Introduction to Numerical Methods and Analysis with Julia (draft)

Brenton LeMesurier (College of Charleston, South Carolina) with

Nov 22, 2022

CONTENTS

Brenton LeMesurier College of Charleston, Charleston, South Carolina lemesurierb@cofc.edu,

with contributions by Stephen Roberts (Australian National University)

Last revised November 22, 2022

Recent changes:

- Improved formating of definitions, theorems, proofs, examples, remarks, etc. (ongoing!)
- Created a small bibliography using a BiBTeX file: see https://jupyterbook.org/en/stable/tutorials/references.html
- Added a draft section *Piecewise Polynomial Approximating Functions: Splines and Hermite Cubics*, based on notes by Stephen Roberts.
- Added a draft section *Choosing the collocation points: the Chebyshev method*, based on notes by Stephen Roberts.
- Added notes on the stiff limit $K = D \gg 1$ of the damped mass-spring system (use this for future stiff-stability examples.)
- Expanded the notes on Adams-Bashforth methods, and revised somewhat the notes on leapfrog.
- Added first drafts of some sections on multi-step methods: leapfrog, Adams-Bashforth methods and Adams-Moulton methods.

This Jupyter book is a sibling of "… with Python", both of which are based on Elementary Numerical Analysis with Python, my notes for the course *Elementary Numerical Analysis* at the University of Northern Colorado in Spring 2021.)

The material is based in part on Jupyter notebooks and other materials for the courses MATH 245, MATH 246, MATH 445 and MATH 545 at the College of Charleston, South Carolina, and MATH 375 at the University of Northern Colorado.

To do:

- Add the basics of boundary value probems for ODEs, with both finite diference and (finite) element mthods.
- Positive definiite matrices and the Cholesky factorization.
- More on minimization
- Predicror-corrector methods for ODE IVP's.
- Stability analysis for multi-step methods.
- Continue updating the formating of definitions, theorems, proofs, examples, remarks, etc. and MyST notation for references, from *An Introduction to Multistep Methods* onward. For details https://jupyterbook.org/en/stable/tutorials/references.html or here in the Jupyterbook documentation. For example
  - label a section heading with preceeding line "(label-text)="
  - link to such a label with "{ref}(label-text)" or "`[link text](label-text)`"
- More exercises
- Gather exercises at the end of each section, with links to them from their current location.
- Expand the notes on "Julia for people who know Matlab or Python", *Notes on the Julia Language*

This is published at http://lemesurierb.people.cofc.edu/introduction-to-numerical-methods-and-analysis-julia/

and its Python sibling at http://lemesurierb.people.cofc.edu/introduction-to-numerical-methods-and-analysis-python/

About Julia

On the topic of programming, this book tries to address two audiences: people who already know Julia (only a basic familiarity is needed), and people who are familiar with either Matlab or "Python Scientific" (Python + NumPy + Matplotlib) and are Julia-curious.

Julia is designed with a fair degree of backward compatability with Matlab, and NumPy and Matplotlib often mimic Matlab syntax too, so the jump is not so big.

To help, there are two appendices

- *Installing Julia and some useful add-ons* and
- *Notes on the Julia Language*

Some Further Reading on Numerical Methods and Analysis

- [Sauer, 2019] *Numerical Analysis* by Timothy Sauer, 2nd or 3rd edition.
- [Burden *et al.*, 2016] *Numerical Analysis* by Richard L. Burden and J. Douglas Faires, 9th edition.
- [Chenney and Kincaid, 2012] *Numerical Mathematics and Computing* by Ward Chenney and David Kincaid.
- [Kincaid and Chenney, 1990] *Numerical Analysis* by David Kincaid and Ward Chenney.

CHAPTER

ONE

INTRODUCTION

This book addresses the design and analysis of methods for computing numerical values for solutions to mathematical problems. Most often, only accurate approximations are possible rather than exact solutions, so a key mathematical goals is to assess the accuracy of such approximations.

Given that most numerical methods allow any degree of accuracy to be achieved by working hard enough, the next level of analysis is assessing *cost*, or equivalently *speed*, or more generally the *efficiency of resource usage*. The most natural question then is how much time and other resources are needed to achieve a given degree of accuracy.

1.1 Topics

The main areas of interest are:

1. Finding the zeros of a function: solving $f(x) = 0$.
2. Solving systems of simultaneous linear equations; in matrix-vector notation, solving $Ax = b$ for x.
3. Fitting polynomials to a collection of data points, either exactly (collocation) or approximately (by least-squares).
4. Approximating a function by a polynomial, or several polynomials.
5. Approximating derivatives and definite integrals.
6. Finding the minimum of a function.
7. Solving initial value problems for ordinary differential equations.

Although it is the last major topic, the numerical solution of differential equations will often be mentioned earlier as a motivation for other topics. However, we start in a simpler setting: the problem of finding the zeros of a real-valued function: solving $f(x) = 0$.

1.2 Julia: a New Alternative to Matlab and Python

The two dominant dynamic programming language environments for numerical computing and graphics are

- the open-source combination of Python with Numpy, Matplotlib, SciPy and other packages, sometime claled *Python Scientific*, and
- the older, commercial product Matlab(TM).

Though I generally recommend the former, Matlab is long-established, especially in engineering fields, and has advantages in a few respects, like notation for working with matrices and vectors that is cleaner and closer to standard mathematical notation.

One more recent innovation is the Julia language, which for one thing combines many of the virtues of both:

- Like Python, it is open source with some nice modern programming language features (some newer and improving on Python), while also being
- more compatability in its syntax and notation with existing Matlab code and standard linear algebra notation.
- Also, Julia code usually runs faster that either Python or Matlab.

P.S. What does *Jupyter* mean?

"JuPyteR" is a portmanteau of the names of three important modern open-source programming tools for scientific computing:

- Julia,
- Python, and
- R (which is for statistical computing).

So in particular, Jupyter notebooks and Jupyter books work fine with Julia, and a Jupyter book can use a mix of different languages in different sections.

CHAPTER

TWO

ROOT-FINDING

2.1 Root Finding by Interval Halving (Bisection)

References:

- Section 1.1 *The Bisection Method* in *Numerical Analysis* by Sauer [Sauer, 2019]
- Section 2.1 *The Bisection Method* in *Numerical Analysis* by Burden, Faires and Burden [Burden *et al.*, 2016]

(See the *Bibliography*.)

2.1.1 Introduction

One of the most basic tasks in numerical computing is finding the roots (or "zeros") of a function — solving the equation $f(x) = 0$ where $f : \mathbb{R} \to \mathbb{R}$ is a continuous function from and to the real numbers. As with many topics in this course, there are multiple methods that work, and we will often start with the simplest and then seek improvement in several directions:

- **reliability** or *robustness* — how good it is at avoiding problems in hard cases, such as division by zero.
- *accuracy* and guarantees about accuracy like estimates of how large the error can be — since in most cases, the result cannot be computed exactly.
- *speed* or *cost* — often measured by minimizing the amount of arithmetic involved, or the number of times that a function must be evaluated.

We use the package PyPlot; see the notes on plotting graphs and on using package and modules in *Notes on the Julia Language*.

```
using PyPlot
```

Example 2.1 (Solve $x = \cos x$)

This is a simple equation for which there is no exact formula for a solution, but we can easily ensure that there is a solution, and moreover, a unique one. It is convenient to put the equation into "zero-finding" form $f(x) = 0$, by defining

$$f(x) := x - \cos x.$$

Also, note that $|\cos x| \leq 1$, so a solution to the original equation must have $|x| \leq 1$. So we will start graphing the function on the interval $[a, b] = [-1, 1]$.

```
f(x) = x - cos(x);
```

Remark 2.1 (On Julia)

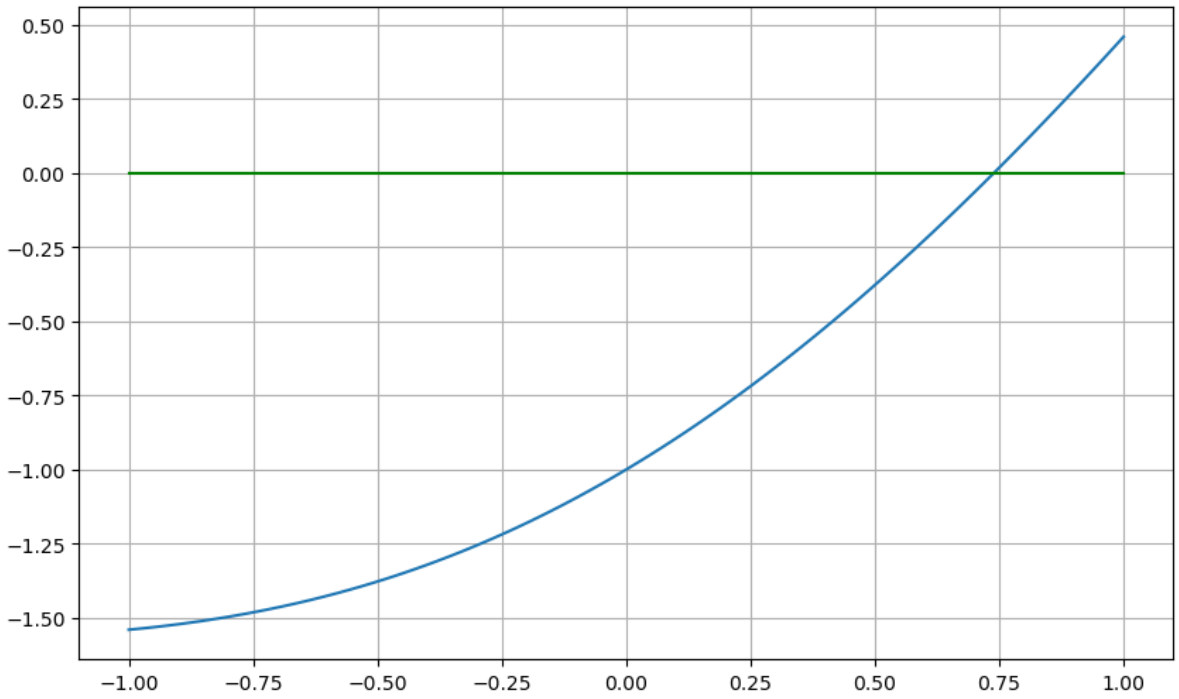
For notes on this compact version of Julia function syntax, see Functions, part 1 in *Notes on the Julia Language*.

```
a = -1.0
b = 1.0;
```

Remark 2.2 (On Julia)

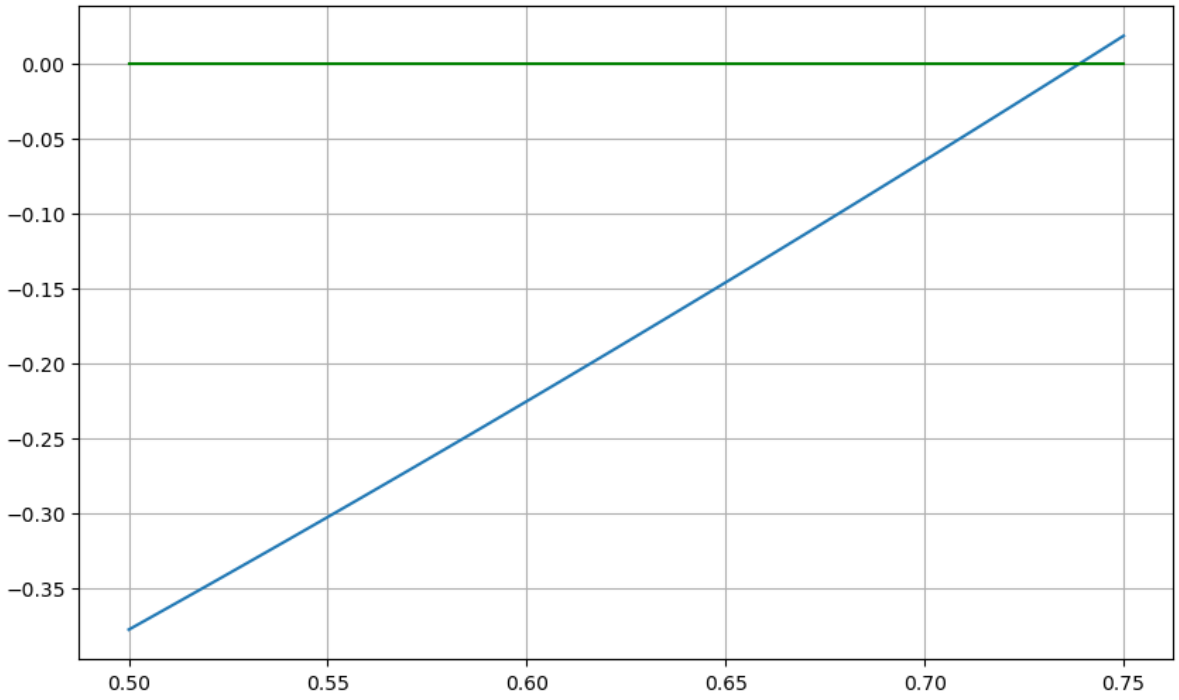
See the notes on the function range used below in *Notes on the Julia Language*.

```
x = range(a, b, 100)
figure(figsize=[10,6])
plot(x, f.(x));
plot([a, b], [0, 0], "g");  # Mark the x-axis in green
grid(true)  # Add a graph paper background
```



This shows that the zero lies between 0.5 and 0.75, so zoom in:

```
a = 0.5
b = 0.75
x = range(a, b, 100)
figure(figsize=[10,6])
plot(x, f.(x))
plot([a, b], [0, 0], "g")
grid(true)
```



And we could repeat, geting an approximation of any desired accuracy.

However this has two weaknesses: it is very inefficient (the function is evaluated about fifty times at each step in order to draw the graph), and it requires lots of human intervention.

To get a procedure that can be efficiently implemented in Julia (or another programming language of your choice), we extract one key idea here: finding an interval in which the function changes sign, and then repeatedly find a smaller such interval within it. The simplest way to do this is to repeatedly divide an interval known to contain the root in half and check which half has the sign change in it.

Graphically, let us start again with interval $[a, b] = [-1, 1]$, but this time focus on three points of interest: the two ends and the midpoint, where the interval will be bisected:

```
a = -1.0
b = 1.0
c = (a+b)/2
println("a=$a, b=$b, c=$c")
```

```
a=-1.0, b=1.0, c=0.0
```

Remark 2.3 (On Julia)

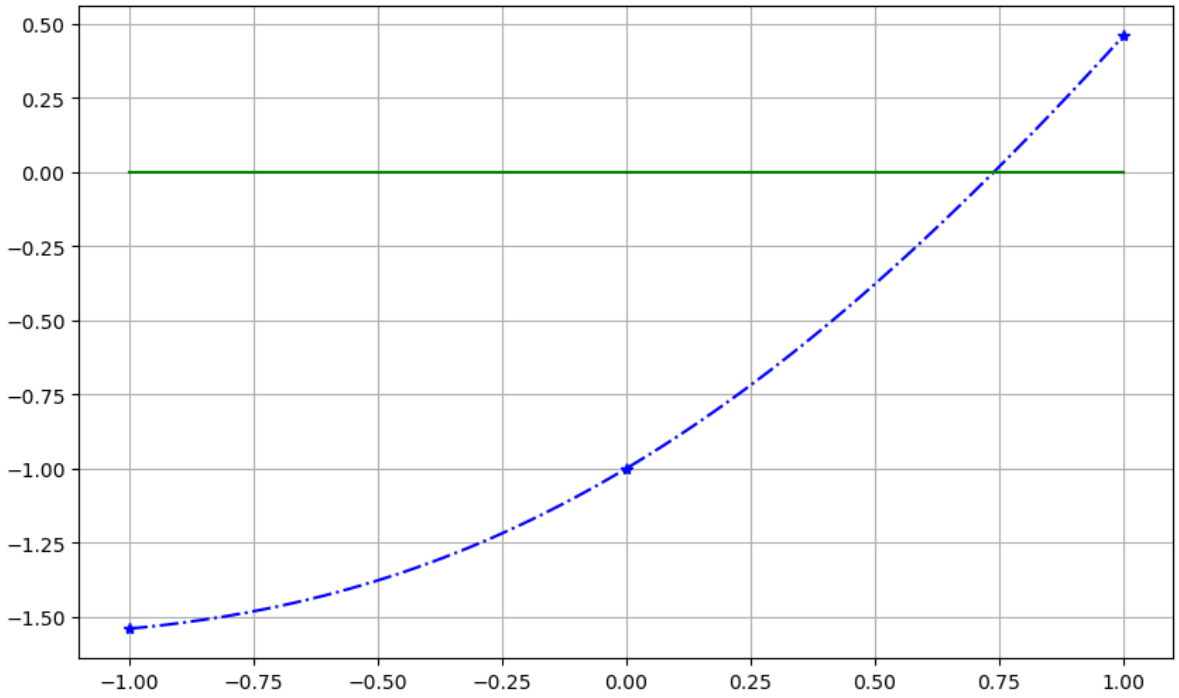
If you are unfamiliar with `println` see the notes on Displaying values in *Notes on the Julia Language*.

```
acb = [a c b]
figure(figsize=[10,6])
plot(acb, f.(acb), "b*")
# And just as a visual aid:
x = range(a, b, 100)
plot(x, f.(x), "b-.")
```

(continues on next page)

(continued from previous page)

```
plot([a, b], [0, 0], "g")
grid(true)
```

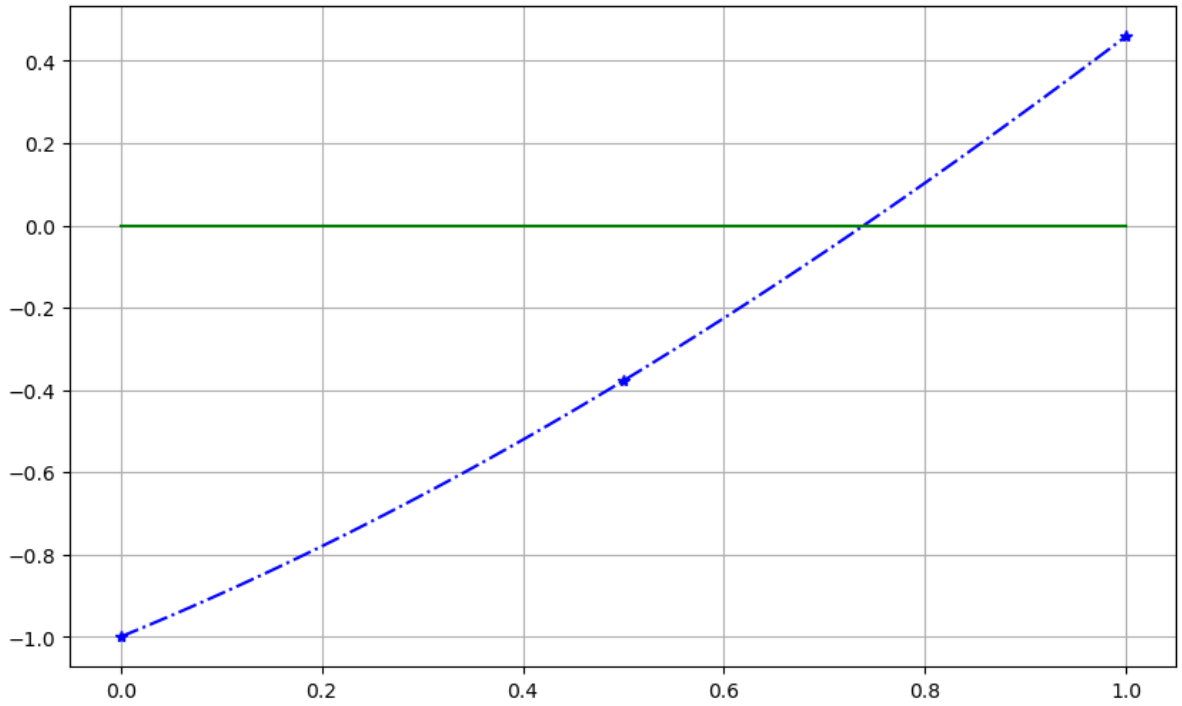


$f(a)$ and $f(c)$ have the same sign, while $f(c)$ and $f(b)$ have opposite signs, so the root is in $[c, b]$; update the a, b, c values and plot again:

```
a = c  # new left end is old center
b = b  # redundant, as the right end is unchanged
c = (a+b)/2
println("a=$a, b=$b, c=$c")
```

```
a=0.0, b=1.0, c=0.5
```

```
acb = [a c b]
figure(figsize=[10,6])
plot(acb, f.(acb), "b*")
x = range(a, b, 100)
plot(x, f.(x), "b-.")
plot([a, b], [0, 0], "g")
grid(true)
```

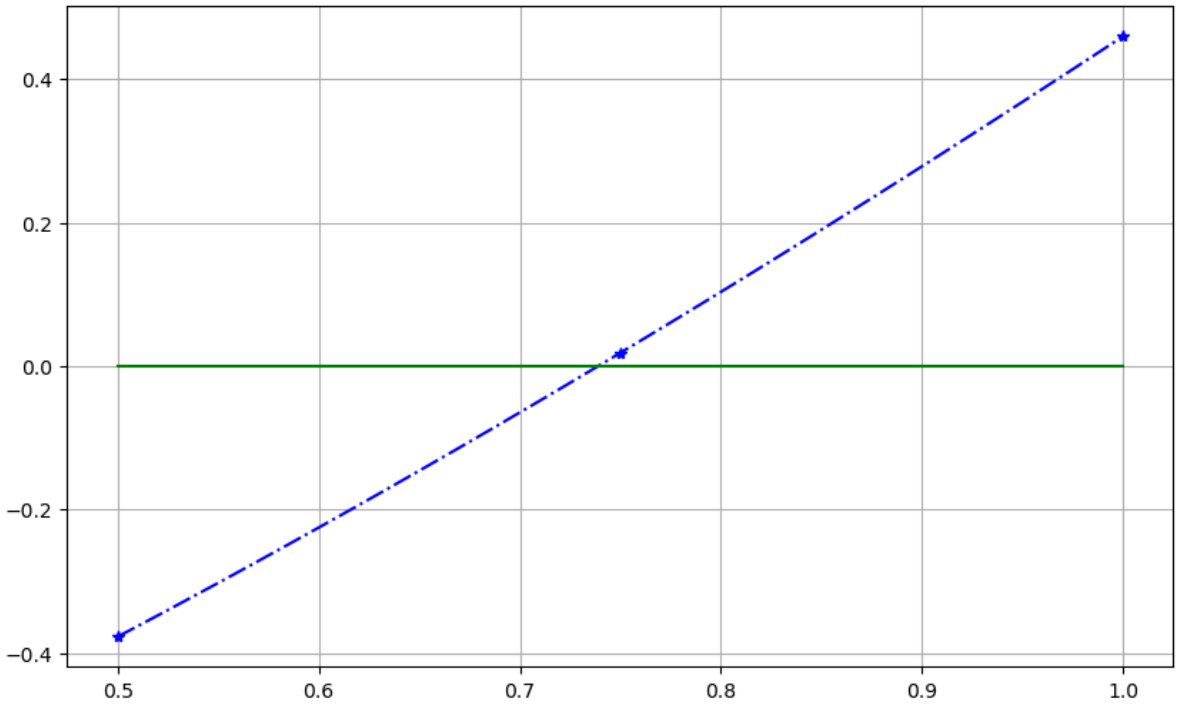


Again $f(c)$ and $f(b)$ have opposite signs, so the root is in $[c, b]$, and …

```
a = c  # new left end is old center again
# skipping the redundant "b = b" this time
c = (a+b)/2
println("a=$a, b=$b, c=$c")
```

```
a=0.5, b=1.0, c=0.75
```

```
acb = [a c b]
figure(figsize=[10,6])
plot(acb, f.(acb), "b*")
x = range(a, b, 100)
plot(x, f.(x), "b-.")
plot([a, b], [0, 0], "g")
grid(true)
```

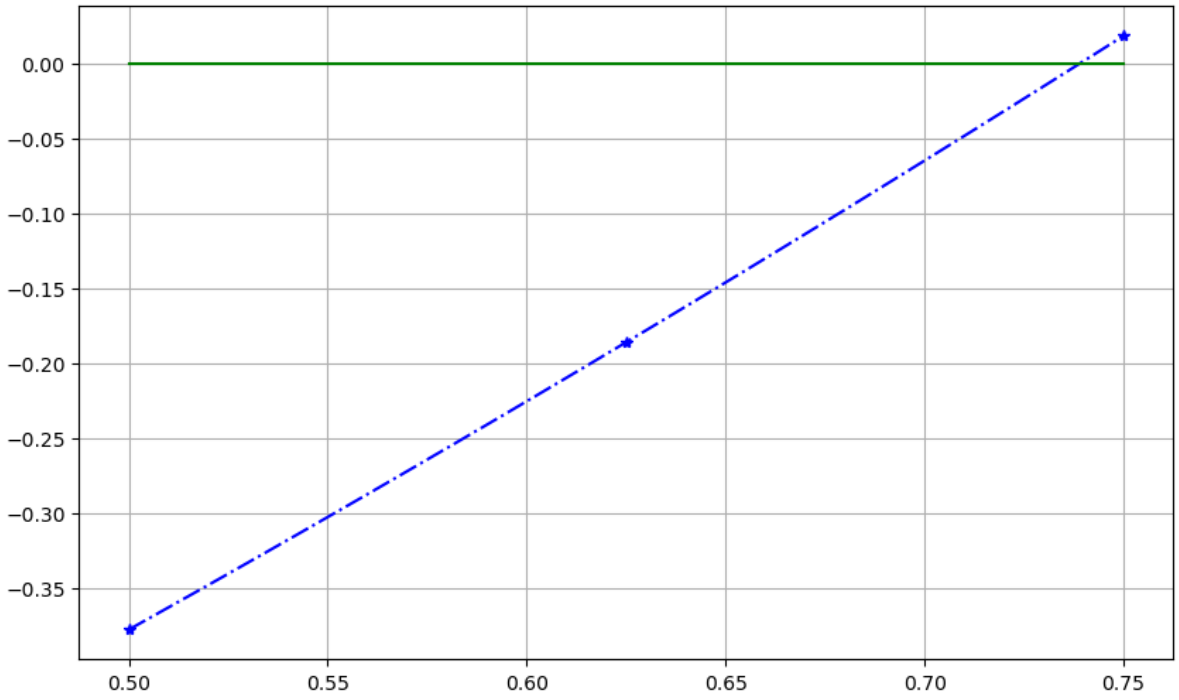


This time $f(a)$ and $f(c)$ have opposite sign, so the root is at left, in $[a, c]$:

```
# this time, the value of a does not need to be updated ...
b = c  # ... and the new right end is the former center
c = (a+b)/2
println("a=$a, b=$b, c=$c")
```

```
a=0.5, b=0.75, c=0.625
```

```
acb = [a c b]
figure(figsize=[10,6])
plot(acb, f.(acb), "b*")
x = range(a, b, 100)
plot(x, f.(x), "b-.")
plot([a, b], [0, 0], "g")
grid(true)
```



2.1.2 A first algorithm for the bisection method

Now it is time to dispense with the graphs, and describe the procedure in mathematical terms:

- if $f(a)$ and $f(c)$ have opposite signs, the root is in interval $[a, c]$, which becomes the new version of interval $[a, b]$.
- otherwise, $f(c)$ and $f(b)$ have opposite signs, so the root is in interval $[c, b]$

Pseudo-code for describing algorithms

As a useful bridge from the mathematical desciption of an algorithm with words and formulas to actual executable code, these notes will often describe algorithms in *pseudo-code* — a mix of words and mathematical formulas with notation that somewhat resembles code in a language like Julia.

This is also preferable to going straight to code in a particular programming language (such as Julia) because it makes it easier if, later, you wish to implement algorithms in a different language.

Note well one feature of the pseudo-code used here: **assignment** is denoted with a left arrow:

$x \leftarrow a$

is the instruction to cause the value of variable `x` to become the current value of a.

This is to distinguish from

$x = a$

which is a **comparison**: the true-or-false assertion that the two quantities *already* have the same value.

Unfortunately however, Julia (like most programming languages) does not use this notation: instead assignment is done with `x = a` so that asserting equality needs a differnt notation: this is done with `x == a`; note well that double equal sign!

With that notational issue out of the way, the key step in the bisection strategy is the update of the interval:

Algorithm 2.1 (one step of bisection)

$c \leftarrow \dfrac{a+b}{2}$ if $f(a)f(c) < 0$ then $b \leftarrow c$ else $a \leftarrow c$ end

This needs to be repeated a finite number of times, and the simplest way is to specify the number of iterations. (We will consider more refined methods soon.)

Algorithm 2.2 (bisection, first version)

- Get an initial interval $[a, b]$ with a sign-change: $f(a)f(b) < 0$.
- Choose N, the number of iterations.
- for i from 1 to N $c \leftarrow \dfrac{a+b}{2}$ if $f(a)f(c) < 0$ then $b \leftarrow c$ else: $a \leftarrow c$ end end
- The approximate root is the final value of c.

A Julia version of the iteration is not a lot different:

```
for i in 1:N
    c = (a+b)/2
    if f(a) * f(c) < 0
        b = c
    else
        a = c
    end
end
```

Remark 2.4 (On Julia)

See the notes on Iteration and Conditionals on the syntax seen here for first time.

See *Exercise A*.

2.1.3 Error bounds, and a more refined algorithm

The above method of iteration for a fixed number of times is simple, but usually not what is wanted in practice. Instead, a better goal is to get an approximation with a guaranteed maximum possible error: a result consisting of an approximation $\tilde{r}$ to the exact root r and also a bound E_{max} on the maximum possible error; a guarantee that $|r - \tilde{r}| \leq E_{max}$. To put it another way, a guarantee that the root r lies in the interval $[\tilde{r} - E_{max}, \tilde{r} + E_{max}]$.

In the above example, each iteration gives a new interval $[a, b]$ guaranteed to contain the root, and its midpoint $c = (a + b)/2$ is with a distance $(b - a)/2$ of any point in that interval, so at each iteration, we can have:

- $\tilde{r}$ is the current value of $c = (a + b)/2$
- $E_{max} = (b - a)/2$

2.1.4 Error tolerances and stopping conditions

The above algorthm can *passively* state an error bound, but it is better to be able to solve to a desired degree of accuracy; for example, if we want a result "accurate to three decimal places", we can specify $E_{max} \leq 0.5 \times 10^{-3}$.

So our next goal is to *actively* set an accuracy target or *error tolerance* E_{tol} and keep iterating until it is met. This can be achieved with a `while` loop; here is a suitable algorithm:

Algorithm 2.3 (bisection with error tolerance)

- Input function f, interval endpoints a and b, and an error tolerance E_{tol}
- Evaluate $E_{max} = (b-a)/2$
- while $E_{max} > E_{tol}$: $c \leftarrow (a+b)/2$ if $f(a)f(c) < 0$ then $b \leftarrow c$ else $a \leftarrow c$ end $E_{max} \leftarrow (b-a)/2$ end
- Output $\tilde{r} = c$ as the approximate root and E_{max} as a bound on its absolute error.

