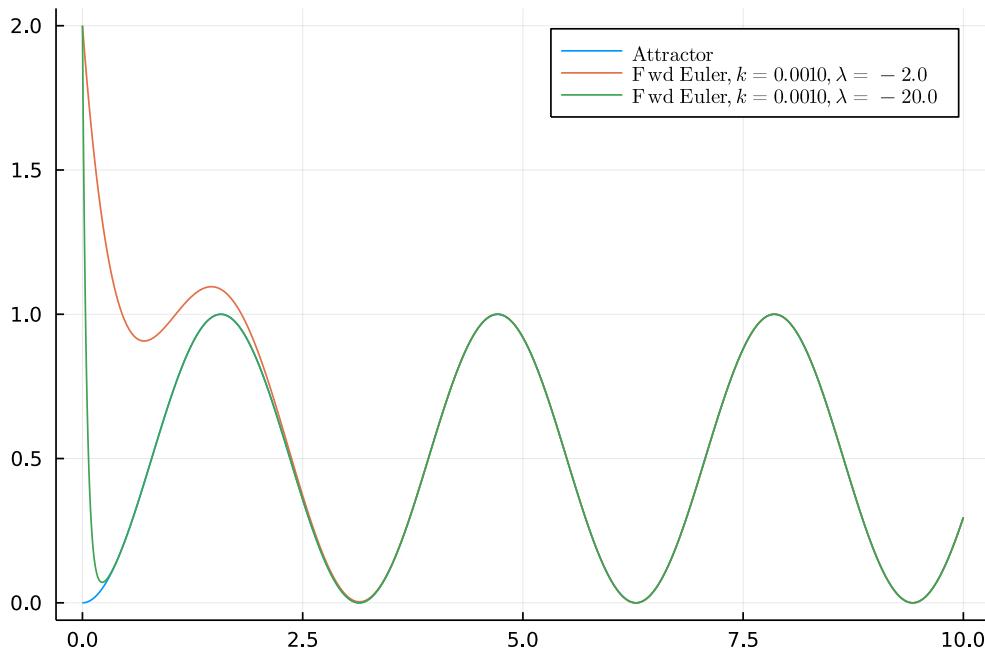
```

T = 10.# Final time.
t = 0:.01:T
p = plot(t,map(h,t),label=@sprintf("\mathrm{Attractor}") |> latexstring)

# Forward Euler
λ = -2;
k = 0.001 # Step size
u₀ = 2.
n = convert(Int64,ceil(T/k))# Number of time steps, converted to Int64
U = zeros(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-1])
    t[i] = t[i-1] + k
end
plot!(t,U,label=@sprintf("\mathrm{Fwd\sim Euler}, k = %0.4f,
    \lambda = %3.1f",k,λ) |> latexstring)

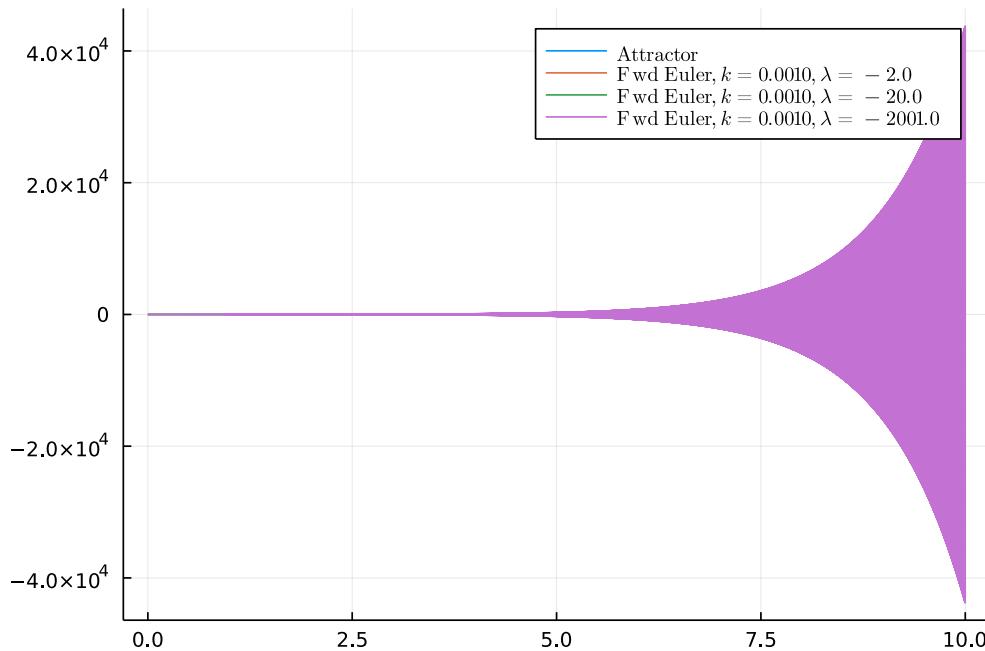
# Forward Euler again
λ = -20;
T = 10.# Final time.
k = 0.001 # Step size
u₀ = 2.
n = convert(Int64,ceil(T/k))# Number of time steps, converted to Int64
U = zeros(n+1) # To save the solution values
U[1] = u₀
t = zeros(n+1) # To save times
t[1] = 0.
for i = 2:n+1
    U[i] = U[i-1] + k*f(U[i-1],t[i-1])
    t[i] = t[i-1] + k
end
plot!(t,U,label=@sprintf("\mathrm{Fwd\sim Euler}, k = %0.4f,
    \lambda = %3.1f",k,λ) |> latexstring)

```



That seems fine, but let's increase $|\lambda|$ a little more.

```
# Forward Euler one more time
λ = -2001;
T = 10. # Final time.
k = 0.001 # Step size
u₀ = 2.
n = convert(Int64,ceil(T/k))# Number of time steps, converted to Int64
U = zeros(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-1])
    t[i] = t[i-1] + k
end
plot!(t,U,label=@sprintf("\mathrm{Fwd~Euler}, k = %0.4f,
    \lambda = %3.1f",k,λ) |> latexstring)
```

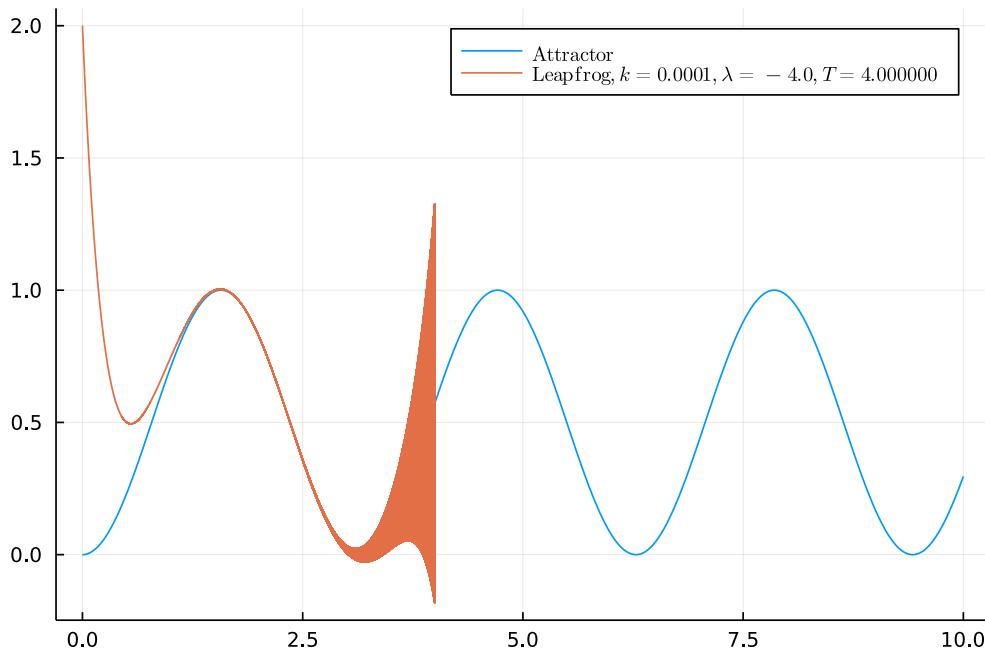


Now, let's try some second-order methods. First up is leapfrog.

```

λ = -4.;
T = 4.# Final time.
k = 0.0001 # Step size
u₀ = 2.
n = convert(Int64,ceil(T/k))# Number of time steps, converted to Int64
U = zeros(n+1) # To save the solution values
U[1] = u₀
t = zeros(n+1)
t[1] = 0.
U[2] = U[1] + k*f(U[1],t[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],t[i-1]) #Leapfrog
    t[i] = t[i-1] + k
end
plot!(t,U,label=@sprintf("\mathrm{Leapfrog}, k = %0.4f,
\lambda = %3.1f, T = %f",k,λ,T) |> latexstring)

```



It is clear that this method fails for a smaller value of $|\lambda|$ than even forward Euler! So, let's now try a second-order implicit method, trapezoid.

```

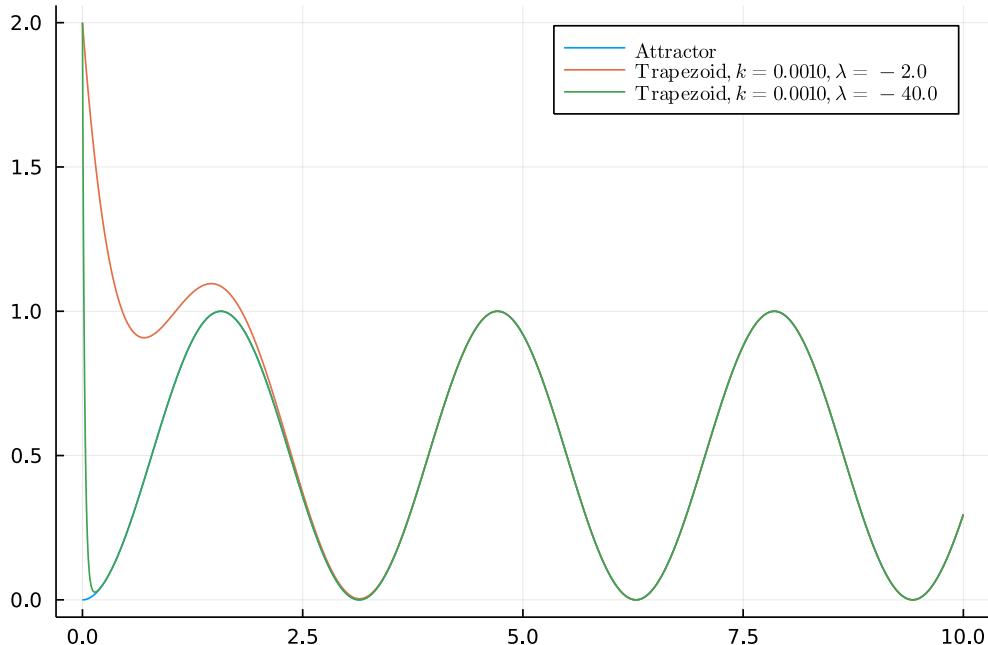
T = 10.# Final time.
t = 0:.01:T
p = plot(t,map(h,t),label=@sprintf("\mathrm{Attractor}") |> latexstring)

λ = -2.#
T = 10.# Final time.
k = 0.001 # Step size
u₀ = 2.
n = convert(Int64,ceil(T/k))
U = zeros(n+1) # To save the solution values
U[1] = u₀
t = zeros(n+1)
t[1] = 0.
max_iter = 10
for i = 2:n+1
    t[i] = t[i-1] + k
    U[i] = Newton(U[i-1],u -> g(u,U[i-1],t[i],t[i-1]), Dg; tol = k^3/10)
end
plot!(t,U,label=@sprintf("\mathrm{Trapezoid}, k = %0.4f,
    \lambda = %3.1f",k,λ) |> latexstring)

λ = -40.#
T = 10.# Final time.
k = 0.001 # Step size
u₀ = 2.
n = convert(Int64,ceil(T/k))
U = zeros(n+1) # To save the solution values
U[1] = u₀
t = zeros(n+1)
t[1] = 0.
max_iter = 10
for i = 2:n+1
    t[i] = t[i-1] + k
    U[i] = Newton(U[i-1],u -> g(u,U[i-1],t[i],t[i-1]), Dg; tol = k^3/10)
end
plot!(t,U,label=@sprintf("\mathrm{Trapezoid}, k = %0.4f,
    \lambda = %3.1f",k,λ) |> latexstring)

```

```
\lambda = %3.1f",k,lambda) |> latexstring)
```

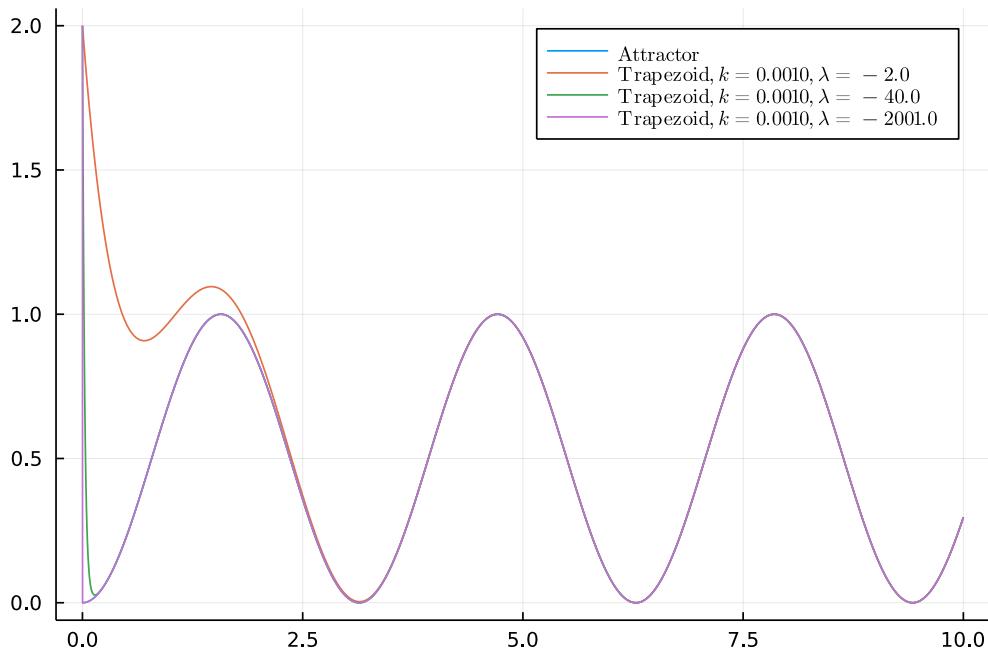


We can go to larger $|\lambda|$, where forward Euler failed.

```

λ = -2001.;
T = 10.# Final time.
k = 0.001 # Step size
u₀ = 2.
n = convert(Int64,ceil(T/k))
U = zeros(n+1) # To save the solution values
U[1] = u₀
t = zeros(n+1)
t[1] = 0.
max_iter = 10
for i = 2:n+1
    t[i] = t[i-1] + k
    U[i] = Newton(U[i-1],u -> g(u,U[i-1],t[i],t[i-1]), Dg; tol = k^3/10)
end
plot!(t,U,label=@sprintf("\mathrm{Trapezoid}, k = %0.4f",
    \lambda = %3.1f",k,lambda) |> latexstring)

```

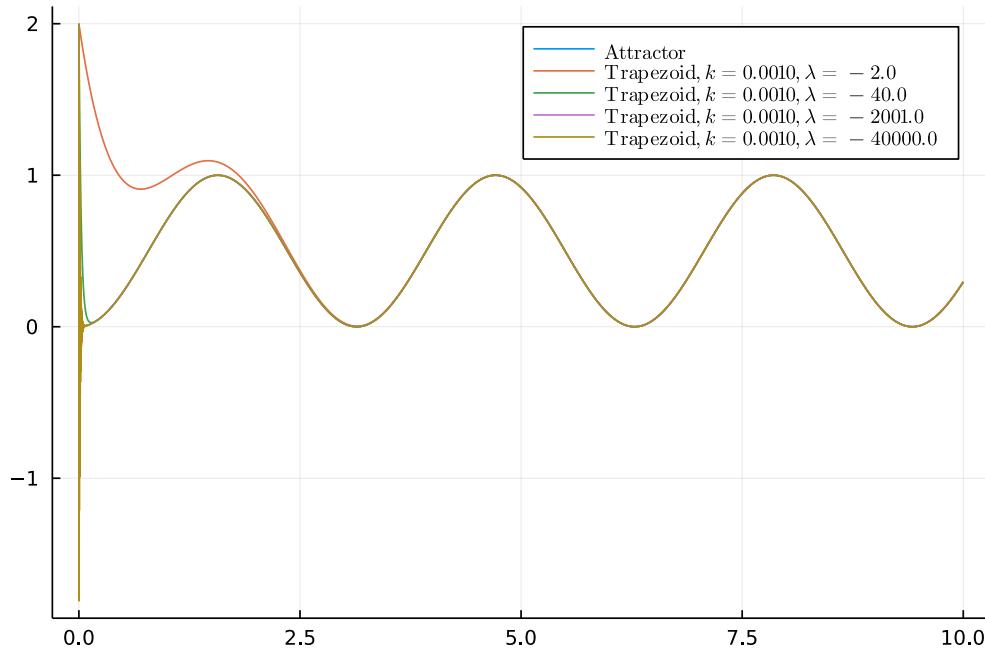


We can even go to extreme situations where the method fails to represent the solution initially, yet it does not blow up.

```

λ = -40000.;
T = 10. # Final time.
k = 0.001 # Step size
u₀ = 2.
n = convert(Int64,ceil(T/k))
U = zeros(n+1) # To save the solution values
U[1] = u₀
t = zeros(n+1)
t[1] = 0.
max_iter = 10
for i = 2:n+1
    t[i] = t[i-1] + k
    U[i] = Newton(U[i-1],u -> g(u,U[i-1],t[i],t[i-1]), Dg; tol = k^3/10)
end
plot!(t,U,label=@sprintf("\mathrm{Trapezoid}, k = %0.4f",
    "\lambda = %3.1f",k,λ) |> latexstring)

```



3.4.2 ■ Regions of absolute stability

We begin with the definition of the region of absolute stability.

