Advanced Topics in Numerical Analysis

Thomas Trogdon
University of Washington
trogdon@uw.edu

Contents

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 .+ scal
```

Instead, one needs to use `.+`:

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

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

```
α = 1.
```

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

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

0.2 ▪ Installing packages

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

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

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

```
add NewPackage
```

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

0.3 ▪ Loading packages

To load `NewPackage.jl` simply enter

```
using NewPackage
```

0.4 ▪ Loops and conditionals

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

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

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

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

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

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

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

```
10
```

JULIA allows you to loop through an array as well.

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

```
0.12998325979929964
```

If statements are nearly the same as in MATLAB.

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

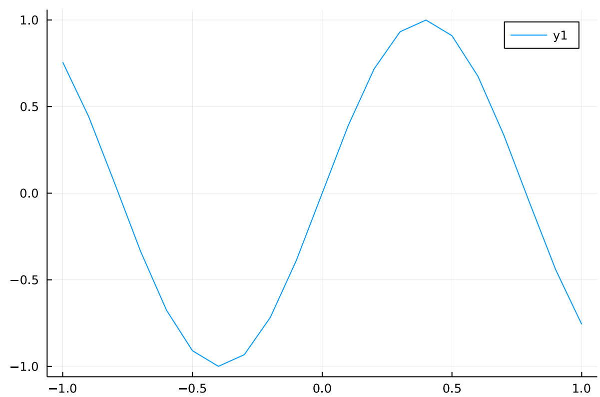
0.5 ▪ Plotting

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