2.1.5 Exercises

Exercise A

Create a Julia function `bisection1` which implements the first algorithm for bisection above, which performs a fixed number N of iterations; the usage should be: `root = bisection1(f, a, b, N)`

Test it with the above example: $f(x) = x - \cos x = 0$, $[a, b] = [-1, 1]$

Julia newcomers: see the notes introducing Julia Functions.

Exercise B

Create a Julia function implementing this better algorithm, with usage `root = bisection2(f, a, b, E_tol)`

Test it with the above example: $f(x) = x - \cos x$, $[a, b] = [-1, 1]$, this time accurate to within 10^{-4}.

Use the fact that there is a solution in the interval $(-1, 1)$.

2.2 Solving Equations by Fixed Point Iteration (of Contraction Mappings)

References:

- Section 1.2 *Fixed-Point Iteration* of [Sauer, 2019]
- Section 2.2 *Fixed-Point Iteration* of [Burden *et al.*, 2016]

2.2.1 Introduction

In the next section we will meet *Newton's Method for Solving Equations* for root-finding, which you might have seen in a calculus course. This is one very important example of a more general strategy of **fixed-point iteration**, so we start with that.

```
using PyPlot
```

2.2.2 Fixed-point equations

A variant of stating equations as *root-finding* ($f(x) = 0$) is *fixed-point* form: given a function $g : \mathbb{R} \to \mathbb{R}$ or $g : \mathbb{C} \to \mathbb{C}$ (or even $g : \mathbb{R}^n \to \mathbb{R}^n$; a later topic), find a *fixed point* of g. That is, a value p for its argument such that

$$g(p) = p$$

Such problems are interchangeable with root-finding. One way to convert from $f(x) = 0$ to $g(x) = x$ is functionining

$$g(x) := x - w(x)f(x)$$

for any "weight function" $w(x)$.

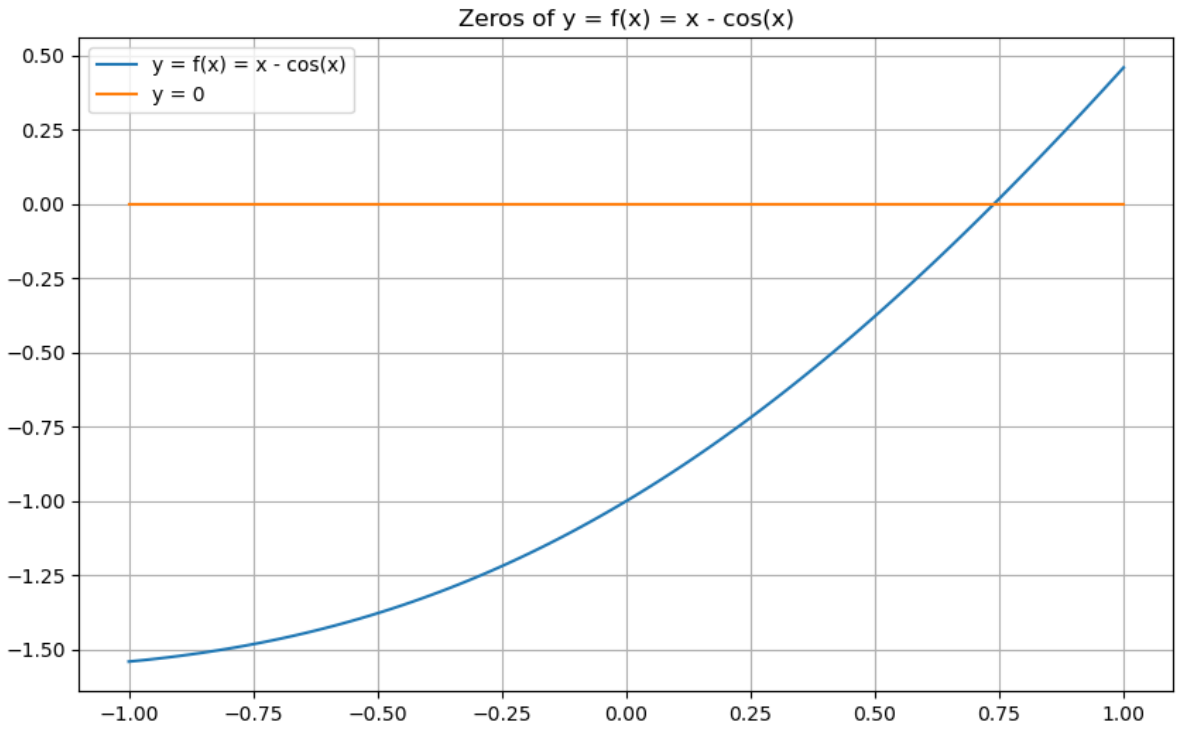
One can convert the other way too, for example functionining $f(x) := g(x) - x$. We have already seen this when we converted the equation $x = \cos x$ to $f(x) = x - \cos x = 0$.

Compare the two setups graphically: in each case, the x value at the intersection of the two curves is the solution we seek.

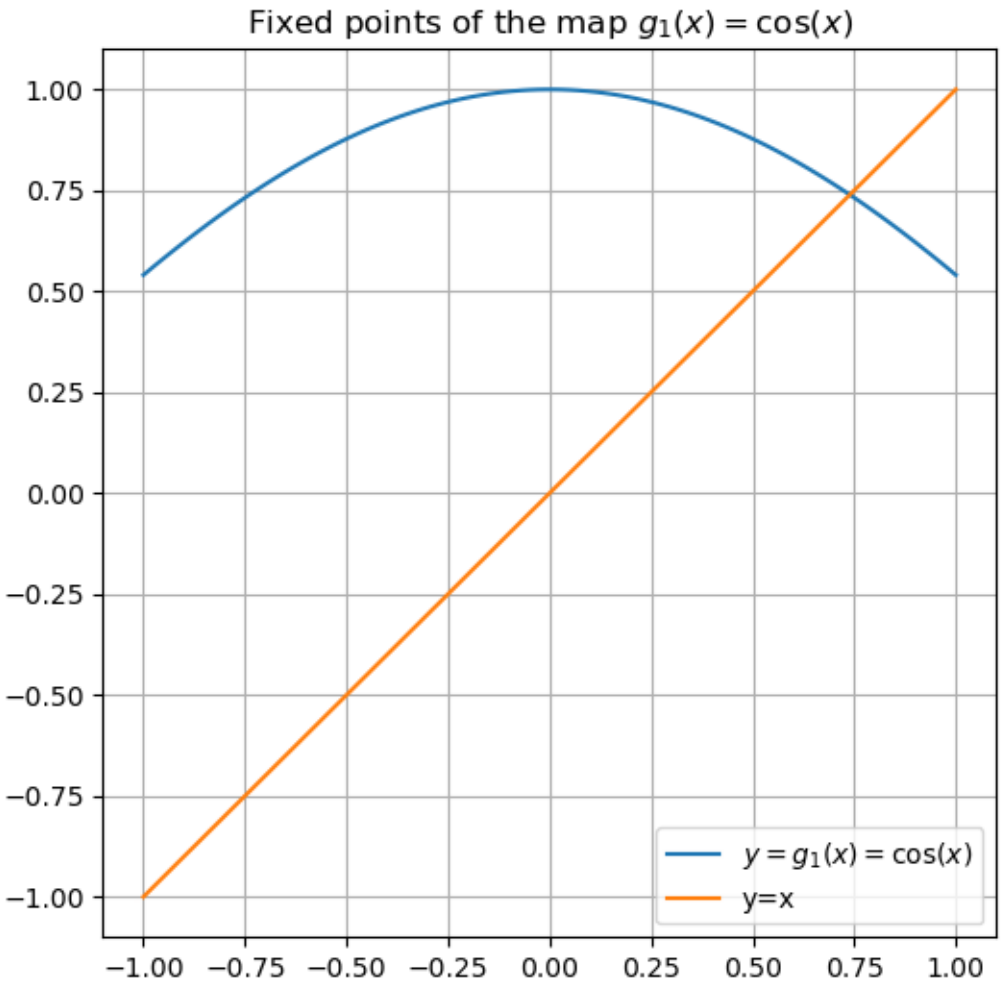
```
f_1(x) = x - cos(x)
g_1(x) = cos(x);
```

```
a = -1.0
b = 1.0
x = range(a, b, 100);
```

```
figure(figsize=[10,6])
title("Zeros of y = f(x) = x - cos(x)")
plot(x, f_1.(x), label="y = f(x) = x - cos(x)")
plot([a, b], [0, 0], label="y = 0")
legend()
grid(true)
```




```
figure(figsize=[6,6])
title(L"Fixed points of the map $g_1(x) = \cos(x)$")
plot(x, g_1.(x), label=L"y = g_1(x) = \cos(x)")
plot(x, x, label="y=x")
legend()
grid(true);
```



The fixed point form can be convenient partly because we almost always have to solve by successive approximations, or *iteration*, and fixed point form suggests *one* choice of iterative procedure: start with any first approximation x_0, and iterate with

$$x_1 = g(x_0),\ x_2 = g(x_1), \dots, x_{k+1} = g(x_k), \dots$$

Proposition 2.1

If g is continuous, and **if** the above sequence $\{x_0, x_1, \dots\}$ converges to a limit p, then that limit is a fixed point of function g: $g(p) = p$.

Proof. From $\lim_{k\to\infty} x_k = p$, continuity gives

$$\lim_{k\to\infty} g(x_k) = g(p).$$

On the other hand, $g(x_k) = x_{k+1}$, so

$$\lim_{k\to\infty} g(x_k) = \lim_{k\to\infty} x_{k+1} = p.$$

Comparing gives $g(p) = p$.

That second "if" is a big one. Fortunately, it can often be resolved using the idea of a *contraction mapping*.

Definition 2.1 (Mapping)

A function $g(x)$ defined on a closed interval $D = [a, b]$ which sends values back into that interval, $g : D \to D$, is sometimes called a *map* or *mapping*.

(*Aside:* The same applies for a function $g : D \to D$ where D is a subset of the complex numbers, or even of vectors $\mathbb{R}^n$ or $\mathbb{C}^n$.)

A mapping is sometimes thought of as moving a region S within its domain D to another such region, by moving each point $x \in S \subset D$ to its image $g(x) \in g(S) \subset D$.

A very important case is mappings that shrink the region, by reducing the distance between points:

Proposition 2.2

Any continuous mapping on a closed interval $[a, b]$ has at least one fixed point.

Proof. Consider the "root-finding cousin", $f(x) = x - g(x)$.

First, $f(a) = a - g(a) \leq 0$, since $g(a) \geq a$ so as to be in the domain $[a, b]$ — similarly, $f(b) = b - g(b) \geq 0$.

From the Intermediate Value Theorem, f has a zero p, where $f(p) = p - g(p) = 0$.

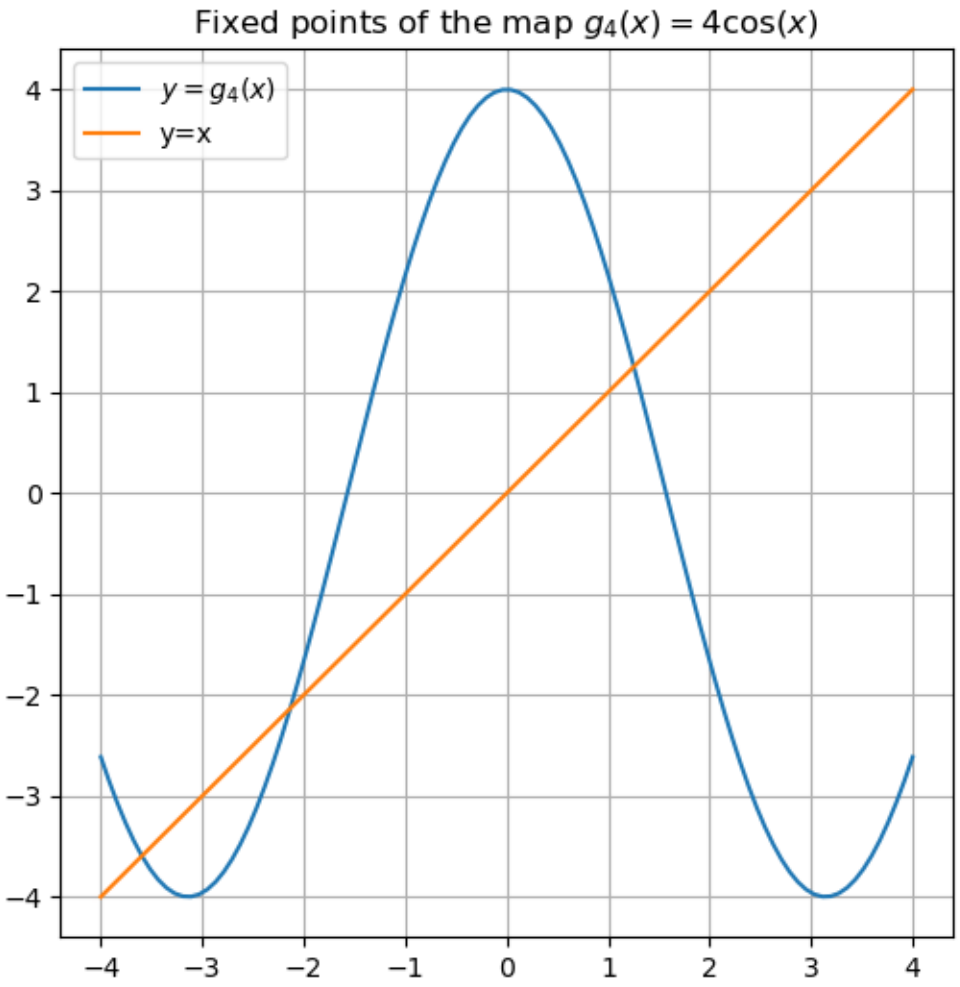
In other words, the graph of $y = g(x)$ goes from being above the line $y = x$ at $x = a$ to below it at $x = b$, so at some point $x = p$, the curves meet: $y = x = p$ and $y = g(p)$, so $p = g(p)$.

Example 2.2

Let us illustrate this with the mapping $g4(x) := 4 \cos x$, for which the fact that $|g4(x)| \leq 4$ ensures that this is a map of the domain $D = [-4, 4]$ into itself:

```
g_4(x) = 4cos(x)
a = -4.0
b = 4.0;
```

```
x = range(a, b, 100);
figure(figsize=[6,6])
title(L"Fixed points of the map $g_4(x) = 4 \cos(x)$")
plot(x, g_4.(x), label=L"y = g_4(x)")
plot(x, x, label="y=x")
legend()
grid(true);
```



This example has multiple fixed points (three of them). To ensure both the existence of a **unique** solution, and covergence of the iteration to that solution, we need an extra condition.

Definition 2.2 (Contraction Mapping)

A mapping $g : D \to D$, is called a **contraction** or **contraction mapping** if there is a constant $C < 1$ such that

$$|g(x) - g(y)| \leq C|x - y|$$

for any x and y in D. We then call C a *contraction constant*.

(*Aside:* The same applies for a domain in $\mathbb{R}^n$: just replace the absolute value $|\ldots|$ by the vector norm $\|\ldots\|$.)

Remark 2.5

It is not enough to have $|g(x) - g(y)| < |x - y|$ or $C = 1$! We need the ratio $\dfrac{|g(x) - g(y)|}{|x - y|}$ to be *uniformly* less than one for all possible values of x and y.

Theorem 2.1 (A Contraction Mapping Theorem)

Any contraction mapping on a closed, bounded interval $D = [a, b]$ has exactly one fixed point p in D. Further, this can be calculated as the limit $p = \lim_{k\to\infty} x_k$ of the iteration sequence given by $x_{k+1} = g(x_k)$ for *any* choice of the starting point $x_0 \in D$.

Proof. The main idea of the proof can be shown with the help of a few pictures.

First, uniqeness: between any two of the multiple fixed points above — call them p_0 and p_1 — the graph of $g(x)$ has to rise with secant slope 1: $(g(p_1) - g(p_0)/(p_1 - p_0) = (p_1 - p_0)/(p_1 - p_0) = 1$, and this violates the contraction property.

So instead, for a contraction, the graph of a contraction map looks like the one below for our favorite example, $g(x) = \cos x$ (which we will soon verify to be a contraction on interval $[-1, 1]$):

The second claim, about **convergence** to the fixed point from any initial approximation x_0, will be verified below, once we have seen some ideas about measuring errors.

An easy way of checking whether a differentiable function is a contraction

With differentiable functions, the contraction condition can often be easily verified using derivatives:

Theorem 2.2 (A derivative-based fixed point theorem)

If a function $g : [a, b] \to [a, b]$ is differentiable and there is a constant $C < 1$ such that $|g'(x)| \leq C$ for all $x \in [a, b]$, then g is a contraction mapping, and so has a unique fixed point in this interval.

Proof. Using the Mean Value Theorem, $g(x) - g(y) = g'(c)(x - y)$ for some c between x and y. Rhen taking absolute values,

$$|g(x) - g(y)| = |g'(c)| \cdot |(x - y)| \leq C|(x - y)|.$$

Example 2.3 ($g(x) = \cos(x)$ is a contraction on interval $[-1, 1]$)

Our favorite example $g(x) = \cos(x)$ is a contraction, but we have to be a bit careful about the domain.

For all real x, $g'(x) = -\sin x$, so $|g'(x)| \leq 1$; this is almost but not quite enough.

However, we have seen that iteration values will settle in the interval $D = [-1, 1]$, and considering g as a mapping of this domain, $|g'(x)| \leq \sin(1) = 0.841\cdots < 1$: that is, now we have a contraction, with $C = \sin(1) \approx 0.841$.

And as seen in the graph above, there is indeed a unique fixed point.

The contraction constant C as a measure of how fast the approximations improve (the smaller the better)

It can be shown that if C is small (at least when one looks only at a reduced domain $|x-p| < R$) then the convergence is "fast" once $|x_k - p| < R$.

To see this, we define some jargon for talking about errors. (For more details on error concepts, see section *Measures of Error and Order of Convergence*.)

Definition 2.3 (Error)

The *error* in $\tilde{x}$ as an approximation to an exact value x is

$$\text{error} := (\text{approximation}) - (\text{exact value}) = \tilde{x} - x$$

This will often be abbreviated as E.

Definition 2.4 (Absolute Error)

The *absolute error* in $\tilde{x}$ an approximation to an exact value x is the magnitude of the error: the absolute value $|E| = |\tilde{x} - x|$.

(*Aside:* This will later be extended to x and $\tilde{x}$ being vectors, by again using the vector norm in place of the absolute value. In fact, I will sometimes blur the distinction by using the "single line" absolute value notation for vector norms too.)

In the case of x_k as an approximation of p, we name the error $E_k := x_k - p$. Then C measures a worst case for how fast the error decreases as k increases, and this is "exponentially fast":

Proposition 2.3

$|E_{k+1}| \le C|E_k|$, or $|E_{k+1}|/|E_k| \le C$, and so

$$|E_k| \le C^k |x_0 - p|$$

That is, the error decreases at worst in a geometric sequence, which is exponential decrease with respect to the variable k.

Proof. $E_{k+1} = x_{k+1} - p = g(x_k) - g(p)$, using $g(p) = p$. Thus the contraction property gives

$$|E_{k+1}| = |g(x_k) - g(p)| \le C|x_k - p| = C|E_k|$$

Applying this again,

$$|E_k| \le C|E_{k-1}| \le C \cdot C|E_{k-2}| = C^2|E_{k-2}|$$

and repeating $k-2$ more times,

$$|E_k| \le C^k |E_0| = C^k |x_0 - p|.$$

Remark 2.6

We will often use this "recursive" strategy of relating the error in one iterate to that in the previous iterate.

We can now complete the proof of the above contraction mapping theorem *Theorem 2.1*

Proof. This now follows from *Proposition 2.3*

For **any** initial approximation x_0, we know that $|E_k| \leq C^k|x_0 - p|$, and with $C < 1$, this can be made as small as we want by choosing a large enough value of k. Thus

$$\lim_{k\to\infty} |E_k| = \lim_{k\to\infty} |x_k - p| = 0,$$

which is another way of saying that $\lim_{k\to\infty} x_k = p$, or $x_k \to p$, as claimed.

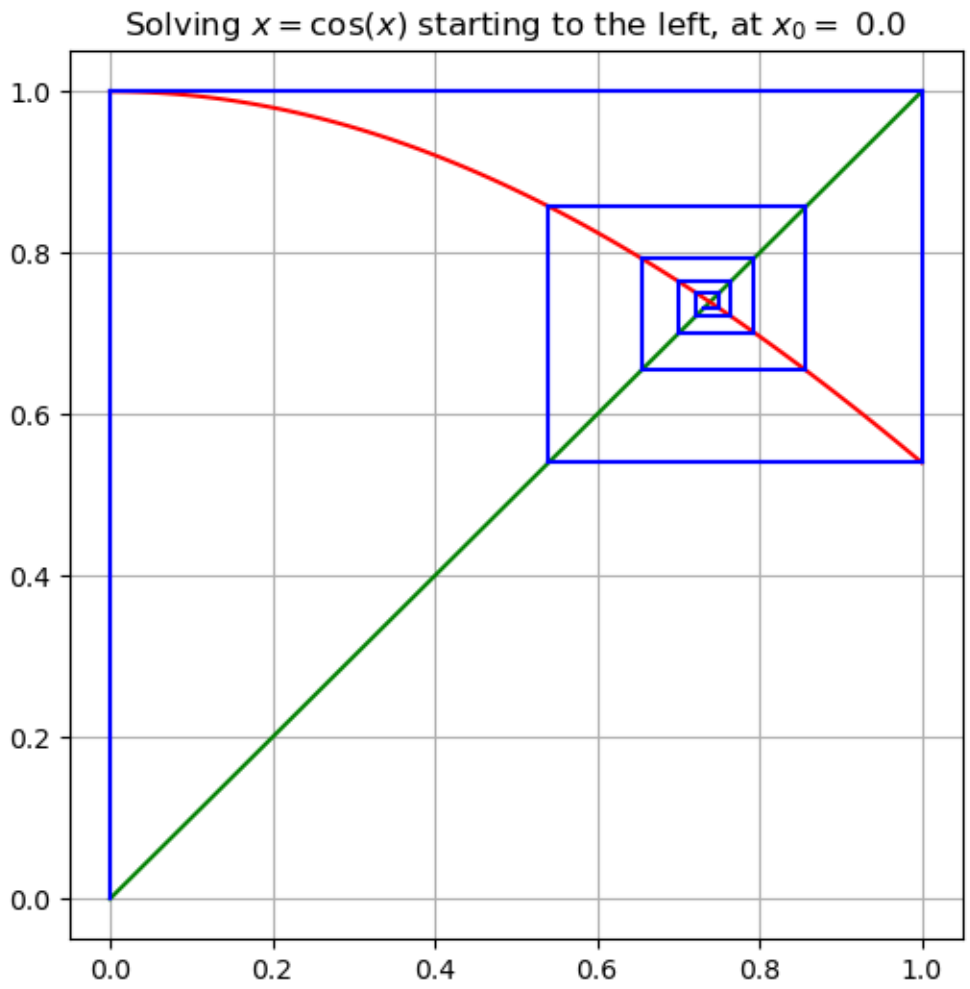
Example 2.4 (Solving $x = \cos x$ with a naive fixed point iteration)

We have seen that *one* way to convert the example $f(x) = x - \cos x = 0$ to a fixed point iteration is $g(x) = \cos x$, and that this is a contraction on $D = [-1, 1]$

Here is what this iteration looks like:

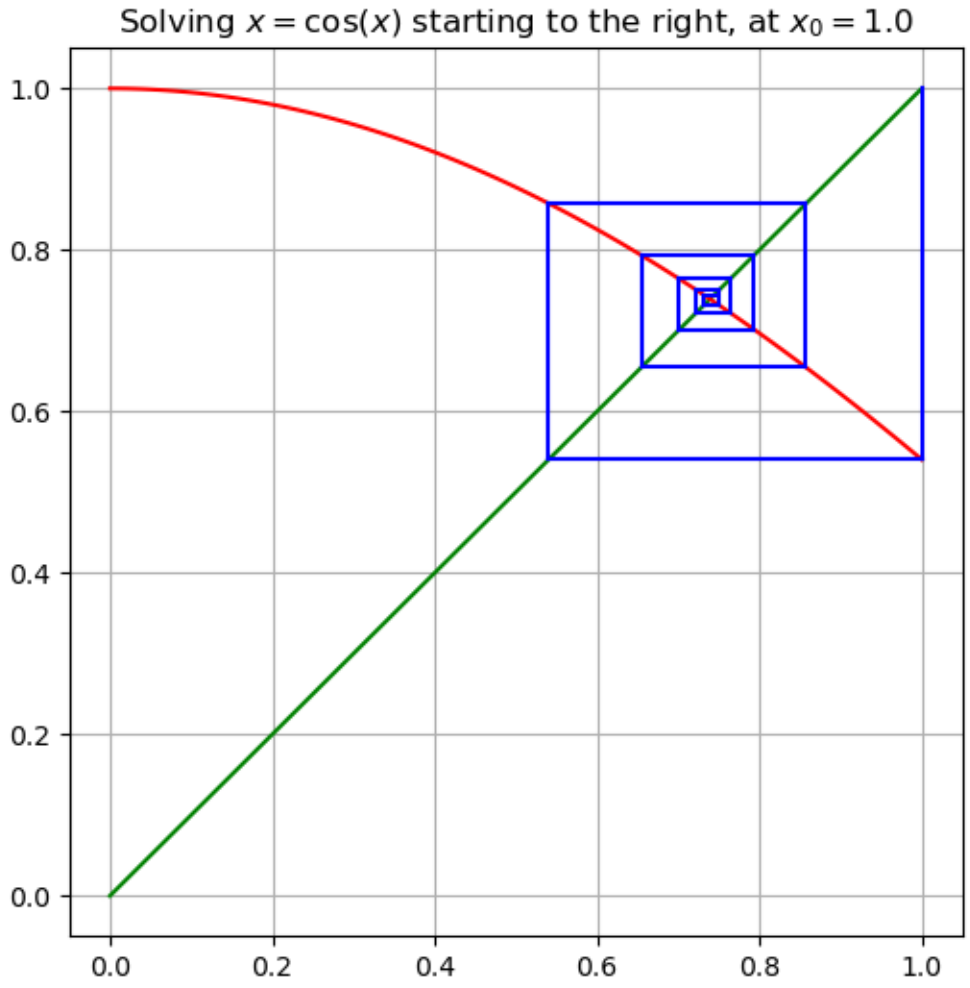
```
a = 0.0
b = 1.0
x = range(a, b, 100)
iterations = 10

# Start at left
x_k = a
figure(figsize=[6,6])
title(L"Solving $x = \cos(x)$ starting to the left, at $x_0 =$"*" $a")
plot(x, x, "g")
plot(x, g_1.(x), "r")
grid(true)
println("x_0 = $x_k")
for k in 1:iterations
    g_x_k = g_1(x_k)
    # Graph evalation of g(x_k) from x_k:
    plot([x_k, x_k], [x_k, g_1(x_k)], "b")
    x_k_plus_1 = g_1(x_k)
    #Connect to the new x_k on the line y = x:
    plot([x_k, g_1(x_k)], [x_k_plus_1, x_k_plus_1], "b")
    # Update names: the old x_k+1 is the new x_k
    x_k = x_k_plus_1
    println("x_$(k) = $x_k")
end
```

```
x_0 = 0.0
x_1 = 1.0
x_2 = 0.5403023058681398
x_3 = 0.8575532158463934
x_4 = 0.6542897904977791
x_5 = 0.7934803587425656
x_6 = 0.7013687736227565
x_7 = 0.7639596829006542
x_8 = 0.7221024250267077
x_9 = 0.7504177617637605
x_10 = 0.7314040424225098
```




```
# Start at right
x_k = b
figure(figsize=[6,6])
title("Solving " * L"x = \cos(x)" * " starting to the right, at " * L"x_0 = " * "$b")
# Julia note: "*" is concatenation of strings
plot(x, x, "g")
plot(x, g_1.(x), "r")
grid(true)
println("x_0 = $x_k")
for k in 1:iterations
    g_x_k = g_1(x_k)
    # Graph evalation of g(x_k) from x_k:
    plot([x_k, x_k], [x_k, g_1(x_k)], "b")
    x_k_plus_1 = g_1(x_k)
    #Connect to the new x_k on the line y = x:
    plot([x_k, g_1(x_k)], [x_k_plus_1, x_k_plus_1], "b")
    # Update names: the old x_k+1 is the new x_k
    x_k = x_k_plus_1
    println("x_$(k) = $x_k")
end
```

```
x_0 = 1.0
x_1 = 0.5403023058681398
x_2 = 0.8575532158463934
x_3 = 0.6542897904977791
x_4 = 0.7934803587425656
x_5 = 0.7013687736227565
x_6 = 0.7639596829006542
x_7 = 0.7221024250267077
x_8 = 0.7504177617637605
x_9 = 0.7314040424225098
x_10 = 0.744237354900557
```



In each case, one gets a "box spiral" in to the fixed point. It always looks like this when g is *decreasing* near the fixed point.

If instead g is *increasing* near the fixed point, the iterates approach monotonically, either from above or below:

Example 2.5 (Solving $f(x) = x^2 - 5x + 4 = 0$ in interval $[0, 3]$)

The roots are 1 and 4; for now we aim at the first of these, so we chose a domain $[0, 3]$ that contains just this root.