Definition 3.11. *The region of absolute stability S for a method*

$$U^{n+r} = F(f, k, U^n, \dots, U^{n+r}),$$

are the values of $z := k\lambda \in \mathbb{C}$ such that the method applied to

$$\begin{cases} u'(t) = \lambda u(t), \\ u(0) = \eta \neq 0, \end{cases}$$

is absolutely stable if and only if $z \in S$.

Note that the ODE here is linear, so the choice η will play no role.

Example 3.12 (Forward Euler). Forward Euler applied to the test problem gives

$$U^{n+1} = U^n + k\lambda U^n = (1 + k\lambda)U^n = (1 + z)U^n.$$

Solutions are bounded for all $n > 0$ if and only if $|1 + z| \leq 1$. Write $z = \operatorname{Re} z + i \operatorname{Im} z$ and then

$$|1 + z|^2 = (1 + \operatorname{Re} z)^2 + (\operatorname{Im} z)^2 = 1$$

is a circle centered at $z = -1$ with radius 1.

3.5 • Plotting regions of absolute stability

For LMMs

To characterize the regions of absolute stability for LMMs, we need to consider the method applied to $u'(t) = \lambda u(t)$,

$$\sum_{j=0}^r \alpha_j U^{n+j} = k \sum_{j=0}^r \beta_j f(U^{n+j}) = \underbrace{\lambda k}_{z} \sum_{j=0}^r \beta_j U^{n+j}.$$

We arrive at the relation

$$\sum_{j=0}^r (\alpha_j - z\beta_j) U^{n+j} = 0.$$

Now, suppose $U^n = \zeta^n$, and we arrive at

$$\zeta^n \underbrace{\sum_{j=0}^r (\alpha_j - z\beta_j) \zeta^j}_{\rho(\zeta) - z\sigma(\zeta)},$$

Or,

$$0 = \rho(\zeta) - z\sigma(\zeta) = \pi(\zeta; z), \quad \rho(\zeta) = \sum_{j=0}^r \alpha_j \zeta^j, \quad \sigma(\zeta) = \sum_{j=0}^r \beta_j \zeta^j.$$

As we know, the general solution of the recurrence is given by a linear combination of roots of $\pi(\zeta; z)$,

$$U^n = \sum_{j=1}^r c_j \zeta_j^n.$$

We arrive at the following.

Theorem-Definition 3.13. *The region of absolute stability S for a LMM is the region in the complex z -plane where the roots of $\pi(\zeta; z)$ satisfy the root condition.*

Remark 3.14. *The LMM is zero stable if the roots of $\pi(\zeta; 0)$ satisfy the root condition, i.e., if $0 \in S$.*

To plot the region of stability, we make the observation that if $\pi(e^{i\theta}; z) = 0$, i.e., if $\zeta = e^{i\theta}$ is root of π and is of modulus 1 (a possible point on the boundary of S), then

$$z = \frac{\rho(e^{i\theta})}{\sigma(e^{i\theta})}, \quad \theta \in [0, 2\pi).$$

This gives the parameterization of the (possible) boundary of S .

We now provide some functionality to compute the region. We first define the functions using the coefficients

```

ρ = (α, z) -> (z.^ (length(α)-1:-1:0))' * α
σ = (β, z) -> (z.^ (length(β)-1:-1:0))' * β
R = (α, β, z) -> ρ(α, z) / σ(β, z)

```

Then we define some functions to first compute the roots of a polynomial, using eigenvalue computations, and then to check if the root condition is satisfied

```

function find_roots(c) # supposing that the leading order coefficient is 1
    # c contains the remaining coefficients
    r = length(c)
    A = zeros(Complex{Float64}, r, r)
    A[1, :] = -c
    A[2:end, 1:end-1] = A[2:end, 1:end-1] + I # add ones
    return eigvals(A)
end

function check_condition(λ)
    if maximum(abs.(λ)) > 1
        return 0
    else
        for i = 1:length(λ)
            if abs(λ[i]) ≈ 1. && sum(map(t -> λ[i] ≈ t, λ)) > 1
                return 0
            end
        end
    end
    return 1
end

```

We then assemble this to work with a LMM.

```

function compute_roots(α, β, z)
    r = length(α)-1
    c = α-z*β
    if α[1]-z*β[1] ≈ 0.
        λ = find_roots(c[3:end]/c[2])
    else
        λ = find_roots(c[2:end]/c[1]) # suppose 1st, 2nd coefficients not zero
    end
    return λ
end

function root_condition(α, β, z)
    return compute_roots(α, β, z) |> check_condition
end

```

Lastly, we build a function to plot the region of absolute stability and its boundary.

```

function convergence_stability(α, β)
    check_convergence(α, β)
    θ = 0:0.01:(1+rand()/10):2*π # random perturbation to avoid singularities
    z = map(t -> R(α, β, exp(1im*t)), θ);

    if abs(minimum(real(z)) - maximum(real(z))) < .1
        xrange = [-4., 4.]
    else
        xrange = [minimum(real(z))-1, maximum(real(z))+1]
    end

    if abs(minimum(imag(z)) - maximum(imag(z))) < .1
        yrange = [-4., 4.]
    else
        yrange = [minimum(imag(z))-1, maximum(imag(z))+1]
    end

```

```

if xrange[1] < -10
    xrange[1] = -4
end
if xrange[1] < -10
    xrange[1] = -4
end
if xrange[2] > 10
    xrange[2] = 4
end
if xrange[2] > 10
    xrange[2] = 4
end

contourf(xrange[1]:0.01:xrange[2],yrange[1]:0.01(1+rand()/10):yrange[2],
          (x,y)-> root_condition(alpha,beta,x+1im*y),colorbar=false)
plot!(real(z),imag(z),xlim=xrange,ylim=yrange,aspectratio=1,
      legend=false,lw=4,linecolor=:orange)
end

```

Example 3.15. Consider the 3rd-order Adams-Moulton method

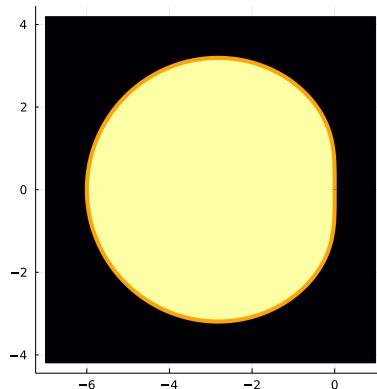
$$U^{n+2} = U^{n+1} + k \left[\frac{5}{12}U^{n+2} + \frac{8}{12}U^{n+1} - \frac{1}{12}U^n \right].$$

We can easily check the consistency of the method and its region of absolute stability.

```

# A-M 3rd order
alpha = [1, -1, 0]
beta = [5/12, 8/12, -1/12]
convergence_stability(alpha, beta)

```



For onestep methods

When applying any onestep (potentially multistage) method to $u'(t) = \lambda u(t)$, we find

$$U^{n+1} = R(z)U^n, \quad z = \lambda k,$$

where the rational function $R(z)$ is a polynomial if the method is explicit.

Theorem-Definition 3.16. *The region of absolute stability S for a onestep method is*

$$S = \{z \in \mathbb{C} : |R(z)| \leq 1\}.$$

3.6 • Heuristic implications of regions of absolute stability

Consider solving the linear system of ODEs

$$\begin{cases} u'(t) = Au(t) + b(t), \\ u(0) = \eta \in \mathbb{R}^n. \end{cases}$$

Suppose A has eigenvalues $\lambda_1, \dots, \lambda_n$ and set

$$z_j = k\lambda_j.$$

If we suppose that $z_j \in S$, where S is the region of absolute stability for a method we are applying to the system, we should expect the method to be absolutely stable on $u'(t) = Au(t) + b(t)$.

Now, consider solving

$$\begin{cases} u'(t) = f(u(t)), \\ u(0) = \eta \in \mathbb{R}^n. \end{cases}$$

Supposing that f is differentiable, we can write

$$f(u) = f(u^*) + D_u f(u^*)(u - u^*) + o(\|u - u^*\|).$$

We now examine what things look like for t small:

$$u'(t) = \underbrace{f(\eta) - D_u f(\eta)\eta}_b + \underbrace{D_u f(\eta)}_A u(t) + o(\|u(t) - \eta\|).$$

So, if the eigenvalues of our Jacobian $D_u f(u(t))$, $\lambda_1(t), \dots, \lambda_n(t)$, satisfy

$$k\lambda_j(t) \in S,$$

for a given method, we might conjecture that a method is absolutely stable on this problem for this initial condition. This is a good guess, but the fact is that the numerical method will not encounter $D_u f(u(t_n))$ but rather $D_u f(U^n)$, which may differ from the former significantly, especially after many iterations. Furthermore, even if the eigenvalues are all within the stability region, the eigenvectors can conspire to cause instability.

3.6.1 • Examining the eigenvalues of the Jacobian for a test problem

Recall the test problem

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

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

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

is a solution where $\operatorname{cn}(x, k)$ is the Jacobi cosine function and k is the elliptic modulus. Some notations use $\operatorname{cn}(x, m)$ where $m = k^2$. The corresponding initial conditions are

$$v(0) = \beta_3, \quad v'(0) = 0, \quad v''(0) = -\frac{(\beta_3 - \beta_1)(\beta_3 - \beta_2)}{6}.$$

Let us write the equation as a system and compute the Jacobian. For $\beta_1 = 0$, $\beta_2 = 1$, and $\beta_3 = 10$, based on an analysis of the Jacobian, we will see if we can suggest methods to solve the problem. As before, $u_1 = v(t)$, $u_2 = v'(t)$, and $u_3 = v''(t)$, we can express the nonlinear system as $u' = f(u)$ where

$$f(u) = \begin{bmatrix} u_2 \\ u_3 \\ \frac{\beta_1 + \beta_2 + \beta_3}{3} u_2 - u_1 u_2 \end{bmatrix}.$$

The Jacobian is then given by

$$D_u f(u) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -u_2 & \frac{\beta_1 + \beta_2 + \beta_3}{3} - u_1 & 0 \end{bmatrix}.$$

The range of possible eigenvalues for different values of t are now plotted. Observe that the plots turn out better, in this case, if the eigenvalues at each given value of t are sorted according to their imaginary parts. The `sort` command does not initially apply to complex numbers, but if we overload the `isless` function, it will extend.

```
import Base.isless
function isless(a::ComplexF64,b::ComplexF64)
    return imag(a) < imag(b)
end
```

We choose some parameters and plot the eigenvalues parameterized by t .

```

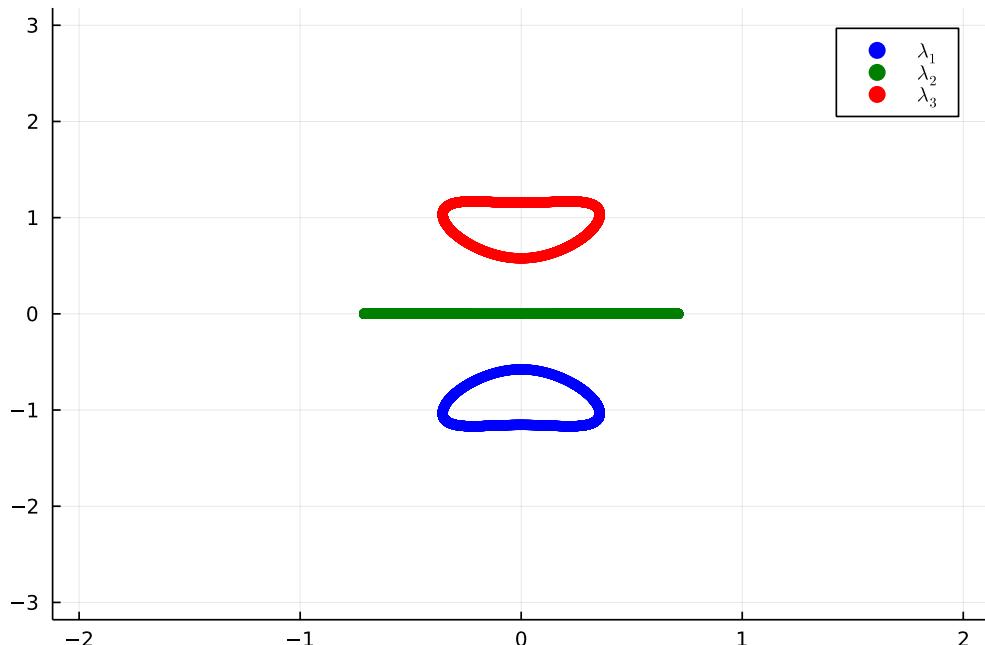
β₁ = -1.
β₂ = 1.
β₃ = 2.
c = (β₁ + β₂ + β₃)/3
vcn = t -> cn(sqrt((β₃-β₁)/12)*t, (β₃-β₂)/(β₃-β₁) )
vsn = t -> sn(sqrt((β₃-β₁)/12)*t, (β₃-β₂)/(β₃-β₁) )
vdn = t -> dn(sqrt((β₃-β₁)/12)*t, (β₃-β₂)/(β₃-β₁) )

function Jacobian(t)
    v = β₂ + (β₃ - β₂)*vcn(t)^2
    dv = -2*(β₃ - β₂)*vcn(t)*vsn(t)*vdn(t)
    return [0.0 1.0 0.0; 0.0 0.0 1.0; -dv c-v 0.0]
end

t = 0:.001:40;
λs = map( t -> eigvals(Jacobian(t)), t )
λs = [ sort(i) for i in λs]
λ₁ = [i[1] for i in λs][1:10:end]
λ₂ = [i[2] for i in λs][1:10:end]
λ₃ = [i[3] for i in λs][1:10:end];

```

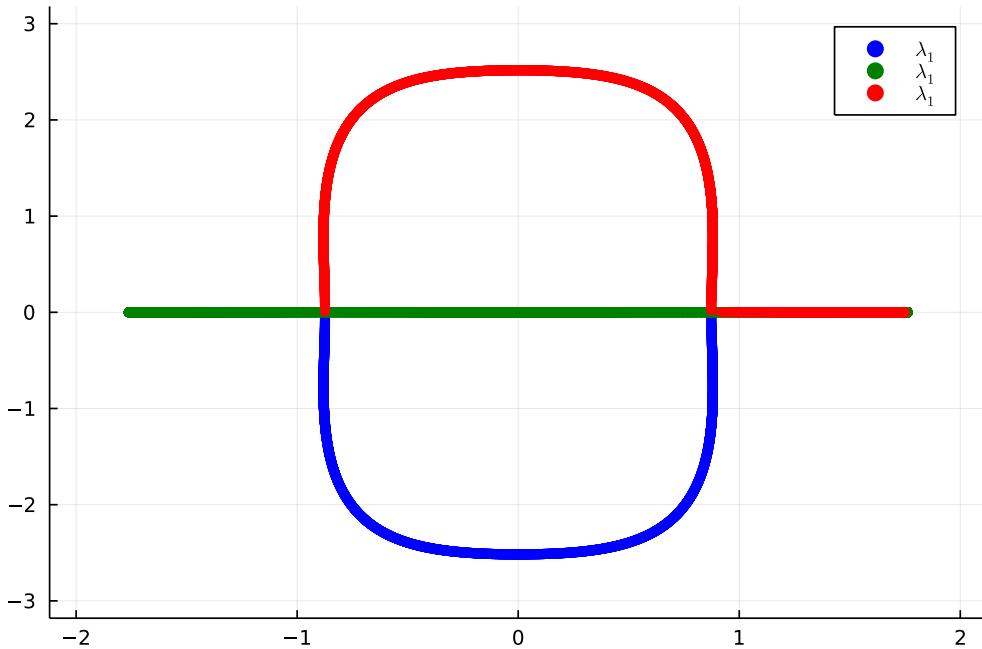
```
scatter!(λ1 |> real, λ1 |> imag, markercolor = :blue, markerstrokewidth=0,
yaxis = [-3,3], xaxis = [-2,2], label = L"\lambda_1");
scatter!(λ2 |> real, λ2 |> imag, markercolor = :green, markerstrokewidth=0,
yaxis = [-3,3], xaxis = [-2,2], label = L"\lambda_2");
scatter!(λ3 |> real, λ3 |> imag, markercolor = :red, markerstrokewidth=0,
yaxis = [-3,3], xaxis = [-2,2], label = L"\lambda_3")
```



Due to the location of the eigenvalues it is not possible (see HW3!) to make it so that all the eigenvalues lie within the stability region for any method. So, it is about maximizing. With something like trapezoid, the eigenvalues are in the stability region half of the time. With backward Euler, this is true of λ_2 , but the others are within the stability region more than half of the time. Something like leapfrog would be a bad choice here.