```
using Plots
```

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

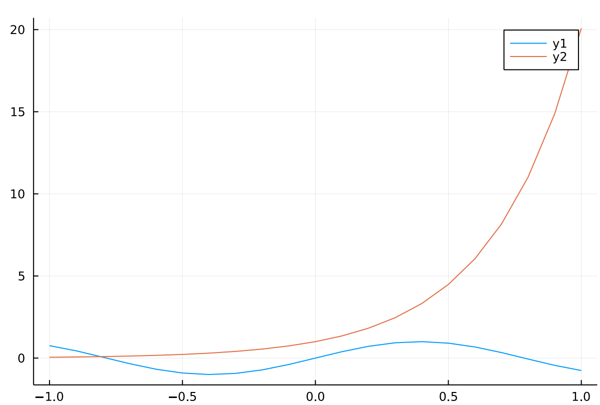
```
x = -1:0.1:1
f = x -> sin.(4x)
p = plot(x,f(x))
```




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

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

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



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

Part I

Numerical linear algebra

Part II

Approximation theory

Part III

Numerical solution of evolution problems

Chapter 1

Review of the theory of ordinary differential equations

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

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

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

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

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

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

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

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

Define

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

Then we have

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

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

Assemble the vector

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

Then

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

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

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

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

Example 1.3. Consider

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

Define

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

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

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

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

1.1.1 ▪ The matrix exponential

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

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

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

Example 1.4.

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

Example 1.5.

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

Three important properties of the matrix exponential are

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

We now use the matrix exponential to solve the IVP

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

The main calculation we make here is that

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

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

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

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

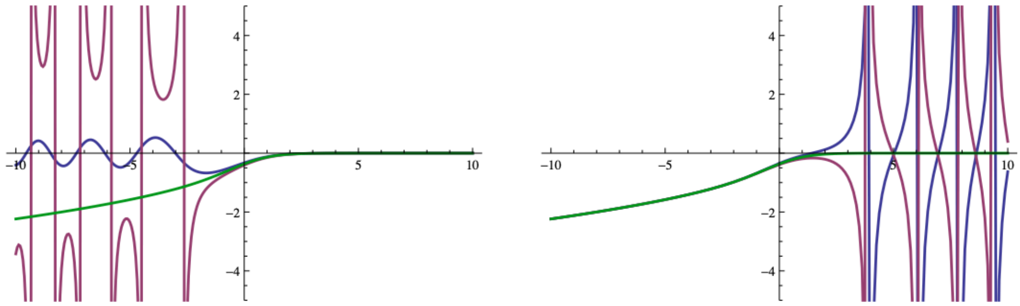
1.1.2 ▪ A cautionary tale in ODE theory

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

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

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

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



(a) (b)

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.

1.1.3 ▪ ODE existence and uniqueness theory

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

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

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

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

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

if

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The chain rule implies

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

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

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

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

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

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

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

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

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

for

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

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:

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

From this we find that

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

Solving for u, we find

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

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

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

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

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

which is valid for all t.

Exercise 1.11. *Suppose the ODE system*

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

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

1.1.4 ▪ The importance of the Lipschitz constant

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

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

We now want to see how the difference evolves by computing

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

Therefore

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

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

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

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

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

So, this is a decreasing function and we find that

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

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

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

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

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

Chapter 2

Numerical solution of ordinary differential equations

2.1 ▪ The Euler methods

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

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

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

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

for a sequence of times

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

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

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

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

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

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

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

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

Thus, we arrive at

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

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

2.2 ▪ Newton's method

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

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

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

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

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

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

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

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

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

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

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

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

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

Using Taylor's theorem

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

This gives

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

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

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

We are left with the relation

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

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

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

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

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

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

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

This implies the theorem.

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

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

2.3 ▪ A nonlinear ODE with the Euler methods

Consider

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

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

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

So,

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

First, define `f`.

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

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

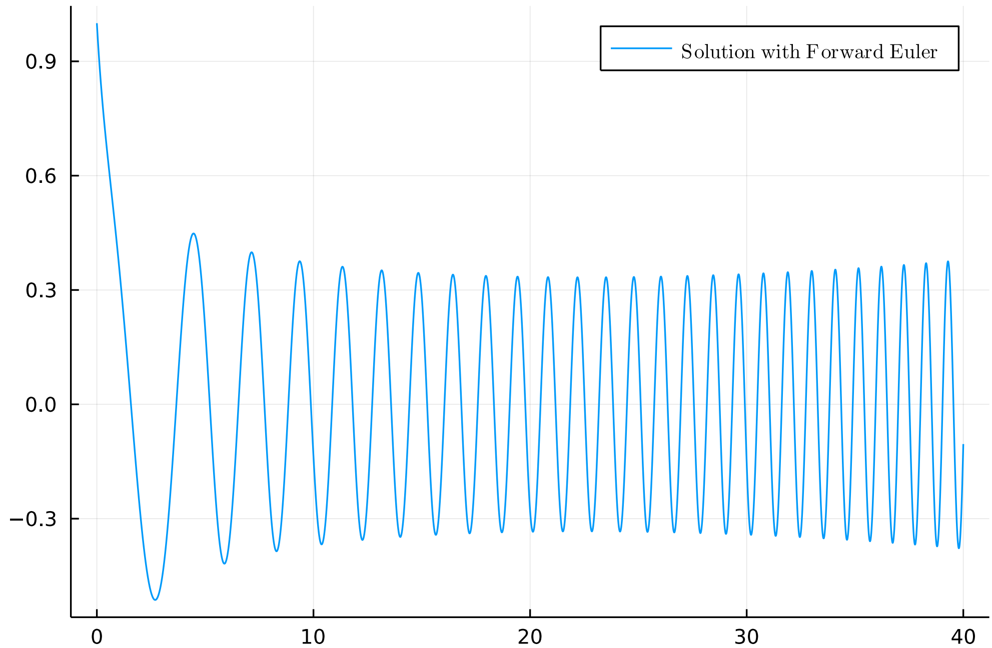
```
T = 40.# Final time.
k = 0.001 # Step size
```

The implementation of forward Euler is straighforward.

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

The result is then plotted.

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



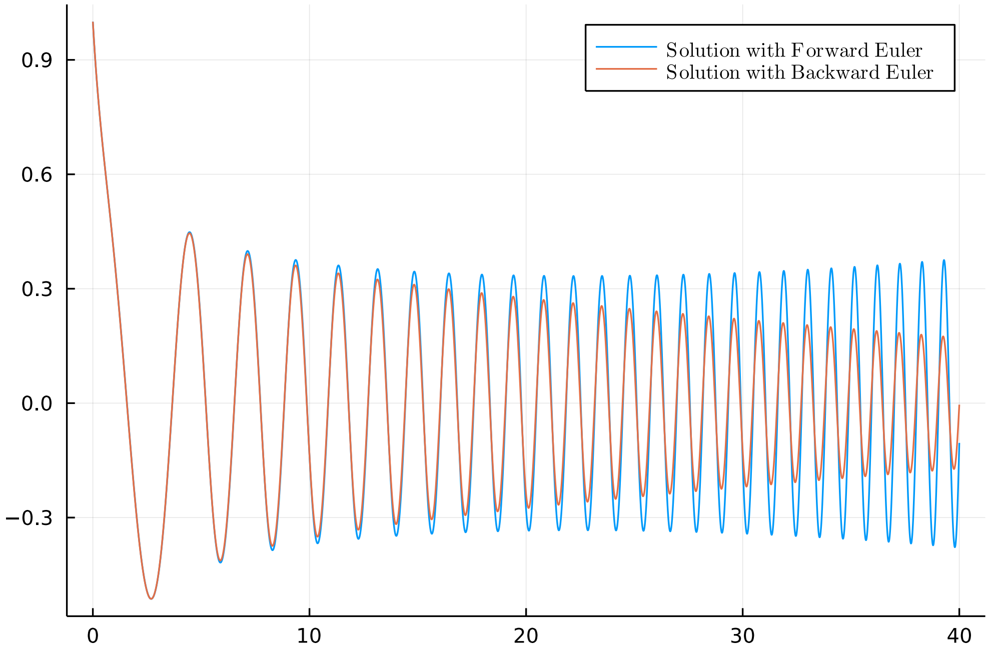
The implementation of backward Euler is more involved, but not too much because we have `Newton!` defined already. Set up the function `g` that we will find the roots of:

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

Then we simply run the iteration.

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

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


Solution with Forward Euler
Solution with Backward Euler
0.9
0.6
0.3
0.0
−0.3
0
10
20
30
40

Bibliography

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

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

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