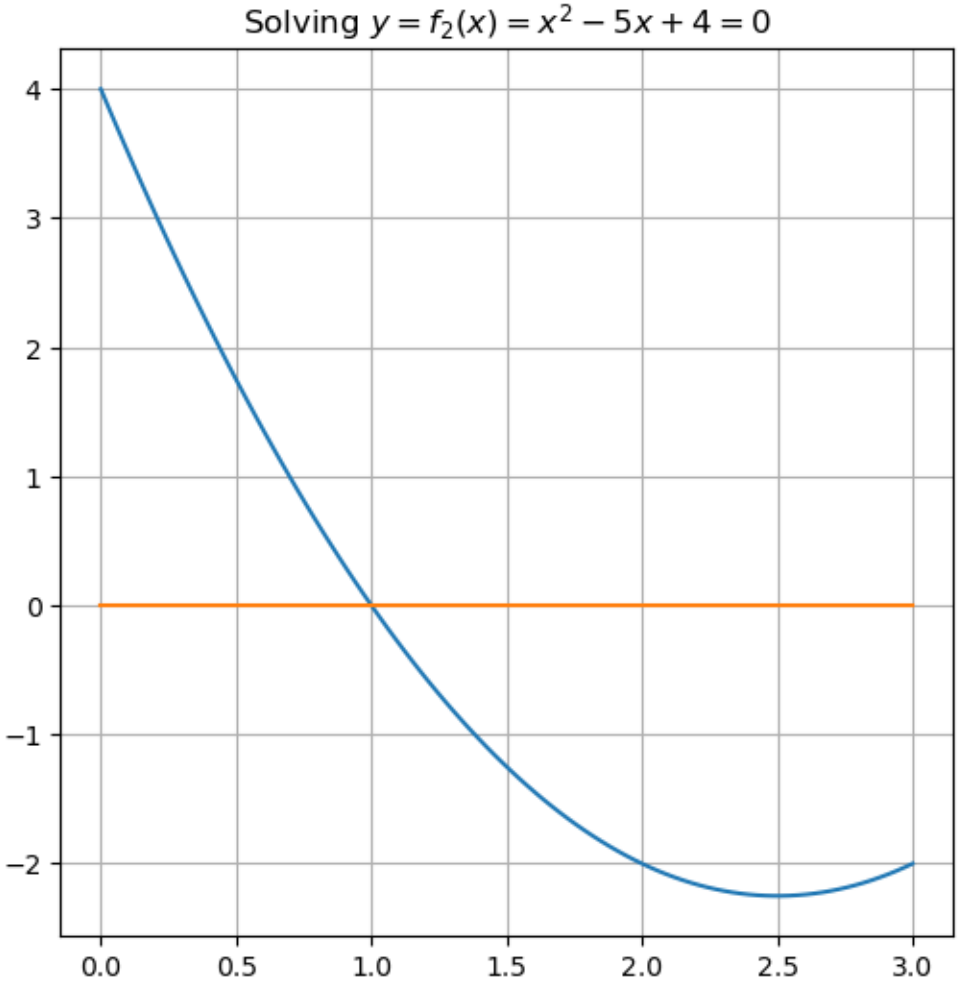
Let us get a fixed point for by "partially solving for x": solving for the x in the $5x$ term:

$$x = g(x) = (x^2 + 4)/5$$

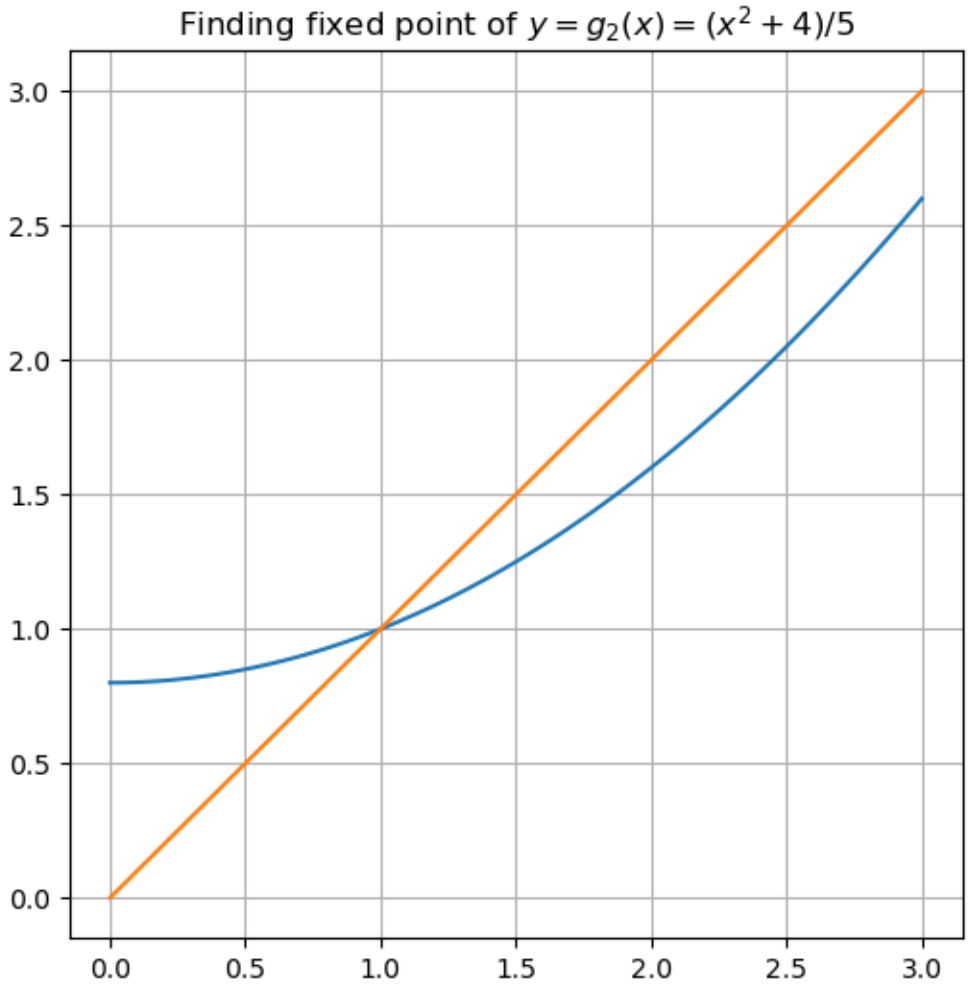
```
f_2(x) = x^2 - 5*x + 4;
g_2(x) = (x^2 + 4)/5;
```

```
a = 0.0;
b = 3.0;
x = range(a, b, 100);
```

```
figure(figsize=[6,6])
title(L"Solving $y = f_2(x) = x^2-5x+4 = 0$")
plot(x, f_2.(x))
plot([a, b], [0, 0])
grid(true)
```



```
figure(figsize=[6,6])
title(L"Finding fixed point of $y = g_2(x) = (x^2 + 4)/5$")
plot(x, g_2.(x))
plot(x, x)
grid(true)
```




```
iterations = 10
a = 0.0
b = 1.5
x = range(a, b, 100);
```

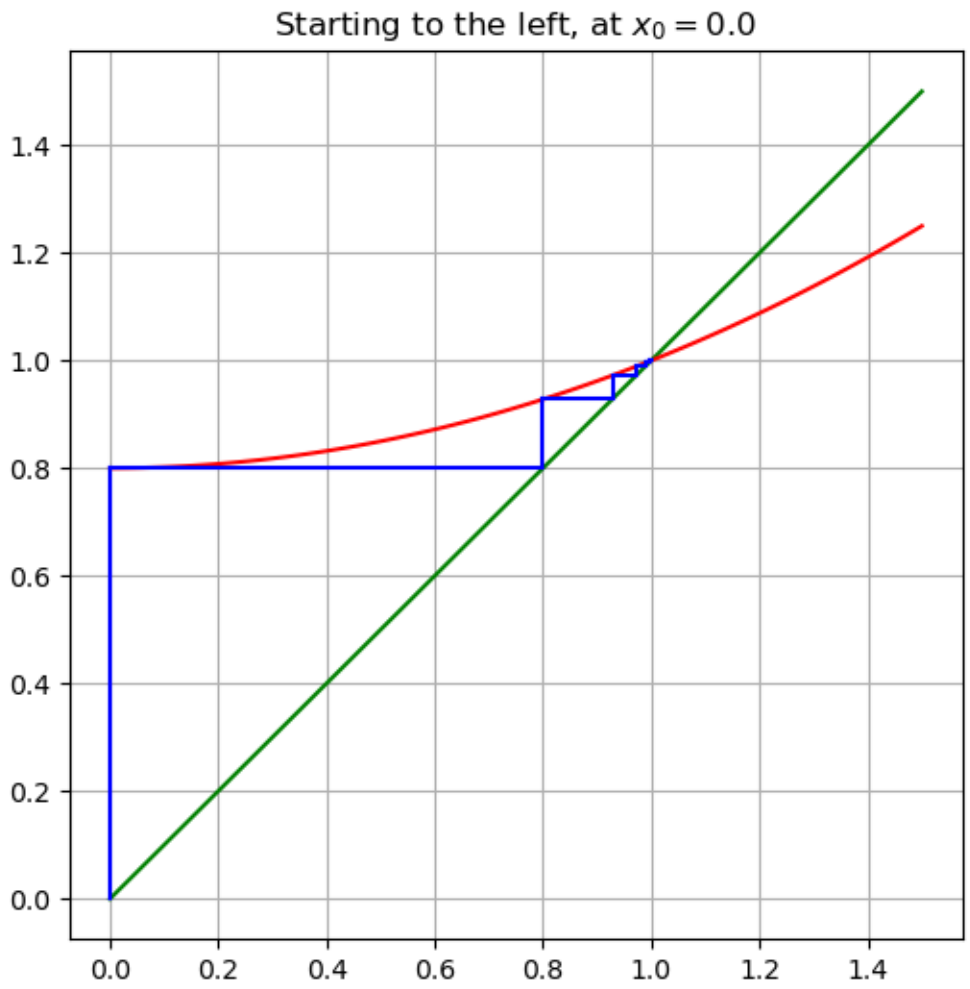
```
# Start at left
x_k = a
figure(figsize=[6,6])
title(L"Starting to the left, at $x_0 =$"*"$a")
grid(true)
plot(x, x, "g")
plot(x, g_2.(x), "r")
println("x_0 = $x_k")
for k in 1:iterations
```

(continues on next page)

(continued from previous page)

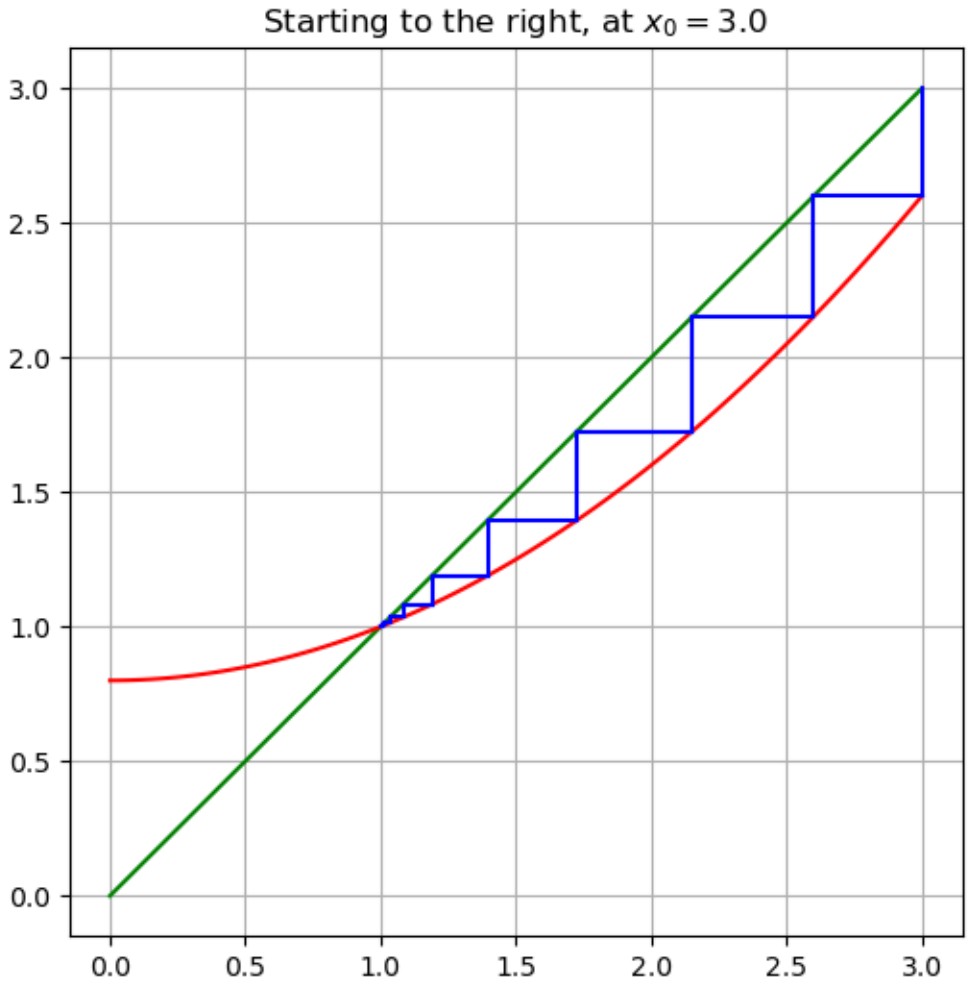
```
    g_x_k = g_2(x_k)
    # Graph evalation of g(x_k) from x_k:
    plot([x_k, x_k], [x_k, g_2(x_k)], "b")
    x_k_plus_1 = g_2(x_k)
    #Connect to the new x_k on the line y = x:
    plot([x_k, g_2(x_k)], [x_k_plus_1, x_k_plus_1], "b")
    # Update names: the old x_k+1 is the new x_k
    x_k = x_k_plus_1
    println("x_$(k) = $x_k")
end;
```

```
x_0 = 0.0
x_1 = 0.8
x_2 = 0.9280000000000002
x_3 = 0.9722368000000001
x_4 = 0.9890488790548482
x_5 = 0.9956435370319303
x_6 = 0.9982612105666906
x_7 = 0.9993050889044148
x_8 = 0.999722132142052
x_9 = 0.9998888682989302
x_10 = 0.999955549789623
```




```
# Start at right
a = 0.0;
b = 3.0;
x = range(a, b, 100)
x_k = b;
figure(figsize=[6,6])
title(L"Starting to the right, at $x_0 =$"*"$b")
grid(true)
plot(x, x, "g")
plot(x, g_2.(x), "r")
println("x_0 = $x_k")
for k in 1:iterations
    g_x_k = g_2(x_k)
    # Graph evalation of g(x_k) from x_k:
    plot([x_k, x_k], [x_k, g_2(x_k)], "b")
    x_k_plus_1 = g_2(x_k)
    #Connect to the new x_k on the line y = x:
    plot([x_k, g_2(x_k)], [x_k_plus_1, x_k_plus_1], "b")
    # Update names: the old x_k+1 is the new x_k
    x_k = x_k_plus_1
    println("x_$(k) = $x_k")
end;
```

```
x_0 = 3.0
x_1 = 2.6
x_2 = 2.152
x_3 = 1.7262208
x_4 = 1.3959676500705283
x_5 = 1.1897451360086866
x_6 = 1.0830986977312658
x_7 = 1.0346205578054328
x_8 = 1.014087939726725
x_9 = 1.0056748698998388
x_10 = 1.0022763887896116
```



2.2.3 Exercises

Exercise A

The equation $x^3 - 2x + 1 = 0$ can be written as a fixed point equation in many ways, including

1. $x = \dfrac{x^3 + 1}{2}$

   and

2. $x = \sqrt[3]{2x - 1}$

For each of these options:

(a) Verify that its fixed points do in fact solve the above cubic equation.

(b) Determine whether fixed point iteration with it will converge to the solution $r = 1$. (assuming a "good enough" initial approximation).

Note: computational experiments can be a useful start, but prove your answers mathematically!

2.3 Newton's Method for Solving Equations

References:

- Section 1.4 *Newton's Method* in [Sauer, 2019]
- Section 2.3 *Newton's Method and Its Extensions* in [Burden *et al.*, 2016]

2.3.1 Introduction

Newton's method for solving equations has a number of advantages over the bisection method:

- It is usually faster (but not always, and it can even fail completely!)
- It can also compute complex roots, such as the non-real roots of polynomial equations.
- It can even be adapted to solving systems of non-linear equations; that topic wil be visited later.

```
using PyPlot
```

2.3.2 Derivation as a contraction mapping with "very small contraction coefficient C"

You might have previously seen Newton's method derived using tangent line approximations. That derivation is presented below, but first we approach it another way: as a particularly nice contraction mapping.

To compute a root r of a differentiable function f, we design a contraction mapping for which the contraction constant C becomes arbitrarily small when we restrict to iterations in a sufficiently small interval around the root: $|x - r| \leq R$.

That is, the error ratio $|E_{k+1}|/|E_k|$ becomes ever smaller as the iterations get closer to the exact solution; the error is thus reducing ever faster than the above geometric rate C^k.

This effect is in turn achieved by getting $|g"(x)|$ arbitrarily small for $|x - r| \leq R$ with R small enough, and then using the above connection between $g"(x)$ and C. This can be achieved by ensuring that $g"(r) = 0$ at a root r of f — so long as the root r is *simple*: $f"(r) \neq 0$ (which is generically true, but not always).

To do so, seek g in the above form $g(x) = x - w(x)f(x)$, and choose $w(x)$ appropriately. At the root r,

$$g'(r) = 1 - w'(r)f(r) - w(r)f'(r) = 1 - w(r)f'(r) \quad \text{(using } f(r) = 0, \text{)}$$

so we ensure $g'(r) = 0$ by requiring $w(r) = 1/f'(r)$ (hence the problem if $f'(r) = 0$).

We do not know r, but that does not matter! We can just choose $w(x) = 1/f"(x)$ for all x values. That gives

$$g(x) = x - f(x)/f'(x)$$

and thus the iteration formula

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

(That is, $g(x) = x - f(x)/f'(x)$.)

You might recognize this as the formula for Newton's method.

To explore some examples of this, here is a function implementing Newton's method.

Remark (On Julia)

This function uses optional keyword parameters with default values for the first time; these will be described in Functions, part 2 in *Notes on the Julia Language* once they have been written.

Remark (A Julia module)

This and many other functions defined in this book are also gathered in the module `NumericalMethods`; see the appendix *Module NumericalMethods*. It can be made available with the commands

```
include("NumericalMethods.jl")
using .NumericalMethods: newtonmethod
```

The dot at the start of the module name is a litle surprising; that is needed whe the module is defined locally; in this case by that `include` line.

```
function newtonmethod(f, Df, x0, errortolerance; maxiterations=20, demomode=false)
    # Basic usage is:
    # (rootapproximation, errorestimate, iterations) = newton(f, Df, x0,␣
↪errortolerance)
    # There is an optional input parameter "demomode" which controls whether to
    # - println intermediate results (for "study" purposes), or to
    # - work silently (for "production" use).
    # The default is silence.

    if demomode
        println("Solving by Newton's Method.")
        println("maxiterations = $maxiterations")
        println("errortolerance = $errortolerance")
    end
    x = x0
    global errorestimate  # make it global to this function; without this it would be␣
↪local to the "for" loop.
    for iteration in 1:maxiterations
        fx = f(x)
```

(continues on next page)

(continued from previous page)

```
        Dfx = Df(x)
        # Note: a careful, robust code would check for the possibility of division by
↪zero here,
        # but for now I just want a simple presentation of the basic mathematical
↪idea.
        dx = fx/Dfx
        x -= dx  # Aside: this is shorthand for "x = x - dx"
        errorestimate = abs(dx);
        if demomode
            println("At iteration $iteration, x = $x with estimated error
↪$errorestimate and backward error $(abs(f(x)))")
        end
        if errorestimate <= errortolerance
            if demomode
                println("Done!")
            end
            return (x, errorestimate, iteration)
        end
    end
    # Note: if we get to here (no "return" above), it completed maxIterations
↪iterations without satisfying the accuracy target,
    # but we still return the information that we have.
    return (x, errorestimate, maxiterations)
end;
```

Example

Let's start with our favorite equation, $x = \cos x$.

Remark (On Julia style)

Recommended style for Julia code is that function names be alpha-numeric (possibly with the underscore _ as a "special guest letter"), so I will avoid primes in notation for derivatives as much as possible: from now on, the derivative of f is most often denoted as Df rather than f.

```
f1(x) = x - cos(x)
Df1(x) = 1 + sin(x);
```

```
x0 = 0.0
errortolerance = 1e-8
rei = newtonmethod(f1, Df1, x0, errortolerance; maxiterations=4, demomode=true)
root = rei[1]
errorestimate = rei[2]
iterations = rei[3]
println()
if errorestimate > errortolerance
    println("Warning: the error tolerance was not achieved!")
    println()
end
println("The root is approximately $root")
println("The estimated absolute error is $errorestimate")
```

(continues on next page)

(continued from previous page)

```
println("The backward error is $(abs(f1(root)))")
println("This required $iterations iterations")
```

```
Solving by Newton's Method.
maxiterations = 4
errortolerance = 1.0e-8
At iteration 1, x = 1.0 with estimated error 1.0 and backward error 0.
 ↪45969769413186023
At iteration 2, x = 0.7503638678402439 with estimated error 0.24963613215975608␣
 ↪and backward error 0.018923073822117442
At iteration 3, x = 0.7391128909113617 with estimated error 0.011250976928882236␣
 ↪and backward error 4.6455898990771516e-5
At iteration 4, x = 0.739085133385284 with estimated error 2.77575260776869e-5 and␣
 ↪backward error 2.847205804457076e-10

Warning: the error tolerance was not achieved!

The root is approximately 0.739085133385284
The estimated absolute error is 2.77575260776869e-5
The backward error is 2.847205804457076e-10
This required 4 iterations
```

Here we have introduced another way of talking about errors and accuracy, which is further discussed in *Measures of Error and Order of Convergence*.

Definition (Backward Error)

- The **backward error** in $\tilde{x}$ as an approximation to a root of a function f is $f(\tilde{x})$.
- The **absolute backward** error is its absolute value, $|f(\tilde{x})|$. However sometimes the latter is simply called the backward error — as the above code does.

This has the advantage that we can actually compute it without knowing the exact solution!

The backward error also has a useful geometrical meaning: if the function f were changed by this much to a nearbly function $\tilde{f}$ then $\tilde{x}$ could be an exact root of $\tilde{f}$. Hence, if we only know the values of f to within this backward error (for example due to rounding error in evaluating the function) then $\tilde{x}$ could well be an exact root, so there is no point in striving for greater accuracy in the approximate root.

We will see this in the next example.

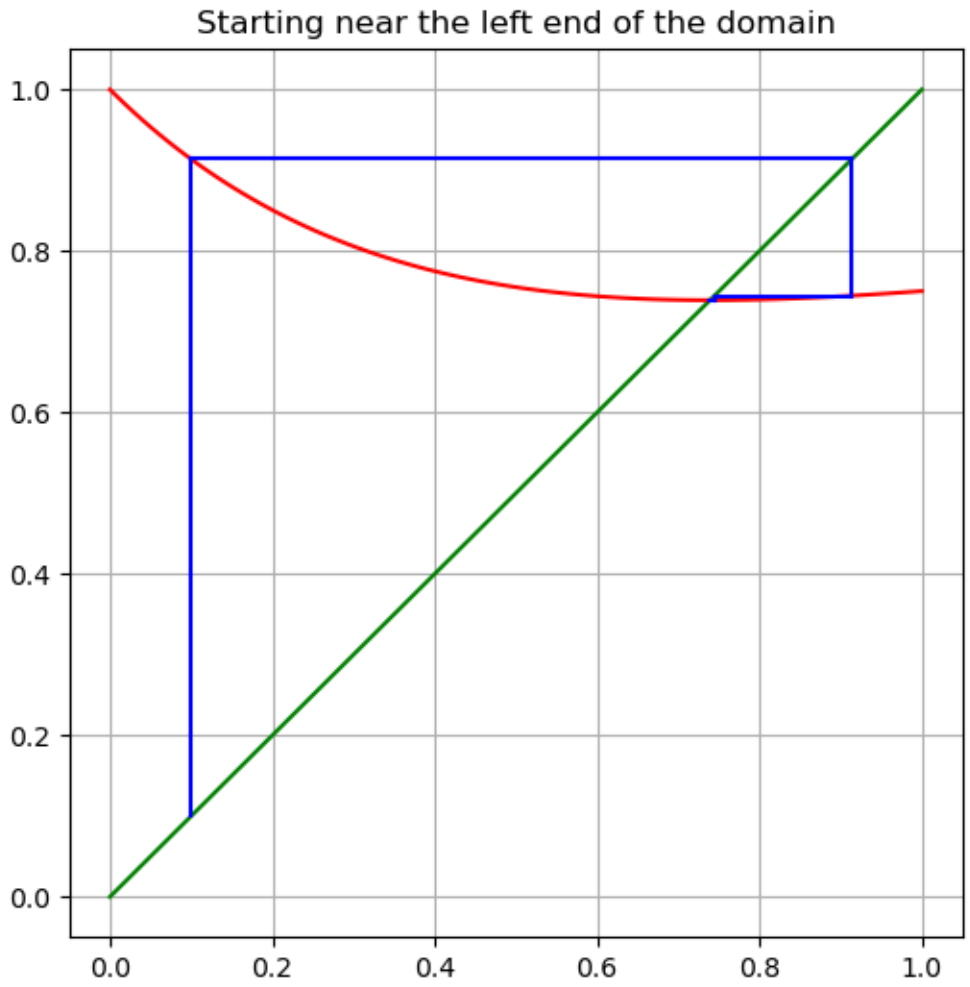
Graphing Newton's method iterations as a fixed point iteration

Since this is a fixed point iteration with $g(x) = x - (x - \cos(x)/(1 + \sin(x))$, let us compare its graph to the ones seen in *Solving Equations by Fixed Point Iteration (of Contraction Mappings)*. Now g is neither increasing nor decreasing at the fixed point, so the graph has an unusual form.

```
g(x) = x - (x - cos(x))/(1 + sin(x))
a = 0.0
b = 1.0
# An array of x values for graphing
x = range(a, b, 100)
iterations = 4;  # Not so many are needed now!
```

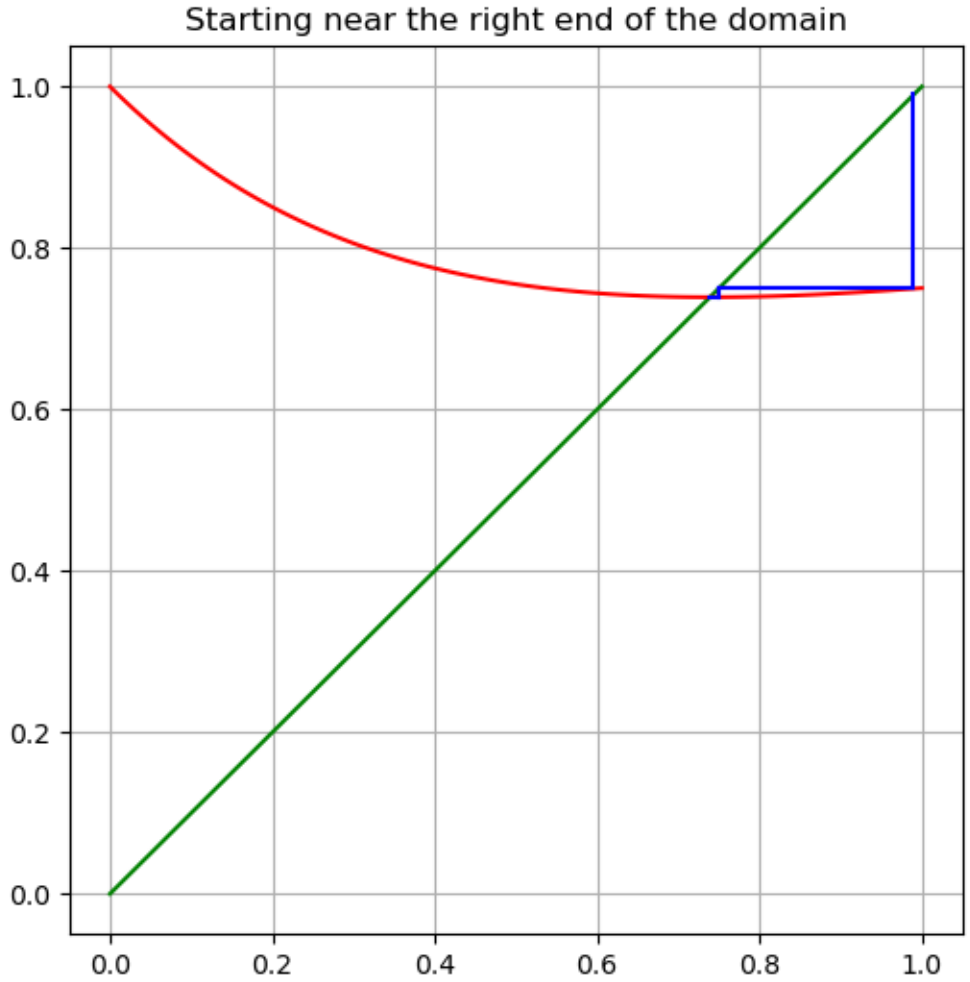
```
# Start at left
description = "Starting near the left end of the domain"
println(description)
x_k = 0.1

figure(figsize=[6,6])
title(description)
grid(true)
plot(x, x, "g")
plot(x, g.(x), "r")
println("x_0 = $x_k")
for k in 1:iterations
    g_x_k = g(x_k)
    # Graph evalation of g(x_k) from x_k:
    plot([x_k, x_k], [x_k, g(x_k)], "b")
    x_k_plus_1 = g(x_k)
    #Connect to the new x_k on the line y = x:
    plot([x_k, g(x_k)], [x_k_plus_1, x_k_plus_1], "b")
    # Update names: the old x_k+1 is the new x_k
    x_k = x_k_plus_1
    println("x_$k = $x_k")
end;
```

```
Starting near the left end of the domain
x_0 = 0.1
x_1 = 0.9137633861014282
x_2 = 0.7446642419816996
x_3 = 0.7390919659607759
x_4 = 0.7390851332254692
```




```
# Start at right
description = "Starting near the right end of the domain"
println(description)
x_k = 0.99
figure(figsize=[6,6])
title(description)
grid(true)
plot(x, x, "g")
plot(x, g.(x), "r")
println("x_0 = $x_k")
for k in 1:iterations
    g_x_k = g(x_k)
    # Graph evalation of g(x_k) from x_k:
    plot([x_k, x_k], [x_k, g(x_k)], "b")
    x_k_plus_1 = g(x_k)
    #Connect to the new x_k on the line y = x:
    plot([x_k, g(x_k)], [x_k_plus_1, x_k_plus_1], "b")
    # Update names: the old x_k+1 is the new x_k
    x_k = x_k_plus_1
    println("x_$k = $x_k")
end;
```

```
Starting near the right end of the domain
x_0 = 0.99
x_1 = 0.7496384013287254
x_2 = 0.7391094534708724
x_3 = 0.7390851333457581
x_4 = 0.7390851332151607
```



In fact, wherever you start, all iterations take you to the right of the root, and then approach the fixed point monotonically — and very fast. We will see an explanation for this in *The Convergence Rate of Newton's Method*.

Example (Pushing to the limits of standard 64-bit computer arithmetic)

Next, demand more accuracy; this time silently. As we will see in a later section, 10^{-16} is about the limit of the precision of standard (IEE64) computer arithmetic with 64-bit numbers.