We try some other parameters.

```
β₁ = 0.
β₂ = 1.
β₃ = 10.
```



The method one should choose for this problem is maybe less clear. But similar considerations as those given for the previous choice of β_i 's still are good justification. But for both problems, Runge–Kutta 4 is actually probably the best choice – onestep, accurate, and for sufficiently small step size the eigenvalues will be within the stability region approximately half the time.

3.7 • Relative stability regions

Sometimes the notion of absolute stability is insufficient. We may be interested in growing solutions and we want our numerical method to grow with the solution but not over or underestimate the growth. Or similarly, we could have decay and we want to sufficiently approximate the rate of decay. The notion of relative stability helps with this.

Definition 3.17. *The region of relative stability S_{rel} for a onestep method is given by*

$$S_{\text{rel}} = \left\{ z \in \mathbb{C} : \left| \frac{R(z)}{e^z} \right| \leq 1 \right\},$$

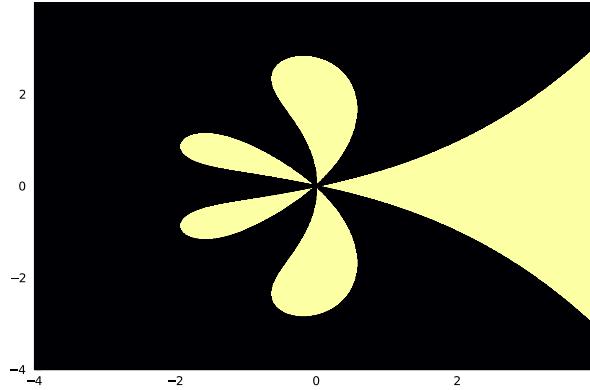
where $R(z)$ is the rational function such that

$$U^{n+1} = R(z)U^n, \quad z = \lambda k,$$

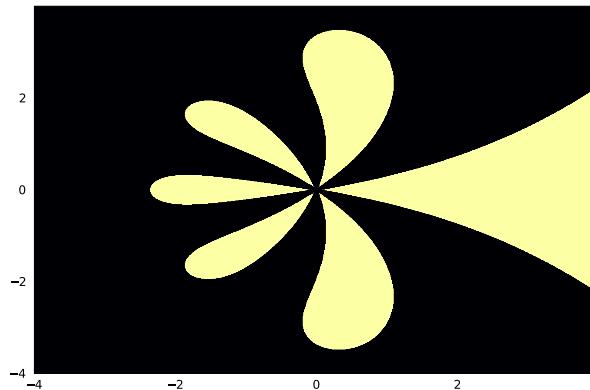
when the method is applied to $u'(t) = \lambda u(t)$.

This may be a more accurate reflection of which methods are successful on a given problem, but it is much harder to use in practice.

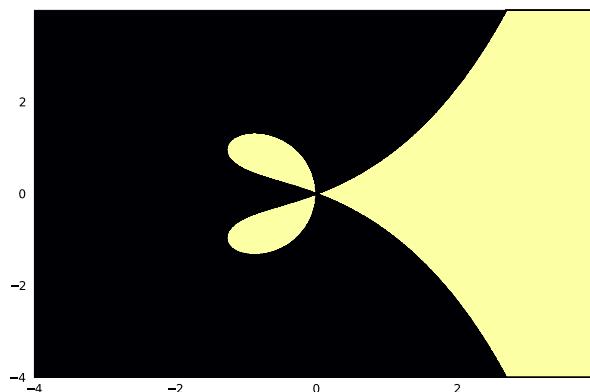
```
## For fourth-order explicit Runge--Kutta
R = (z) -> 1 + z + z^2/2 + z^3/6 + z^4/24
xrange = [-4,4]; yrange = [-4,4]
contourf(xrange[1]:0.01(1+rand()/10):xrange[2],yrange[1]:0.01(1+rand()/10):yrange[2],(x,y)-> sign(-(
```



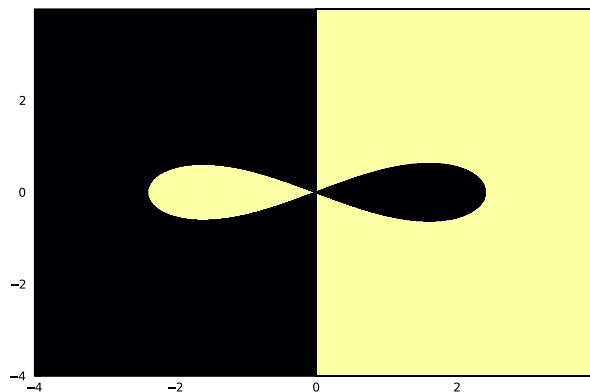
```
## For a fifth-order Taylor series method
R = (z) -> 1 + z + z^2/2 + z^3/6 + z^4/24 + z^5/(24*5)
```



```
## For a second-order Taylor series method
R = (z) -> 1 + z + z^2/2
```



```
## For an implicit method
R = (z) -> (1+z/2) / (1-z/2)
```



3.8 • Systems of stiff ordinary differential equations

We have to criteria that determine our step size:

- Accuracy goal: With a fourth order method, we could choose $k = 10^{-4}$ if we want to attempt an error on the order of 10^{-16} .
- Stability: We want to shrink k so that $z = k\lambda$ is within the region of absolute stability S .

The maximum value of k for our accuracy goal is k_{acc} and the maximum value for stability is k_{stab} . In practice, it is important to choose a method that allows

$$k_{\text{acc}} < k_{\text{stab}}. \quad (3.4)$$

Because if, one has the opposite inequality it indicates that there is computation being wasted, and potentially, a better method could alleviate that restriction. Stiff systems of differential equations are an example of systems of ODEs for which one does not have (3.4), unless a good method is chosen.

3.8.1 ▪ An example of a stiff system of ODEs

Consider the Lotka-Volterra system

$$\begin{cases} u'_1(t) = u_1(t) - u_1(t)u_2(t), \\ u'_2(t) = u_1(t)u_2(t) - u_2(t). \end{cases}$$

For $u_1(0) = 5, u_2(0) = 0.8$. We add a third variable to this model that gives some interesting dynamics:

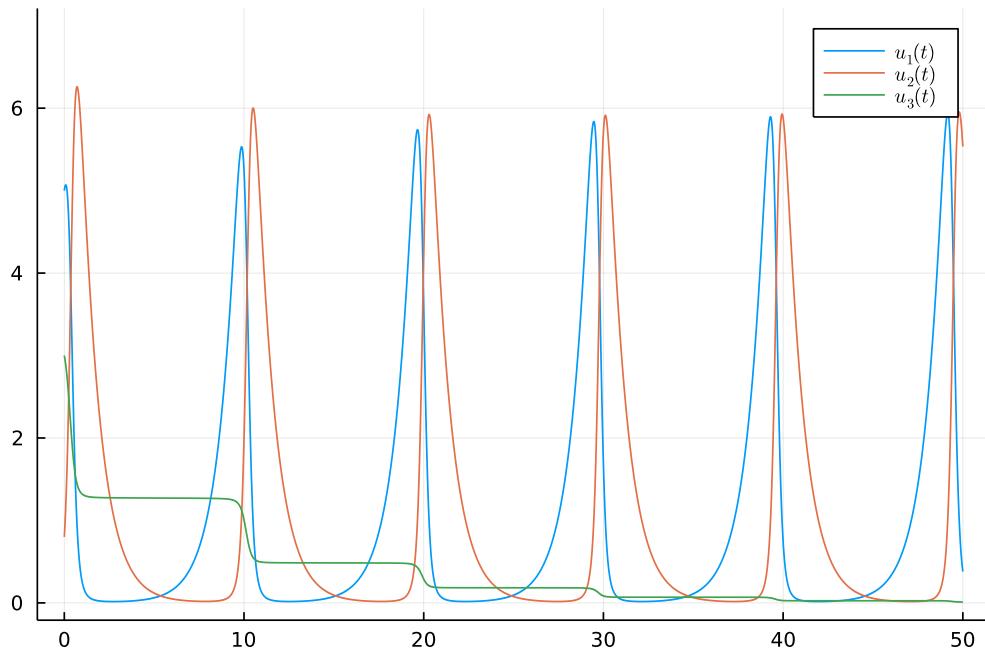
$$\begin{cases} u'_1(t) = u_1(t) - u_1(t)u_2(t) + \kappa_1 u_1(t)u_2(t)u_3(t), \\ u'_2(t) = u_1(t)u_2(t) - u_2(t) + \kappa_2 u_1(t)u_2(t)u_3(t), \\ u'_3(t) = -\kappa_3 u_1(t)u_2(t)u_3(t). \end{cases}$$

We then suppose that $\kappa_1 + \kappa_2 = \kappa_3$. This forces the third variable $u_3(t)$ to be severely damped with $u_1(t)u_2(t)$ is large. For simplicity, we then choose $\kappa_1 = \kappa/2 = \kappa_2, \kappa_3 = \kappa$. We first run forward Euler on this problem with small κ .

```
## Small κ
κ = .1;
f = u -> [u[1] - u[1]*u[2] + (κ/2)*u[1]*u[2]*u[3], u[1]*u[2] - u[2] + (κ/2)*u[1]*u[2]*u[3], -κ*u[1]*u[2]*u[3]]
```

```
# Forward Euler
κ = 0.001
n = convert(Int64, 50/κ)
U = zeros(3, n+1)
t = zeros(n+1)
U[:, 1] = [5., 0.8, 3.]
t[1] = 0.
for i = 2:n+1
    U[:, i] = U[:, i-1] + κ*f(U[:, i-1])
    t[i] = t[i-1] + κ
end
println("Maximum value of u_1(t): ", maximum(U[1, :]))
```

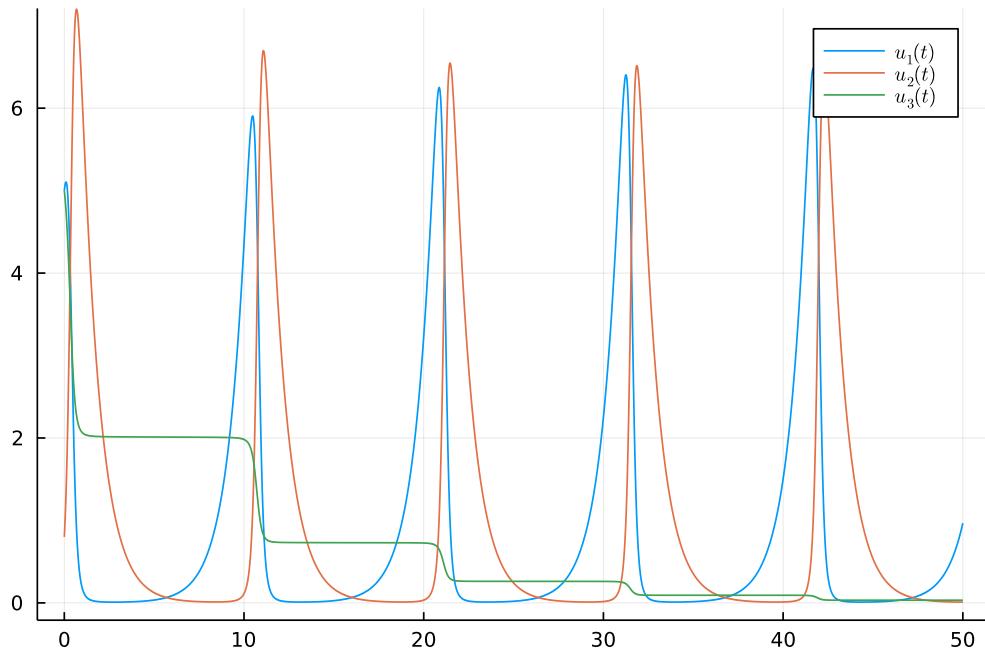
Maximum value of u_1(t): 5.935784479580017



We might think that because $u_3(t)$ is damped, the approximation of it is irrelevant in the long term. But if we start with a different initial condition for $u_3(0)$, we get a different limit cycle:

```
# Forward Euler
k = 0.001
n = convert(Int64, 50/k)
U = zeros(3,n+1)
t = zeros(n+1)
U[:,1] = [5., 8, 5.] # 5 instead of 3
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
println("Maximum value of u_1(t): ", maximum(U[1,:]))
```

Maximum value of $u_1(t)$: 6.481976371728525



We see that the amplitude of the oscillations has increased. So, if we accumulate error in our approximation of $u_3(t)$, it will necessarily effect the quality of the solution for all time. This is different than for the equation

$$u'(t) = \lambda(u(t) - h(t)) + h'(t),$$

where initial errors will be damped out overtime and the limiting solution $u(t) = h(t)$ does not depend on the choice of initial condition.

Next, we increase κ .

```
 $\kappa = 200.$ ; # Increase  $\kappa$ 
f = u -> [u[1] - u[1]*u[2] + ( $\kappa/2$ )*u[1]*u[2]*u[3], u[1]*u[2] - u[2] + ( $\kappa/2$ )*u[1]*u[2]*u[3], - $\kappa$ *u[1]*u[3]]
```

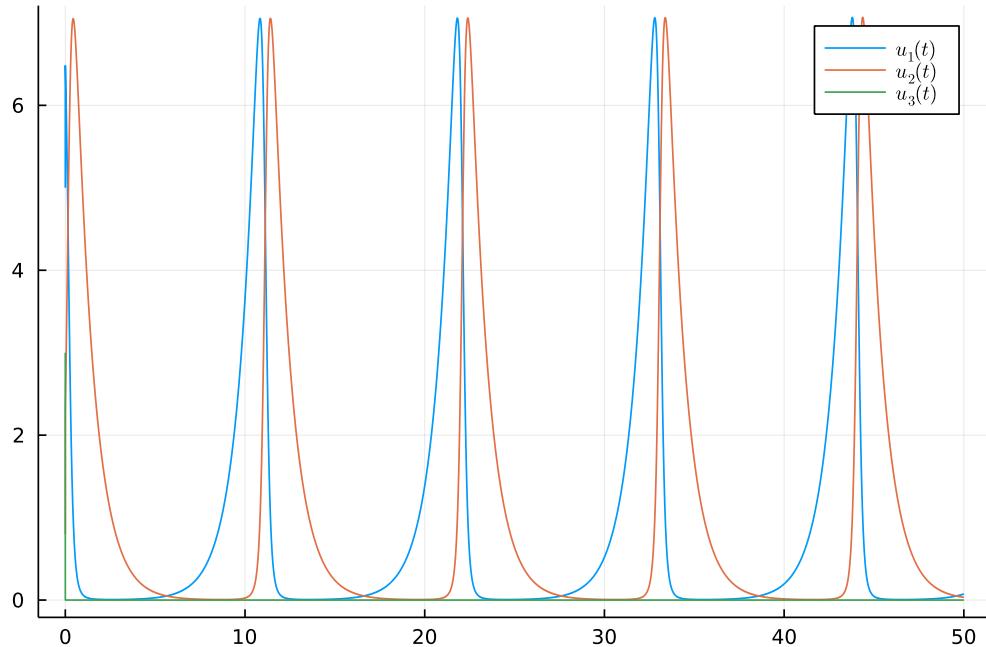
```
# Forward Euler
k = 0.001
n = convert(Int64, 50/k)
U = zeros(3, n+1)
t = zeros(n+1)
U[:, 1] = [5., 8, 3.]
t[1] = 0.
for i = 2:n+1
    U[:, i] = U[:, i-1] + k*f(U[:, i-1])
    t[i] = t[i-1] + k
end
println("Maximum value of u_1(t): ", maximum(U[1, :]))
```

Maximum value of $u_1(t)$: NaN

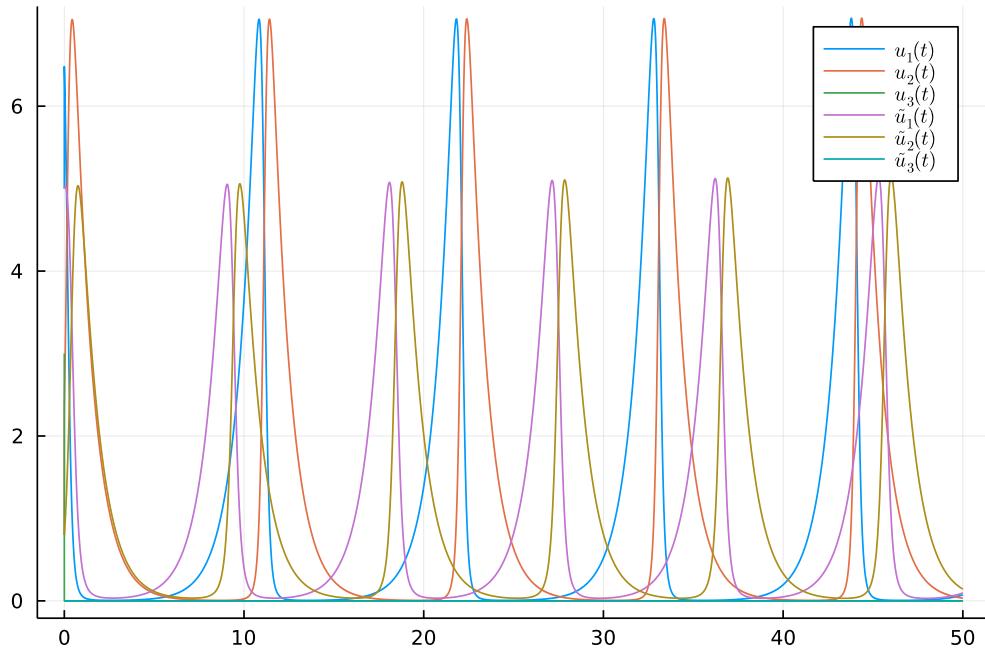
And forward Euler becomes unstable.

```
# Forward Euler with smaller step size
k = 0.0001
n = convert(Int64, 50/k)
U = zeros(3,n+1)
t = zeros(n+1)
U[:,1] = [5., .8, 3.]
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
println("Maximum value of u_1(t) : ", maximum(U[1,:]))
```

Maximum value of $u_1(t)$: 7.066414513377226



Compare this to the solution where $u_3(t) = 0$, which we call $\tilde{u}_1, \tilde{u}_2, \tilde{u}_3$.



So, capturing the initial transient is extremely important in determining the behavior of the solution. But we have to shrink the time step just to capture this transient. This is stiffness — severe differences in timescales.

3.8.2 • On a definition of stiffness

Considering only the numerical solution of $u'(t) = \lambda u(t)$ can hide some difficulties that occur systems. One can see that increasing $|\lambda|$ is the same as scaling the time variable. So, for larger values of λ , we expect to need to decrease the time step to obtain quality solutions. But consider a system

$$u'(t) = Au(t),$$

where A has eigenvalues of vastly different sizes. This indicates both fast and slow time scales. And to truly converge, we need to choose our timestep according to the fastest timescale.

But, what if the fast timescale is that of exponential decay? Then capturing it, while important, maybe is not as important as other situations. And it would be unfortunate if rapid dynamics that damp out of the problem almost immediately would force our timestep to be very small for all time.

Definition 3.18. *The stiffness ratio for $f(u)$ is given by*

$$\frac{\max_j |\lambda_j|}{\min_j |\lambda_j|},$$

where the maximum is taken over all eigenvalues λ_j of $D_u f(u)$.

This may lead us to think that scalar problems cannot be stiff but recall that a non-autonomous scalar ODE is actually a system!

There is no clear definition of what it means for a problem to be stiff, but often you know it when you see it. One more restrictive notion is $u'(t) = f(u(t))$ is stiff if

$D_u f(u)$ has eigenvalues with vastly different real parts.

We distinguish between real and imaginary here because if the imaginary part of an eigenvalue is large, one expects rapid oscillations and a small stepsize k will be required to resolve these oscillations anyway. But then again, there are settings where the solution of an ODE has small amplitude, very rapid oscillations and maybe they can be roughly approximated without destroying the overall quality of the solution. Here one might want to think the system is stiff if the stiffness ratio is large.

3.9 • L-stability and $A(\alpha)$ -stability

Methods that are particularly well suited to stiff problems are those that are called A -stable.

Definition 3.19. An ODE method is said to be A -stable if its region of stability S contains the entire left half-plane $\{z \in \mathbb{C} : \operatorname{Re} z \leq 0\}$.

It is important to note that Dahlquist's second barrier theorem states that an A -stable LMM method is at most second-order accurate. This limitation does not apply to (implicit) RK methods.

For this reason, it is convenient to introduce a more general definition.

Definition 3.20. An ODE method is said to be $A(\alpha)$ -stable if its region of stability S contains the sector $\{z \in \mathbb{C} : \pi - \alpha \leq \arg zz \leq \pi + \alpha\}$.

Another notion, is L-stability, something that differentiates trapezoid and backward Euler.

Definition 3.21. A onestep method is L-stable if it is A -stable and $\lim_{z \rightarrow \infty} |R(z)| = 0$, where $R(z)$ is the stability function found by applying one step of the method to $u'(t) = \lambda u(t)$.

Chapter 4

Numerical solution of diffusion problems

The first problem we will address is that of computing solutions of

$$u_t = \kappa u_{xx}, \quad \kappa, t > 0, \quad 0 < x < 1.$$

This is the one-dimensional heat equation. we need to specify the rest of the problem to make it well-posed:

$$\begin{cases} u_t = \kappa u_{xx}, & \kappa, t > 0, \quad 0 < x < 1, \\ u(x, 0) = \eta(x), \\ u(0, t) = g_0(t), \\ u(1, t) = g_1(t). \end{cases} \quad (4.1)$$

Here we have specified *Dirichlet data*. If you want to know more about solutions of this problem, see [Eva10]. In this course we will largely ignore issues related to the regularity of the actual solution $u(x, t)$. Some things to note:

1. If $g_0(0) \neq \eta(0)$, or $g_1(0) \neq \eta(1)$, then the solution is guaranteed to have discontinuity in the corner of the domain.
2. The heat equation is smoothing. Even if η has a discontinuity at $t = 0$, for $t > 0$, typically, the solution will be smooth.
3. But if g_1 or g_0 are not smooth, this is no longer true — the solution can fail to be differentiable in time.

The boundary functions $g_0(t), g_1(t)$ are incredibly important contributing factors to the solution.

4.1 • Notation

We now introduce the notation that we will use to describe the numerical solution of time-dependent PDEs. As before, k will represent the time step. Here h will denote the grid spacing parameter:

$$x_j = jh, \quad t_n = n_k, \quad (4.2)$$

where $h = 1/(1 + m)$ so that $x_0 = 0, x_{m+1} = 1$. And we then use the notation

$$U_j^n \approx u(x_j, t_n).$$

Then the first thing you might try is

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{u_{xx}}{h^2} = \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

This then gives an explicit (in time) scheme:

$$U_i^{n+1} = U_i^n + \frac{k}{h^2} [U_{i+1}^n - 2U_i^n + U_{i-1}^n], \quad (4.3)$$

$$i = 1, 2, \dots, m. \quad (4.4)$$

Note that U_0^n and U_{m+1}^n do not appear here because these values are determined by the boundary conditions. This is effectively a forward Euler-based discretization.

Let us now examine why using this method may present some difficulties. In practice, we might need h to be quite small, and we may hope that we can take $k \propto h$, so that

$$\frac{k}{h^2} \propto \frac{1}{h}.$$

It is convenient to write (4.3) in matrix-vector notation

$$\begin{aligned} \begin{bmatrix} U_j^{n+1} \end{bmatrix} &= \begin{bmatrix} U_j^n \end{bmatrix} + \frac{k}{h^2} \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & 1 \\ & & & & 1 & -2 \end{bmatrix} \begin{bmatrix} U_j^n \end{bmatrix} + \frac{k}{h^2} \begin{bmatrix} g_0(t_n) \\ 0 \\ \vdots \\ 0 \\ g_1(t_n) \end{bmatrix}, \\ \begin{bmatrix} U_j^{n+1} \end{bmatrix} &= (I - D_h) \begin{bmatrix} U_j^n \end{bmatrix} + \frac{k}{h^2} g(t_n). \end{aligned} \quad (4.5)$$

So, it is important here to note that k/h^2 could be quite large. Intuitively, we should expect something here with large eigenvalues because we are discretizing the second derivative operator, which is an unbounded operator. To understand why this might be problematic, we consider the eigenvalues of

$$A = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & -1 \\ & & & & 2 \end{bmatrix}. \quad (4.6)$$

The Gershgorin circle theorem is a relatively simple theorem that allows us to say something immediately about eigenvalues after inspecting a matrix. For some matrices it is effectively useless but it can be quite useful for others, such as this matrix above. Before we state the theorem, we have a definition.

Definition 4.1. Let $A \in \mathbb{R}^{n \times n}$ (or $\mathbb{C}^{n \times n}$). For each $i = 1, 2, \dots, n$ define the Gershgorin disks

$$D_j = \{z \in \mathbb{C} : |z - a_{ii}| \leq r_j\}, \quad r_j = \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|.$$

The Gershgorin domain D_A is defined to be the union of all the disks:

$$D_A = \bigcup_{i=1}^n D_j \subset \mathbb{C}.$$

Theorem 4.2 (Gershgorin Circle Theorem). All the eigenvalues of A lie in its Gershgorin domain.

Proof. Suppose λ is an eigenvalue of A :

$$Av = \lambda v, \quad v \neq 0.$$

Suppose the i th entry of v is largest: $|v_j| = \|v\|_\infty$. And since we can divide both sides of this equation by scalars, we can suppose that $v_j = 1$. Looking at the i th row of this equation gives

$$\sum_{j=1}^n a_{ij}v_j = \lambda v_j \quad \Rightarrow \quad a_{ii} + \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij}v_j = \lambda.$$

This then implies that

$$|\lambda - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}v_j| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|$$

because $|v_j| \leq 1$. This shows that every eigenvalue must be in at least one disk and the conclusion follows.

The Gershgorin domain for A is the disk of radius 2 centered at $z = 2$. And because A is symmetric, we know all the eigenvalues lie in the interval $[0, 4]$. Indeed, we can compute the eigenvalues explicitly, see Appendix TBD, giving

$$\begin{aligned} \lambda_{\max}(A) &= 2 \cos\left(\frac{\pi}{m+1}\right) + 2, \\ \lambda_{\min}(A) &= 2 \cos\left(\frac{m\pi}{m+1}\right) + 2. \end{aligned}$$

So, for k/h^2 large, $I - D_h$ will have an eigenvalue outside the interval $[-1, 1]$, indicating blowup. Another way to see this, and connect it with stability regions, is to view (4.5) as a discretization of a system of ODEs with the forward Euler method. This viewpoint has a name.

4.2 • Method of lines discretization

In the above derivation of the method (4.5), we discretized both space and time at once. This can be useful. But another approach is to first discretize space, and convert the PDE problem to an ODE problem. This approach is called the method of lines discretization. Consider

$$\begin{aligned} u_t &= u_{xx}, \\ U_j(t) &= \frac{U_{j+1}(t) - 2U_j(t) + U_{j-1}(t)}{h^2}, \end{aligned}$$

for $i = 1, 2, \dots, m$. Here we are using the notation that $U_j(t) \approx U(x_j, t)$. Incorporating the boundary condition we are left with the ODE system

$$\left[\begin{array}{c} U'_j(t) \\ U_j(t) \end{array} \right] = \underbrace{\frac{1}{h^2} \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & & 1 & -2 \\ & & & & 1 \end{bmatrix}}_{-h^{-2}A} \left[\begin{array}{c} U_j(t) \\ U_{j+1}(t) \end{array} \right] + \underbrace{\frac{1}{h^2} \begin{bmatrix} g_0(t) \\ 0 \\ \vdots \\ 0 \\ g_1(t) \end{bmatrix}}_{h^{-2}g(t)}. \quad (4.7)$$

If we discretize this with forward Euler, we will obtain exactly (4.5). So, the stability of the method is contingent upon the eigenvalues of $-k/h^2 A = -D_h$ being within the unit disk. Based on the fact that the eigenvalues of A fill out the interval $[0, 4]$ as $h \rightarrow 0$, stability requires that the eigenvalues of $-h^{-2}A$ are within $[-2, 0]$, i.e., $k/h^2 \leq 1/2$. This enforces a very small step size. While this method will work, look to another method that will be far more efficient.

4.3 • The Crank-Nicolson method

The Crank-Nicolson method is derived by applying the trapezoidal method to the method of lines discretization (4.7), giving

$$\left[\begin{array}{c} U_j^{n+1} \\ U_j^n \end{array} \right] = \left[\begin{array}{c} U_j^n \\ U_j^n \end{array} \right] - \frac{k}{2h^2} \left[\begin{array}{c} A \left[\begin{array}{c} U_j^n \\ U_j^n \end{array} \right] + A \left[\begin{array}{c} U_j^{n+1} \\ U_j^{n+1} \end{array} \right] + g(t_n) + g(t_{n+1}) \\ A \left[\begin{array}{c} U_j^n \\ U_j^n \end{array} \right] + A \left[\begin{array}{c} U_j^{n+1} \\ U_j^{n+1} \end{array} \right] + g(t_n) + g(t_{n+1}) \end{array} \right].$$

Defining $r = \frac{k}{2h^2}$, and rearranging this, we find

$$(I + rA) \left[\begin{array}{c} U_j^{n+1} \\ U_j^n \end{array} \right] = (I - rA) \left[\begin{array}{c} U_j^n \\ U_j^n \end{array} \right] + r [g(t_n) + g(t_{n+1})].$$

The matrix $I + rA$ is invertible for any $r \geq 0$ and this linear system can be solved efficiently at each timestep because $I + rA$ is a tridiagonal matrix.

Stability

For any k , the eigenvalues of $-\frac{k}{h^2}A$ are in the left-half plane, within the stability region for trapezoid. Thus this numerical method will be stable for any choice of step size. We will see how this influences the convergence proof.