So let's try to compute the root as accurately as we can within these limits:

```
x0 = 0.0
errortolerance=1e-16
rei = newtonmethod(f1, Df1, x0, errortolerance)
```

(continues on next page)

(continued from previous page)

```
root = rei[1]
errorestimate = rei[2]
iterations = rei[3]
println()
println("The root is approximately $root")
println("The estimated absolute error is $errorestimate")
println("The backward error is $(abs(f1(root)))")
println("This required $iterations iterations")
```

```
The root is approximately 0.7390851332151607
The estimated absolute error is 0.0
The backward error is 0.0
This required 6 iterations
```

Observations:

- It only took one more iteration to meet the demand for twice as many decimal places of accuracy.
- The result is "exact" as fas as the computer arithmeric can tell, as shown by the zero backward error: we have indeed reached the accuracy limits of computer arithmetic.

2.3.3 Newton's method works with complex numbers too

This book will work almost entirely with real values and vectors in $\mathbb{R}^n$, but actually, everything above also works for complex numbers. In particular, Newton's method works for finding roots of functions $f : \mathbb{C} \to \mathbb{C}$; for example when seeking all roots of a polynomial.

(See the notes on *Complex number in Julia*.)

Example (All roots of a cubic)

As an example, let us seek all three cube roots of 8, by solving $x^3 - 8 = 0$ and trying different initial values x_0.

```
f2(x) = x^3 - 8
Df2(x) = 3x^2;
```

First, $x_0 = 1$

```
x0 = 1.0;
errortolerance = 1e-8;
rei1 = newtonmethod(f2, Df2, x0, errortolerance; demomode=true)
root1 = rei1[1]
errorestimate1 = rei1[2]
iterations1 = rei1[3]
println()
println("The first root is approximately $root1")
println("The estimated absolute error is $errorestimate1")
println("The backward error is $(abs(f2(root1)))")
println("This required $iterations1 iterations")
```

```
Solving by Newton's Method.
maxiterations = 20
```

(continues on next page)

(continued from previous page)

```
errortolerance = 1.0e-8
At iteration 1, x = 3.3333333333333335 with estimated error 2.3333333333333335 and␣
 ↪backward error 29.037037037037045
At iteration 2, x = 2.462222222222222 with estimated error 0.8711111111111113 and␣
 ↪backward error 6.92731645541838
At iteration 3, x = 2.081341247671579 with estimated error 0.380880974550643 and␣
 ↪backward error 1.0163315496105625
At iteration 4, x = 2.003137499141287 with estimated error 0.07820374853029163 and␣
 ↪backward error 0.03770908398584538
At iteration 5, x = 2.000004911675504 with estimated error 0.003132587465783101␣
 ↪and backward error 5.894025079733467e-5
At iteration 6, x = 2.0000000000120624 with estimated error 4.911663441917921e-6␣
 ↪and backward error 1.447482134153688e-10
At iteration 7, x = 2.0 with estimated error 1.2062351117801901e-11 and backward␣
 ↪error 0.0
Done!

The first root is approximately 2.0
The estimated absolute error is 1.2062351117801901e-11
The backward error is 0.0
This required 7 iterations
```

Next, start at $x_0 = i$ (a.k.a. $x_0 = im$):

```
x0 = im;
rei2 = newtonmethod(f2, Df2, x0, errortolerance; demomode=true)
root2 = rei2[1]
errorestimate2 = rei2[2]
iterations2 = rei2[3]
println()
println("The second root is approximately $root2")
println("The estimated absolute error is $errorestimate2")
println("The backward error is $(abs(f2(root2)))")
println("This required $iterations2 iterations")
```

```
Solving by Newton's Method.
maxiterations = 20
errortolerance = 1.0e-8
At iteration 1, x = -2.6666666666666665 + 0.6666666666666667im with estimated␣
 ↪error 2.6874192494328497 and backward error 27.23670564570405
At iteration 2, x = -1.4663590926566703 + 0.6105344098423685im with estimated␣
 ↪error 1.2016193667222377 and backward error 10.211311837398133
At iteration 3, x = -0.23293230984230884 + 1.1571382823138845im with estimated␣
 ↪error 1.3491172750968106 and backward error 7.206656519642179
At iteration 4, x = -1.9202321953438537 + 1.5120026439880303im with estimated␣
 ↪error 1.7242127533456901 and backward error 13.405769067901167
At iteration 5, x = -1.1754417924325344 + 1.4419675366055338im with estimated␣
 ↪error 0.7480759724352086 and backward error 3.7583743808280228
At iteration 6, x = -0.9389355523964149 + 1.7160019741718O7im with estimated error␣
 ↪0.36198076543966584 and backward error 0.7410133693135651
At iteration 7, x = -1.0017352527552088 + 1.7309534907089796im with estimated␣
 ↪error 0.06455501693838851 and backward error 0.02463614729973853
At iteration 8, x = -0.9999988050398477 + 1.7320490713246675im with estimated␣
 ↪error 0.0020531798639315357 and backward error 2.529258531285453e-5
At iteration 9, x = -1.0000000000014002 + 1.7320508075706016im with estimated␣
 ↪error 2.107719871290353e-6 and backward error 2.6654146785452274e-11
```

(continues on next page)

(continued from previous page)

```
At iteration 10, x = -1.0 + 1.7320508075688774im with estimated error 2.
↪2211788987828177e-12 and backward error 1.9860273225978185e-15
Done!

The second root is approximately -1.0 + 1.7320508075688774im
The estimated absolute error is 2.2211788987828177e-12
The backward error is 1.9860273225978185e-15
This required 10 iterations
```

This root is in fact $-1 + i\sqrt{3}$.

Finally, $x_0 = 1 - i$

```
x0 = 1-im
rei3 = newtonmethod(f2, Df2, x0, errortolerance; demomode=false)
root3 = rei3[1]
errorestimate3 = rei3[2]
iterations3 = rei3[3]
println()
println("The third root is approximately $root3")
println("The estimated absolute error is $errorestimate3")
println("The backward error is $(abs(f2(root3)))")
println("This required $iterations3 iterations")
```

```
The third root is approximately -1.0 - 1.7320508075688772im
The estimated absolute error is 3.62974830495685e-15
The backward error is 1.9860273225978185e-15
This required 10 iterations
```

This root is in fact $-1 - i\sqrt{3}$.

2.3.4 Newton's method derived via tangent line approximations: linearization

The more traditional derivation of Newton"s method is based on the very widely useful idea of *linearization*; using the fact that a differentiable function can be approximated over a small part of its domain by a straight line — its tangent line — and it is easy to compute the root of this linear function.

So start with a first approximation x_0 to a solution r of $f(x) = 0$.

Step 1: Linearize at x_0.

The tangent line to the graph of this function wih center x_0, also know as the *linearization of* f *at* x_0, is

$$L_0(x) = f(x_0) + f'(x_0)(x - x_0).$$

(Note that $L_0(x_0) = f(x_0)$ and $L_0'(x_0) = f'(x_0)$.)

Step 2: Find the zero of this linearization

Hopefully, the two functions f and L_0 are close, so that the root of L_0 is close to a root of f; close enough to be a better approximation of the root r than x_0 is.

Give the name x_1 to this root of L_0: it solves $L_0(x_1) = f(x_0) + f'(x_0)(x_1 - x_0) = 0$, so

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

Step 3: Iterate

We can then use this new value x_1 as the center for a new linearization $L_1(x) = f(x_1) + f'(x_1)(x - x_1)$, and repeat to get a hopefully even better approximate root,

$$x_2 = x_1 - f(x_1)/f'(x_1)$$

And so on: at each step, we get from approximation x_k to a new one x_{k+1} with

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

And indeed this is the same formula seen above for Newton's method.

Illustration: a few steps of Newton's method for $x - \cos(x) = 0$.

This approach to Newton's method via linearization and tangent lines suggests another graphical presentation; again we use the example of $f(x) = x - \cos(x)$. This has $Df(x) = 1 + \sin(x)$, so the linearization at center a is

$$L(x) = (a - \cos(a)) + (1 + \sin(a))(x - a)$$

For Newton's method starting at $x_0 = 0$, this gives

$$L_0(x) = -1 + x$$

and its root — the next iterate in Newton"s method — is $x_1 = 1$

Then the linearization at center x_1 is

$$L_1(x) = (1 - \cos(1) + (1 + \sin(1))(x - 1), \approx 0.4596 + 1.8415(x - 1)$$

giving $x_2 \approx 1 - 0.4596/1.8415 \approx 0.7504$.

Let's graph a few steps.

```
L_0(x) = -1.0 + x;
```

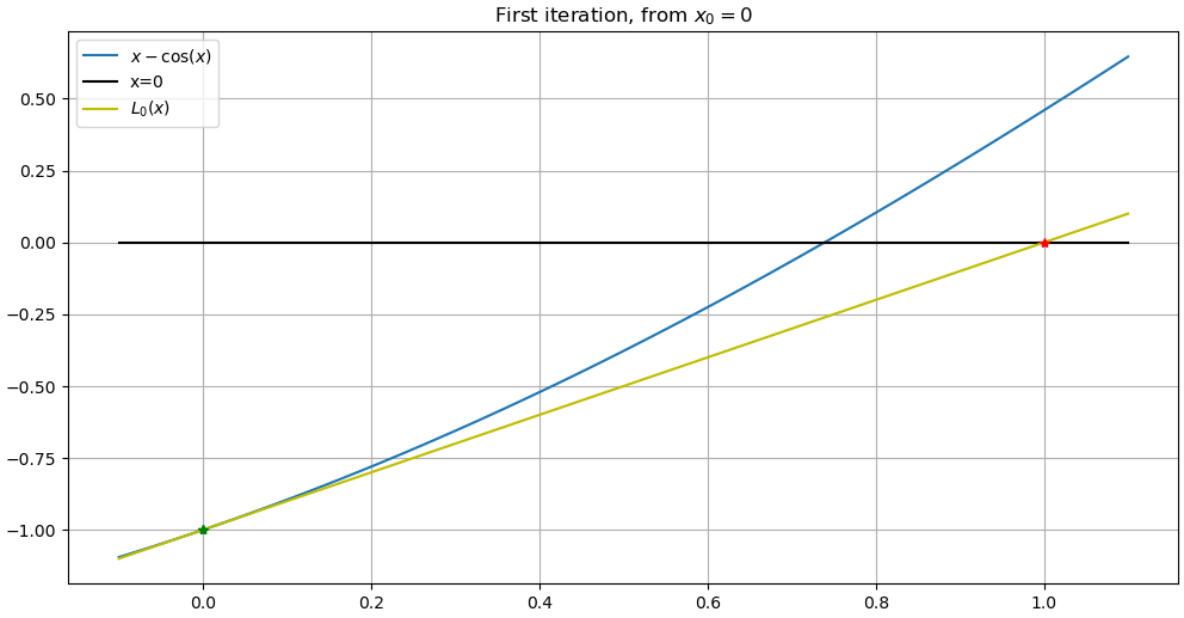
```
figure(figsize=[12,6])
title(L"First iteration, from $x_0 = 0$")
left = -0.1
right = 1.1
x = range(left, right, 100)
plot(x, f1.(x), label=L"x - \cos(x)")
plot([left, right], [0, 0], "k", label="x=0")  # The x-axis, in black
x_0 = 0
plot([x_0], [f1(x_0)], "g*")
plot(x, L_0.(x), "y", label=L"L_0(x)")
plot([x_0], [f1(x_0)], "g*")
```

(continues on next page)

(continued from previous page)

```
x_1 = x_0 - f1(x_0)/Df1(x_0)
println("x_1 = $x_1")
plot([x_1], [0], "r*")
legend()
grid(true)
```

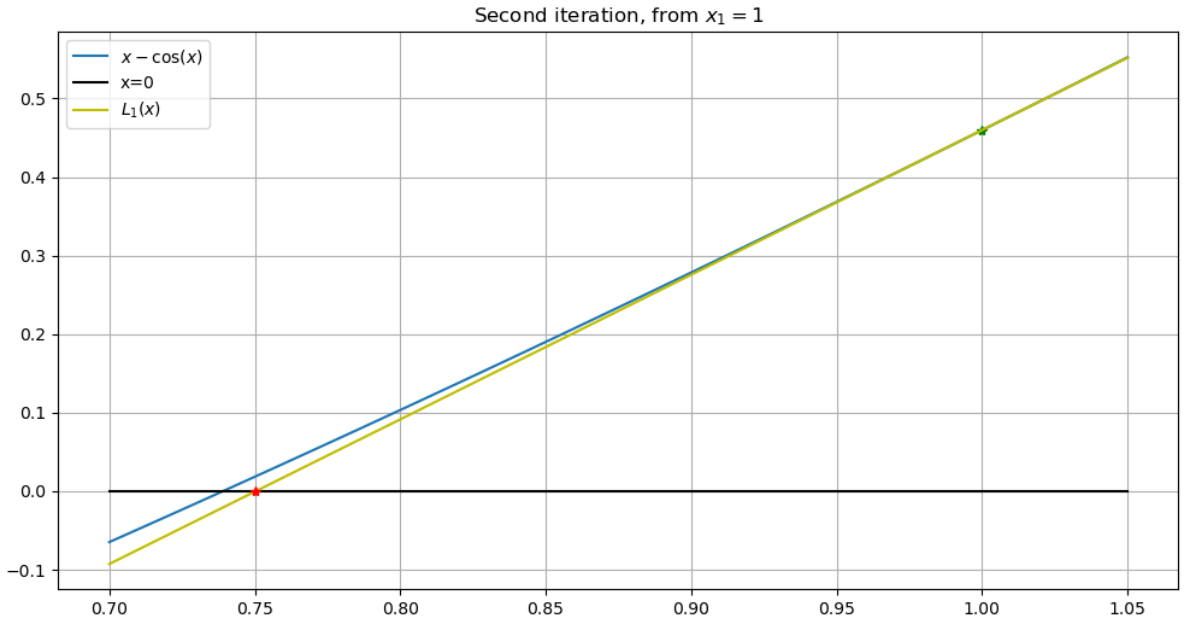
```
x_1 = 1.0
```




```
L_1(x) = (x_1 - cos(x_1)) + (1 + sin(x_1))*(x - x_1);
```

```
figure(figsize=[12,6])
title(L"Second iteration, from $x_1 = 1$")
# Shrink the domain
left = 0.7
right = 1.05
x = range(left, right, 100)

plot(x, f1.(x), label=L"x - \cos(x)")
plot([left, right], [0, 0], "k", label="x=0")  # The x-axis, in black
plot([x_1], [f1(x_1)], "g*")
plot(x, L_1.(x), "y", label=L"L_1(x)")
x_2 = x_1 - f1(x_1)/Df1(x_1)
println("x_2 = $x_2")
plot([x_2], [0], "r*")
legend()
grid(true)
```

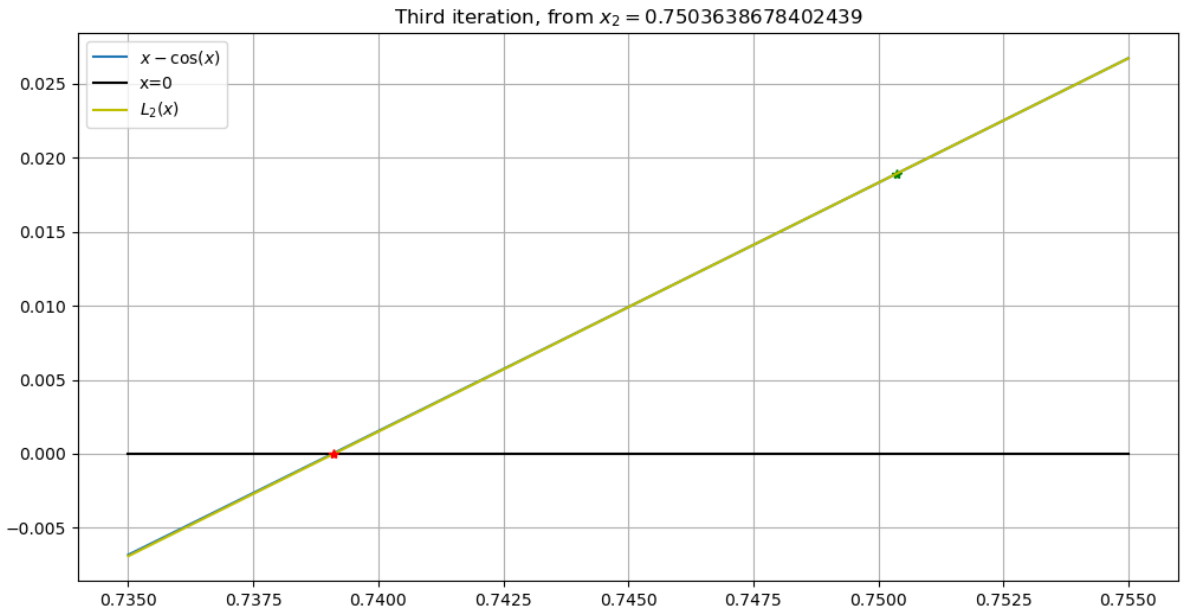
```
x_2 = 0.7503638678402439
```




```
L_2(x) = (x_2 - cos(x_2)) + (1 + sin(x_2))*(x - x_2);
```

```
figure(figsize=[12,6])
title(L"Third iteration, from $x_2=$"*"$x_2")
# Shrink the domain some more
left = 0.735
right = 0.755
x = range(left, right, 100)
plot(x, f1.(x), label=L"x - \cos(x)")
plot([left, right], [0, 0], "k", label="x=0")  # The x-axis, in black
plot([x_2], [f1(x_2)], "g*")
plot(x, L_2.(x), "y", label=L"L_2(x)")
x_3 = x_2 - f1(x_2)/Df1(x_2)
println("x_3 = $x_3")
plot([x_3], [0], "r*")
legend()
grid(true)
```

```
x_3 = 0.7391128909113617
```



2.3.5 How accurate and fast is this?

For the bisection method, we have seen in *Root Finding by Interval Halving (Bisection)* a fairly simple way to get an upper limit on the absolute error in the approximations.

For absolute guarantees of accuracy, things do not go quite as well for Newton's method, but we can at least get a very "probable" *estimate* of how large the error can be. This requires some calculus, and more specifically Taylor's theorem, reviewed in *Taylor's Theorem and the Accuracy of Linearization*.

So we will return to the question of both the speed and accuracy of Newton's method in *The Convergence Rate of Newton's Method*.

On the other hand, the example graphs above illustrate that the successive linearizations become ever more accurate as approximations of the function f itself, so that the approximation x_3 looks "perfect" on the graph — the speed of Newton's method looks far better than for bisection. This will also be explained in *The Convergence Rate of Newton's Method*.

2.3.6 Exercises

Exercise A

a) Show that Newton's method applied to

$$f(x) = x^k - a$$

leads to fixed point iteration with function

$$g(x) = \frac{(k-1)x + \dfrac{a}{x^{k-1}}}{k}.$$

b) Then verify mathematically that the iteration $x_{k+1} = g(x_k)$ has super-linear convergence.

Exercise B

a) Create a Julia function for Newton's method, with usage

```
(root, errorEstimate, iterations, functionEvaluations) = newtonMethod(f, Df, x_0,␣
↪errorTolerance, maxIterations)
```

(The last input parameter `maxIterations` could be optional, with a default like `maxIterations=100`.)

b) based on your function `bisection2` create a third (and final!) version with usage

```
(root, errorBound, iterations, functionEvaluations) = bisection(f, a, b,␣
↪errorTolerance, maxIterations)
```

c) Use both of these to solve the equation

$$f_1(x) = 10 - 2x + \sin(x) = 0$$

i) with [estimated] absolute error of no more than 10^{-6}, and then

ii) with [estimated] absolute error of no more than 10^{-15}.

Note in particular how many iterations and how many function evaluations are needed.

Graph the function, which will help to find a good starting interval $[a, b]$ and initial approximation x_0.

d) Repeat, this time finding the unique real root of

$$f_2(x) = x^3 - 3.3x^2 + 3.63x - 1.331 = 0$$

Again graph the function, to find a good starting interval $[a, b]$ and initial approximation x_0.

e) This second case will behave differently than for f_1 in part (c): describe the difference. (We will discuss the reasons in class.)

2.4 Taylor's Theorem and the Accuracy of Linearization

References:

- Theorem 0.8 in Section 0.5 *Review of Calculus* in [Sauer, 2019].
- Section 1.1 *Review of Calculus* in [Burden *et al.*, 2016], from Theorem 1.14 onward.

2.4.1 Taylor's theorem

Taylor's theorem is most often stated in the form

Theorem (Taylor's Theorem, with center a)

When all the relevant derivatives exist,

$$\begin{aligned} f(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 \cdots \frac{f^{(k)}(a)}{k!}(x-a)^k + \cdots \\ + \frac{f^{(n)}(a)}{n!}(x-a)^n + R_n(x) \end{aligned} \tag{2.1}$$

The polynomial part of this,

$$T_n(x) = f(a) + f'(a) + \cdots \frac{f^{(k)}(a)}{k!}(x-a)^k + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{2.2}$$

is the **Taylor polynomial of degree n with center a for function f**, and the *remainder* is

$$R_n(x) = \frac{f^{(n+1)}(c_x)}{(n+1)!}(x-a)^{n+1} \tag{2.3}$$

with the value c_x lying between a and x, and so depending on x.

This gives information about the absolute error in the polynomial $T_n(x)$ as an approximation of $f(x)$:

$$|f(x) - T_n(x)| \leq \frac{M_{n+1}}{(n+1)!}|x-a|^{n+1}$$

where M_{n+1} is the maximum absolute value of $f^{(n+1)}$ over the relevant interval between a and x.

Of course we typically do not know much about that constant M_{n+1}, so often the most important thing is the power law rate $|x-a|^{n+1}$ at which the error reduces as x approaches a.

Taylor polynomials are therefore most useful when the quantity $h := x - a$ is small, and we will most often use them in situations where the limit as $h \to 0$ is relevant. It is convenient to change the notation a bit, treating h as the variable:

Theorem (Taylor's Theorem, h form)

When all the relevant derivatives exist,

$$T_n(h) = f(a) + f'(a)h + \cdots \frac{f^{(k)}(a)}{k!}h^k + \cdots + \frac{f^{(n)}(a)}{n!}h^n \tag{2.4}$$

with this polynomial in h approximating $f(a+h)$ in that

$$f(a+h) - T_n(h) = R_n(h) = \frac{f^{(n+1)}(c_h)}{(n+1)!}h^{n+1} \text{ with } |c_h - a| < |h|. \tag{2.5}$$

(Note: h may be negative!)

This gives a bound on the absolute error

$$|f(a+h) - T_n(h)| \leq \frac{M_{n+1}}{(n+1)!}|h|^{n+1} \tag{2.6}$$

with

$$M_{n+1} := \max_{|x-a| \leq |h|} |f^{(n+1)}(x)|$$

2.4.2 Error formula for linearization

A very common use of Taylor's Theorem is the rather simple case $n = 1$; *linearization*, to approximate a twice diferentiable function by a linear one. (This will be even more so when we come to system of equations, since the only such systems that we can systematically solve exactly are linear systems.)

Taylor's Theorem for the linearization $L(x) = f(a) + f'(a)(x-a)$ of f at a then says that

$$f(x) - L(x) = \frac{f''(c_x)}{2}h^2, \quad |c_x - a| < |x-a|$$

or in terms of h,

$$f(a+h) = f(a) + f'(a)h + \frac{f''(c_h)}{2}h^2, \quad |c_h - a| < |h|$$

Thus there is an error bound

$$|f(a+h) - (f(a) + f'(a)h)| \leq \frac{M_2}{2}h^2, \text{ where } M_2 = \max_{|x-a|<|h|} |f''(x)|$$

Of course sometimes it is enough to use the maximum over the whole domain, $M_2 = \max |f''(x)|$.

2.5 Measures of Error and Order of Convergence

References:

- Section 1.3.1 *Forward and backward error* of [Sauer, 2019], on *measures of error*;
- Section 2.4 *Error Analysis for Iterative Methods* of [Burden *et al.*, 2016], on *order of convergence.*

These notes cover a number of small topics:

- Measures of error: absolute, relative, forward, backward, etc.
- Measuring the rate of convergence of a sequence of approximations.
- Big-O and little-o notation for describing how small a quantity (usually an error) is.

2.5.1 Error measures

Several of these have been mentioned before, but they are worth gathering here.

Consider a quantity $\tilde{x}$ considered as an approximation of an exact value x. (This can be a number or a vector.)

Definition 2.6 (Error)

The **error** in $\tilde{x}$ is $\tilde{x} - x$ (or $x - \tilde{x}$; different sources use both versions and either is fine so long as you are **consistent**.)

Definition 2.7 (Absolute Error)

The **absolute error** is the absolute value of the error: $|\tilde{x} - x|$. For vector quantities this means the norm $\|\tilde{x} - x\|$, and it can be any norm, so long as we again choose one and use it consistently. Two favorites are the *Euclidean norm* $\|x\| = \sqrt{\sum |x_i|}$, denoted $\|x\|_2$, and the *maximum norm* (also mysteriously known at the *infinity norm*):

$$\|x\|_{\max} = \|x\|_\infty = \max_i |x_i|.$$

For real-valued quantities, the absolute error is related to the number of **correct decimal places**: p decimal places of accuracy corresponds roughly to absolute error no more than 0.5×10^{-p}.

Definition 2.8 (Relative Error)

The **relative error** is the ratio of the absolute error to the size of the exact quantity: $\dfrac{\|\tilde{x} - x\|}{\|x\|}$ (again possibly with vector norms).

This is often more relevant than absolute error for inherently positive quantities, but is obviously unwise where $x = 0$ is a possibility. For real-valued quantities, this is related to the number of **significant digits**: accuracy to p significant digits corresponds roughly to relative error no more than 0.5×10^{-p}.

When working with computer arithmetic, p significant **bits** corresponds to relative error no more than $2^{-(p+1)}$.

Backward error (and forward error)

An obvious problem is that we usually do not know the exact solution x, so cannot evaluate any of these; instead we typically seek **upper bounds** on the absolute or relative error. Thus, when talking of approximate solutions to an equation $f(x) = 0$ the concept of

backward error *Definition 2.5* OR *backward error* OR *Definition 2.5*

introduced in section *Newton's Method for Solving Equations* can be very useful, for example as a step in getting bounds on the size of the error; to recap

---

Definition 2.9 (Backward Error)

The **backward error** in $\tilde{x}$ as an approxiate solution to the equation $f(x) = 0$ is $f(\tilde{x})$; the amount by which function f would have to be changed in order for $\tilde{x}$ to be an exact root.

For the case of solving simultaneous linear equations in matrix-vector form $Ax = b$, this is $b - A\tilde{x}$, also known as the **residual**.

Definition 2.10 (Absolute Backward Error)

The absolute backward error is — as you might have guessed — the absolute value of the backward error: $|f(\tilde{x})|$. This is sometimes also called simply the backward error. (The usual disclaimer about vector quantities applies.)

For the case of solving simultaneous linear equations in matrix-vector form $Ax = b$, this is $\|b - A\tilde{x}\|$, also known as the **residual norm**.

Remark 2.10

- One obvious advantage of the backward error concept is that you can actually evaluate it without knowing the exact solution x.
- Also, one significance of backward error is that if the values of $f(x)$ are only known to be accurate within an absolute error of E then any approximation with absolute backward error less than E could in fact be exact, so there is no point in seeking greater accuracy.
- The names *forward error* and *absolute forward error* are sometimes used as synonyms for *error* etc. as defined above, when they need to be distinguished from backward errors.

2.5.2 Order of convergence of a sequence of approximations

Definition 2.11

We have seen that for the sequence of approximations x_k to a quantity x given by the fixed point iteration $x_{k+1} = g(x_k)$, the absolute errors $E_k := |x_k - x|$ typically have

$$\frac{E_{k+1}}{E_k} \to C = |g'(x)|.$$

so that eventually the errors diminsh in a roughly geometric fashion: $E_k \approx KC^k$. This is called **linear convergence**.

Aside: Why "linear" rather than "geometric"? Because there is an approximately linear relationship between consecutive error values,

$$E_{n+1} \approx CE_n.$$

This is a very common behavior for iterative numerical methods, but we will also see that a few methods do even better; for example, when Newton's method converges to a simple root r of f (one with $f'(r) \neq 0$)

$$E_{k+1} \approx CE_k^2$$

This is called **quadratic convergence**. More generally:

Definition 2.12 (convergence of order p)

This is when

$$E_{k+1} \approx CE_k^p, \text{ or more precisely, } \lim_{k\to\infty} \frac{E_{k+1}}{E_k^p} \text{ is finite.}$$

We have already observed experimentally the intermediate result that "$C = 0$" for Newton's method in this case; that is,

$$\frac{E_{k+1}}{E_k} \to 0. \tag{2.7}$$

Definition 2.13 (super-linear convergence)

When the successive errors behave as in Equation (2.7) the convergence is **super-linear**. This includes any situation with order of convergence $p > 1$.

For most practical purposes, if you have established super-linear convergence, you can be happy, and not worry much about refinements like the particular order p.

2.5.3 Big-O and little-o notation

Consider the error formula for approximation of a function f with the Taylor polynomial of degree n, center a:

$$|f(a+h) - T_n(h)| \leq \frac{M_{n+1}}{(n+1)!}|h|^{n+1} \text{ where } M_{n+1} = \max|f^{(n+1)}(x)|.$$

Since the coefficient of h^{n+1} is typicaly not known in practice, it is wise to focus on the power law part, and for this the "big-O" and little-o" notation is convenient.

If a function $E(h)$ goes to zero at least as fast as h^p, we say that it is *of order* h^p, written $O(h^p)$.

More precisely, $E(h)$ is no bigger than a multiple of h^p for h small enough; that is, there is a constant C such that for some positive number δ

$$\frac{|E(h)|}{|h|^p} \leq C \text{ for } |h| < \delta.$$

Another way to say this is in terms of the *lim-sup*, if you have seen that jargon:

$$\limsup_{h \to 0} \frac{|E(h)|}{|h|^p} \text{ is finite.}$$

This can be used to rephrase the above Taylor's theorem error bound as

$$f(x) - T_n(x) = O(|x-a|^{n+1})$$

or

$$f(a+h) - T_n(h) = O(h^{n+1}),$$

and for the case of the linearization,

$$f(a+h) - L(x) = f(a+h) - (f(a) + f'(a)h) = O(h^2).$$

Little-o notation, for "negligibly small terms"

Sometimes it is enough to say that some error term is small enough to be neglected, at least when h is close enough to zero. For example, with a Taylor series we might be able to neglect the powers of $x-a$ or of h higher than p.

We will thus say that a quantity $E(h)$ is *small of order* h^p, written $o(h^p)$ when

$$\lim_{h \to 0} \frac{|E(h)|}{|h|^p} = 0.$$

Note the addition of the word **small** compared to the above description of the big-O case!

With this, the Taylor's theorem error bound can be stated as

$$f(x) - T_n(x) = o(|x-a|^n),$$

or

$$f(a+h) - T_n(h) = o(h^n),$$

and for the case of the linearization,

$$f(a+h) - L(x) = f(a+h) - (f(a) + f'(a)h) = o(h).$$

2.6 The Convergence Rate of Newton's Method

References:

- Section 1.4.1 *Quadratic Convergence of Newton's Method* in [Sauer, 2019].
- Theorem 2.9 in Section 2.4 *Error Analysis of Iterative Methods* in [Burden *et al.*, 2016], but done quite differently.