4.4 • Local truncation errors

4.4.1 • Forward Euler method

To see how we compute the LTE for a PDE discretization, consider the simple method we first derived, which is forward Euler applied to (4.7). Define

$$\tau(x, t) = \frac{u(x, t+k) - u(x, t)}{k} - \frac{u(x+h, t) - 2u(x, t) + u(x-h, t)}{h^2}.$$

Our goal is to then Taylor expand every term at (x, t) and determine the order of this expression. For convenience, we will drop function arguments. For example,

$$\frac{u(x, t+k) - u(x, t)}{k} = u_t + \frac{k}{2}u_{tt} + \frac{k^2}{6}u_{ttt} + O(k^3).$$

Then

$$\frac{u(x+h, t) - 2u(x, t) + u(x-h, t)}{h^2} = u_{xx} + \frac{1}{12}h^2u_{xxxx} + O(h^3)$$

It is convenient to note that $u_t = u_{xx}$ implies that $u_{tt} = u_{xxxx}$. We find

$$t(x, t) = \left(\frac{k}{2} - \frac{h^2}{12} \right) u_{xxx} + O(k^2 + h^3) = O(k + h^2).$$

The reveals something that is undesirable, and not unexpected. The contribution to the LTE from k is larger (order wise) than the contribution from h . And sometimes, we may use the notation

$$\tau_j^n = \tau(x_j, t_n).$$

4.4.2 • The Crank-Nicolson method

The LTE formula is given by

$$\begin{aligned} \tau(x, t) &= \frac{u(x, t+h) - u(x, t)}{k} \\ &\quad - \frac{1}{2h^2} [u(x-h, t) - 2u(x, t) + u(x+h, t) + u(x-h, t+k) - 2u(x, t+k) + u(x+h, t+k)]. \end{aligned}$$

We expand each term again

$$\begin{aligned} \frac{u(x, t+h) - u(x, t)}{k} &= u_t + \frac{k}{2}u_{tt} + \frac{k^2}{6}u_{ttt} + O(k^3), \\ \frac{1}{2h^2} [u(x-h, t) - 2u(x, t) + u(x+h, t)] &= \frac{1}{2}u_{xx} + \frac{1}{24}h^2u_{xxxx} + O(h^3). \end{aligned} \tag{4.8}$$

When expanding terms that have both arguments shifted, there are a couple ways to proceed. One could use the multivariate formula for Taylor series, or just expand in one variable at a time. We will do the latter:

$$\begin{aligned} u(x+h, t+k) &= u(x, t) + hu_x(x, t+k) + \frac{h^2}{2}u_{xx}(x, t+k) + O(h^3) \\ &= u + hu_x + hku_{xt} + \frac{h^2}{2}u_{xx} + O(h^2k + hk^2). \end{aligned}$$

But to use this principle, to get the last term we need, we can just replace t with $t + k$ in (4.8) and expand each term:

$$\begin{aligned} \frac{1}{2h^2} [u(x - h, t + k) - 2u(x, t + k) + u(x + h, t + k)] \\ = \frac{1}{2}u_{xx}(x, t + k) + \frac{1}{24}h^2u_{xxxx}(x, t + k) + O(h^3) \\ = \frac{1}{2}u_{xx} + \frac{k}{2}u_{xxt} + \frac{k^2}{4}u_{xxtt} + \frac{1}{24}h^2u_{xxxx} + O(k^3 + h^3 + h^2k). \end{aligned}$$

Combining everything by using that $u_t = u_{xx}$, $u_{xxt} = u_{tt}$, etc., we find

$$\tau(x, t) = -\left(\frac{k^2}{12}u_{6x} + \frac{h^2}{12}u_{xxxx}\right) + O(k^3 + h^3 + h^2k).$$

4.5 • Stability and convergence for methods for the heat equation

To understand the subtleties of the convergence of numerical methods for PDEs, the method of lines discretization is a good tool. If we fix m , and hence h , we know that our numerical methods for ODEs will converge:

$$\lim_{\substack{k \rightarrow 0 \\ Nk=T}} \left\| \begin{bmatrix} U_j^N \\ \vdots \\ U_j^N \end{bmatrix} - \begin{bmatrix} U_j(T) \\ \vdots \\ U_j(T) \end{bmatrix} \right\|_\infty = 0.$$

Then, if we could establish that method of lines discretization converged to the solution of the PDE

$$\lim_{\substack{h \rightarrow 0 \\ h=(m+1)^{-1}}} \left\| \begin{bmatrix} U_j(T) \\ \vdots \\ U_j(T) \end{bmatrix} - \begin{bmatrix} u(x_j, T) \\ \vdots \\ u(x_j, T) \end{bmatrix} \right\|_\infty = 0,$$

this would imply

$$\lim_{\substack{h \rightarrow 0 \\ h=(m+1)^{-1}}} \lim_{\substack{k \rightarrow 0 \\ Nk=T}} \left\| \begin{bmatrix} U_j^N \\ \vdots \\ U_j^N \end{bmatrix} - \begin{bmatrix} u(x_j, T) \\ \vdots \\ u(x_j, T) \end{bmatrix} \right\|_\infty = 0.$$

The issue is that taking a limit with finite computational resources is impossible. What we really have in the first case is a bound of the form (for fixed T).

$$\left\| \begin{bmatrix} U_j^N \\ \vdots \\ U_j^N \end{bmatrix} - \begin{bmatrix} U_j(T) \\ \vdots \\ U_j(T) \end{bmatrix} \right\|_\infty \leq C_h k^r.$$

And for each h we have a new system of ODEs and therefore new truncation errors, which is captured in the constant C_h . Then we suppose

$$\left\| \begin{bmatrix} U_j(T) \\ \vdots \\ U_j(T) \end{bmatrix} - \begin{bmatrix} u(x_j, T) \\ \vdots \\ u(x_j, T) \end{bmatrix} \right\|_\infty \leq Ch^\ell.$$

The triangle inequality gives

$$\left\| \begin{bmatrix} U_j^N \\ u(x_j, T) \end{bmatrix} - \begin{bmatrix} u(x_j, T) \\ u(x_j, T) \end{bmatrix} \right\|_{\infty} \leq C_h k^r + Ch^{\ell}.$$

And we need to send $k, h \rightarrow 0$ in a way that the right-hand side goes to zero. The issue at hand is that if a method is not sufficiently stable, then C_h could grow as $h \rightarrow 0$. If it grows at a polynomial rate, one can possibly overcome this (but should you?). But if it is exponential, the method will always fail to work in practice.

4.5.1 • Grid norms

The constants C_h and C above, and possibly r, ℓ , may be influenced by which norm one considers. We will discuss this more in what follows, but, if you use the built-in `norm` function in JULIA you will be using the 2-norm. But we have not just one 2-norm, we have an infinite family of 2-norms, one for each m . So, when considering convergence, it is important to scale the norms appropriately:

$$\|V_j\|_p^p = h \sum_{i=1}^m |V_j|^p. \quad (4.9)$$

The scaling by h is there so that if V_j is an approximation of a function $v(x_j)$, then this norm becomes an approximation of the continuous $L^p([0, 1])$ norm

$$\|v\|_p^p = \int_0^1 |v(x)|^p dx.$$

4.6 • Complexity considerations

Before we go further to discuss the convergence of our discretization of the heat equation, we will do a calculation to examine their efficiency.

Forward Euler discretization

- The LTE for the forward Euler method is $O(k+h^2)$. We need $k \leq h^2/2$ for stability considerations, $k \propto h^2$.
- Thus the number of time steps is $\propto 1/k \propto m^2$.
- Each iteration requires one multiplication by a tridiagonal matrix and one sparse vector addition $\approx 5m$ FLOPs.
- Thus we have $O(m^3)$ FLOPs to compute to $O(1)$ times with $O(m^{-2})$ error.

Crank-Nicolson discretization

- The LTE f is $O(k^2 + h^2)$. We have unconditional stability.
- Thus the number of time steps is $\propto 1/k \propto m$.
- Each iteration requires effectively the same calculation forward Euler, followed by a linear solve involving a tridiagonal linear system, $\approx 11m$ FLOPs.
- Thus we have $O(m^2)$ FLOPs to compute to $O(1)$ times with $O(m^{-2})$ error.

4.6.1 • Convergence theory

In this section, we suppose that $h = h(k)$ is a prescribed function of k , and $h(k) \rightarrow 0$ as $k \rightarrow 0$. And we suppose our method for the PDE under consideration takes the form

$$\begin{bmatrix} U_j^{n+1} \end{bmatrix} = B(k) \begin{bmatrix} U_j^n \end{bmatrix} + g^n(k). \quad (4.10)$$

Example 4.3. For Crank-Nicolson, we have with $r(k) = k/(2h(k)^2)$

$$B(k) = (I + r(k)A)^{-1}(I - r(k)A), \quad g^n(k) = (I + r(k)A)^{-1}r(k)[g(t_n) + g(t_{n+1})].$$

Definition 4.4. The method (4.10) is said to be Lax–Richtmyer stable if, for each time T , there is a constant $C_T > 0$ such that

$$\|B(k)^n\| \leq C_T, \quad (4.11)$$

for all k sufficiently small ($k > 0$) and integers n satisfying $kn \leq T$.

Remark 4.5. As h shrinks, the dimension of (4.10) grows, and so (4.11) is using a different norm for each h . So, when speaking about stability we need to specify which sequence of norms we want to consider. Often, it is convenient to use the grid norm $\|\cdot\|_2$.

Remark 4.6 (Operator norm induced by a grid norm). Consider the induced matrix norm $\|A\|_p$ where $\|\cdot\|_p$ is the grid norm (4.9),

$$\|A\|_p = \max_{x \in \mathbb{R}^m : \|x\|_p=1} \|Ax\|_p.$$

Let $\|\cdot\|$ be the standard ℓ^p norm, $\|x\|^p = \sum_{i=1}^m |x_i|^p$. Then $\|x\|_p = 1 \Leftrightarrow \|x\| = h^{-1/p}$ and

$$\|A\|_p = \max_{x \in \mathbb{R}^m : \|x\|=h^{-1/p}} h^{1/p} \|Ax\| \stackrel{y=h^{1/p}x}{=} \max_{y \in \mathbb{R}^m : \|y\|=1} \|Ay\|.$$

So, we see that this matrix norm coincides with the standard ℓ^p operator norm, and, in particular, $\|A\|_2$ is the largest singular value of A .

Theorem 4.7. A consistent method of the form (4.10) is convergent if and only if it is Lax–Richtmyer stable.

Proof. Set

$$E^n = \begin{bmatrix} U_j^n \end{bmatrix} - \begin{bmatrix} u(x_j, t_n) \end{bmatrix}.$$

Then

$$E^{n+1} = B(k)E^n - k\tau^n, \quad \tau^n = \begin{bmatrix} \tau_j^n \\ \vdots \\ \tau_1^n \end{bmatrix}.$$

Then

$$E^n = B(k)^N E^0 - k \sum_{n=1}^N B(k)^{N-n} \tau^{n-1}.$$

This gives

$$\|E^n\| \leq C_T (\|E^0\| + Nk\|\tau^n\|_\infty).$$

Here

$$\|\tau^n\|_\infty = \max_{0 \leq n \leq N} \|\tau^n\|.$$

And one needs to appeal to PDE theory to show that this can be bounded above by something independent of k, h .

Note that the use of absolute stability regions is required for establishing Lax–Richtmyer stability in practical settings. Since the derivative is an unbounded operator is not reasonable to assume that $k\lambda \rightarrow 0$ for every eigenvalue of the coefficient matrix of the method of lines discretization.

Example 4.8. The forward Euler method for the heat equation is Lax–Richtmyer stable in the grid norm $\|\cdot\|_2$ provided that $k/h^2 \leq 1/2$. The Crank–Nicolson method is unconditionally Lax–Richtmyer stable.

4.7 • Using the methods

Before we discuss the specifics of implementations, it is important to note that the use of the right data type for a tridiagonal matrix is important. In PYTHON and MATLAB you want to use *spdiags* to construct a sparse matrix.

```
import numpy as np
from scipy.sparse import spdiags

m = 4
data = np.array([np.ones(m), -2.0*np.ones(m), np.ones(m)]);
diags = np.array([-1, 0, 1])
A = spdiags(data, diags, m, m)

A.toarray() # just to see what it looks like
```

JULIA has a special symmetric tridiagonal data type that we will make extensive use of in this demo.

```
h = 0.01;
m = convert(Int64, 1/h)-1;
k = 0.01;
```

```
T = 10;
A = SymTridiagonal(fill(2.0,m),fill(-1.0,m-1))
r = k/(2*h^2)
A1 = I + r*A
Ar = I - r*A;
```

We are considering the heat equation (4.1) with Dirichlet boundary conditions.

```
g0 = t -> t.^2/(1 .+ t.^2)*sin.(4*t)
g1 = t -> 0.
η = x -> exp.(-20*(x .-1/2).^2)
```

Note that $\eta(1) = \eta(0) \neq 0$, so there is some small degree of incompatibility of the initial and boundary data. The following code creates a simple gif using Crank-Nicolson.

```
plot()
anim = Animation()
n = convert(Int64,ceil(T/k))
x = h:h:1-h
U = η(x)
t = 0.0
plot(x, U, xaxis = [0,1], yaxis = [-1,2],lw=3,
      label = @sprintf("u(x,t), t = %1.2f",t))
plot!([0,1],[g0(t),g1(t)], label = "BCs", seriestype = :scatter)
frame(anim)

fr = 100 #frames/unit time
tb = convert(Int64,ceil(n/(fr*T)))
for i = 2:n+1
    t += k
    U = Ar*U
    U[1] += r*(g0(t)+g0(t-k))
    U[end] += r*(g1(t)+g1(t-k))
    U = A1\U
    if mod(i-1,tb) ≈ 0.0
        plot(x, U, xaxis = [0,1], yaxis = [-1,2],lw=3,
              label = @sprintf("u(x,t), t = %1.2f",t))
        plot!([0,1],[g0(t),g1(t)], label = "BCs", seriestype = :scatter)
        frame(anim)
    end
end
gif(anim,"heat_CN.gif")
```

To see just how efficient this method is, we now do some timing.

```
function CN_heat(η,g0,g1,T,k,h)
    m = convert(Int64,1/h)-1;
    A = SymTridiagonal(fill(-2.0,m),fill(1.0,m-1))
    r = k/(2*h^2)
    A1 = I - r*A
    Ar = I + r*A;
    n = convert(Int64,ceil(T/k))
    x = h:h:1-h
    U = η(x)
    t = 0.0

    for i = 2:n+1
        t += k
        U = Ar*U
        U[1] += r*(g0(t)+g0(t-k))
        U[2] += r*(g1(t)+g1(t-k))
        U = A1\U
    end
    U
end
```

```
@time CN_heat(η,g₀,g₁,10.,0.0001,0.0001);
```

```
0.212549 seconds (40.01 k allocations: 310.167 MiB, 23.36% gc time)
```


Chapter 5

von Neumann stability analysis: The heat equation

We begin with a review of Fourier analysis because von Neumann stability analysis is derived from it.

5.1 • The (periodic) Fourier transform

Recall Fourier series for a smooth, periodic function $f : [-\pi, \pi] \rightarrow \mathbb{C}$:

$$f(x) = \frac{1}{\sqrt{2\pi}} \sum_{j=-\infty}^{\infty} c_j(f) e^{ix},$$
$$c_j(f) = \frac{1}{\sqrt{2\pi}} \int_{-\pi}^{\pi} e^{-ix} f(x) dx, \quad n = 0, \pm 1, \dots$$

When f is implied from context, we may write $c_n = c_n(f)$. An important relation is the *Plancharel theorem*.