Jumping to the punch line, we will see that when the iterates x_k given by Newton's method converge to a *simple root* r (that is, *a solution of* $f(r) = 0$ *with* $f'(r) \neq 0$) then the errors $E_k = x_k - r$ satisfy

$$E_{k+1} = O(E_k^2) \text{ and therefore } E_{k+1} = o(E_k)$$

In words, the error at each iteration is *of the order of* the square of the previous error, and so is *small of order* the previous error.

(Yes, it this a slight abuse of the notation as defined above, but all will become clear and rigorous soon.)

The first key step is getting a recursive relationship between consecutive errors E_k and E_{k+1} from the recursion formula for Newton's method,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Start by subtracting r:

$$E_{k+1} = x_{k+1} - r = x_k - \frac{f(x_k)}{f'(x_k)} - r = E_k - \frac{f(x_k)}{f'(x_k)}$$

The other key step is to show that the two terms at right are very close, using the linearization of f at x_k with the error E_k as the small term h, and noting that $r = x_k - E_k$:

$$0 = f(r) = f(x_k - E_k) = f(x_k) - f'(x_k)E_k + O(E_k^2)$$

Solve for $f(x_k)$ to insert into the numerator above: $f(x_k) = f'(x_k)E_k + O(E_k^2)$. (There is no need for a minus sign on that last term; big-O terms can be of either sign, and this new one is a different but still small enough quantity!)

Inserting above,

$$E_{k+1} = E_k - \frac{f'(x_k)E_k + O(E_k^2)}{f'(x_k)} = E_k - E_k + \frac{O(E_k^2)}{f'(x_k)} = \frac{O(E_k^2)}{f'(x_k)} \to \frac{O(e_k^2)}{f'(r)} = O(E_k^2)$$

As $k \to \infty$, $f'(E_k) \to f'(r) \neq 0$, so the term at right is still no larger than a multiple of E_k^2: it is $O(E_k^2)$, as claimed.

If you wish to verify this more carefully, note that

- this $O(E_k^2)$ term is no bigger than $\frac{M}{2}E_k^2$ where M is an upper bound on $|f''(x)|$, and
- once E_k is small enough, so that x_k is close enough to r, $|f'(x_k)| \geq |f'(r)|/2$.

Thus the term $\dfrac{O(E_k^2)}{f'(x_k)}$ has magnitude no bigger than $\dfrac{M/2}{|f'(r)|/2}E_k^2 = \dfrac{M}{|f'(r)|}E_k^2$, which meets the definition of being of order E_k^2.

A more careful calculation actually shows that

$$\lim_{k\to\infty} \frac{|E_{k+1}|}{E_k^2} = \left|\frac{f''(r)}{2f'(r)}\right|,$$

which is the way that this result is often stated in texts. For either form, it then easily follows that

$$\lim_{k\to\infty} \frac{|E_{k+1}|}{|E_k|} = 0,$$

giving the *super-linear convergence* already seen using the Contraction Mapping Theorem, now restated as $E_{k+1} = o(E_k)$.

2.6.1 A Practical error estimate for fast-converging iterative methods

One problem for Newton's Method (and many other numerical methods we will see) is that there is not a simple way to get a guaranteed upper bound on the absolute error in an approximation. Our best hope is finding an interval that is guaranteed to contain the solution, as the Bisection Method does, and we can *sometimes* also do that with Newton's Method for a real root. But that approach fails as soon as the solution is a complex number or a vector.

Fortunately, when convergnce is "fast enough" is some sense, the following *heuristic* or "rule of thumb" applies in many cases:

The error in the latest approximation is typically smaller than the difference between the two most recent approximations.

When combined with the *backward error*, this can give a fairly reliable measure of accuracy, and so can serve as a fairly reliable *stopping condition* for the loop in an iterative calculation.

When is a fixed point iteration "fast enough" for this heuristic?

This heuristic can be shown to be reliable in several important cases:

Proposition

For the iterations x_k given by a contraction mapping that has $C \leq 1/2$,

$$|E_k| \leq |x_k - x_{k-1}|,$$

or in words *the error in x_k is smaller than the change from x_{k-1} to x_k*, so the above guideline is valid.

Proposition

For a super-linearly convergent iteration, eventually $|E_{k+1}|/|E_k| < 1/2$, and from that point onwards in the iterations, the above applies again.

I leave verification as an exercise, or if you wish, to discuss in class.

2.7 Root-finding without Derivatives

References:

- Section 1.5 *Root-finding without Derivatives* of [Sauer, 2019].
- The second part of Section 2.3 *Newton's Method and Its Extensions* in [Burden *et al.*, 2016], about the Secant method.

2.7.1 Introduction

In section *Root Finding by Interval Halving (Bisection)*) we have seen one method for solving $f(x) = 0$ without needing to know any derivatives of f: the *Bisection Method* a.k.a. *Interval Halving*.

However, we have also seen that that method is far slower then Newton's Method; here we explore methods that are almost the best of both worlds: about as fast as Newton's method but not needing derivatives.

The first of these is the Secant Method. Later in this course we will see how this has been merged with the Bisection Method and *Polynomial Interpolation* to produce the current state-of-the-art approach; only perfected in the 1960's.

```
using PyPlot
```

2.7.2 Using Linear Approximation Without Derivatives

One quirk of the *Bisection Method* is that it only used the sign of the values $f(a)$ and $f(b)$, not their magnitudes. If one of these is far smaller than the other, one might guess that the root is closer to that end of the interval. This leads to the idea of:

- starting with an interval $[a, b]$ known to contain a zero of f,
- connecting the two points $(a, f(a))$ and $(b, f(b))$ with a straight line, and
- finding the x-value c where this line crosses the x-axis. In the words, aproximating the function by a *secant line*, in place of the *tangent line* used in Newton's Method.

First Attempt: The Method of False Position

The next step requires some care. The first idea (from almost a millenium ago) was to use this new approximation c as done with bisection: check which of the intervals $[a, c]$ and $[c, b]$ has the sign change and use it as the new interval $[a, b]$; this is called *The Method of False Position* (or *Regula Falsi*, since the academic world used latin in those days.)

The secant line between $(a, f(a))$ and $(b, f(b))$ is

$$L(x) = \frac{f(a)(b-x) + f(b)(x-a)}{b-a}$$

and its zero is at

$$c = \frac{af(b) - f(a)b}{f(b) - f(a)}$$

This is easy to implement, and an example will show that it sort of works, but with a weakness that hampers it a bit:

Some helper functions for displaying numbers

```
roundoff(x) = round(x, sigdigits=12);  # Rounding off output to 12 significant digits,
 ↪ to clean up output
round3(x) = round(x, sigdigits=3);  # For rounding off errors,wher only a few␣
 ↪significant digits are needed
```

```
function falseposition(f, a, b, errortolerance; maxiterations=10, demomode=false)
    # Solve f(x)=0 in the interval [a, b] by the Method of False Position.
    # This code also illustrates a few ideas that I encourage, such as:
    # - Avoiding infinite loops, by using for loops sand break
    # - Avoiding repeated evaluation of the same quantity
    # - Use of descriptive variable names
    # - Use of "camelCase" to turn descriptive phrases into valid Julia variable names
    # - An optional "demonstration mode" to display intermediate results.

    if demomode
        println("Solving by the Method of False Position.")
    end;
    fa = f(a)
    fb = f(b)
    global c, errorbound  # So that they persist after the end of the following for
↪loop!
    for iteration in 1:maxiterations
        if demomode
            println("\nIteration $iteration:")
        end;
        c = (a * fb - fa * b)/(fb - fa)
        fc = f(c)
        if fa * fc < 0
            b = c
            fb = fc  # N.B. When b is updated, so must be fb = f(b)
        else
            a = c
            fa = fc
        end;
        errorbound = b - a
        if demomode
            println("The root is in interval [$(roundoff(a)), $(roundoff(b))]")
            println("The new approximation is $(roundoff(c)) with error bound
↪$(round3(errorbound)), backward error $(round3(abs(fc)))")
        end
        if errorbound <= errortolerance
            break
        end;
    end;
    # Whether we got here due to accuracy of running out of iterations,
    # return the information we have, including an error bound:
    root = c  # the newest value is probably the most accurate
    return (root, errorbound)
end;
```

```
f(x) = x - cos(x)
a = -1.0
b = 1.0;
```

```
errortolerance = 1e-12
(root, errorbound) = falseposition(f, a, b, errortolerance; demomode=true)
println()
println("The Method of False Position gave approximate root $root")
println(" with estimate error $errorbound and backward error $(abs(f(root)))")
```

```
Solving by the Method of False Position.

Iteration 1:
The root is in interval [0.540302305868, 1.0]
The new approximation is 0.540302305868 with error bound 0.46, backward error 0.317

Iteration 2:
The root is in interval [0.728010361468, 1.0]
The new approximation is 0.728010361468 with error bound 0.272, backward error 0.
 ↪0185

Iteration 3:
The root is in interval [0.738527006242, 1.0]
The new approximation is 0.738527006242 with error bound 0.261, backward error 0.
 ↪000934

Iteration 4:
The root is in interval [0.739057166678, 1.0]
The new approximation is 0.739057166678 with error bound 0.261, backward error 4.
 ↪68e-5

Iteration 5:
The root is in interval [0.739083732278, 1.0]
The new approximation is 0.739083732278 with error bound 0.261, backward error 2.
 ↪34e-6

Iteration 6:
The root is in interval [0.739085063039, 1.0]
The new approximation is 0.739085063039 with error bound 0.261, backward error 1.
 ↪17e-7

Iteration 7:
The root is in interval [0.7390851297, 1.0]
The new approximation is 0.7390851297 with error bound 0.261, backward error 5.88e-
 ↪9

Iteration 8:
The root is in interval [0.739085133039, 1.0]
The new approximation is 0.739085133039 with error bound 0.261, backward error 2.
 ↪95e-10

Iteration 9:
The root is in interval [0.739085133206, 1.0]
The new approximation is 0.739085133206 with error bound 0.261, backward error 1.
 ↪48e-11

Iteration 10:
The root is in interval [0.739085133215, 1.0]
The new approximation is 0.739085133215 with error bound 0.261, backward error 7.
 ↪39e-13

The Method of False Position gave approximate root 0.7390851332147188
 with estimate error 0.2609148667852812 and backward error 7.394085344003543e-13
```

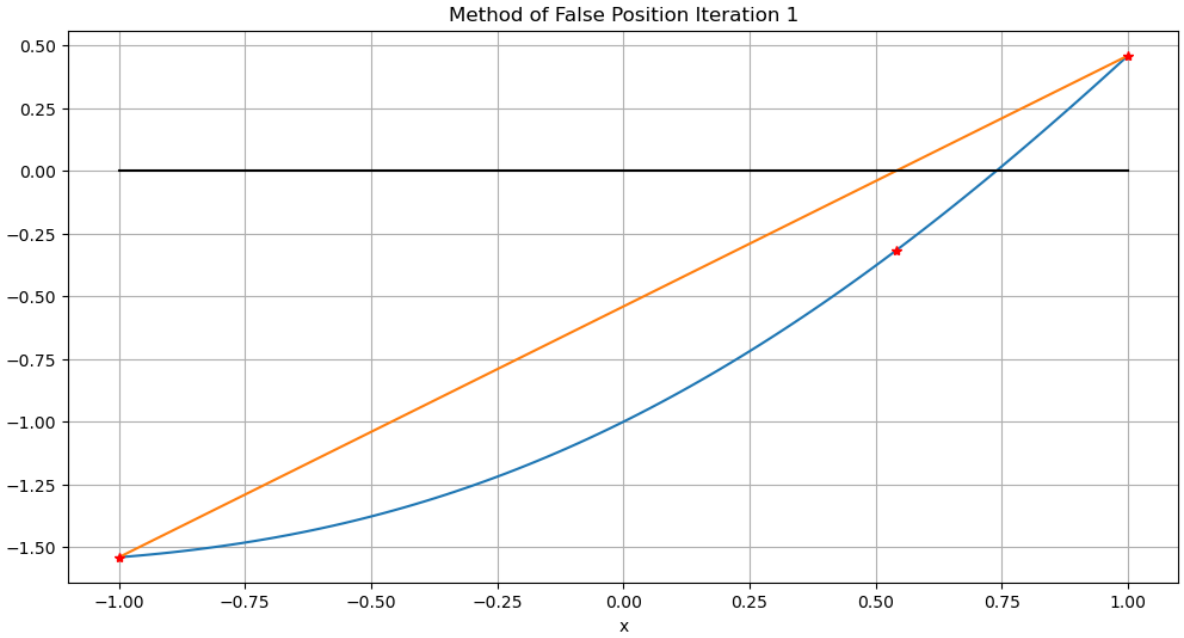
The good news is that the approximations are approaching the zero reasonably fast — far faster than bisection — as indicated by the backward errors improving by a factor of better than ten at each iteration.

The bad news is that one end gets "stuck", so the interval does not shrink on both sides, and the error bound stays large.

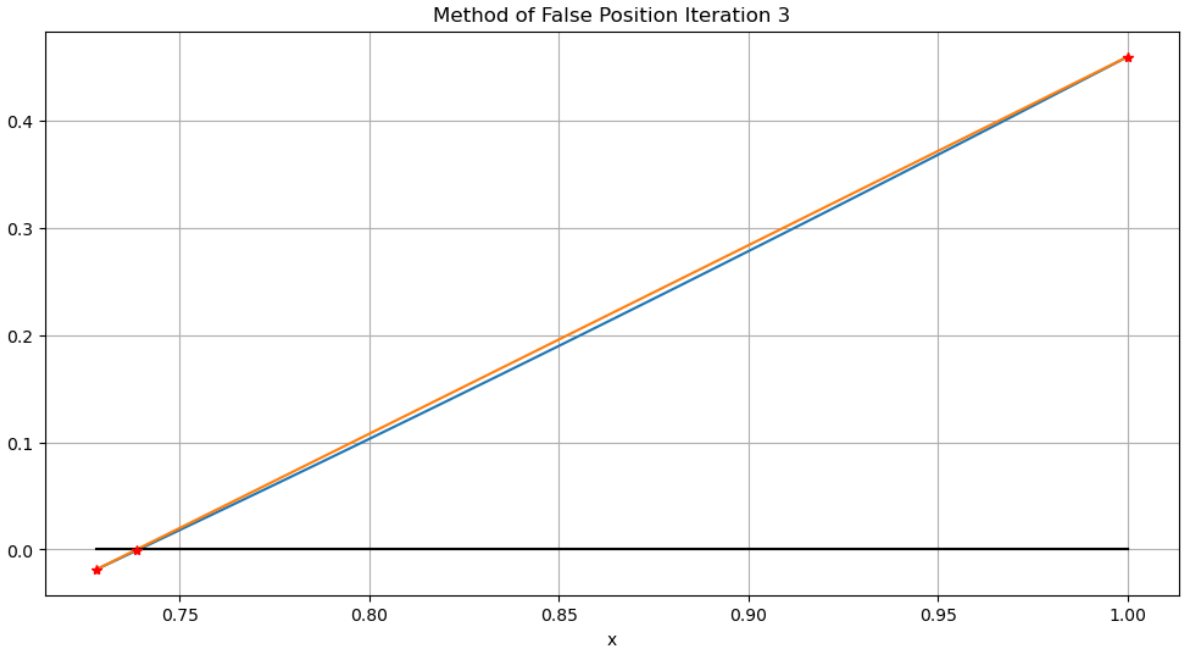
This behavior is generic: with function f of the same convexity on the interval $[a, b]$, the secant line will always cross on the same side of the zero, so that one end-point persists; in this case, the curve is concave up, so the secant line always crosses to the left of the root, as seen in the following graphs.

```
function graphfalseposition(f, a, b; maxiterations=3)
    # Graph a few iterations of the Method of False Position for solving f(x)=0 in␣
 ↪the interval [a, b].

    fa = f(a)
    fb = f(b)
    for iteration in 1:maxiterations
        c = (a * fb - fa * b)/(fb - fa)
        fc = f(c)
        abc = [a b c]
        # TODO: find min, max in julia!
        left = min(a, b)
        right = max(a, b)
        xplot = range(left, right, 100);
        figure(figsize=[12,6])
        grid(true)
        title("Method of False Position Iteration $iteration")
        xlabel("x")
        plot(xplot, f.(xplot))
        plot([left, right], [f(left), f(right)])  # the secant line
        plot([left, right], [0, 0], "k")  # the x-axis line
        plot(abc, f.(abc), "r*")
        #show()  # The Windows version of JupytLab might need this command; it is␣
 ↪harmless anyway.
        if fa * fc < 0
            b = c
            fb = fc  # N.B. When b is updated, so must be fb = f(b)
        else
            a = c
            fa = fc
        end;
    end;
end;
```

```
graphfalseposition(f, a, b)
```


Method of False Position Iteration 1
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x


Method of False Position Iteration 2
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
0.6
0.7
0.8
0.9
1.0
x



Refinement: Alway Use the Two Most Recent Approximations — The Secant Method

The basic solution is to always discard the oldest approximation — at the cost of not always having the zero surrounded! This gives the Secant Method.

For a mathemacal description, one typically enumerates the successive approximations as x_0, x_1, etc., so the notation above gets translated with $a \to x_{k-2}$, $b \to x_{k-1}$, $c \to x_k$; then the formula becomes the recursive rule

$$x_k = \frac{x_{k-2}f(x_{k-1}) - f(x_{k-2})x_{k-1}}{f(x_{k-1}) - f(x_{k-2})}$$

Two difference from above:

- previously we could assume that $a < b$, but now we do not know the order of the various x_k values, and
- the root is not necessarily bewtween the two most recent values, so we no longer have tht simple error bound. (In fact, we will see that the zero is typically surrounded two-thirds of the time!)

Instead, we use the *magnitude* of $b - a$ which is now $|x_k - x_{k-1}|$, and this is only an *estimate* of the error. This is the same as used for Newton's Method; as there, it is still useful as a condition for ending the iterations and indeed tends to be pessimistic, so that we typically do one more iteration than needed — but it is not on its own a complete guarantee of having achieved the desired accuracy.

Pseduo-code for a Secant Method Algorithm

Algorithm 2.4 (Secant Method)

Input function f, interval endpoints x_0 and x_1, an error tolerance E_{tol}, and an iteration limit N

for k from 2 to N

$$x_k \leftarrow \frac{x_{k-2}f(x_{k-1}) - f(x_{k-2})x_{k-1}}{f(x_{k-1}) - f(x_{k-2})}$$

Evaluate the error estimate $E_{est} \leftarrow |x_k - x_{k-1}|$

if $E_{est} \leq E_{tol}$

End the iterations

else

Go around another time

end

end

Output the final x_k as the approximate root and E_{est} as an estimate of its absolute error.

Julia Code for this Secant Method Algorithm

We could write Julia code that closely follows this notation, accumulating a list of the values x_k.

However, since we only ever need the two most recent values to compute the new one, we can instead just store these three, in the same way that we recylced the variables `a`, `b` and `c`. Here I use more descriptive names though:

```
function secantmethod(f, a, b, errortolerance; maxiterations=20, demomode=false)
    # Solve f(x)=0 in the interval [a, b] by the Secant Method.

    if demomode
        print("Solving by the Secant Method.")
    end;
    # Some more descriptive names
    x_older = a
    x_more_recent = b
    f_x_older = f(x_older)
    f_x_more_recent = f(x_more_recent)
    for iteration in 1:maxiterations
        global x_new, errorestimate
        if demomode
            println("\nIteration $(iteration):")
        end;
        x_new = (x_older * f_x_more_recent - f_x_older * x_more_recent)/(f_x_more_
↪recent - f_x_older)
        f_x_new = f(x_new)
        (x_older, x_more_recent) = (x_more_recent, x_new)
        (f_x_older, f_x_more_recent) = (f_x_more_recent, f_x_new)
        errorestimate = abs(x_older - x_more_recent)
        if demomode
            println("The latest pair of approximations are $(roundoff(x_older)) and
↪$(roundoff(x_more_recent)),")
            println("where the function's values are $(roundoff(f_x_older)) and
↪$(roundoff(f_x_more_recent)) respectively.")
            print("The new approximation is $(roundoff(x_new)),")
            println("with estimated error $(round3(errorestimate)) and backward error
↪$(round3(abs(f_x_new)))")
        end;
        if errorestimate < errortolerance
            break
        end;
    end;
```

(continues on next page)

(continued from previous page)

```
    # Whether we got here due to accuracy of running out of iterations,
    # return the information we have, including an error estimate:
    return (x_new,  errorestimate)
end;
```

```
errortolerance = 1e-12
(root, errorestimate) = secantmethod(f, a, b, errortolerance, demomode=true)

println()
println("The Secant Method gave approximate root $root,")
print("with estimated error $errorestimate, backward error $(abs(f(root)))")
```

```
Solving by the Secant Method.
Iteration 1:
The latest pair of approximations are 1.0 and 0.540302305868,
where the function's values are 0.459697694132 and -0.317250909978 respectively.
The new approximation is 0.540302305868,with estimated error 0.46 and backward␣
 ↪error 0.317

Iteration 2:
The latest pair of approximations are 0.540302305868 and 0.728010361468,
where the function's values are -0.317250909978 and -0.0184893945776 respectively.
The new approximation is 0.728010361468,with estimated error 0.188 and backward␣
 ↪error 0.0185

Iteration 3:
The latest pair of approximations are 0.728010361468 and 0.739627012631,
where the function's values are -0.0184893945776 and 0.000907004400407␣
 ↪respectively.
The new approximation is 0.739627012631,with estimated error 0.0116 and backward␣
 ↪error 0.000907

Iteration 4:
The latest pair of approximations are 0.739627012631 and 0.739083800783,
where the function's values are 0.000907004400407 and -2.22997338051e-6␣
 ↪respectively.
The new approximation is 0.739083800783,with estimated error 0.000543 and backward␣
 ↪error 2.23e-6

Iteration 5:
The latest pair of approximations are 0.739083800783 and 0.739085133056,
where the function's values are -2.22997338051e-6 and -2.66740518562e-10␣
 ↪respectively.
The new approximation is 0.739085133056,with estimated error 1.33e-6 and backward␣
 ↪error 2.67e-10

Iteration 6:
The latest pair of approximations are 0.739085133056 and 0.739085133215,
where the function's values are -2.66740518562e-10 and 0.0 respectively.
The new approximation is 0.739085133215,with estimated error 1.59e-10 and backward␣
 ↪error 0.0

Iteration 7:
The latest pair of approximations are 0.739085133215 and 0.739085133215,
where the function's values are 0.0 and 0.0 respectively.
```

(continues on next page)

(continued from previous page)

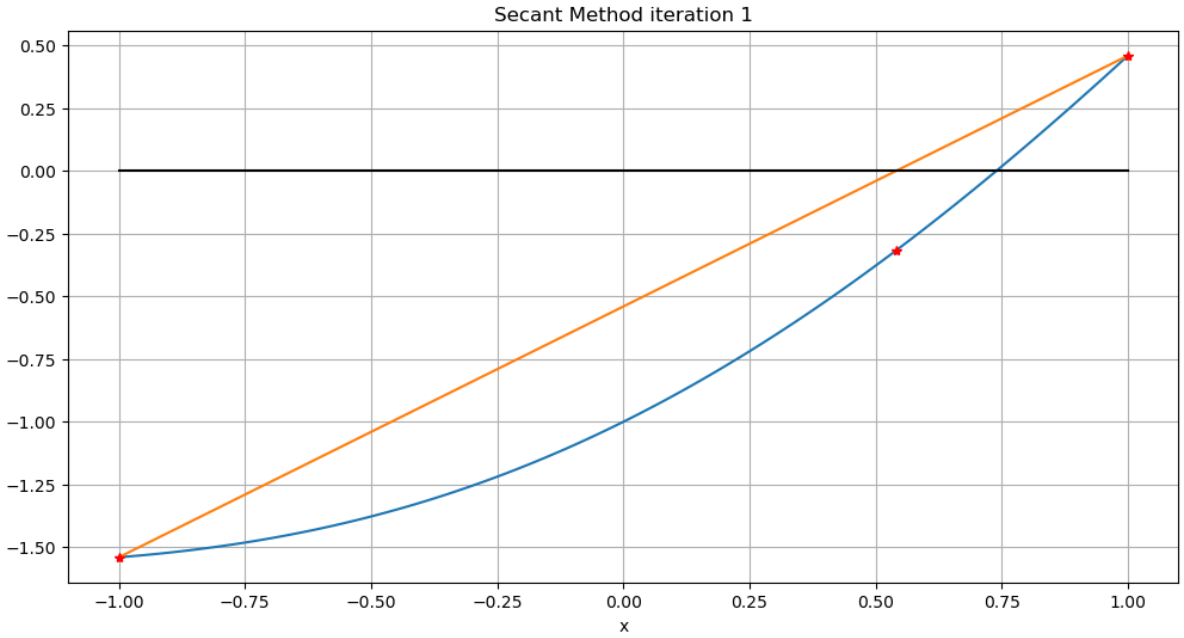
```
The new approximation is 0.739085133215,with estimated error 0.0 and backward␣
 ↪error 0.0

The Secant Method gave approximate root 0.7390851332151607,
with estimated error 0.0, backward error 0.0
```

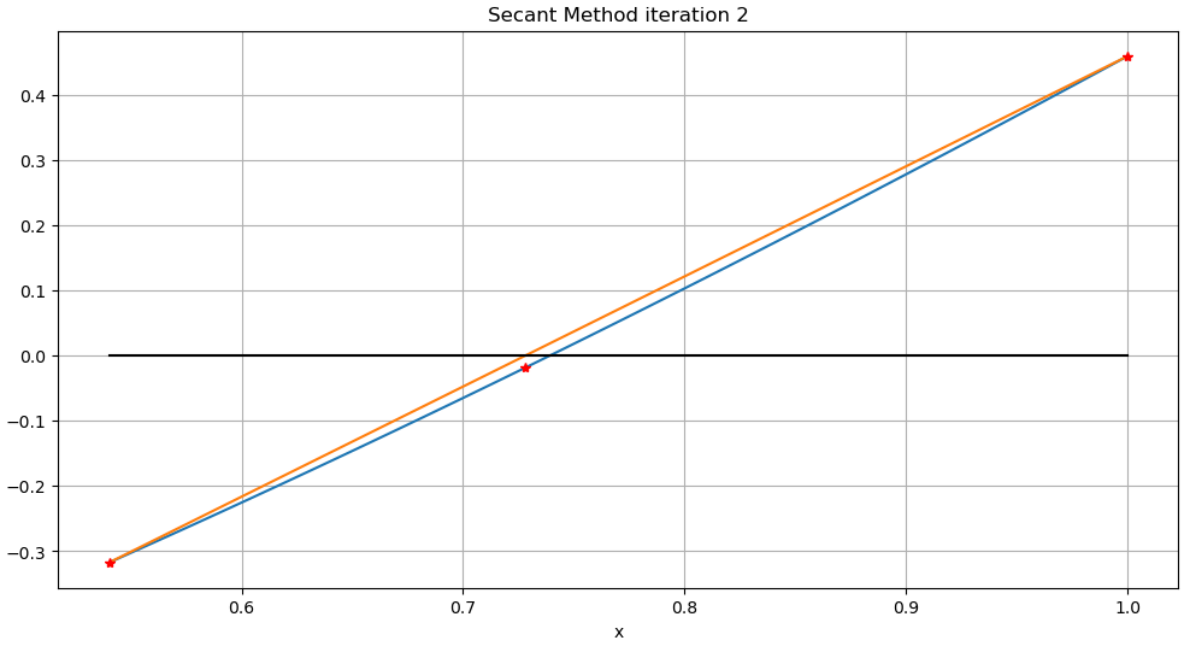
Alternatively, you get a version of `secantmethod` from the module `NumericalMethods` with

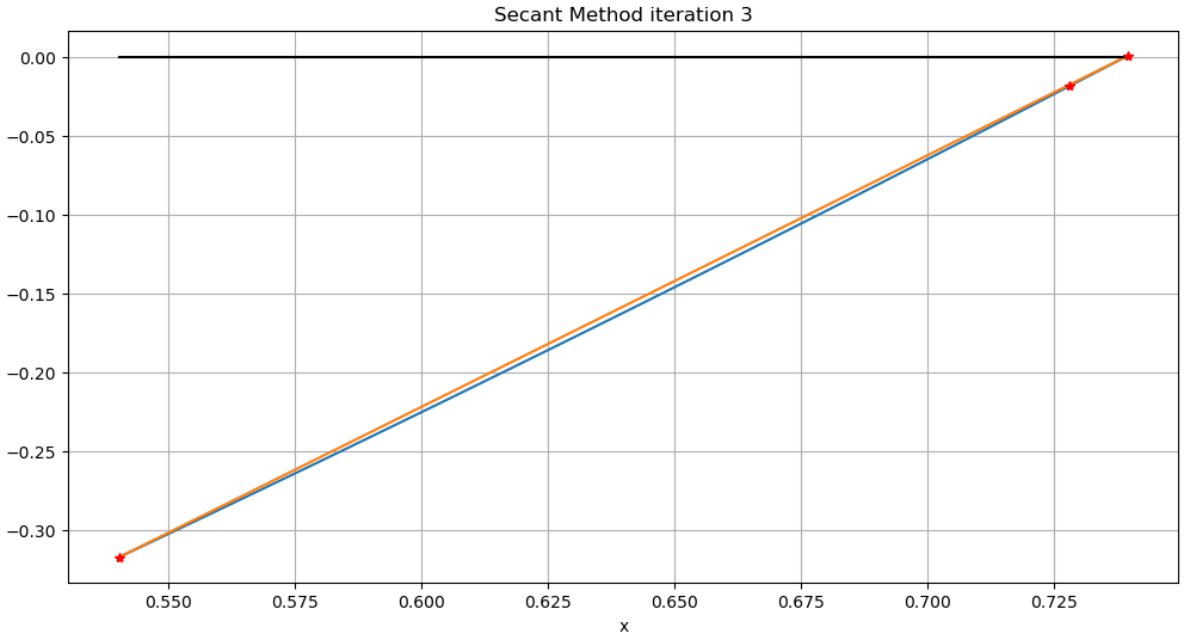
```
import .NumericalMethods: secantmethod
```

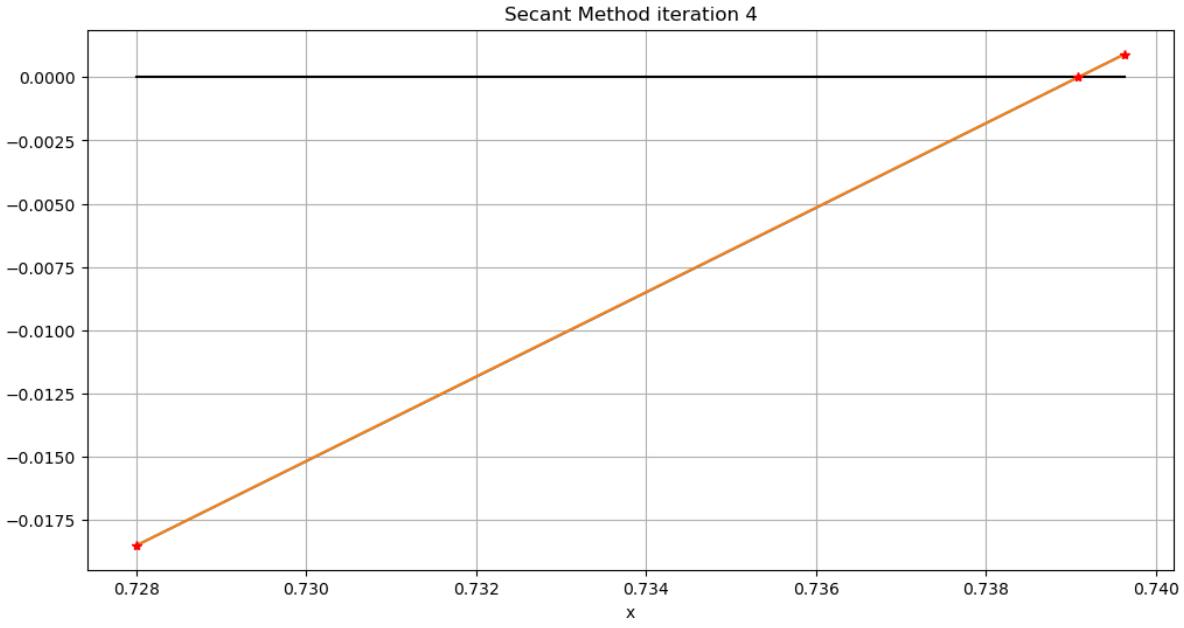
```
function graphsecantmethod(f, a, b; maxiterations=5)
    # Graph a few iterations of the Secant Method for solving f(x)=0 in the interval␣
 ↪[a, b].

    x_older = a
    x_more_recent = b
    f_x_older = f(x_older)
    f_x_more_recent = f(x_more_recent)
    for iteration in 1:maxiterations
        x_new = (x_older * f_x_more_recent - f_x_older * x_more_recent)/(f_x_more_
 ↪recent - f_x_older)
        f_x_new = f(x_new)
        latest_three_x_values = [x_older x_more_recent x_new]
        left = min(x_older, x_more_recent, x_new)
        right = max(x_older, x_more_recent, x_new)
        xplot = range(left, right, 100);
        figure(figsize=[12,6])
        grid(true)
        title("Secant Method iteration $iteration")
        xlabel("x")
        plot(xplot, f.(xplot))
        plot([left, right], [f(left), f(right)])  # the secant line
        plot([left, right], [0, 0], "k")  # the x-axis line
        plot(latest_three_x_values, f.(latest_three_x_values), "r*")
        x_older = x_more_recent
        f_x_older = f_x_more_recent
        x_more_recent = x_new
        f_x_more_recent = f_x_new
        errorEstimate = abs(x_older - x_more_recent)
    end;
end;
```

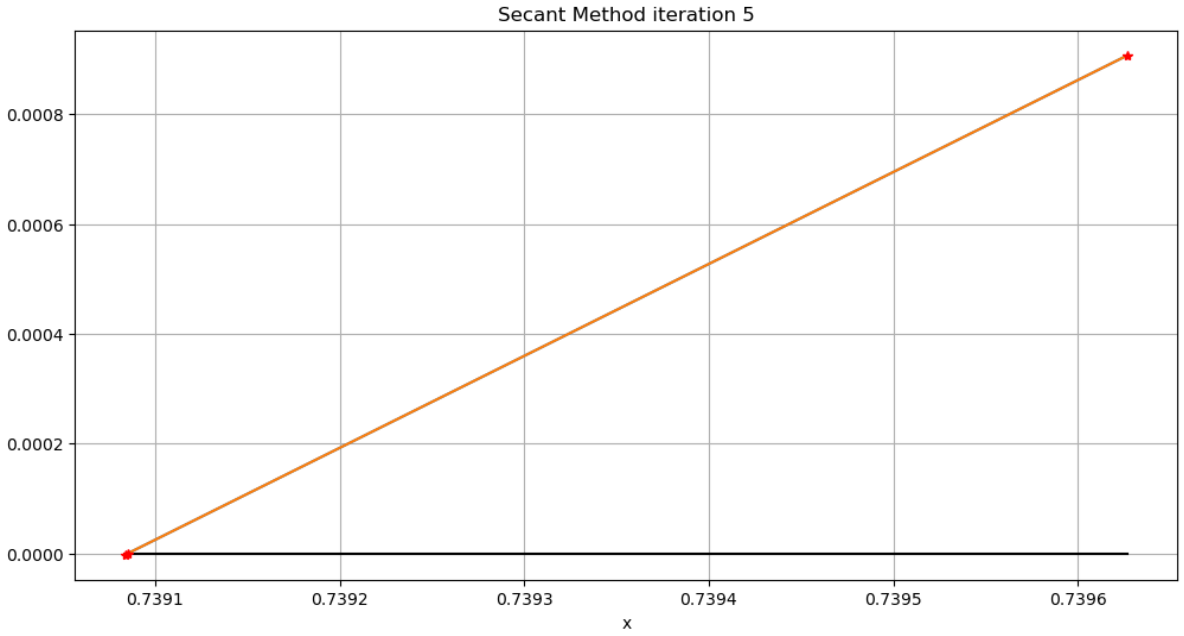
```
graphsecantmethod(f, a, b)
```


Secant Method iteration 1
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x


Secant Method iteration 2
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
0.6
0.7
0.8
0.9
1.0
x


Secant Method iteration 3
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
−0.30
0.550
0.575
0.600
0.625
0.650
0.675
0.700
0.725
x


Secant Method iteration 4
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
−0.0125
−0.0150
−0.0175
0.728
0.730
0.732
0.734
0.736
0.738
0.740
x



Observation 2.1

- This converges faster than the *Method of False Position* (and far faster than Bisection).
- The majority of iterations have the root surrounded (sign-change in f), but every third one — the second and fifth — do not.
- Comparing the error estimate to the backward error, the error estimate is quite pessimistic (and so fairly trustworthy); in fact, it is typically of similar size to the backward error at the *previous* iteration.

The last point is a quite common occurence: the available error estimates are often "trailing indicators", closer to the error in the previous approximation in an iteration. For example, recall that we saw the same thing with Newton's Method when we used $|x_k - x_{k-1}|$ to estimate the error $E_k := x_k - r$ and saw that it is in fact closer to the previous error, E_{k-1}.

CHAPTER

THREE

LINEAR ALGEBRA AND SIMULTANEOUS EQUATIONS

3.1 Row Reduction/Gaussian Elimination

References:

- Section 2.1.1 *Naive Gaussian elimination* of [Sauer, 2019].
- Section 6.1 *Linear Systems of Equations* of [Burden *et al.*, 2016].
- Section 7.1 of [Chenney and Kincaid, 2012].

3.1.1 Introduction

The problem of solving a system of n simultaneous linear equations in n unknowns, with matrix-vector form $Ax = b$, is quite thoroughly understood as far as having a good general-purpose methods usable with any $n \times n$ matrix A: essentially, Gaussian elimination (or row-reduction) as seen in most linear algebra courses, combined with some modifications to stay well away from division by zero: *partial pivoting*. Also, good robust software for this general case is readily available, for example in the Julia package `LinearAlgebra`.

Nevertheless, this basic algorithm can be very slow when n is large – as it often is when dealing with differential equations (even more so with *partial* differential equations). We will see that it requires about $n^3/3$ arithmetic operations.

Thus I will summarise the basic method of row reduction or Gaussian elimination, and then build on it with methods for doing things more robustly, and on methods for doing it faster in some important special cases:

1. When one has to solve many systems $Ax^{(m)} = b^{(m)}$ with the same matrix A but different right-hand side vectors $b^{(m)}$.
2. When A is *banded*: most elements are zero, and all the non-zero elements $a_{i,j}$ are near the main diagonal: $|i - j|$ is far less than n. (*Aside on notation:* "far less than" is sometimes denoted $\ll$, as in $|i - j| \ll n$.)
3. When A is *strictly diagonally dominant*: each diagonal element $a_{i,i}$ is larger in magnitude that the sum of the magnitudes of all other elements in the same row.

Other cases not (yet) discussed in this text are

1. When A is *positive definite*: symmetric ($a_{i,j} = a_{j,i}$) and with all eigenvalues positive. This last condition would seem hard to verify, since computing all the eigenvalues of A is harder that solving $Ax = b$, but there are important situations where this property is automatically guaranteed, such as with *Galerkin* and *finite-element methods* for solving boundary value problems for differential equations.
2. When A is *sparse*: most elements are zero, but not necessarily with all the non-zero elements near the main diagonal.

3.1.2 Strategy for getting from mathematical facts to a good algorithm and then to its implentation in [Julia] code

Here I take the opportunity to illustrate some useful strategies for getting from mathematical facts and ideas to good algorithms and working code for solving a numerical problem. The pattern we will see here, and often later, is:

Step 1. Get a basic algorithm:

1. Start with mathematical facts (like the equations $\sum_{j=1}^{n} a_{ij}x_j = b_i$).
2. Solve to get an equation for each unknown — or for an updated aproximation of each unknown — in terms of other quantitities.
3. Specify an order of evaluation in which all the quantities at right are evaluated earlier.

In this, it is often best to start with a verbal description before specifying the details in more precise and detailed mathematical form.

Step 2. Refine to get a more robust algorithm:

1. Identify cases that can lead to failure due to division by zero and such, and revise to avoid them.
2. Avoid inaccuracy due to problems like severe rounding error. One rule of thumb is that anywhere that a zero value is a fatal flaw (in particular, division by zero), a very small value is also a hazard when rounding error is present. So **avoid very small denominators**. (We will soon examine this through the phenomenon of **loss of significance** and it extreme case **catastrophic cancellation**.)

Step 3. Refine to get a more efficient algorithm

For example,

- Avoid repeated evaluation of exactly the same quantity.
- Avoid redundant calculations, such as ones whose value can be determnied in advance; for example, values that can be shown in advance to be zero.
- Compare and choose between alternative algorithms.

3.1.3 Gaussian elimination, a.k.a. row reduction

We start by considering the most basic algorithm, based on ideas seen in a linear algebra course.

The problem is best stated as a collection of equations for individual numerical values:

Given coefficients $a_{i,j}1 \leq i \leq n$, $1 \leq j \leq n$ and right-hand side values b_i, $1 \leq i \leq n$, solve for the n unknowns x_j, $1 \leq j \leq n$ in the equations $$\sum_{j=1}^{n} a_{i,j}x_j = b_i,\ 1 \leq i \leq n.$$

In verbal form, the basic strategy of *row reduction* or *Gaussian elimination* is this:

- **Choose** one equation and use it to eliminate one **chosen** unknown from all the other equations, leaving that chosen equation plus $n-1$ equations in $n-1$ unknowns.
- Repeat recursively, at each stage using one of the remaining equations to eliminate one of the remaining unknowns from all the other equations.
- This gives a final equation in just one unknown, preceeded by an equation in that unknown plus one other, and so on: solve them in this order, from last to first.

Determining those choices, to produce a first algorithm: "naive gaussian elimination"

A precise algorithm must include rules specifying all the choices indicated above. The simplest "naive" choice, which works in most but not all cases, is to eliminate from the top to bottom and left to right:

- Use the first equation to eliminate the first unknown from all other equations.
- Repeat recursively, at each stage using the first remaining equation to eliminate the first remaining unknown. Thus, at step k, equation k is used to eliminate unknown x_k.
- This gives one equation in just the last unknown x_n; another equation in the last two unknowns x_{n-1} and x_n, and so on: solve them in this reverse order, evaluating the unknowns from last to first.

This usually works, but can fail because at some stage the (updated) k-th equation might not include the k-th unknown: that is, its coefficient might be zero, leading to division by zero.

We will refine the algorithm to deal with that in the later section *Partial Pivoting*.

3.1.4 The general case of solving $Ax = b$

The problem of solving $Ax = b$ in general, when all you know is that A is an $n \times n$ matrix and b is an n-vector, can in most cases be handled well by using standard software rather than by writing your own code. Here is an example in Julia, solving

$$\begin{bmatrix} 4 & 2 & 7 \\ 3 & 5 & -6 \\ 1 & -3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

```
A = [4.0 2.0 7.0; 3.0 5.0 -6.0; 1.0 -3.0 2.0]
println("A =\n$(A)")
b = [2.0; 3.0; 4.0]
println("b = $(b)")
println("A*b = $(A*b)")
```

```
A =
[4.0 2.0 7.0; 3.0 5.0 -6.0; 1.0 -3.0 2.0]
b = [2.0, 3.0, 4.0]
A*b = [42.0, -3.0, 1.0]
```

Remark 3.1 (On Julia)

- See the notes on *Arrays* or *Arrays* in *Notes on the Julia Language*.
- Julia mimics Matlab's notation for "dividing from the left": the solution of $Ax = b$ is $x = A^{-1}b$ and given by `A\b`; it is not bA^{-1} which is what you get from the usual "divide from the right" notation of `b/A`.

See the notes on *Arrays* or *Julia arrays* or *Arrays* in *Notes on the Julia Language*.

```
x = A\b;
```

```
println("Julia says that the solution of Ax=b is x=$(x)")
```

```
Julia says that the solution of Ax=b is x=[1.8116883116883116, -1.0324675324675323,
↪ -0.45454545454545453]
```

Check the **residual** $b - Ax$, a measure of *backward error*:

```
r = b-A*x;
```

```
println()
println("As a check, the residual is")
println("    r = b - Ax = $(r)")
println("and its infinity (or 'maximum') norm is")
println("    ||r|| = $(maximum(abs.(r)))")
```

```
As a check, the residual is
    r = b - Ax = [0.0, 0.0, 8.881784197001252e-16]
and its infinity (or 'maximum') norm is
    ||r|| = 8.881784197001252e-16
```

Remark 3.2 (Not quite zero values and rounding)

Some values here that you might hope to be zero are instead very small non-zero numbers, with exponent 10^{-16}, due to rounding error in computer arithmetic. For details on this (like why "-16" in particular) see *Machine Numbers, Rounding Error and Error Propagation*.

3.1.5 The naive Gaussian elimination algorithm, in pseudo-code

Here the elements of the transformed matrix and vector after step k are named $a_{i,j}^{(k)}$ and $b_k^{(k)}$, so that the original values are $a_{i,j}^{(0)} = a_{i,j}$ and $b_i^{(0)} = b_i$.

The name $l_{i,k}$ is given to the multiple of row k that is subtracted from row i at step k. This naming might seem redundant, but it becomes very useful later.

Algorithm 3.1 (naive Gaussian elimination)

for k from 1 to n-1 *Step k: get zeros in column k below row k:*

for i from k+1 to n

Evaluate the multiple of row k to subtract from row i:

$l_{i,k} = a_{i,k}^{(k-1)} / a_{k,k}^{(k-1)}$ **If** $a_{k,k}^{(k-1)} \neq 0$!

Subtract ($l_{i,k}$ *times row k) from row i in matrix A …:*

for j from 1 to n

$a_{i,j}^{(k)} = a_{i,j}^{(k-1)} - l_{i,k} a_{k,j}^{(k-1)}$

end

… and at right, subtract ($l_{i,k}$ times b_k) from b_i:

$b_i^{(k)} = b_i^{(k-1)} - l_{i,k} b_k^{(k-1)}$

end

The rows before $i = k$ are unchanged, so they are ommited from the update; however, in a situation where we need to complete the definitions of $A^{(k)}$ and $b^{(k)}$ we would also need the following inside the `for k` loop:

Algorithm 3.2 (Inserting the zeros below the main diagonal)

for i from 1 to k

for j from 1 to n

$a_{i,j}^{(k)} = a_{i,j}^{(k-1)}$

end

$b_i^{(k)} = b_i^{(k-1)}$

end

However, the algorithm will usually be implemented by overwriting the previous values in an array with new ones, and then this part is redundant.

The next improvement in efficiency: the updates in the first k columns at step k give zero values (that is the key idea of the algorithm!), so there is no need to compute or store those zeros, and thus the only calculations needed in the above `for j from 1 to n` loop are covered by `for j from k+1 to n`. Thus from now on we use only the latter—except when, for demonstration purposes, we need those zeros.

Thus, the standard algorithm looks like this:

Algorithm 3.3 (basic Gaussian elimination)

for k from 1 to n-1 *Step k: Get zeros in column k below row k:*

for i from k+1 to n *Update only the rows that change: from k+1 on:*

Evaluate the multiple of row k to subtract from row i:

$l_{i,k} = a_{i,k}^{(k-1)} / a_{k,k}^{(k-1)}$ **If** $a_{k,k}^{(k-1)} \neq 0$!

Subtract ($l_{i,k}$ *times row k) from row i in matrix A, in the columns that are not automaticaly zero:*

for j from k+1 to n

$a_{i,j}^{(k)} = a_{i,j}^{(k-1)} - l_{i,k} a_{k,j}^{(k-1)}$

end

and at right, subtract ($l_{i,k}$ *times* b_k) *from* b_i*:*

$b_i^{(k)} = b_i^{(k-1)} - l_{i,k} b_k^{(k-1)}$

end

3.1.6 The naive Gaussian elimination algorithm, in Julia

Conversion to actual Julia code is now quite straightforward; there is litle more to be done than:

- Change the way that indices are described, from b_i to `b[i]` and from $a_{i,j}$ to `A[i,j]`.
- Use case consistently in array names, since the quirk in mathematical notation of using upper-case letters for matrix names but lower case letters for their elements is gone! In these notes, matrix names will be upper-case and vector names will be lower-case (even when a vector is considered as 1-column matrix).
- Rather than create a new array for each matrix $A^{(0)}$, $A^{(1)}$, etc. and each vector $b^{(0)}$, $b^{(1)}$, we overwite each in the same array.

```
for k in 1:n
    for i in k+1:n
        L[i,k] = A[i,k] / A[k,k]
        for j in k+1:n
            A[i,j] -= L[i,k] * A[k,j]
        end
        b[i] -= L[i,k] * b[k]
    end
end
```

To demonstrate this, some additions are needed:

- Putting this algorithm into a function.
- Getting the value n needed for the loop, using the fact that it is the length of vector `b`.
- Creating the array L.
- Copying the input arrays `A` and `b` into new ones, `U` and `c`, so that the original arrays are not changed. That is, when the row reduction is completed, `U` contains $A^{(n-1)}$ and `c` contains $b^{(n-1)}$.

Also, for some demonstrations, the zero values below the main diagonal of `U` are inserted, though usually they would not be needed.

```
function rowreduce(A, b)
    # To avoid modifying the matrix and vector specified as input,
    # they are copied to new arrays, with the function copy().
    # Warning: it does not work to say "U = A" and "c = b";
    # this makes these names synonyms, referring to the same stored data.

    U = copy(A)  # not "U=A", which makes U and A synonyms
    c = copy(b)
    n = length(b)
    L = zeros(n, n)
    for k in 1:n-1
        for i in k+1:n
            # compute all the L values for column k:
            L[i,k] = U[i,k] / U[k,k]  # Beware the case where U[k,k] is 0
            for j in k+1:n
                U[i,j] -= L[i,k] * U[k,j]
            end
            # Put in the zeros below the main diagonal in column k of U;
            # this is not important for calculations,
            # since those elements of U are not used in backward substitution,
            # but it helps for displaying results and for checking the results via␣
 ↪residuals.
            U[i,k] = 0.

            c[i] -= L[i,k] * c[k]
        end
    end
    for i in 2:n
        for j in 1:i-1
            U[i,j] = 0.
        end
    end
    return (U, c)
end;
```

Here is a helper function for displaying matrices. Since it will be used in several sections, it is also in the module `NumericalMethods`, along with the above function; see *Module NumericalMethods*.

Remark 3.3 (Julia Modules)

Modules and their usage are introduced in the notes on Using modules and packages. As explained there, these two functions can be obtained with

```
include("NumericalMethods.jl")
using .NumericalMethods: rowreduce
using .NumericalMethods: printmatrix
```

Details about creating your own modules will be given in creating modules once those notes are written; meanwhile, examining *Module NumericalMethods* could help: the actual module definition file is just the Julia code from there with the interspersed explanatory comments removed.

Some helper functions to cleanup output

```
function printmatrix(A)
    # A helper function to "pretty print" matrices

    (rows, cols) = size(A)
    print("[ ")
    for row in 1:rows
        if row > 1
            print("  ")
        end
        for col in 1:cols
            print(A[row,col], " ")
        end
        if row < rows;
            println()
        else
            println("]")
        end
    end
end;
```

```
# A shorthand for rounding off to n significant digits.
import Base: round
round(x, n::Integer) = round(x, sigdigits=n);
```

```
println("A is")
printmatrix(A)
println("b = $(b)")
```

```
A is
[ 4.0 2.0 7.0
  3.0 5.0 -6.0
  1.0 -3.0 2.0 ]
b = [2.0, 3.0, 4.0]
```

```
(U, c) = rowreduce(A, b);
```

```
println("Row reduction gives")
println("U =")
printmatrix(U)
println("c = $(c)")
```

```
Row reduction gives
U =
[ 4.0 2.0 7.0
  0.0 3.5 -11.25
  0.0 0.0 -11.0 ]
c = [2.0, 1.5, 5.0]
```

Let's take advantage of the fact that we have used Julia's built-in linear algebra command `b\A` to get a very accurate approximation of the solution x to $Ax = b$; this should also solve $Ux = c$, so check the backward error, a.k.a. the *residual*:

```
r = c - U * x
println("The residual (backward error) r = c-Ux is $(round.(r,4))," *
    " with maximum norm $(round(maximum(abs.(r)),4))")
```

```
The residual (backward error) r = c-Ux is [0.0, -2.22e-16, 0.0], with maximum norm␣
↪2.22e-16
```

Remark 3.4 (Array slicing in Julia)

Many operations in linear algebra can be expressed more concisely using array slicing and vectorization as described in *Notes on the Julia Language* allowing the loops above to be expressed as

```
for k in 1:n
    L[k+1:end,end] = A[k+1:end,k] / A[k,k]
    A[k+1:end,k+1:end] -= L[k+1:end,k] * A[[k],k+1:end]
    b[k+1:end] -= L[k+1:end,k] * b[k]
    end
end
```

Remark 3.5 (Matrix slicing in Julia)

One subtlety here, as mentioned in the notes on slicing: that row slice at the end of line 3 has to be done as `A[[k], k+1:end]` with brackets around the row index, in order to make it a 1-row matrix; using `A[k,k+1:end]` instead would give a vector, and then the product would fail.

I will break my usual guideline of non-repetition by redefining `rowreduce`, since this is just a restatement of exactly the same algorithm with different Julia notation.

While I am about it, I add a `demomode`, for display of intermediate results.

```
function rowreduce(A, b; demomode=false)
    # To avoid modifying the matrix and vector specified as input,
    # they are copied to new arrays, with the method .copy()
    # Warning: it does not work to say "U = A" and "c = b";
    # this makes these names synonyms, referring to the same stored data.
```

(continues on next page)

(continued from previous page)

```
    # This version vectorizes the inner loops, and all of the "i, j" loop for␣
↪updating U.

    if demomode
        println("rowreduce version 2: some loops vectorized")
    end
    U = copy(A)  # not "U=A", which makes U and A synonyms
    c = copy(b)
    n = length(b)
    L = zeros(n, n)
    for k in 1:n-1
        if demomode; println("Step $(k):"); end
        # compute all the L values for column k:
        L[k+1:end,k] = U[k+1:end,k] / U[k,k]  # Beware the case where U[k,k] is 0
        U[k+1:end,k+1:end] -= L[k+1:end,k] * U[[k],k+1:end]
        c[k+1:end] -= L[k+1:end,k] * c[k]

        # Insert the below-diagonal zeros in column k;
        # this is not important for calculations,
        # since those elements of U are not used in backward substitution,
        # but it helps for displaying results and for checking the results via␣
↪residuals.
        U[k+1:end,k] .= 0.0

        if demomode
            println("After step $k the matrix is")
            printmatrix(U)
            println("and the right-hand side is $c")
        end
    end
     return (U, c)
end;
```

Repeating the above testing:

```
U = ones(1,3)
```

```
1×3 Matrix{Float64}:
 1.0  1.0  1.0
```

```
(U, c) = rowreduce(A, b, demomode=true);
println("Row reduction gives U=")
printmatrix(U)
println("and right-hand side $c")
```

```
rowreduce version 2: some loops vectorized
Step 1:
After step 1 the matrix is
[ 4.0 2.0 7.0
  0.0 3.5 -11.25
  0.0 -3.5 0.25 ]
and the right-hand side is [2.0, 1.5, 3.5]
Step 2:
```

(continues on next page)

(continued from previous page)

```
After step 2 the matrix is
[ 4.0 2.0 7.0
  0.0 3.5 -11.25
  0.0 0.0 -11.0 ]
and the right-hand side is [2.0, 1.5, 5.0]
Row reduction gives U=
[ 4.0 2.0 7.0
  0.0 3.5 -11.25
  0.0 0.0 -11.0 ]
and right-hand side [2.0, 1.5, 5.0]
```

3.1.7 Backward substitution with an upper triangular matrix

The transformed equations have the form

$$\begin{aligned} u_{1,1}x_1 + u_{1,2}x_2 + u_{1,3}x_3 + \cdots + u_{1,n}x_n &= c_1 \\ &\vdots \\ u_{i,i}x_i + u_{i+1,i+1}x_{i+1} + \cdots + u_{i,n}x_n &= c_i \\ &\vdots \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= c_{n-1} \\ u_{nn}x_n &= c_n \end{aligned}$$

and can be solved from bottom up, starting with $x_n = c_n/u_{n,n}$.

All but the last equation can be written as

$$u_{i,i}x_i + \sum_{j=i+1}^{n} u_{i,j}x_j = c_i,\ 1 \le i \le n-1$$

and so solved as

$$x_i = \frac{c_i - \sum_{j=i+1}^{n} u_{i,j}x_j}{u_{i,i}}, \qquad \textbf{If } u_{i,i} \neq 0$$

This procedure is **backward substitution**, giving the algorithm

---

Algorithm 3.4 (Backward substitution)

$x_n = c_n/u_{n,n}$ for i from n-1 down to 1 $\quad x_i = \dfrac{c_i - \sum_{j=i+1}^{n} u_{i,j}x_j}{u_{i,i}}$ end

---

This works so long as none of the main diagonal terms $u_{i,i}$ is zero, because when done in this order, everything on the right hand side is known by the time it is evaluated.

For future reference, note that the elements $u_{k,k}$ that must be non-zero here, the ones on the **main diagonal** of U, are the same as the elements $a_{k,k}^{(k)}$ that must be non-zero in the row reduction stage above, because after stage k, the elements of row k do not change any more: $a_{k,k}^{(k)} = a_{k,k}^{(n-1)} = u_{k,k}$.

Remark 3.6 (Summing in Julia)

For an n-element single-index array `v`, the sum of its elements $\sum_{i=1}^{n} v_i$ is given by `sum(v)`.

Thus $\sum_{i=a}^{b} v_i$, the sum over a subset of indices $[a, b]$, is given by `sum(v[a:b])`.

The backward substitution algorithm in Julia

With all the above Julia details, the core code for backward substitution is:

```
x[end] = c[end]/U[end,end]
for i in n-1:-1:1
    x[i] = (c[i] - U[i,i+1:end] * x[i+1:end])) / U[i,i]
end
```

Observation 3.1

Note that the backward substitution algorithm and its Julia coding have a nice mathematical advantage over the row reduction algorithm above: the precise mathematical statement of the algorithm does not need any intermediate quantities distinguished by superscripts $^{(k)}$ and correspondingly, all variables in the code have fixed meanings, rather than changing at each step.

In other words, all uses of the equal sign are mathematically correct as equations!

This can be advantageous in creating algorithms and code that is more understandable and more readily verified to be correct, and is an aspect of the *functional programming* approach. We will soon go part way to that *functional* ideal, by rephrasing Gaussian elimination in a form where all variables have clear, fixed meanings, corresponding to the natural mathematical description of the process: the method of **LU factorization** introduced in *Solving Ax = b with LU factorization*.

As a final demonstration, we put this second version of the code into a complete working Julia function and test it:

```
function backwardsubstitution(U, c; demomode=false)
    n = length(c)
    x = zeros(n)
    x[end] = c[end]/U[end,end]
    if demomode
        println("x_$n = $(x[n])")
    end
    for i in n-1:-1:1
        if demomode
            println("i=$i")
        end
        x[i] = ( c[i] - sum(U[i,i+1:end] .* x[i+1:end]) ) / U[i,i]
        if demomode
            println("x_$i = $(x[i])")
        end
    end
    return x
end;
```

Remark 3.7

As usual, this is also available via

```
using .NumericalMethods: backwardsubstitution
```

```
x = backwardsubstitution(U, c, demomode=true)
println("x = $x")
```

```
x_3 = -0.45454545454545453
i=2
x_2 = -1.0324675324675323
i=1
x_1 = 1.8116883116883116
x = [1.8116883116883116, -1.0324675324675323, -0.45454545454545453]
```

```
println("x = $x")
r = b - A*x
println("The residual b - Ax = $(round.(r,4)), with maximum norm $(round(maximum(abs.
 ↪(r)),4)))")
```

```
x = [1.8116883116883116, -1.0324675324675323, -0.45454545454545453]
The residual b - Ax = [0.0, 0.0, 8.882e-16], with maximum norm 8.882e-16)
```

Since one is often just interested in the solution given by the two steps of row reduction and then backward substitution, they can be combined in a single function by composition:

```
solvelinearsystem(A, b) = backwardsubstitution(rowreduce(A, b)...);
```

Remark 3.8 (The Julia "splat" operation)

The **splat** notation "…" takes the item to its left (here the tuple `(U, c)` returned by `rowreduce`) and unpacks it into separate items [here `U` and `c`, as needed for input to `backwardsubstitution`.

3.1.8 Two code testing hacks: starting from a known solution, and using randomly generated examples

An often useful strategy in developing and testing code is to create a test case with a known solution; another is to use random numbers to avoid accidently using a test case that in unusually easy.