Theorem 5.1. Suppose $g, f \in L^2([-\pi, \pi])$, then

$$\int_{-\pi}^{\pi} f(x) \overline{g(x)} dx = \sum_{j=-\infty}^{\infty} c_j(f) \overline{c_j(g)}.$$

This implies that, in particular,

$$\int_{-\pi}^{\pi} |f(x)|^2 dx = \sum_{j=-\infty}^{\infty} |c_j(f)|^2.$$

This is probably the context in which you have seen Fourier series — you take a function of a space variable and transform it to a function of a discrete frequency variable. We will turn this around, and since we discretize space for a PDE, we will take a function of a discrete space variable and transform it to a function of a continuous frequency variable. Define $\ell^2(\mathbb{Z})$ to be the set of all such bi-infinite vectors V

$$V = \begin{bmatrix} V_j \end{bmatrix} = (V_j)_{j=-\infty}^{\infty}$$

such that

$$\|V\|_2 := \left(h \sum_{j=-\infty}^{\infty} |V_j|^2 \right)^{1/2} < \infty.$$

We will think of $V_j = v(x_j)$ being a function evaluated on the grid $x_j = jh$, $j = 0, \pm 1, \pm 2, \dots$. Now define, for $\xi \in [-\pi/h, \pi/h]$

$$\hat{V}(\xi) = \frac{h}{\sqrt{2\pi}} \sum_{j=-\infty}^{\infty} V_j e^{-ijh\xi}.$$

An important calculation is to then compute

$$\frac{1}{\sqrt{2\pi}} \int_{-\pi/h}^{\pi/h} \hat{V}(\xi) \left[\frac{h}{\sqrt{2\pi}} e^{ijh\xi} \right] d\xi = \frac{h^2}{2\pi} \sum_{j=-\infty}^{\infty} \int_{-\pi/h}^{\pi/h} V_j e^{i(\ell-j)\xi h} d\xi = hV_j.$$

The new statement of the Plancharel theorem becomes

$$\|\hat{V}\|_2^2 := \int_{-\pi/h}^{\pi/h} |\hat{V}(\xi)| d\xi = h \sum_{j=-\infty}^{\infty} |V_j|^2 = \|V\|_2^2.$$

5.2 • The discretization of the Cauchy problem on the line

Now, consider solving the heat equation,

$$u_t = u_{xx},$$

for all $x \in \mathbb{R}$, $t > 0$ with initial data $u(x, 0) = \eta(x)$. Let us assume that η has compact support, that is, on the complement of $[-R, R]$, for R sufficiently large, η vanishes identically. We now construct the bi-infinite method of lines discretization ($U_j(t) \approx u(x_j, t)$)

$$U'_j(t) = \frac{U_{j-1}(t) - 2U_j(t) + U_{j+1}(t)}{h^2}, \quad j = 0, \pm 1, \pm 2, \dots$$

Without some boundary conditions, this system of ODEs is not closed, and you must consider all integers j ranging from $-\infty$ to $+\infty$. But we can discretize this with, for example, forward Euler to find

$$\begin{cases} U_j^0 = \eta(x_j), & j = 0, \pm 1, \pm 2, \dots, \\ U_j^{n+1} = U_j^n + \frac{k}{h^2} [U_{j-1}^n - 2U_j^n + U_{j+1}^n]. \end{cases} \quad (5.1)$$

Since we have assumed that η has compact support, $U_j^0 = 0$ if $|jh| > R$, and this system is closed and we can iterate it, and we want to. The behavior of this system should indicate something about the stability of methods based on this discretization, even though we have not incorporated any boundary conditions. But note that if $U_j^n = 0$ but $U_{j-1}^n \neq 0$, then $U_j^{n+1} \neq 0$. This might make one feel like solving (5.1) is difficult, but fortunately, Fourier series simplifies things.

We should also point out that (5.1) can be written in the form

$$\begin{bmatrix} U_j^{n+1} \end{bmatrix} = B(k) \begin{bmatrix} U_j^n \end{bmatrix}, \quad h = h(k), \quad (5.2)$$

for a matrix $B(k)$.

Lemma 5.2. *Consider the system of the form (5.2). Then*

$$\left\| \begin{bmatrix} U_j^{n+1} \end{bmatrix} \right\|_2 \leq (1 + \alpha k) \left\| \begin{bmatrix} U_j^n \end{bmatrix} \right\|_2 \Leftrightarrow \left\| \hat{U}^{n+1} \right\|_2 \leq (1 + \alpha k) \left\| \hat{U}^n \right\|_2,$$

And, if the second inequality is satisfied for all $(U_j^n)_{j=-\infty}^\infty \in \ell^2(\mathbb{Z})$, then the iteration (5.2) is Lax-Richtmyer stable.

Proof. The first claim is just a direct consequence of the Plancharel theorem. For the latter claim, we see that

$$\begin{aligned} & \left\| \hat{U}^{n+1} \right\|_2 \leq (1 + \alpha k) \left\| \hat{U}^n \right\|_2, \\ & \Rightarrow \left\| \begin{bmatrix} U_j^{n+1} \end{bmatrix} \right\|_2 \leq (1 + \alpha k) \left\| \begin{bmatrix} U_j^n \end{bmatrix} \right\|_2, \\ & \Rightarrow \left\| B(k) \begin{bmatrix} U_j^n \end{bmatrix} \right\|_2 \leq (1 + \alpha k) \left\| \begin{bmatrix} U_j^n \end{bmatrix} \right\|_2, \\ & \Rightarrow \|B(k)\|_2 \leq (1 + \alpha k), \\ & \Rightarrow \|B(k)^n\|_2 \leq \|B(k)\|_2^n \leq (1 + \alpha k)^n \leq e^{\alpha k n} \leq e^{\alpha T}. \end{aligned}$$

To solve (5.1) we use the fact that every translation invariant operator has a *Fourier symbol*. Define the shift operator S_ℓ by

$$\begin{bmatrix} W_j \end{bmatrix} = S_\ell \begin{bmatrix} V_j \end{bmatrix}, \quad W_j = V_{j-\ell}.$$

Then it is clear that $S_{-\ell} = S_\ell^{-1}$. A bi-infinite bounded linear operator L on $\ell^2(\mathbb{Z})$ is translation invariant if

$$\begin{bmatrix} W_j \end{bmatrix} = S_{-\ell} L S_\ell \begin{bmatrix} V_j \end{bmatrix},$$

for all vectors V in $\ell^2(\mathbb{Z})$ and all integers ℓ .

Theorem 5.3. If a bounded linear operator L on $\ell^2(\mathbb{Z})$ is translation invariant then there exists a bounded^a function $s(\xi)$, the Fourier symbol, such that for

$$\begin{bmatrix} W_j \\ V_j \end{bmatrix} = L \begin{bmatrix} V_j \\ V_j \end{bmatrix} \Leftrightarrow \hat{W}(\xi) = s(\xi) \hat{V}(\xi).$$

Furthermore, $s(\xi) = L \widehat{\begin{bmatrix} Z_j \\ Z_j \end{bmatrix}}(\xi)$ where $Z_0 = 1, Z_j = 0, j \neq 0$.

^aTechnically, the function is essentially bounded.

Proof. We apply L to the standard basis vectors $e_j, j = 0, \pm 1, \pm 2, \dots$. We know that $Le_j \in \ell^2(\mathbb{Z})$ for each j . And we also know that $S_{-\ell}LS_\ell e_j = Le_j$. Then

$$S_\ell e_j = e_{j-\ell}.$$

We obtain the relation

$$Le_{j-\ell} = S_\ell Le_j. \quad (5.3)$$

First, we recall that for any vector

$$\widehat{S_\ell \begin{bmatrix} V_j \\ V_j \end{bmatrix}}(\xi) = \frac{h}{\sqrt{2\pi}} \sum_{j=-\infty}^{\infty} V_{j-\ell} e^{-i j h \xi} = e^{-i \ell h \xi} \hat{V}(\xi).$$

This implies that

$$\widehat{Le_{j-\ell}}(\xi) = e^{-i \ell h \xi} \widehat{Le_j}(\xi),$$

and, in particular,

$$\widehat{Le_\ell}(\xi) = e^{i \ell h \xi} \widehat{Le_0}(\xi).$$

So now if we apply L to a vector V , we first decompose

$$V = \sum_{j=-\infty}^{\infty} V_j e_j,$$

and linearity implies that

$$\widehat{LV}(\xi) = \sum_{j=-\infty}^{\infty} \widehat{Le_0}(\xi) V_j e^{i j h \xi} = s(\xi) \hat{V}(\xi),$$

where $s(\xi) = \widehat{Le_0}(\xi)$. Now, if $s(\xi)$ was not essentially bounded, it would contradict the fact that the operator L is bounded.

Remark 5.4. The above theorem gives a method to compute $s(\xi)$ and one can easily use this. But, in practice, it is often easier to use other methods.

5.2.1 • Calculating the Fourier symbol

Now, the operator D_2 defined by

$$D_2 \begin{bmatrix} V_j \\ W_j \end{bmatrix} = \begin{bmatrix} W_j \\ V_j \end{bmatrix}, \quad W_j = \frac{V_{j-1} - 2V_j + V_{j+1}}{h^2},$$

is translation invariant and we look for its symbol. This is clear because we are actually considering

$$\frac{1}{h^2}(S_1 - 2S_0 + S_{-1}).$$

We know that the symbol of S_ℓ is $e^{-i\ell h\xi}$, and therefore linearity gives us the symbol

$$\frac{1}{h^2} [e^{-ih\xi} - 2 + e^{ih\xi}] = \frac{2}{h^2} [\cos(h\xi) - 1].$$

Computing things this way can be a bit non-intuitive, especially when we go to higher spatial dimensions, so one method is to note that

$$S_\ell \begin{bmatrix} e^{ij\xi h} \\ e^{i(j-\ell)\xi h} \end{bmatrix} = \begin{bmatrix} e^{ij\xi h} \\ e^{i(j-\ell)\xi h} \end{bmatrix} = e^{-i\ell h\xi} \begin{bmatrix} e^{ij\xi h} \\ e^{ij\xi h} \end{bmatrix},$$

which is the right symbol for the shift. So, we just substitute $V_j = e^{ij\xi h}$ into our difference relation:

$$\frac{V_{j-1} - 2V_j + V_{j+1}}{h^2} = e^{ij\xi h} \frac{1}{h^2} [e^{-ih\xi} - 2 + e^{ih\xi}],$$

giving us the same result.

5.2.2 • Solving the discretized Cauchy problem

Consider solving (5.1). Fourier transform the equation

$$\begin{aligned} U_j^{n+1} &= U_j^n + \frac{k}{h^2} [U_{j-1}^n - 2U_j^n + U_{j+1}^n] \\ &\Leftrightarrow \\ \hat{U}^{n+1}(\xi) &= \hat{U}^n(\xi) + \frac{k}{h^2} [\cos(h\xi) - 1] \hat{U}^n(\xi) \\ &= \left[1 + \frac{2k}{h^2} [\cos(h\xi) - 1] \right] \hat{U}^n(\xi). \end{aligned}$$

So, the Fourier transform of the solution at time step n is

$$\hat{U}^n(\xi) = \left[1 + \frac{2k}{h^2} [\cos(h\xi) - 1] \right]^n \hat{U}^0(\xi).$$

So, we see that stability requires that

$$\left| 1 + \frac{2k}{h^2} [\cos(h\xi) - 1] \right| \leq 1,$$

or $\frac{k}{h^2} \leq \frac{1}{2}$. This proves Lax-Richtmyer stability of (5.2), which may or may not have anything to do with stability for a problem on a finite interval with boundary conditions. This is the *von Neumann stability analysis* for the forward Euler discretization of the heat equation.

5.3 • von Neumann analysis in multiple space dimensions

The value of the last observation that we can just choose $V_j = e^{ijh\xi}$ to compute the symbol makes calculations in higher dimensions much simpler. Consider the heat equation

$$u_t = u_{xx} + u_{yy}.$$

We now use $U_{j,\ell}(t)$ to denote our approximation of $u(x_j, y_\ell, t)$. In principle, we could have two grid spacings h_1 for x_j and h_2 for y_ℓ , but we take $h_1 = h_2 = h$, for simplicity. Then, the most basic method of lines discretization gives

$$U_{j,\ell}(t) = \frac{U_{j+1,\ell}(t) - 2U_{j,\ell}(t) + U_{j-1,\ell}(t)}{h^2} + \frac{U_{j,\ell+1}(t) - 2U_{j,\ell}(t) + U_{j,\ell-1}(t)}{h^2}. \quad (5.4)$$

We could think of the right-hand side as defining an operator

$$\frac{V_{j+1,\ell} - 2V_{j,\ell} + V_{j-1,\ell}}{h^2} + \frac{V_{j,\ell+1} - 2V_{j,\ell} + V_{j,\ell-1}}{h^2}, \quad (5.5)$$

on $\ell^2(\mathbb{Z}^2)$ and repeat the arguments above using multidimensional Fourier series, i.e.,

$$\hat{V}(\xi_1, \xi_2) = \frac{h^2}{2\pi} \sum_{j,\ell} V_{j,\ell} e^{-i h \xi_1 j} e^{-i h \xi_2 \ell}, \quad -\frac{\pi}{h} \leq \xi_1, \xi_2 < \frac{\pi}{h}.$$

Or, we can just substitute $V_{j,\ell} = e^{ih\xi_1 j} e^{ih\xi_2 \ell}$ into (5.5) to find

$$e^{ih\xi_1 j} e^{ih\xi_2 \ell} \underbrace{\left[\frac{2 \cos(h\xi_1) - 2}{h^2} + \frac{2 \cos(h\xi_2) - 2}{h^2} \right]}_{s(\xi_1, \xi_2)}.$$

If we discretize (5.4) with forward Euler, we find

$$\hat{U}^{n+1}(\xi_1, \xi_2) = \left[1 + \frac{2k}{h^2} [\cos(h\xi_2) + \cos(h\xi_1) - 2] \right] \hat{U}^n(\xi_1, \xi_2).$$

If $\xi_1 = \xi_2 = -\pi/h$ then the expression in brackets is $1 - \frac{8k}{h^2}$ and $1 - \frac{8k}{h^2} \geq -1$ implies $k \leq \frac{h^2}{4}$.

Chapter 6

Introduction to wave propagation: The advection equation

We now consider what is arguably the simplest PDE

$$\begin{cases} u_t + au_x = 0, & a \in \mathbb{R}, \\ u(x, 0) = \eta(x), & x \in \mathbb{R}. \end{cases} \quad (6.1)$$

This equation is hyperbolic (see [LeV07, Appendix E]). If η is continuously differentiable, then it is clear that

$$u(x, t) = \eta(x - at).$$

This is just purely translation of the initial profile with speed a . This is always the solution of this problem, as posed for $x \in \mathbb{R}$. If η is not continuously differentiable, then one needs to be careful with the way in which it solves (6.1). One approach to solving (6.1) is to use a first-order discretization in time and a second-order, centered difference in space:

$$\frac{U_j^{n+1} - U_j^n}{k} = -a \frac{U_{j+1}^n - U_{j-1}^n}{2h}$$

This, of course, gives

$$U_j^{n+1} = U_j^n - \frac{ak}{2h} [U_{j+1}^n - U_{j-1}^n]. \quad (6.2)$$

And we know that the LTE is $O(k + h^2)$. One can realize this method as a forward Euler discretization of the MOL discretization

$$U'_j(t) = -\frac{a}{2h} [U_{j+1}(t) - U_{j-1}(t)].$$

It will turn out that method (6.2) can be improved by replacing U_j^n on the right-hand side with the average of its neighbors. This gives the *Lax-Friedrichs method*

$$U_j^{n+1} = \frac{U_{j+1}^n + U_{j-1}^n}{2} - \frac{ak}{h} \left[\frac{U_{j+1}^n - U_{j-1}^n}{2} \right]. \quad (6.3)$$

We will discuss the LTE for this method soon, but it is $O(k + h^2)$, if $k \propto h$. There is a nuance here coming from the fact that this is not a method coming from MOL and we cannot take k as small as we like.

Set $r = \frac{ak}{h}$, as this is the coefficient that appears in the right-hand side. If we take $k \approx h$ then $r = O(1)$ (recall the that the analogous parameter for the heat equation would be $O(h^{-1})$ in this setting. This indicates that explicit methods are back in the picture! We will see that (6.3) is Lax-Richtmyer stable if

$$\left| \frac{ak}{h} \right| \leq 1.$$

But, before we go to do this in more detail, let us be more explicit with the MOL discretization.

The advection equation is first-order in space. This implies that only one boundary condition needs to be/can be specified. Which side of the domain? The main heuristic for this is that we impose boundary conditions on the problem and therefore they act as forcings, not in response. So, the boundary condition should be placed at the *in-flow boundary* – if $a > 0$, at $x = 0$, and if $a < 0$ at $x = 1$. In the case where $a > 0$, the MOL discretization becomes

$$\begin{aligned} U_0(t) &= g_0(t), \\ U'_1(t) &= -a \frac{U_2(t) - U_0(t)}{2h} = -a \frac{U_2(t) - g_0(t)}{2h}, \\ U'_2(t) &= -a \frac{U_3(t) - U_1(t)}{2h}, \\ &\vdots \\ U'_{m+1}(t) &= -a \frac{U_{m+2}(t) - U_m(t)}{2h}. \end{aligned}$$

This last equation becomes an issue because it does not close the system. One option to fix this is to instead set,

$$U'_{m+1}(t) = -a \frac{3U_{m+1}(t) - 4U_m(t) + U_{m-1}(t)}{2h},$$

i.e., to instead use backward differentiation at the boundary. But one should note that this will change the structure of the matrix that needs to be considered to establish Lax-Richtmyer stability, making establishing it much more difficult.

6.1 • Periodic boundary conditions

One setting where the implementation of boundary conditions is straightforward is in the case of periodic boundary conditions:

$$\begin{cases} u_t = -au_x, \\ u(x, 0) = \eta(x), \\ u(0, t) = u(1, t). \end{cases}$$

This implies that $U_{m+1}(t) = U_0(t)$ in any MOL discretization. Periodically extending this we also get that

$$U_{m+2}(t) = U_1(t), \quad U_{m+3}(t) = U_2(t), \quad \dots$$

Then when formulating the MOL system we assemble $U_1(t), U_2(t), \dots, U_{m+1}(t)$ in a vector to find

$$\begin{bmatrix} U'_j(t) \end{bmatrix} = -\frac{a}{2h} S U(t), \quad S = \begin{bmatrix} 0 & 1 & & & -1 \\ -1 & 0 & 1 & & \\ & -1 & 0 & 1 & \\ & & -1 & \ddots & \ddots \\ 1 & & & -1 & 0 \end{bmatrix}$$

This is a structure-preserving discretization. The derivative operator is formally skew-adjoint and this matrix is skew-symmetric (or anti-symmetric):

$$\int \varphi(x) u_x(x) dx = - \int \varphi_x(x) u(x) dx.$$

if we use integration by parts and assume no contributions from the boundary. This gives $\langle \varphi, u_x \rangle = -\langle \varphi_x, u \rangle$ — skew symmetry.

This means that A has purely imaginary eigenvalues. Let's find these eigenvalues. Since A encodes a periodic operator then it is often successful to look for eigenvectors that come from periodic grid functions

$$\begin{bmatrix} V_j \end{bmatrix} : V_j = e^{2\pi i j h}, \quad j = 1, 2, \dots, m + 1.$$

A generalization of this is

$$\begin{bmatrix} V_j^{(\ell)} \end{bmatrix} : V_j^{(\ell)} = e^{2\pi i j \ell h} \quad j = 1, 2, \dots, m + 1.$$

We then compute

$$S \begin{bmatrix} V_j^{(\ell)} \end{bmatrix} = \begin{bmatrix} e^{2\pi i(2)h} - e^{2\pi i(m+1)h} \\ e^{2\pi i(3)h} - e^{2\pi i(1)h} \\ e^{2\pi i(4)h} - e^{2\pi i(2)h} \\ \vdots \\ e^{2\pi i(1)h} - e^{2\pi i(m)h} \end{bmatrix} = \begin{bmatrix} e^{2\pi i(1)h}(e^{2\pi i(1)h} - e^{2\pi i(-1)h}) \\ e^{2\pi i(2)h}(e^{2\pi i(1)h} - e^{2\pi i(-1)h}) \\ \vdots \\ e^{2\pi i(m+1)h}(e^{2\pi i(1)h} - e^{2\pi i(-1)h}) \end{bmatrix} = 2i \sin(2\pi \ell h) \begin{bmatrix} V_j^{(\ell)} \end{bmatrix}.$$

This shows that the eigenvalues of S are given by

$$2i \sin(2\pi \ell), \quad \ell = 1, 2, \dots, m + 1.$$

Applying forward Euler to the periodic MOL discretization gives

$$\begin{bmatrix} U_j^{n+1} \end{bmatrix} = \begin{bmatrix} U_j^n \end{bmatrix} - \frac{ak}{2h} S \begin{bmatrix} U_j^n \end{bmatrix}.$$

For Lax-Richtmyer stability we need to examine the eigenvalues of $I - \frac{ak}{2h} S$. These are on the line $1 + i[-ak/h, ak/h]$ in the complex plane. Note that for any finite choice of k , there will be an eigenvalue outside the unit circle at a distance

$$\left| 1 + i \frac{ak}{h} \right| = \sqrt{1 + \frac{a^2 k^2}{h^2}} \approx 1 + \frac{a^2 k^2}{2h^2} + O(k^4/h^4),$$

from the origin. If we choose $k \propto h^2$ then this is of the form $1 + \alpha k$ and previous considerations give Lax-Richtmyer stability in the 2-norm.

But the issue is that we have to take $k \propto h^2$ and the method is not efficient.

6.2 • Stability of the Lax-Friedrichs method

Appendix A

Functions of matrices

One way to define the (square!) matrix exponential is through its Taylor series

$$\begin{aligned} e^A &= I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots \\ &= \sum_{j=0}^{\infty} \frac{A^j}{j!}. \end{aligned} \tag{A.1}$$

Now, let $\|\cdot\|$ be any norm on \mathbb{R}^n and then the induced operator norm satisfies

$$\|A^j\| \leq \|A\|^j, \quad j = 0, 1, 2, \dots.$$

This implies that for any $m > 0$

$$\left\| \sum_{j=0}^m \frac{A^j}{j!} \right\| \leq \sum_{j=0}^m \frac{\|A\|^j}{j!} \leq e^{\|A\|} < \infty,$$

and therefore the series converges absolutely and is therefore convergent because of the following fact.

Theorem A.1. *A normed vector space is complete if and only if every absolutely convergent series converges.*

And the vector space of $n \times n$ matrices with the operator norm is complete.

The representation (A.1) is useful to derive many of the important properties of e^A . First, we recall that one can always compute the derivative of a convergent Taylor series by term-by-term differentiation within its radius of convergence. The same principle applies here

$$\frac{d}{dt} e^{tA} = \sum_{j=0}^{\infty} \frac{d}{dt} \frac{(tA)^j}{j!} = A e^{tA}.$$

Two other useful facts are

$$\begin{aligned} e^{0A} &= I, \\ e^{sA} e^{tA} &= e^{(s+t)A}. \end{aligned}$$

We then have some remaining questions:

1. Does the Taylor series give a viable method to compute $e^{tA} \eta$?
2. What about other functions like \sqrt{A} ?

The answer to the first question is a resounding *sometimes*.

We now introduce some tools from complex analysis to assist in the analysis more general classes of functions of matrices.

Definition A.2. Let $\Gamma \subset \mathbb{C}$ be a smooth simple curve. Suppose $f : \Omega \rightarrow \mathbb{C}$ is analytic in an open set Ω and $\Gamma \subset \Omega$. If Γ encloses all of the eigenvalues of A , we set

$$f(A) := \frac{1}{2\pi i} \int_{\Gamma} f(z)(zI - A)^{-1} dz. \quad (\text{A.2})$$

Example A.3. Consider the 2×2 case

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}.$$

Then

$$(zI - A)^{-1} = \frac{1}{(z - a_{11})(z - a_{22}) - a_{12}a_{21}} \begin{bmatrix} z - a_{22} & a_{12} \\ a_{21} & z - a_{11} \end{bmatrix}.$$

Then we can understand the integral formula for $f(A)$ and just the componentwise contour integral

$$f(A) = \frac{1}{2\pi i} \int_{\Gamma} \frac{f(z)}{(z - a_{11})(z - a_{22}) - a_{12}a_{21}} \begin{bmatrix} z - a_{22} & a_{12} \\ a_{21} & z - a_{11} \end{bmatrix} dz.$$

Now, as we go forward, we want to get a handle on what (A.2) does by understanding what it gives with f is particularly simple. Consider

$$I_k = \frac{1}{2\pi i} \int_{\Gamma} z^k (zI - A)^{-1} dz \quad \left(\stackrel{?}{=} A^k \right).$$

We put A into Jordan canonical form

$$A = R J R^{-1},$$

where

$$J = \begin{bmatrix} J(\lambda_1, k_1) & & & \\ & J(\lambda_2, k_2) & & \\ & & \ddots & \\ & & & J(\lambda_s, k_s) \end{bmatrix},$$

$$J(\lambda, \ell) = \begin{bmatrix} \lambda & 1 & & & \\ & \lambda & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1 \\ & & & & \lambda \end{bmatrix} \in \mathbb{R}^{\ell \times \ell},$$

and $\sum_j k_j = n$.

We first assume that A is diagonalizable, implying that $k_j = 1$ for all j . Then

$$I_k = R \left(\frac{1}{2\pi i} \int_{\Gamma} z^k (zI - J)^{-1} dz \right) R^{-1},$$

$$(zI - J)^{-1} = \begin{bmatrix} (z - \lambda_1)^{-1} & & \\ & \ddots & \\ & & (z - \lambda_n)^{-1} \end{bmatrix}.$$

Applying the residue theorem, we see that

$$\frac{1}{2\pi i} \int_{\Gamma} z^k (z - \lambda_j)^{-1} dz = \lambda_j^k,$$

and therefore

$$I_K = R J^k R^{-1} = A^k,$$

as we expect.

Now, with an eye towards the general case, compute

$$zI - J(\lambda, \ell) = \begin{bmatrix} z - \lambda & -1 & & & \\ & z - \lambda & -1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & -1 \\ & & & & z - \lambda \end{bmatrix} = (z - \lambda) \left(I - \frac{N}{z - \lambda} \right),$$

where

$$N = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \in \mathbb{R}^{\ell \times \ell}.$$

Note that $N^\ell = 0$ implying that N is nilpotent. We now have lemma to help in the computations.

Lemma A.4. *For $z \neq \lambda$,*

$$\left(I - \frac{N}{z - \lambda} \right)^{-1} = \sum_{j=0}^{\ell-1} \frac{N^j}{(z - \lambda)^j}.$$

To see why this follows, we note that if it ever happens that for $M \in \mathbb{R}^{n \times n}$,

$$\sum_{j=0}^{\infty} M^j$$

converges, then

$$(I - M)^{-1} = \sum_{j=0}^{\infty} M^j.$$

And we see that if we take M proportional to N then this series clearly converges because only a finite number of the terms are nonzero. The lemma now directly implies

$$(zI - J(\lambda, \ell))^{-1} = \frac{1}{z - \lambda} \sum_{j=0}^{\ell-1} \frac{N^j}{(z - \lambda)^j},$$

and therefore

$$\frac{1}{2\pi i} \int_{\Gamma} z^k (zI - J(\lambda, \ell))^{-1} dz = \frac{1}{2\pi i} \int_{\Gamma} \frac{z^k}{z - \lambda} \sum_{j=0}^{\ell-1} \frac{N^j}{(z - \lambda)^j} dz.$$

We can compute this integral via the residue theorem, but we need a Taylor expansion of z^k at $z = \lambda$. To do this, write, by the binomial theorem

$$z^k = (z - \lambda + \lambda)^k = \sum_{p=0}^k \binom{k}{p} \lambda^{k-p} (z - \lambda)^p.$$

Thus

$$\frac{1}{2\pi i} \int_{\Gamma} z^k (zI - J(\lambda, \ell))^{-1} dz = \frac{1}{2\pi i} \int_{\Gamma} \sum_{p=0}^k \sum_{j=0}^{\ell-1} \binom{k}{p} \lambda^{k-p} (z - \lambda)^p \frac{N^j}{(z - \lambda)^j} \frac{dz}{z - \lambda}.$$

Most of these terms integrate to zero. The only nonzero values result when $j = p$, giving

$$\begin{aligned} \frac{1}{2\pi i} \int_{\Gamma} z^k (zI - J(\lambda, \ell))^{-1} dz &= \frac{1}{2\pi i} \int_{\Gamma} \sum_{p=0}^{\min\{k, \ell-1\}} \binom{k}{p} \lambda^{k-p} \frac{N^p}{z - \lambda} dz \\ &= \sum_{p=0}^{\min\{k, \ell-1\}} \binom{k}{p} \lambda^{k-p} N^p = \sum_{p=0}^k \binom{k}{p} \lambda^{k-p} N^p. \end{aligned}$$

Note that we can replace $\min\{k, \ell-1\}$ with the, possibly larger, value k because $N^\ell = 0$. Then we just notice that

$$\sum_{p=0}^k \binom{k}{p} \lambda^{k-p} N^p = (\lambda I - N)^k = J(\lambda, \ell)^k.$$

This argument applied block-by-block, shows that

$$A^k = \frac{1}{2\pi i} \int_{\Gamma} z^k (zI - A)^{-1} dz,$$

again, provided that Γ encircles the eigenvalues of A . So, we get a consistent extension of notion of a *function of a matrix* that generalizes polynomials, and therefore coincides with any function that has a convergent Taylor series in a disk that contains the eigenvalues.

Appendix B

A proof of some of Dahlquist's theorem

The goal of this section is to prove the portion of Dahlquist's theorem that gives the convergence of LMMs. We do not consider the necessity of the conditions we state.

Consider solving

$$\begin{cases} u'(t) = f(u(t)), \\ u(0) = \eta \in \mathbb{R}^n, \end{cases}$$

on the domain $\mathcal{D} = \{(u, t) : \|u - \eta\| \leq a, |t| \leq T\}$. We will have two norms to consider for which we will use the same notation

$$\|u\|_\infty = \max_{0 \leq t \leq T} |u(t)|, \quad \|f\|_\infty = \max_{\eta-a \leq u \leq \eta+a} \|f(u)\|_2.$$

Which norm is being used will be clear from context. The method we will use is a LMM

$$\sum_{j=0}^r \alpha_j U^{n+j} = k \sum_{j=0}^r \beta_j f(U^{n+j}), \quad \alpha_r = 1. \quad (\text{B.1})$$

Suppose that the LTE satisfies

$$\|\tau_{\text{LMM}}^n\|_2 \leq C_1 \|u^{(p+1)}\|_\infty k^p,$$

for a constant $C_1 > 0$. For to be finite, one needs that

$$f^{(j)}, \quad j = 0, 1, 2, \dots, p, \text{ is continuous on } [\eta - a, \eta + a].$$

B.1 • A proof in the scalar case

We will assume that $u(t) \in \mathbb{R}$ and comment on how one extends the arguments. Define

$$\underline{U}^n = \begin{bmatrix} U^{n+r-1} \\ \vdots \\ U^{n+1} \\ U^n \end{bmatrix}, \quad f(\underline{U}^n) = \begin{bmatrix} f(U^{n+r-1}) \\ \vdots \\ f(U^{n+1}) \\ f(U^n) \end{bmatrix}.$$

Then the LMM (B.1) can be written as

$$\underline{U}^{n+1} = \underbrace{\begin{bmatrix} -\alpha_{r-1} & -\alpha_{r-2} & \cdots & -\alpha_0 \\ 1 & & & 0 \\ & 1 & & 0 \\ & & \ddots & \vdots \\ & & & 1 & 0 \end{bmatrix}}_A \underline{U}^n + kF(\underline{U}^{n+1}, \underline{U}^n), \quad (\text{B.2})$$

where

$$F(\underline{U}^{n+1}, \underline{U}^n) = [\beta_r e_1^T f(\underline{U}^{n+1}) + [\beta_{r-1} \ \beta_{r-2} \ \cdots \ \beta_0] f(\underline{U}^n)] e_1,$$

and e_1 is the first standard basis vector. For the case where \underline{U}^n is a vector, this would have to be a block matrix with blocks of the same form down the diagonal. We also use the notation

$$\underline{u}(t_n) = \begin{bmatrix} u(t_{n+r-1}) \\ \vdots \\ u(t_n) \end{bmatrix}.$$

The assumption on the LTE implies that

$$\underline{u}(t_{n+1}) = A\underline{u}(t_n) + kF(\underline{u}(t_{n+1}), \underline{u}(t_n)) + k\tau_{\text{LTE}}^n e_1.$$

Now, define

$$\underline{E}^n = \underline{U}^n - \underline{u}(t_n),$$

which satisfies the iteration

$$\underline{E}^{n+1} = A\underline{E}^n + \underbrace{k(F(\underline{U}^{n+1}, \underline{U}^n) - F(\underline{u}(t_{n+1}), \underline{u}(t_n)))}_{k\Delta_F^n} - k\tau_{\text{LMM}}^n e_1.$$