Remark 3.9 (function rand! from module Random)

- The preferred style is to have all `import` and `using` statements near the top, but since this is the first time we've heard of module `Random` I did not want it to be mentioned mysteriously above.
- The exclamation point in `rand!` indicates that the function modifies its input. This will be discussed in Functions part 2 when that section gets written.

```
using Random: rand!
```

```
n = length(b)
xrandom = zeros(n)
rand!(xrandom)  # fill with random values, uniform in [0, 1)
(xrandom *= 2.0) .-= 1.0  # double and then subtract 1: now uniform in [-1, 1)
println("xrandom = $xrandom")
```

```
xrandom = [0.13386638477695145, -0.7305805960100351, -0.8909949807354525]
```

Create a right-hand side b that automatically makes `xrandom` the correct solution:

```
brandom = A * xrandom;
```

```
println("A is"); printmatrix(A)
println("\nbrandom is $brandom")
(U, crandom) = rowreduce(A, brandom)
println("\nU is"); printmatrix(U)
println("\nResidual crandom - U*xrandom  = $(round.(crandom - U*xrandom,4))")
xcomputed = backwardsubstitution(U, crandom)
println("\nxcomputed is $(round.(xcomputed,12))")
r = brandom -  A*xcomputed
println("\nResidual brandom-A*xcomputed is $(round.(r,4))")
println("\nBackward error is $(round(maximum(abs.(r)),4))")
xerror = xrandom - xcomputed
println("\nError xrandom - xcomputed is $(round.(xerror,4))")
println("\nAbsolute error |xrandom-xcomputed| is $(round(maximum(abs.(xerror)),4))")
```

```
A is
[ 4.0 2.0 7.0
  3.0 5.0 -6.0
  1.0 -3.0 2.0 ]

brandom is [-7.162660518060432, 2.0946660586933934, 0.5436182113361516]

U is
[ 4.0 2.0 7.0
  0.0 3.5 -11.25
  0.0 0.0 -11.0 ]

Residual crandom - U*xrandom  = [0.0, 0.0, 0.0]

xcomputed is [0.133866384777, -0.73058059601, -0.890994980735]

Residual brandom-A*xcomputed is [0.0, 4.441e-16, -4.441e-16]

Backward error is 4.441e-16

Error xrandom - xcomputed is [0.0, 1.11e-16, 0.0]

Absolute error |xrandom-xcomputed| is 1.11e-16
```

3.1.9 What can go wrong? Some examples

Example 3.1 (An obvious division by zero problem)

Consider the system of two equations

$$\begin{aligned} x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned}$$

It is easy to see that this has the solution $x_1 = x_2 = 1$; in fact it is already in "reduced form". However when put into matrix form

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

the above algorithm fails, because the fist *pivot element* a_{11} is zero:

```
A1 = [0.0 1.0 ; 1.0 1.0]
b1 = [1.0 ; 1.0]

println("A1 is)");   printmatrix(A1)
println("b1 is $(b1)")
```

```
A1 is)
[ 0.0 1.0
  1.0 1.0 ]
b1 is [1.0, 1.0]
```

```
(U1, c1) = rowreduce(A1, b1)
x1 = backwardsubstitution(U1, c1)

println("U1 is"); printmatrix(U1)
print("c1 is $c1")
print("x1 is $x1")
```

```
U1 is
[ 0.0 1.0
  0.0 -Inf ]
c1 is [1.0, -Inf]x1 is [NaN, NaN]
```

Remark 3.10 (IEEE fake numbers Inf and NaN)

- `Inf`, meaning "infinity", is a special value given as the result of calculations like division by zero. Surprisingly, it can have a sign!
- `NaN`, meaning "Not a Number", is a special value given as the result of a calculation like `0/0`.

Example 3.2 (A less obvious division by zero problem)

Next consider this system

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 2 \\ 1 & 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$$

The solution is $x_1 = x_2 = x_3 = 1$, and this time none of th diagonal elements is zero, so it is not so obvious that a division by zero problem will occur, but:

```
A2 = [1.0  1.0  1.0 ; 1.0  1.0  2.0 ; 1.0 2.0 2.]
b2 = [3.0 ; 4.0 ; 5.]

println("A2 is"); printmatrix(A2)
println("b2 is $b2")
```

```
A2 is
[ 1.0 1.0 1.0
  1.0 1.0 2.0
  1.0 2.0 2.0 ]
b2 is [3.0, 4.0, 5.0]
```

```
(U2, c2) = rowreduce(A2, b2)
x2 = backwardsubstitution(U2, c2)

println("U2 is"); printmatrix(U2)
println("c2 is $c2")
println("x2 is $x2")
```

```
U2 is
[ 1.0 1.0 1.0
  0.0 0.0 1.0
  0.0 0.0 -Inf ]
c2 is [3.0, 1.0, -Inf]
x2 is [NaN, NaN, NaN]
```

What happens here is that the first stage subtracts the first row from each of the others …

```
A2[2,:] -= A2[1,:]
b2[2] -= b2[1]
A2[3,:] -= A2[1,:]
b2[3] -= b2[1];
```

… and the new matrix has the same problem as above at the next stage:

```
println("Now A2 is"); printmatrix(A2)
println("and b2 is $(b2)")
```

```
Now A2 is
[ 1.0 1.0 1.0
  0.0 0.0 1.0
  0.0 1.0 1.0 ]
and b2 is [3.0, 1.0, 2.0]
```

Thus, the second and third equations are

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

with the same problem as in *Example 3.1*.

Example 3.3 (Problems caused by inexact arithmetic: "divison by almost zero")

The equations

$$\begin{bmatrix} 1 & 10^{16} \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 + 10^{16} \\ 2 \end{bmatrix}$$

again have the solution $x_1 = x_2 = 1$, and the only division that happens in the above algorithm for row reduction is by that pivot element $a_{11} = 1, \neq 0$, so with exact arithmetic, all would be well. But:

```
A3 = [1.0  1e16 ; 1.0 1.0]
b3 = [1.0 + 1e16 ; 2.0]

println("A3 is"); printmatrix(A3)
println("b3 is $b3")
```

```
A3 is
[ 1.0 1.0e16
  1.0 1.0 ]
b3 is [1.0e16, 2.0]
```

```
(U3, c3) = rowreduce(A3, b3)
x3 = backwardsubstitution(U3, c3)

println("U3 is"); printmatrix(U3)
println("c3 is $c3")
println("x3 is $x3")
```

```
U3 is
[ 1.0 1.0e16
  0.0 -1.0e16 ]
c3 is [1.0e16, -9.999999999999998e15]
x3 is [2.0, 0.9999999999999998]
```

This gets $x_2 = 1$ fairly accurately, but x_1 is completely wrong!

One hint is that b_1, which should be $1 + 10^{16} = 10000000000000001$, is instead just given as 10^{16}.

On the other hand, all is well with less large values, like 10^{15}:

```
A3a = [1.0  1e15 ; 1.0  1.0]
b3a = [1.0 + 1e15 ; 2.0]

println("A3a is"); printmatrix(A3a)
println("b3a is $b3a")
```

```
A3a is
[ 1.0 1.0e15
  1.0 1.0 ]
b3a is [1.000000000000001e15, 2.0]
```

```
(U3a, c3a) = rowreduce(A3a, b3a)
x3a = backwardsubstitution(U3a, c3a)

println("U3a is"); printmatrix(U3a)
println("c3a is $c3a")
println("x3a is $x3a")
```

```
U3a is
[ 1.0 1.0e15
  0.0 -9.99999999999999e14 ]
c3a is [1.000000000000001e15, -9.99999999999999e14]
x3a is [1.0, 1.0]
```

Example 3.4 (Avoiding small denominators)

The first equation in *Example 3.3* can be divided by 10^{16} to get an equivalent system with the same problem:

$$\begin{bmatrix} 10^{-16} & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 + 10^{-16} \\ 2 \end{bmatrix}$$

Now the problem is more obvious: this system differs from the system in *Example 3.1* just by a tiny change of 10^{-16} in that pivot elements a_{11}, and the problem is *division by a value very close to zero.*

```
A4 = [1e-16  1.0 ; 1.0 1.0]
b4 = [1.0 + 1e-16 ; 2.0]

println("A4 is"); printmatrix(A4)
println("b4 is $b4")
```

```
A4 is
[ 1.0e-16 1.0
  1.0 1.0 ]
b4 is [1.0, 2.0]
```

```
(U4, c4) = rowreduce(A4, b4)
x4 = backwardsubstitution(U4, c4)

println("U4 is"); printmatrix(U4)
println("c4 is $c4")
println("x4 is $x4")
```

```
U4 is
[ 1.0e-16 1.0
  0.0 -1.0e16 ]
c4 is [1.0, -9.999999999999998e15]
x4 is [2.220446049250313, 0.9999999999999998]
```

One might think that there is no such small denominator in *Example 3.3*, but what counts for being "small" is magnitude relative to other values — 1 is very small compared to 10^{16}.

To understand these problems more (and how to avoid them) we will explore *Machine Numbers, Rounding Error and Error Propagation* in the next section.

3.1.10 When naive Guassian elimination is safe: diagonal dominance

There are several important cases when we can guarantee that these problem do not occur. One obvious case is when the matrix A is diagonal and non-singular (so with all non-zero elements); then it is already row-reduced and with all denominators in backward substitution being non-zero.

A useful measure of being "close to diagonal" is *diagonal dominance*:

Definition 3.1 (Strict Diagonal Dominance)

A matrix A is **row-wise strictly diagonally dominant**, sometimes abbreviated as just **strictly diagonally dominant** or **SDD**, if

$$\sum_{1 \le k \le n, k \neq i} |a_{i,k}| < |a_{i,i}|$$

Loosely, each main diagonal "dominates" in size over all other elements in its row.

Definition 3.2 (Column-wise Strict Diagonal Dominance)

If instead

$$\sum_{1 \le k \le n, k \neq i} |a_{k,i}| < |a_{i,i}|$$

(so that each main diagonal element "dominates its column") the matrix is called **column-wise strictly diagonally dominant**.

Note that this is the same as saying that the transpose A^T is SDD.

Aside: If only the corresponding non-strict inequality holds, the matrix is called *diagonally dominant*.

Theorem 3.1

For any strictly diagonally dominant matrix A, each of the intermediate matrices $A^{(k)}$ given by the naive Gaussan elimination algorithm is also strictly diagonally dominant, and so the final upper triangular matrix U is. In particular, all the diagonal elements $a_{i,i}^{(k)}$ and $u_{i,i}$ are non-zero, so no division by zero occurs in any of these algorithms, including the backward substitution solving for x in $Ux = c$.

The corresponding fact also true if the matrix is column-wise strictly diagonally dominant: that property is also preserved at each stage in naive Guassian elimination.

Thus in each case the diagonal elements — the elements divided by in both row reduction and backward substitution — are in some sense safely away from zero. We will have more to say about this in the sections on *Partial Pivoting* and *Solving Ax = b with LU factorization*

For a column-wise SDD matrix, more is true: at stage k, the diagonal dominance says that the pivot elemet on the diagonal, $a_{k,k}^{(k-1)}$, is larger (in magnitude) than any of the elements $a_{i,k}^{(k-1)}$ below it, so the multipliers $l_{i,k}$ have

$$|l_{i,k}| = |a_{i,k}^{(k-1)} / a_{k,k}^{(k-1)}| < 1.$$

As we will see when we look at the effects of rounding error in *Machine Numbers, Rounding Error and Error Propagation* and *Error bounds for linear algebra, condition numbers, matrix norms, etc.*, keeping intermediate value small is generally good for accuracy, so this is a nice feature.

Remark 3.11 (Positive definite matrices)

Another class of matrices for which naive Gaussian elimination works well is **positive definite matrices** which arise in any important situations; that property is in some sense more natural than diagonal dominance. However that topic will be left for later.

3.2 Machine Numbers, Rounding Error and Error Propagation

References:

- Sections 0.3 *Floating Point Represenation of Real Numbers* and 0.4 **Loss of Sinnificance* in [Sauer, 2019].
- Section 1.2 *Round-off Errors and Computer Arithmetic* of [Burden *et al.*, 2016].
- Sections 1.3 and 1.4 of [Chenney and Kincaid, 2012].

3.2.1 Overview

The naive Gaussian elimination algorithm seen in *Row Reduction/Gaussian Elimination*. has several related weaknesses which make it less robust and flexible than desired.

Most obviously, it can fail even when the equations are solvable, due to its naive insistence on always working from the top down. For example, as seen in *Example 3.1* of that section, it fails with the system

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

because the formula for the first multiplier $l_{2,1} = a_{2,1}/a_{1,1}$ gives $1/0$.

Yet the equations are easily solvable, indeed with no reduction needed: the first equation just says $x_2 = 1$, and then the second gives $x_1 = 2 - x_2 = 1$.

All one has to do here to avoid this problem is change the order of the equations. Indeed we will see that such reordering is *all that one ever needs to do,* so long as the original equation has a unique solution.

However, to develop a good strategy, we will also take account of errors introduced by rounding in computer arithmetic, so that is our next topic.

3.2.2 Robustness and well-posedness

The above claim raises the concept of **robustness** and the importance of both existence and uniqueness of solutions.

Definition 3.3 (Well-Posed)

A problem is **well-posed** if it is stated in a way that it has a unique solution. (Note that this might include asking for the set of all solutions, such as asking for all roots of a polynomial.)

For example, the problem of finding the root of a continuous, monotonic function $f : [a, b] \to \mathbb{R}$ with $f(a)$ and $f(b)$ of opposite sign is well-posed. Note the care taken with details to ensure both existence and uniqueness of the solution.

Definition 3.4 (Robust)

An algorithm for solving a class of problems is **robust** if it is guaranteed to solve any well-posed problem in the class.

For example, the bisection method is robust for the above class of problems. On the other hand, Newton's method is not, and if we dropped the specification of monotonicity (so allowing multiple solutons) then the bisection method in its current form would not be robust: it would fail whenever there is more that one solution in the interval $[a, b]$.

3.2.3 Rounding error and accuracy problems due to "loss of significance"

There is a second slightly less obvious problem with the naive algorithm for Guassian elimination, closely related to the first. As soon as the algorithm is implemented using any rounding in the arithmetic (rather than, say, working with exact arithmetic on rational numbers) division by values that are very close to zero can lead to very large intermediate values, which thus have very few correct decimals (correct bits); that is, very large absolute errors. These large errors can then propagate, leading to low accuracy in the final results, as seen in *Example 3.2* and *Example 3.4* of *Row Reduction/Gaussian Elimination*

This is the hazard of *loss of significance*, discussed in Section 0.4 of [Sauer, 2019] and Section 1.4 of [Chenney and Kincaid, 2012].

So it is time to take Step 2 of the strategy described in the previous notes:

2. Refine to get a more robust algorithm

1. Identify cases that can lead to failure due to division by zero and such, and revise to avoid them.
2. Avoid inaccuracy due to problems like severe rounding error. One rule of thumb is that anywhere that a zero value is a fatal flaw (in particular, division by zero), a very small value is also a hazard when rounding error is present. So avoid very small denominators. …

3.2.4 The essentials of machine numbers and rounding in machine arithmetic

As a very quick summary, standard computer arithmetic handles real numbers using *binary machine numbers* with p significant bits, and rounding off of other numbers to such *machine numbers* introduces a relative error of at most 2^{-p}. The current dominant choice for machine numbers and arithmetic is IEEE-64, using 64 bits in total and with $p = 53$ significant bits, so that $1/2^p \approx 1.11 \cdot 10^{-16}$, giving about fifteen significant digits. (The other bits are used for an exponent and the sign.)

(Note: in the above, I ignore the extra problems with real numbers whose magnitude is too large or too small to be represented: *underflow* and *overflow*. Since the allowable range of magnitudes is from $2^{-1022} \approx 2.2 \cdot 10^{-308}$ to $2^{1024} \approx 1.8 \cdot 10^{308}$, this is rarely a problem in practice.)

With other systems of binary machine numbers (like older 32-bit versions, or higher precision options like 128 bits) the significant differences are mostly encapsulated in that one number, the **machine unit**, $u = 1/2^p$.

Binary floating point machine numbers

The basic representation is a binary version of the familiar *scientific* or *decimal floating point* notation: in place of the form $\pm d_0.d_1d_2 \ldots d_{p-1} \times 10^e$ where the *fractional part* or *mantissa* is $f = d_0.d_1d_2 \ldots d_{p-1} = d_0 + \frac{d_1}{10} + \cdots + \frac{d_{p-1}}{10^{p-1}}$.

Binary floating point machine numbers with p significant bits can be described as

$$\pm(b_0.b_1b_2 \ldots b_{p-1})_2 \times 2^e = \pm \left(b_0 + \frac{b_1}{2} + \frac{b_2}{2^2} + \cdots \frac{b_{p-1}}{2^{p-1}} \right) \times 2^e$$

Just as decimal floating point numbers are typically written with the exponent chosen to have non-zero leading digit $d_0 \neq 0$, **normalized** binary floating point machine numbers have exponent e chosen so that $b_0 \neq 0$. Thus in fact $b_0 = 1$ — and so it need not be stored; only $p - 1$ bits are needed to stored for the mantissa.

Worst case rounding error

It turns out that the relative errors are determined solely by the number of significant bits in the mantissa, regardless of the exponent, so we look at that part first.

Rounding error in the mantissa, $(1.b_1b_2 \ldots b_{p-1})_2$

The spacing of consecutive mantissa values $(1.b_1b_2 \ldots b_{p-1})_2$ is one in the last bit, or 2^{1-p}. Thus rounding of any intermediate value x to the nearest number of this form introduces an absolute error of at most half of this: $u = 2^{-p}$, which is called the *machine unit*

How large can the *relative* error be? It is largest for the smallest possible denominator, which is $(1.00 \ldots 0)_2 = 1$, so the relative error due to rounding is also at most 2^{-p}.

Rounding error in general, for $\pm(1.b_1b_2 \ldots b_{p-1})_2 \cdot 2^e$.

The sign has no effect on the absolute error, and the exponent changes the spacing of consecutive machine numbers by a factor of 2^e. This scales the maximum possible absolute error to 2^{e-p}, but in the relative error calculation, the smallest possible denominator is also scaled up to 2^e, so the largest possible relative error is again the machine unit, $u = 2^{-p}$.

One way to describe the machine unit u (sometimes called *machine epsilon*) is to note that the next number above 1 is $1 + 2^{1-p} = 1 + 2u$. Thus $1 + u$ is at the threshold between rounding down to 1 and rounding up to a higher value.

IEEE 64-bit numbers: more details and some experiments

For completely full details, you could read about the IEEE 754 Standard for Floating-Point Arithmetic and specifically the binary64 case. (For historical reasons, this is known as "Double-precision floating-point format", from the era when computers were typicaly used 32-bit words, so 64-bit numbers needed two words.)

In the standard IEEE-64 number system:

- 64 bit words are used to store real numbers (a.k.a. *floating point* numbers, sometimes called *floats*.)
- There are $p = 53$ bits of precision, so that 52 bits are used to store the mantissa (fractional part).
- The sign is stored with one bit s: effectively a factor of $(-1)^s$, so $s = 0$ for positive, $s = 1$ for negative.
- The remaining 11 bits are use for the exponent, which allows for $2^{11} = 2048$ possibilities; these are chosen in the range $-1023 \leq e \leq 1024$.

- However, so far, this does not allow for the value zero! This is handled by giving a special meaning for the smallest exponent $e = -1023$, so the smallest exponent for *normalized* numbers is $e = -1022$.
- At the other extreme, the largest exponent $e = 1024$ is used to encode "infinite" numbers, which can arise when a calculation gives a value too large to represent (displayed as `inf` and `-inf`). This exponent is also used to encode "Not a Number", for situations like trying to divide zero by zero or multiply zero by `inf` (displayed as `NaN`).
- Thus, the exponential factors for normlaized numbers are in the range $2^{-1022} \approx 2 \times 10^{-308}$ to $2^{1023} \approx 9 \times 10^{307}$. Since the mantissa ranges from 1 to just under 2, the range of magnitudes of normalized real numbers is thus from $2^{-1022} \approx 2 \times 10^{-308}$ to just under $2^{1024} \approx 1.8 \times 10^{308}$.

Some computational experiments:

```
p = 53
u = 2.0^(-p)
println("For IEEE-64 arithmetic, there are $(p) bits of precision and the machine␣
↪unit is u=$(u).")
println("The next numbers above 1 are 1+2u = $(1+2*u), 1+4u = $(1+4*u) and so on.")
for factor in [3, 2, 1.00000000001, 1]
    one_plus_small = 1 + factor * u
    println("1 + $(factor)u rounds to $(one_plus_small)")
    difference = one_plus_small - 1
    println("\tThis is more than 1 by $(difference), which is $(difference/u) times u
↪")
end
```

```
For IEEE-64 arithmetic, there are 53 bits of precision and the machine unit is u=1.
↪1102230246251565e-16.
The next numbers above 1 are 1+2u = 1.0000000000000002, 1+4u = 1.0000000000000004␣
↪and so on.
1 + 3.0u rounds to 1.0000000000000004
        This is more than 1 by 4.440892098500626e-16, which is 4.0 times u
1 + 2.0u rounds to 1.0000000000000002
        This is more than 1 by 2.220446049250313e-16, which is 2.0 times u
1 + 1.00000000001u rounds to 1.0000000000000002
        This is more than 1 by 2.220446049250313e-16, which is 2.0 times u
1 + 1.0u rounds to 1.0
        This is more than 1 by 0.0, which is 0.0 times u
```

```
println("On the other side, the spacing is halved:")
println("the next numbers below 1 are 1-u = $(1-u), 1-2u = $(1-2*u) and so on.")
for factor in [2., 1., 1.00000000001/2, 1/2]
    one_minus_small = 1 - factor * u
    println("1 - $(factor)u rounds to $(one_minus_small)")
    difference = 1 - one_minus_small
    println("\tThis is less than 1 by $(difference), which is $(difference/u) times u
↪")
end
```

```
On the other side, the spacing is halved:
the next numbers below 1 are 1-u = 0.9999999999999999, 1-2u = 0.9999999999999998␣
↪and so on.
1 - 2.0u rounds to 0.9999999999999998
        This is less than 1 by 2.220446049250313e-16, which is 2.0 times u
1 - 1.0u rounds to 0.9999999999999999
        This is less than 1 by 1.1102230246251565e-16, which is 1.0 times u
```

(continues on next page)

(continued from previous page)

```
1 - 0.500000000005u rounds to 0.9999999999999999
        This is less than 1 by 1.1102230246251565e-16, which is 1.0 times u
1 - 0.5u rounds to 1.0
        This is less than 1 by 0.0, which is 0.0 times u
```

Next, look at the extremes of very small and very large magnitudes:

```
println("The smallest normalized positive number is 2^(-1022)=$(2.0^(-1022))")
println("The largest mantissa is binary (1.1111...) with 53 ones: 2 - 2^(-52)=$(2-2.0^
↪(-52))")
println("The largest normalized number is (2 - 2^(-52))*2^1023=$((2 - 2.0^(-52)) * (2.
↪0^1023))")
println("If instead we round that mantissa up to 2 and try again, we get 2*2^1023=$(2.
↪0 * 2.0^1023)")
```

```
The smallest normalized positive number is 2^(-1022)=2.2250738585072014e-308
The largest mantissa is binary (1.1111...) with 53 ones: 2 - 2^(-52)=1.
↪9999999999999998
The largest normalized number is (2 - 2^(-52))*2^1023=1.7976931348623157e308
If instead we round that mantissa up to 2 and try again, we get 2*2^1023=Inf
```

What happens if we compute positive numbers smaller than that smallest normalized positive number 2^{-1022}?

```
for S in [0, 1, 2, 51, 52, 53]
    exponent = -1022-S
    println("2^(-1022-$(S)) = 2^($(exponent)) = $(2.0^(exponent))")
end
```

```
2^(-1022-0) = 2^(-1022) = 2.2250738585072014e-308
2^(-1022-1) = 2^(-1023) = 1.1125369292536007e-308
2^(-1022-2) = 2^(-1024) = 5.562684646268003e-309
2^(-1022-51) = 2^(-1073) = 1.0e-323
2^(-1022-52) = 2^(-1074) = 5.0e-324
2^(-1022-53) = 2^(-1075) = 0.0
```

These extremely small values are called *denormalized numbers*. Numbers with exponent $2^{-1022-S}$ have fractional part with S leading zeros, so only $p - S$ significant bits. So when the shift S reaches $p = 53$, there are no significant bits left, and the value is truly zero.

3.2.5 Propagation of error in arithmetic

The only errors in the results of Gaussian elimination come from errors in the initial data (a_{ij} and b_i) and from when the results of subsequent arithmetic operations are rounded to machine numbers. Here, we consider how errors from either source are propagated — and perhaps amplified — in subsequent arithmetic operations and rounding.

In summary:

- When *multiplying* two numbers, the relative error in the sum is no worse than slightly more than the sum of the relative errors in the numbers multiplied. (the be pedantic, it is at most the sum of those relative plus their product, but that last piece is typically far smaller.)
- When *dividing* two numbers, the relative error in the quotient is again no worse than slightly more than the sum of the relative errors in the numbers divided.

- When *adding* two **positive** numbers, the relative error is no more that the larger of the relative errors in the numbers added, and the absolute error in the sum is no larger than the sum of the absolute errors.
- When *subtracting* two **positive** numbers, the absolute error is again no larger than the sum of the absolute errors in the numbers subtracted, **but the relative error can get far worse!**

Due to the differences between the last two cases, this discussion of error propagation will use "addition" to refer only to adding numbers of the same sign, and "subtraction" when subtracting numbers of the same sign.

More generally, we can think of rewriting the operation in terms of a pair of numbers that are both positive, and assume WLOG that all input values are positive numbers.

Notation: $x_a = x(1 + \delta_x)$ for errors and $fl(x)$ for rounding

Two notations will be useful.

Firstly, for any approximation x_a of a real value x, let $\delta_x = \dfrac{x_a - x}{x}$, so that $x_a = x(1 + \delta_x)$.

Thus, $|\delta_x|$ is the relative error, and δ_x helps keep track of the sign of the error.

Also, introduce the function $fl(x)$ which does rounding to the nearest machine number. For the case of the approximation $x_a = fl(x)$ to x given by rounding, the above results on machine numbers then give the bound $|\delta_x| \leq u = 2^{-p}$.

Propagation of error in products

Let x and y be exact quantities, and $x_a = x(1 + \delta_x)$, $y_a = y(1 + \delta_y)$ be approximations. The approximate product $(xy)_a = x_a y_a = x(1 + \delta_x)y(1 + \delta_y)$ has error

$$x(1 + \delta_x)y(1 + \delta_y) - xy = xy(1 + \delta_x + \delta_y + \delta_x\delta_y), = xy(1 + \delta_{xy})$$

Thus the relative error in the product is

$$|\delta_{xy}| \leq |\delta_x| + |\delta_y| + |\delta_x||\delta_y|$$

For example if the initial errors are due only to rounding, $|\delta_x| \leq u - 2^{-p}$ and similarly for $|\delta_y|$, so the relative error in $x_a y_a$ is at most $2u + u^2 = 2^{1-p} + 2^{-2p}$. In this and most situations, that final "product of errors" term $\delta_x\delta_y$ is far smaller than the first two, giving to a very good approximation

$$|\delta_{xy}| \leq |\delta_x| + |\delta_y|$$

This is the above stated "sum of relative errors" result.

When the "input errors" in x_a and y_a come just from rounding to machine numbers, so that each has p bits of precision, $|\delta_x|, |\delta_y| \leq 1/2^p$ and the error bound for the product is $1/2^{p-1}$: at most one bit of precision is lost.

Exercise 1

Derive the corresponding result for quotients.

Propagation or error in sums (of positive numbers)

With x_a and y_a as above (and positive), the approximate sum $x_a + y_a$ has error

$$(x_a + y_a) - (x + y) = (x_a - x) + (y_a - y)$$

so the absolute error is bounded by $|x_a - x| + |y_a - y|$; the sum of the absolute errors.

For the relative errors, express this error as

$$(x_a + y_a) - (x + y) = (x(1 + \delta_x) + y(1 + \delta_y)) = x\delta_x + y\delta_y$$

Let δ be the maximum or the relative errors, $\delta = \max(|\delta_x|, |\delta_y|)$; then the absolute error is at most $(|x|+|y|)\delta = (x+y)\delta$ and so the relative error is at most

$$\frac{(x + y)\delta}{|x + y|} = \delta = \max(|\delta_x|, |\delta_y|)$$

That is, *the relative error in the sum is at most the sum of the relative errors*, again as advertised above.

When the "input errors" in x_a and y_a come just from rounding to machine numbers, the error bound for the sum is no larger: no precision is lost! Thus, if you take any collection of non-negative numbers, round the to machine numbers so that each has relative error at must u, then the sum of these rounded values also has relative error at most u.

Propagation or error in differences (of positive numbers): loss of significance/loss of precision

The above calculation for the absolute error works fine regardless of the signs of the numbers, so the absolute error of a difference is still bounded by the sum of the absolute errors:

$$|(x_a - y_a) - (x - y)| \leq |x_a - x| + |y_a - y|$$

But for subtraction, the denominator in the relative error formulas can be far smaller. WLOG let $x > y > 0$. The relative error bound is

$$\frac{|(x_a - y_a) - (x - y)|}{|x - y|} \leq \frac{x\delta_x + y\delta_y}{x - y}$$

Clearly if $x - y$ is far smaller than x or y, this can be far larger than the "input" relative errors $|\delta_x|$ and $|\delta_y|$.

The extreme case is where the values x and y round to the same value, so that $x_a - y_a = 0$, and the relative error is 1: "100% error", a case of *catastrophic cancellation*.

Exercise 2

Let us move slightly away from the worst case scenario where the difference is exactly zero to one where it is close to zero; this will illustrate the idea mentioned earlier that *whereever a zero value is a problem in exact aritmetic, a very small value can be a problem in approximate arithmetic.*

For $x = 8.024$ and $y = 8.006$,

- Round each to three significant figures, giving x_a and y_a.
- Compute the absolute errors in each of these approximations, and in their difference as an approximation of $x - y$.
- Compute the relative errors in each of these three approximations.

Then look at rounding to only two significant digits!

Upper and lower bounds on the relative error in subtraction

The problem is worst when x and y are close in relative terms, in that y/x is close to 1. In the case of the errors in x_a and y_a coming just from rounding to machine enumbers, we have:

Theorem 3.2 (Loss of Precision)

Consider $x > y > 0$ that are close in that they agree in at least q significant bits and at most r significant bits:

$$\frac{1}{2^r} < 1 - \frac{y}{x} < \frac{1}{2^q}.$$

Then when rounded to machine numbers which are then subtracted, the relative error in that approximation of the difference is greater than that due to rounding by a factor of between 2^q and 2^r.

That is, subtraction loses between q and r significant bits of precision.

Exercise 3

(a) Illustrate why computing the roots of the quadratic equation $ax^2 + bx + c = 0$ with the standard formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

can sometimes give poor accuracy when evaluated using machine arithmetic such as IEEE-64 floating-point arithmetic. This is not always a problem, so identify specifically the situations when this could occur, in terms of a condition on the coefficents a, b, and c. (It is sufficient to consider real value of the ocefficients. Also as an aside, there is no loss of precision problem when the roots are non-real, so you only need consider quadratics with real roots.)