At this point, it might be tempting to write

$$\|\underline{E}^{n+1}\|_2 \leq \|A\|_2 \|\underline{E}^n\|_2 + k\|\Delta_F^n\|_2,$$

but we can see by example that this will not be sufficient. For Adams-Bashforth

$$A = \begin{bmatrix} -1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ & 1 & \ddots & \vdots \\ & & & 1 & 0 \end{bmatrix},$$

which satisfies $\|A\|_2 = \sqrt{2}$. It turns out that this would give the estimate,

$$\|\underline{E}^n\|_2 \leq 2^{n/2} \|\tau_{\text{LMM}}^n\|_\infty.$$

This will not give convergence. This matrix A has eigenvalues in the closed unit disk but the eigenvector matrix has a condition number that is larger than 1, and this causes the problem. So, one could either find a new norm in which to measure things, not the

2-norm, or change the approach slightly. We will take the latter approach. This approach involves working with equalities for as along as possible:

$$\begin{aligned}\underline{E}^0 & \text{ given} \\ \underline{E}^1 & = A \mathbf{e}^0 + k \Delta_F^0, \\ \underline{E}^2 & = A^2 \mathbf{e}^0 + k A \Delta_F^0 + k \Delta_F^1, \\ \underline{E}^3 & = A^3 \mathbf{e}^0 + k A^2 \Delta_F^0 + k A \Delta_F^1 + k \Delta_F^2, \\ & \vdots \\ \underline{E}^n & = A^n \underline{E}^0 + k \sum_{j=0}^{n-1} A^{n-1-j} \Delta_F^j.\end{aligned}$$

Now, to avoid arguments invoking the implicit function theorem, we will assume that $\beta_r = 0$ and the method under consideration is explicit. Then

$$F(\underline{U}^{n+1}, \underline{U}^n) = G(\underline{U}^n) = ([\beta_{r-1} \quad \beta_{r-2} \quad \cdots \quad \beta_0] f(\underline{U}^n)) e_1.$$

Note that if f is Lipschitz with constant L then so is G , in the 2-norm, with constant

$$L' = L \sum_{j=1}^{r-1} |\beta_j|.$$

This implies that

$$\|\Delta_F^j\|_2 \leq L' \|\underline{E}^n\| + \|\tau_{\text{LMM}}^n\|_\infty,$$

where $\|\tau_{\text{LMM}}^n\|_\infty$ gives the largest truncation error that is encountered in the course of the iteration, up to $n = N$. We then arrive at

$$\|\underline{E}^n\|_2 \leq \|A^n\|_2 \|\underline{E}^0\|_2 + k \sum_{j=0}^{n-1} \|A^{n-1-j}\|_2 (L' \|\underline{E}^j\|_2 + \|\tau_{\text{LMM}}^n\|_\infty). \quad (\text{B.3})$$

Now, suppose the method under consideration is zero-stable. We compute, using a cofactor expansion across the first row

$$\begin{aligned}\det(\lambda I - A) & = \det \begin{bmatrix} \lambda + \alpha_{r-1} & \alpha_{r-2} & \cdots & \alpha_0 \\ -1 & \lambda & & 0 \\ & -1 & \lambda & 0 \\ & & \ddots & \vdots \\ & & & -1 & \lambda \end{bmatrix} \\ & = (\lambda + \alpha_{r-1}) \lambda^{r-1} - \alpha_{r-1} (-\lambda^{r-2}) + \alpha_{r-2} \lambda^{r-2} + \cdots \\ & = \rho(\lambda).\end{aligned}$$

Note that we used that $\alpha_r = 1$ in this calculation. Note that zero-stability then implies that all the eigenvalues of A are within the unit disk, $|\lambda| \leq 1$, and if $|\lambda| = 1$ then the eigenvalue is simple. We now prove a simple but important lemma.

Lemma B.1. *Suppose $A \in \mathbb{R}^n$ satisfies:*

- *If λ is an eigenvalue of A then $|\lambda| \leq 1$, and*

- if $|\lambda| = 1$ is an eigenvalue of A then λ is simple.

Then there exists a constant C such

$$\|A^n\|_2 \leq C,$$

for all $n \geq 0$.

Proof. This proof makes use of the ideas in Appendix A. Let $A = R J R^{-1}$ the matrix decomposition into Jordan canonical form. We can order the columns of R so that

$$J = \left[\begin{array}{cccc|c} \lambda_1 & & & & & \\ & \lambda_2 & & & & \\ & & \ddots & & & \\ & & & \lambda_m & & \\ \hline & & & & & J_1 \end{array} \right],$$

where $\lambda_1, \dots, \lambda_m$ are distinct eigenvalues with modulus 1. Then the remaining eigenvalues, the diagonal entries of J_1 , must lie within the disk $D_\epsilon = \{z : |z| \leq 1 - \epsilon\}$ for some $\epsilon > 0$. Then it is clear that

$$J^n = \left[\begin{array}{cccc|c} \lambda_1^n & & & & & \\ & \lambda_2^n & & & & \\ & & \ddots & & & \\ & & & \lambda_m^n & & \\ \hline & & & & & J_1^n \end{array} \right]$$

We estimate $\|J_1^n\|_2$ by considering

$$\begin{aligned} J_1^n &= \frac{1}{2\pi i} \int_{\partial D_\epsilon} z^n (zI - J_1)^{-1} dz, \\ \|J_1^n\|_2 &\leq \frac{|\partial D_\epsilon|}{2\pi} \max_{z \in \partial D_\epsilon} |z^n| \|(zI - J_1)^{-1}\|_2. \end{aligned}$$

We can bound

$$\begin{aligned} \frac{|\partial D_\epsilon|}{2\pi} &< 1, \\ |z^n| &= 1 - \epsilon, \\ \max_{z \in \partial D_\epsilon} \|(zI - J_1)^{-1}\|_2 &= C_3, \end{aligned}$$

where C_3 is some finite constant. There are many such ways to combine these estimates, but one way is to write

$$J^n = \left[\begin{array}{c|c} & J_1^n \end{array} \right] + \left(J^n - \left[\begin{array}{c|c} & J_1^n \end{array} \right] \right),$$

and bound

$$\left\| \left[\begin{array}{c|c} & J_1^n \end{array} \right] \right\|_2 \leq C_3(1 - \epsilon)^n, \quad \left\| J^n - \left[\begin{array}{c|c} & J_1^n \end{array} \right] \right\|_2 = 1.$$

These combine to give

$$\|A^n\| \leq \|R\|_2 \|R^{-1}\|_2 (1 + C_3(1 - \epsilon)^n),$$

which gives the result.

Returning to (B.3), let $n \leq N, kN = T$

$$\begin{aligned}\|\underline{E}^n\|_2 &\leq C\|\underline{E}^0\|_2 + k \sum_{j=0}^{n-1} [CL'\|\underline{E}^j\|_2 + \|\tau_{\text{LMM}}^n\|_\infty] \\ &\leq C\|\underline{E}^0\|_2 + CT\|\tau_{\text{LMM}}^n\|_\infty + kCL' \sum_{j=0}^{n-1} \|\underline{E}^j\|_2.\end{aligned}$$

This does not give us what we need yet as it is only a bound on \underline{E}^n in terms of the previous \underline{E}^j 's. So, define a sequence that dominates $\|\underline{E}^n\|_2$

$$\begin{aligned}Z^0 &= C\|\underline{E}^0\|_2 + CT\|\tau_{\text{LMM}}^n\|_\infty, \\ Z^n &= C\|\underline{E}^0\|_2 + CT\|\tau_{\text{LMM}}^n\|_\infty + kDT \sum_{j=0}^{n-1} Z^j.\end{aligned}$$

By induction $\|\underline{E}^n\|_2 \leq Z^n$. And we find

$$\begin{aligned}Z^{n+1} - Z^n &= kCL'Z^n \Rightarrow Z^{n+1} = (1 + kCL')Z^n, \\ Z^n &= (1 + kCL')^n Z^0, \\ Z^n &\leq e^{nkCL'} Z^0 \leq e^{TCL'} Z_0.\end{aligned}$$

So, this tells us that

$$\max_{0 \leq j \leq N} \|\underline{E}^j\|_2 \leq e^{TDL} (C\|\underline{E}^0\|_2 + CT\|\tau_{\text{LMM}}^n\|_\infty),$$

which proves the (uniform) convergence of the method at the same rate as the LTE.

Remark B.2. *This method also proves the convergence of onestep methods on the same system, by setting $r = 1$. The extension to systems is straightforward following the remarks made here.*

Appendix C

The discrete Fourier transform

The Fourier series for a smooth, periodic function $f : [-\pi, \pi) \rightarrow \mathbb{C}$ is given by

$$f(x) = \frac{1}{\sqrt{2\pi}} \sum_{j=-\infty}^{\infty} c_j(f) e^{ijx},$$

$$c_j(f) = \frac{1}{\sqrt{2\pi}} \int_{-\pi}^{\pi} e^{-ijx} f(x) dx, \quad n = 0, \pm 1, \dots$$

If we, instead, consider $f : [a, b) \rightarrow \mathbb{C}$, we have

$$f(x) = \frac{1}{\sqrt{b-a}} \sum_{j=-\infty}^{\infty} c_j(f; a, b) e^{2ijx \frac{\pi}{b-a}},$$

$$c_j(f; a, b) = \frac{1}{\sqrt{b-a}} \int_a^b e^{-2ijx \frac{\pi}{b-a}} f(x) dx, \quad n = 0, \pm 1, \pm 2, \dots$$

We will, initially, consider $[-\pi, \pi)$ and make generalizations later on.

We compute the coefficients $c_j(f)$ using the trapezoidal rule with N points

$$c_j(f) \approx \check{c}_j(f) := \frac{1}{N\sqrt{2\pi}} \sum_{\ell=1}^n f(\check{x}_\ell) e^{-ij\check{x}_\ell},$$

$$\check{x}_\ell = 2\pi \frac{\ell - 1}{N} - \pi.$$

We note that the dependence of \check{c}_j and \check{x}_j on n is implicit in all that follows. It is important to note that

$$\exp(-ij\check{x}_\ell) = \exp\left(-2\pi i \frac{j}{N}(\ell - 1) + \pi i j\right) = (-1)^j \exp\left(-2\pi i \frac{j}{N}\ell\right)$$

So, if $j = pN + q$, we find

$$\check{c}_j(f) = (-1)^j \check{c}_q(f).$$

And so, it only makes sense to consider N coefficients \check{c}_j . But which ones? Since we aim to compute approximate the sum as j ranges from $-\infty$ to $+\infty$, then it makes sense to

compute something like \check{c}_j for $j = -N/2, N/2 + 1, \dots, N/2$. This is correct if N is odd. But typically, for reasons we will discuss, one uses $N = 2^n$. So, define

$$N_+ = \lfloor N/2 \rfloor, \quad N_- = \lfloor (N-1)/2 \rfloor.$$

Note that if $N = 2M + 1$ is odd then

$$N_+ + N_- + 1 = M + M + 1 = N.$$

And if $N = 2M$ is even

$$N_+ + N_- + 1 = M + M - 1 + 1 = N.$$

We will then use \check{c}_j for $j = -N_-, -N_- + 1, \dots, N_+$. In matrix-vector notation

$$\begin{bmatrix} \check{c}_j \\ j = -N_- \end{bmatrix}_{\ell=1}^{N_+} = \frac{1}{N\sqrt{2\pi}} \begin{bmatrix} \ddots & \vdots & \ddots \\ \cdots & e^{-ij\check{x}_\ell} & \cdots \\ \ddots & \vdots & \ddots \end{bmatrix}_{\ell=1}^N \begin{bmatrix} f(\check{x}_\ell) \\ \ell=1 \end{bmatrix}_{\ell=1}^N$$

The matrix

$$D_N = \begin{bmatrix} \ddots & \vdots & \ddots \\ \cdots & e^{-ij\check{x}_\ell} & \cdots \\ \ddots & \vdots & \ddots \end{bmatrix}_{\ell=1}^N, \quad j = -N_-$$

is called the discrete Fourier transform (DFT) matrix.

C.1 • Approximation properties of the DFT

From the coefficients \check{c}_j , $j = -N_-, \dots, N_+$, we construct an approximation to $f : [-\pi, \pi] \rightarrow \mathbb{C}$ via

$$f(x) \approx \mathcal{I}_N f(x) := \frac{1}{\sqrt{2\pi}} \sum_{j=-N_-}^{N_+} \check{c}_j(f) e^{ijx}.$$

The first, most basic property of \mathcal{I}_N is given by the following proposition and shows that \mathcal{I}_N constructs an interpolation.

Proposition C.1. *For $\ell = 1, 2, \dots, N$,*

$$f(\check{x}_\ell) = \mathcal{I}_N f(\check{x}_\ell).$$

Furthermore, if $f(x) = \sum_{j=-N_-}^{N_+} c_j e^{ijx}$, then $\mathcal{I}_N f = f$.

Proof.

This proposition, while important, says nothing about $|f(x) - \mathcal{I}_N f(x)|$ for $x \neq \check{x}_\ell$. To understand this difference, we introduce the periodic L^2 -based Sobolev spaces.

Definition C.2. The Sobolev space $H^s([-\pi, \pi])$, $s \in \mathbb{R}$, is given by

$$H^s([-\pi, \pi]) := \left\{ f \in L^2([-\pi, \pi]) : \sum_{j=-\infty}^{\infty} |c_j(f)|^2 (1 + |j|)^{2s} < \infty \right\},$$

and the norm is given by

$$\|f\|_{H^s}^2 := \sum_{j=-\infty}^{\infty} |c_j(f)|^2 (1 + |j|)^{2s}.$$

This is simply the space of functions whose Fourier coefficients behave sufficiently well so that the sum is finite. Note that while $s < 0$ is possible, we will not need this case as it no longer corresponds to a space of functions. It is also important to note that if $f \in H^s([-\pi, \pi])$ and $s > 1/2$ then

$$\sum_{j=-\infty}^j |c_j(f)| = \sum_{j=-\infty}^j |c_j(f)| (1 + |j|)^s (1 + |j|)^{-s} \leq \|f\|_{H^s} \sum_{j=-\infty}^{\infty} (1 + |j|)^{-2s}.$$

This shows the Fourier series for f is absolutely summable and thus f must be continuous (why?). The following gives convergence in the H^t norm for $s > 1/2, t < s$.

Theorem C.3. Suppose $s > 1/2$, $t < s$, then there exists a constant $C_{t,s} > 0$ such that

$$\|f - \mathcal{I}_N f\|_{H^t} \leq N^{t-s} C_{t,s} \|f\|_{H^s}.$$

Proof. TBD To further put this result in context, we have the following, which is an example of a Sobolev embedding result.

Theorem C.4. Suppose $f \in H^{1/2+r+\epsilon}([-\pi, \pi])$ for $\epsilon > 0$ and $r \in \mathbb{N}$. Then f can be taken to be r times continuously differentiable and there exists $A_{r,\epsilon} > 0$ such that

$$\max_{0 \leq j \leq r} \|f^{(j)}\|_{\infty} \leq A_{r,\epsilon} \|f\|_{H^{1/2+r+\epsilon}}.$$

This implies that convergence in Sobolev spaces implies convergence of derivatives.

Lastly, we can ask about how well the DFT approximates the true Fourier coefficients. One result can be derived directly from Theorem C.3.

Corollary C.5. Suppose $f \in H^s([-\pi, \pi])$ for $s > 1/2$. Then

$$|\check{c}_j(f) - c_j(f)| \leq N^{-s} C_{0,s} \sqrt{2\pi} \|f\|_{H^s}.$$

Proof. By the Cauchy-Schwarz inequality

$$|\tilde{c}_j(f) - c_j(f)| = \left| \frac{1}{\sqrt{2\pi}} \int_{-\pi}^{\pi} (f(x) - \mathcal{I}_N f(x)) e^{-i j x} dx \right| \leq \|f - \mathcal{I}_N f\|_2 \leq N^{-s} C_{0,s} \|f\|_{H^s}.$$

Bibliography

- [CLT55] E A Coddington, Norman Levinson, and T. Teichmann, *Theory of Ordinary Differential Equations*, McGraw-Hill, Inc., New York, USA, 1955.
- [Dah56] G Dahlquist, *Convergence and stability in the numerical integration of ordinary differential equations*, MATHEMATICA SCANDINAVICA **4** (1956), 33.
- [Eva10] L C Evans, *Partial differential equations*, second ed., Graduate Studies in Mathematics, vol. 19, American Mathematical Society, Providence, RI, 2010.
- [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.