(b) Then describe a careful procedure for always getting accurate answers. State the procedure first with words and mathematical formulas, and then express it in pseudo-code.

Example 3.5 (Errors when approximating derivatives)

To deal with differential equations, we will need to approximate the derivative of function from just some values of the function itself. The simplest approach is suggested by the definition of the derivative

$$Df(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

by using

$$Df(x) \approx D_h f(x) := \frac{f(x+h) - f(x)}{h}$$

with a small value of h — but this inherently involves the difference of almost equal quantities, and so loss of significance.

Taylor's theorem give an error bound if we assume exact arithmetic — worse for larger h. Then the above results give a measure of rounding error effects — worse for smaller h.

This leads to the need to balance these error sources, to find an optimal choice for h and the corresponding error bound.

Denote the error in approximately calculating $D_h f(x)$ with machine arithmetic as $\tilde{D}_h f(x)$.

The error in this as an approximating of the exact derivative is

$$E = \tilde{D}_h f(x) - Df(x) = (\tilde{D}_h f(x) - D_h f(x)) + (D_h f(x) - Df(x))$$

which we will consider as the sum of two pieces, $E = E_A + E_D$ where

$$E_A = \tilde{D}_h f(x) - D_h f(x)$$

is the error due to machine **A**rithmetic in evaluation of the difference quotient $D_h f(x)$, and

$$E_D = D_h f(x) - Df(x)$$

is the error in this difference quotient as an approximation of the exact derivative $Df(x), = f'(x)$. This error is sometimes called the **discretization error** because it arises whe we replace the derivative by a discrete algebraic calculation.

Bounding the Arithmetic error E_A

The first source of error is rounding of $f(x)$ to a machine number; as seen above, this gives $f(x)(1+\delta_1)$, with $|\delta_1| \leq u$, so absolute error $|f(x)\delta_1| \leq |f(x)|u$.

Similarly, $f(x+h)$ is rounded to $f(x+h)(1+\delta_2)$, absolute error at most $|f(x+h)|u$.

Since we are interested in fairly small values of h (to keep E_D under control), we can assume that $|f(x+h)| \approx |f(x)|$, so this second absolute error is also very close to $|f(x)|u$.

Then the absolute error in the difference in the numerator of $D_h f(x)$ is at most $2|f(x)|u$ (or only a tiny bit greater).

Next the division. We can assume that h is an exact machine number, for example by choosing h to be a power of two, so that division by h simply shifts the power of two in the exponent part of the machine number. This has no effect on on the relative error, but scales the absolute error by the factor $1/h$ by which one is multiplying: the absolute error is now bounded by

$$|E_A| \leq \frac{2|f(x)|u}{h}$$

This is a critical step: the difference has a small absolute error, which conceals a large relative error due to the difference being small; now the absolute error gets amplified greatly when h is small.

Bounding the Discretization error E_D

As seen in *Taylor's Theorem and the Accuracy of Linearization* — for the basic case of linearization — we have

$$f(x+h) - f(x) = Df(x)h + \frac{f''(c_x)}{2}h^2$$

so

$$E_D = \frac{f(x+h) - f(x)}{h} = \frac{f''(c_x)}{2}h$$

and with $M_2 = \max |f''|$,

$$|E_D| \leq \frac{M_2}{2}h$$

Bounding the total absolute error, and minimizing it

The above results combine to give an upper limit on how bad the total error can be:

$$|E| \leq |E_A| + |E_D| \leq \frac{2|f(x)|u}{h} + \frac{M_2}{2}h$$

As aniticipated, the errors go in opposite directions: decreasing h to reduce E_D makes E_A worse, and vice versa. Thus we can expect that there is a "goldilocks" value of h — neither too small nor too big — that gives the best overall bound on the total error.

To do this, let's clean up the notation: let

$$A = 2|f(x)|u, \quad D = \frac{M_2}{2},$$

so that the error bound for a given value of h is

$$E(h) = \frac{A}{h} + Dh$$

This can be minimized with a little calculus:

$$\frac{dE(h)}{dh} = -\frac{A}{h^2} + D$$

which is zero only for the unique critical point

$$h = h^* = \sqrt{\frac{A}{D}} = \sqrt{\frac{2|f(x)|u}{M_2/2}} = 2\sqrt{\frac{|f(x)|}{M_2}}\sqrt{u}, = K\sqrt{u}$$

using the short-hand $K = 2\sqrt{\frac{|f(x)|}{M_2}}$.

This is easily verified to give the global mimimum of $E(h)$; thus, the best error bound we can get is for this value of h:

$$E \leq E^* := E(h^*) = \frac{2|f(x)|u}{K\sqrt{u}} + \frac{M_2}{2}K\sqrt{u} = \left(\frac{2|f(x)|}{K} + K\frac{M_2}{2}\right)\sqrt{u}$$

Conclusions from this example

In practical cases, we do not know the constant K or the coefficient of $\sqrt{u}$ in parentheses — but that does not matter much!

The most important — and somewhat disappointing — observation here is that both the optimal size of h and the resulting error bound is roughly proportional to the square root of the machine unit u. For example with p bits of precision, $u = 2^{-p}$, the best error is of the order of $2^{-p/2}$, or about $p/2$ significant bits: at best we can hope for about half as many signnificant bits as our machine arithmetic gives.

In decimal terms: with IEEE-64 arithmetic $u = 2^{-53} \approx 10^{-16}$, so giving about sixteen significant digits, and $\sqrt{u} \approx 10^{-8}$, so $\tilde{D}_h f(x)$ can only be expected to give about half as many; eight significant digits.

This is a first indication of why machine arithmetic sometimes needs to be so precise — more precise than any physical measurement by a factor of well over a thousand.

It also shows that when we get to computing derivatives and solving differential equations, we will often need to do a better job of approximating derivatives!

3.3 Partial Pivoting

References:

- Section 2.4.1 *Partial Pivoting* of [Sauer, 2019].
- Section 6.2 *Pivoting Stratgies* of [Burden *et al.*, 2016].
- Section 7.1 of [Chenney and Kincaid, 2012].

Remark 3.12

Some references describe the method of *scaled* partial pivoting, but here we present instead a version without the "scaling", because not only is it simpler, but modern research shows that it is esentially always as good, once the problem is set up in a "sane" way.

3.3.1 Introduction

The basic row reduction method can fail due to divisoion by zero (and to have very large rouding errors when a denominator is extremely close to zero. A more robust modification is to swap the order of the equations to avaid these problems: *partial pivotng*. Here we look at a particularly robust version of this strategy, *Maximal Element Partial Pivoting*.

3.3.2 What can go wrong with naive Gaussian elimination?

We have noted two problems with the naive algorithm for Gaussian elimination: total failure due the division be zero, and loss of precision due to dividing by very small values — or more preciselt calculations the lead to intermediate values far larger than the final results. The culprits in all cases are the same: the denominators are first the *pivot elements* $a_{k,k}^{(k-1)}$ in evaluation of $l_{i,k}$ during row reduction and then the $u_{k,k}$ in back substitution. Further, those $a_{k,k}^{(k-1)}$ are the final updated values at indices (k, k), so are the same as $u_{k,k}$. Thus it is exactly these *main diagonal elements* that we must deal with.

3.3.3 The basic fix: partial pivoting

The basic strategy is that at step k, we can swap equation k with any equation i, $i > k$. Note that this involves swapping those rows of array `A` and also those elements of the array `b` for the right-hand side: $b_k \leftrightarrow b_i$.

This approach of swapping equations (swapping rows in arrays `A` and `b`) is called **pivoting**, or more specifically **partial pivoting**, to distinguish from the more elaborate strategy where to columns of `A` are also reordered (which is equivalent to reordeting the unknowns in the equations). The row that is swapped with row k is sometimes called the **pivot row**, and the new denominator is the corresponding **pivot element**.

This approach is robust so long as one is using exact arithmetic: it works for any well-posed system because so long as the $Ax = b$ has a unique solution — so that the original matrix A is non-singular — at least one of the $a_{i,k}^{(k-1)}, i \geq k$ will be non-zero, and thus the swap will give a new element in position (k, k) that is non-zero. (I will stop caring about superscripts to distinguish updates, but if you wish to, the elements of the new row k could be called either $a_{k,j}^{(k)}$ or even $u_{k,j}$, since those values are in their final state.)

3.3.4 Handling rounding error: maximal element partial pivoting

The final refinement is to seek the smallest possible magnitudes for intermediate values, and thus the smallest absolute errors in them, by making the multipliers $l_{i,k}$ small, in turn by making the denominator $a_{k,k}^{(k-1)} = u_{k,k}$ as large as possible in magnitude:

At step k, choose the pivot row $p_k \geq k$ so that $|a_{p_k,k}^{(k-1)}| \geq |a_{i,k}^{(k-1)}|$ for all $i \geq k$. If there is more that one such element of largest magnitude, use the lowest value: in particular, if $p_k = k$ works, use it and do not swap!

A consequence of this is that the mutipliers used in the row operations all have $|l_{i,k}| = \left|\frac{a_{p_i,k}^{(k-1)}}{a_{p_k,k}^{(k-1)}}\right| < 1$.

Remark 3.13 (Swapping values in Julia)

In Julia (as in Python) the value of two variables `a` and `b` can be swapped via tuples with `(a, b) = (b, a)`, and combined with array slicing, this can also be used to swap rows (or columns)

```
a = 1
b = 2
(a, b) = (b, a)
```

```
(2, 1)
```

```
A = [11 12 13 ; 21 22 23 ; 31 32 33]
```

```
3×3 Matrix{Int64}:
 11  12  13
 21  22  23
 31  32  33
```

```
(A[1,:], A[2,:]) = (A[2,:], A[1,:])
A
```

```
3×3 Matrix{Int64}:
 21  22  23
 11  12  13
 31  32  33
```

Exercise 1

Explain why we cannot just swap the relevant elements of rows k and p with:

```
for j in 1:n
    A[k,j] = A[p,j]
    A[p,j] = A[k,j]
end
```

or in vectorized form:

```
A[k,:] = A[p,:]
A[p,:] = A[k,:]
```

Describe what happens instead.

Some demonstrations

No row reduction is done here, so entire rows are swapped rather than just the elements from column k onward:

First, get the matrix pretty-printer seen in *Row Reduction/Gaussian Elimination*; this time from *Module NumericalMethods*:

```
include("NumericalMethods.jl")
using .NumericalMethods: printmatrix
```

```
A = [1 -6 2 ; 3 5 -6 ; 4 2 7]
n = 3
println("Initially A is")
printmatrix(A)
```

```
Initially A is
[ 1 -6 2
  3 5 -6
  4 2 7 ]
```

```
k = 1
p = 3
temp = copy(A[k,:])
A[k,:] = A[p,:]
A[p,:] = temp
println("After swapping rows 1 <-> 3 using slicing and a temporary row, A is")
printmatrix(A)
```

```
After swapping rows 1 <-> 3 using slicing and a temporary row, A is
[ 4 2 7
  3 5 -6
  1 -6 2 ]
```

```
k = 2
p = 3
for j in 1:n
    ( A[k,j] , A[p,j] ) = ( A[p,j] , A[k,j] )
end
println("After swapping rows 2 <-> 3 using a loop and tuples of elements (no temp) A␣
↪is:")
printmatrix(A)
```

```
After swapping rows 2 <-> 3 using a loop and tuples of elements (no temp) A is:
[ 4 2 7
  1 -6 2
  3 5 -6 ]
```

```
k = 1
p = 2
( A[k,:] , A[p,:] ) = ( copy(A[p,:]) , copy(A[k,:]) )
println("After swapping rows 1 <-> 2 using tuples of slices (no loop or temp) A is:")
printmatrix(A)
```

```
After swapping rows 1 <-> 2 using tuples of slices (no loop or temp) A is:
[ 1 -6 2
  4 2 7
  3 5 -6 ]
```

3.3.5 When is it safe to do without pivoting?

Theorem 3.1 shows that diagonal dominance guarantees that pivoting is not necessary because the diagonal elements are never zero.

In the case of the matrix A being column-wise SDD as in *Definition 3.2*, the situation is even better; there is no reason for pivoting:

Theorem 3.3

If matrix A is column-wise SDD, maximal element partial pivoting in fact does no row-swaps; it does the same thing as naive Gaussian elimination.

Being row-wise SDD is more "natural" and common than being column-wise SDD, because the former is a property "within" each of the equations that go into the matrix. This might seem unfortunate, but there is a way to get the benefits of the above nice result also for row-wise SDD matrices, which we will see in the section *Solving $Ax = b$ with LU factorization* with the *Crout decomposition*.

3.4 Solving $Ax = b$ with LU factorization

References:

- Section 2.2 *The LU Factorization* of [Sauer, 2019].
- Section 6.5 *Matrix Factorizations* of [Burden *et al.*, 2016].
- Section 8.1 *Matrix Factorizations* of [Chenney and Kincaid, 2012].

3.4.1 Avoiding repeated calculation, excessive rounding and messy notation: LU factorization

Putting aside pivoting for a while, there is another direction in which the algorithm for solving linear systems $Ax = b$ can be improved. It starts with the idea of being more efficient when solving multiple system with the same right-hand side: $Ax^{(m)} = b^{(m)}, m = 1, 2, ...$

However it has several other benefits:

- allowing a strategy to reduce rounding error, and
- a simpler, more elegant mathematical statement.

We will see how to merge this with partial pivoting in *Solving Ax = b With Both Pivoting and LU Factorization*

Some useful jargon:

Definition 3.5 (Triangular matrix)

A matrix is **triangular** if all its non-zero elements are either on the main diagonal or to one side of it. There are two possibilities:

- Matrix U is **upper triangular** if $u_{ij} = 0$ for all $i > j$.
- Matrix L is **lower triangular** if $l_{ij} = 0$ for all $j > i$.

One important example of an upper triangular matrix is U formed by row reduction; note well that it is much quicker and easier to solve $Ux = c$ than the original system $Ax = b$ exactly because of its triangular form.

We will soon see that the multipliers l_{ij}, $i > j$ for row reduction that were introduced in *Row Reduction/Gaussian Elimination* help to form a very useful lower triangular matrix L.

The key to the LU factorization idea is finding a **lower triangular** matrix L and an **upper triangular** matrix U such that $LU = A$, and then using the fact that it is far quicker to solve a linear system when the corresponding matrix is triangular.

Indeed we will see that, if naive Gaussian elimination for $Ax = b$ succeeds, giving row-reduced form $Ux = c$:

1. The matrix A can be factorized as $A = LU$ with U an $n \times n$ upper triangular matrix and L an $n \times n$ lower triangular matrix.
2. There is a unique such factorization with the further condition that L is **unit lower triangular**, which means the extra requirement that the value on its main diagonal are unity: $l_{k,k} = 1$. This is called the **Doolittle Factorization** of A.
3. In the Doolittle factorization, the matrix U is the one given by naive Gaussian elimination, and the elements of L below its main diagonal are the multipliers arising in naive Gaussian elimination. (The other elements of L, on and above the main diagonal, are the ones and zeros dictated by it being unit lower triangular: the same as for those elements in the $n \times n$ identity matrix.)
4. The transformed right-hand side c arising from naive Gaussian elimination is the solution of the system $Lc = b$, and this is solvable by an procedure caled **forward substitution**, very similar to the backward subsitution used to solve $Ux = c$.

Putting all this together: if naive Gaussian elimination works for A, we can introduce the name c for Ux, and note that $Ax = (LU)x = L(Ux) = Lc = b$. Then solving of the system $Ax = b$ can be done in three steps:

1. Using A, find the Doolittle factors, L and U.
2. Using L and b, solve $Lc = b$ to get c. (Forward substitution)
3. Using U and c, solve $Ux = c$ to get x. (Backward substitution)

3.4.2 The direct method for the Doolittle LU factorization

If you believe the above claims, we already have one algorithm for finding an LU factorization; basically, do naive Gaussian elimination, but ignore the right-hand side b until later. However, there is another "direct" method, which does not rely on anything we have seen before about Gaussian elimination, and has other advantages as we will see.

(If I were to teach linear algebra, I would be tempted to start here and skip Gaussian Elimination!)

This method starts by considering the apparently daunting task of solving the n^2 simultaneous and nonlinear equations for the initially unknown elements of L and U:

$$\sum_{k=1}^{n} l_{i,k}u_{k,j} = a_{i,j} \; 1 \le i \le n, \; 1 \le j \le n.$$

The first step is to insert the known information; the already-known values of elements of L and U. For one thing, the sums above stop when either $k = i$ or $k = j$, whichever comes first, due to all the zeros in L nd U:

$$\sum_{k=1}^{\min(i,j)} l_{i,k}u_{k,j} = a_{i,j} \; 1 \le i \le n, \; 1 \le j \le n.$$

Next, when $i \le j$ — so that the sum ends at $k = i$ and involves $l_{i,i}$ — we can use $l_{i,i} = 1$.

So break up into two cases:

On and above the main diagonal ($i \le j$, so $\min(i,j) = i$):

$$\sum_{k=1}^{i-1} l_{i,k}u_{k,j} + u_{i,j} = a_{i,j} \quad 1 \le i \le n,\ i \le j \le n.$$

Below the main diagonal ($i > j$, so $\min(i,j) = j$):

$$\sum_{k=1}^{j-1} l_{i,k}u_{k,j} + l_{i,j}u_{j,j} = a_{i,j} \quad 2 \le i \le n,\ 1 \le j \le i.$$

In each equation, the last term in the sum has been separated, so that we can use them to "solve" for an unknown:

$$u_{i,j} = a_{i,j} - \sum_{k=1}^{i-1} l_{i,k}u_{k,j} \quad 1 \le i \le n,\ i \le j \le n.$$

$$l_{i,j} = \frac{a_{i,j} - \sum_{k=1}^{j-1} l_{i,k}u_{k,j}}{u_{j,j}} \quad 2 \le i \le n,\ 1 \le j \le i.$$

Here comes the characteristic step that gets us from valid equations to a useful algorithm: we can arrange these equations in an order such that all the values at right are determined by an earlier equation!

First look at what they say for the first row and first column.

With $i = 1$ in the first equation, there is no sum, and so:

$$u_{1,j} = a_{1,j}, \quad 1 \le j \le n,$$

which is the familiar fact that the first row is unchanged in naive Gaussian elimination.

Next, with $j = 1$ in the second equation, there is again no sum:

$$l_{i,1} = \frac{a_{i,1}}{u_{1,1}}, = \frac{u_{i,1}}{u_{1,1}}, \quad 2 \le i \le n,$$

which is indeed the multipliers in the first step of naive Gaussian elimination.

Remember that one way to think of Gaussian elimination is *recursively*: after step k, one just applies the same process recursively to the smaller $n-k \times n-k$ matrix in the bottom-right-hand corner. We can do something similar here; at stage k:

1. First use the first of the above equations to solve first for row k of U, meaning just $u_{k,j}, j \ge k$,
2. Then use the second equation to solve for column k of L: $l_{i,k}, i > k$.

Algorithm 3.5 (Doolittle factorization)

Stage $k = 1$ is handled by the simpler special equations above, and for the rest:

for k from 2 to n

for j from k to n *Get the non-zero elements in row k of U*

$u_{k,j} = a_{k,j} - \sum_{s=1}^{k-1} l_{k,s}u_{s,j}$

end

for i from k+1 to n *Get the non-zero elements in column k of L (except the 1's on its diagonal)*

$$l_{i,k} = \frac{a_{i,k} - \sum_{s=1}^{k-1} l_{i,s}u_{s,k}}{u_{k,k}}$$

end

end

Note well that in the formulas to evaluate at the right,

1. The terms $l_{k,s}$ are for $s < k$, so from a column s that has already been computed for a previous k value.
2. The terms $u_{s,j}$ are for $s < k$, so from a row s that has already been computed for a previous k value.
3. The denominator $u_{k,k}$ in the second inner loop is computed just in time, in the first inner loop for the same k value.

So the only thing that can go wrong is the same as with Gaussian elimination: a zero pivot element $u_{k,k}$.

Remark 3.14 (On this algorithm)

1. For $k = n$, the second inner loop is redundant, so could be eliminated. Indeed it might need to be eliminated in actual code, where "empty loops" might not be allowed. On the other hand, allowing empty loops makes the above correct also for $k = 1$; then the `for k` loop encompasses the entire factorization algorithm.
2. This direct factorization algorithm avoids any intermediate modification of arrays, and thus eliminates all those superscripts like $a_{i,j}^{(k)}$. This is not only nicer mathematically, but can help to avoid mistakes like code that inadvertently modifies the array containing the matrix A and then uses it to compute the residual, $b - Ax$. More generally, such purely mathematical statements of algorithms can help to avoid coding errors; this is part of the philosophy of the *functional programming* approach.
3. Careful examination shows that the product $l_{k,s}u_{s,j}$ that is part of what is subtracted at location (k, j) is the same as what is subtracted there at stage k of Gaussian elimination, just with different names. More generally, *every piece of arithmetic is the same as before,* except arranged in a different order, so that the $k - 1$ changes made to an element in row k are done together, via those sums.

```
include("NumericalMethods.jl")
using .NumericalMethods: printmatrix
```

```
function lu_factorize(A; demomode=false)
    # Compute the Doolittle LU factorization of A.
    # Sums like $\sum_{s=1}^{k-1} l_{k,s} u_{s,j}$ are done as matrix products;
    # in the above case, row matrix L[k, 1:k-1] by column matrix U[1:k-1,j] gives the␣
↪sum for a give j,
    # and row matrix L[k, 1:k-1] by matrix U[1:k-1,k:n] gives the relevant row vector.

    n = size(A)[1]  # First component of the array's size; size(A) returns "(rows,␣
↪columns)"
    # Initialize U as a zero matrix;
    # correct below the main diagonal, with the other entries to be computed and␣
↪filled below.
    U = zeros(n,n)

    # Initialize L as a zero matrix;
    # correct above the main diagonal, with the other entries to be computed and␣
↪filled in below.
    L = zeros(n,n)

    # The first row and columm are special:
    U[1,:] = A[1,:]
    L[1,1] = 1.0
```

(continues on next page)

(continued from previous page)

```
    L[2:end,1] = A[2:end,1]/U[1,1]
    if demomode
        println("After step k=1")
        println("U="); printmatrix(U)
        println("L="); printmatrix(L)
    end;
    for k in 2:n-1
        # Julia note: it is necessary to use indices "[k]" and so on to get one-row␣
↪matrices instead of vectors.
        U[[k],k:end] = A[[k],k:end] - L[[k],1:k] * U[1:k,k:end]
        L[k,k] = 1.0
        L[k+1:end,k] = (A[k+1:end,k] - L[k+1:end,1:k] * U[1:k,k]) / U[k,k]
        if demomode
            println("After step k=$k")
            println("U="); printmatrix(U)
            println("L="); printmatrix(L)
        end;
    end;
    # The last row is also special: not much to do for L.
    L[end,end] = 1.0
    U[end,end] = A[end,end] - sum(L[[n],1:end-1] * U[1:end-1,end])
    if demomode
        println("After step k=$n")
        println("U="); printmatrix(U)
    end;
    return L, U
end;
```

A test case on LU factorization

It will be useful to compute matrix norms as a measure or error; in particular the "maximum" or "infinity" norm of `v` is given by `norm(v, Inf)`

```
using LinearAlgebra: norm
```

```
A = [4.0 2.0 7.0; 3.0 5.0 -6.0; 1.0 -3.0 2.0]
printmatrix(A)
```

```
[ 4.0 2.0 7.0
  3.0 5.0 -6.0
  1.0 -3.0 2.0 ]
```

```
(L, U) = lu_factorize(A, demomode=true);
```

```
After step k=1
U=
[ 4.0 2.0 7.0
  0.0 0.0 0.0
  0.0 0.0 0.0 ]
L=
[ 1.0 0.0 0.0
```

(continues on next page)

(continued from previous page)

```
  0.75 0.0 0.0
  0.25 0.0 0.0 ]
After step k=2
U=
[ 4.0 2.0 7.0
  0.0 3.5 -11.25
  0.0 0.0 0.0 ]
L=
[ 1.0 0.0 0.0
  0.75 1.0 0.0
  0.25 -1.0 0.0 ]
After step k=3
U=
[ 4.0 2.0 7.0
  0.0 3.5 -11.25
  0.0 0.0 -11.0 ]
```

```
println("A is"); printmatrix(A)
println("L is"); printmatrix(L)
println("U is"); printmatrix(U)
println("L times U is"); printmatrix(L*U)
println("The 'residual' or 'backward error' A - LU is"); printmatrix(A - L*U)
```

```
A is
[ 4.0 2.0 7.0
  3.0 5.0 -6.0
  1.0 -3.0 2.0 ]
L is
[ 1.0 0.0 0.0
  0.75 1.0 0.0
  0.25 -1.0 1.0 ]
U is
[ 4.0 2.0 7.0
  0.0 3.5 -11.25
  0.0 0.0 -11.0 ]
L times U is
[ 4.0 2.0 7.0
  3.0 5.0 -6.0
  1.0 -3.0 2.0 ]
The 'residual' or 'backward error' A - LU is
[ 0.0 0.0 0.0
  0.0 0.0 0.0
  0.0 0.0 0.0 ]
```

Forward substitution: solving $Lc = b$ for c

This is the last piece missing. The strategy is very similar to backward substitution, but slightly simplified by the ones on the main didogonal of L. The equations $Lc = b$ can be written much as above, separating off the last term in the sum:

$$\sum_{j=1}^{n} l_{i,j}c_j = b_i,\ 1 \leq i \leq n$$

$$\sum_{j=1}^{i} l_{i,j}c_j = b_i,\ 1 \leq i \leq n$$

$$\sum_{j=1}^{i-1} l_{i,j}c_j + c_i = b_i,\ 1 \leq i \leq n$$

Then solve for c_i:

$$c_i = b_i - \sum_{j=1}^{i-1} l_{i,j}c_j$$

These are already is usable order: the right-hand side in the equation for c_i involves only the c_j values with $j < i$, determined by earlier equations if we run through index i in increasing order.

First, $i = 1$

$$c_1 = b_1 - \sum_{j=1}^{0} l_{1,j}c_j, = b_1$$

Next, $i = 2$

$$c_2 = b_2 - \sum_{j=1}^{1} l_{2,j}c_j, = b_2 - l_{2,1}c_1$$

Next, $i = 3$

$$c_3 = b_3 - \sum_{j=1}^{2} l_{3,j}c_j, = b_3 - l_{3,1}c_1 - l_{3,2}c_2$$

Exercise 1

A) Express this forward substitution strategy as pseudo-code; spell out all the sums in explicit rather than using 'Σ' notation for sums any matrix multiplication short-cut.

B) Then implement it "directly" in a Julia function, with format:

```
function forwardSubstitution(L, b)
    . . .
    return c
```

Again do this with explicit evaluation of each sum rather than using the function `sum` or any matrix multiplication short-cut.

C) Test it, using this often-useful "reverse-engineering" tactic:

1. Create suitable test arrays `L` and `c`. (Use n at least three, and preferably larger.)

2. Compute their product, with `b = L * c`
3. Check if `c_solution = forwardSubstitution(L, b)` gives the correct value (within rounding error.)

As usual, there is also an implementation available from module `NumericalMethods`, at forwardsubstitution, so this is used here. (It is not in the form asked for in the above exercise!)

```
using .NumericalMethods: forwardsubstitution
```

A test case on forward substitution

```
b = [2.0, 3.0, 4.0];
```

```
c = forwardsubstitution(L, b)
print(c)
```

```
[2.0, 1.5, 5.0]
```

```
println("c = $c")
println("The residual b - Lc is $(b - L*c)")
println("\t with maximum norm $(norm(b - L*c,  Inf))")
```

```
c = [2.0, 1.5, 5.0]
The residual b - Lc is [0.0, 0.0, 0.0]
         with maximum norm 0.0
```

Completing the test case, with backward substitution

As this step is unchanged, we can just import the version seen in *Row Reduction/Gaussian Elimination*

```
using .NumericalMethods: backwardsubstitution
```

```
x = backwardsubstitution(U, c);
```

```
residual_cUx = c - U*x
println("The residual c - Ux for the backward substitution step is $residual_cUx")
println("\t with maximum norm $(norm(residual_cUx, Inf))")
residual_bAx = b - A*x
println("The residual b - Ax for the whole solving process is $residual_bAx")
println("\t with maximum norm $(norm(residual_bAx, Inf))")
```

```
The residual c - Ux for the backward substitution step is [0.0, -2.
↪220446049250313e-16, 0.0]
         with maximum norm 2.220446049250313e-16
The residual b - Ax for the whole solving process is [0.0, 0.0, 8.881784197001252e-
↪16]
         with maximum norm 8.881784197001252e-16
```

Exercise 2

(An ongoing activity.)

Start building a Julia module — I suggest the name `MyNumericalMethods` — in a file name by adding suffix ".jl" to the module name (e.g. `MyNumericalMethods.jl`). Put all the functions that you create as you work through this book; for now, just your version of `forwardSubstitution(L, b)`, along with `backwardSubstitution` from a previous section and `luFactorize` from above.

The syntax of the module file is like this:

```
module NumericalMethods
function rowReduce(A, b)
    ...
end;
function forwardSubstitution(L, b)
    ...
end;
function backwardSubstitution(U, c)
    ...
end;
end
```

As an example of creating and using a module, I am creating one for this course, `NumericalMethods.jl`; see *Module NumericalMethods*. For now these two modules will overlap, but your version will contain code that you create in exerices that is not in `NumericalMethodsNumericalMethods`.

3.4.3 When does LU factorization work?

It was seen in the section *Partial Pivoting* that naive Gaussian elimination works (in the sense of avoiding division by zero) so one good result is that

---

Theorem 3.4

Any SDD matrix has a Doolittle factorization $A = LU$, with the diagonal elements of U all non-zero, so backward substitution also works.

For any column-wise SDD matrix, this LU factorization exists and is also "optimal", in the sense that it follows what you would do with maximal element partial pivoting.

This nice second property can be got for SDD matrices via a twist, or actually a transpose.

For an SDD matrix, it transpose $B = A^T$ is column-wise SDD and so has the nice Doolitle factorization described above: $B = L_B U_B$, with L_B being column-wise diagonally dominant and having ones on the main diagonal.

Transposing back, $A = B^T = (L_B U_B)^T = U_B^T L_B^T$, and defining $L = U_B^T$ and $U = L_B^T$,

- L is lower triangular
- U is upper triangular, row-wise diagonally dominant and with ones on it main diagonal: it is "unit upper triangular".
- Thus LU is another LU factorization of A, with U rather than L being the factor with ones on its main diagonal.

3.4.4 Crout decomposition

This sort of LU factorization is called the **Crout decomposition**; as with the Doolittle version, if such a factorization exists, it is unique.

Theorem 3.5

Every SDD matrix has a Crout decomposition, and the factor U is SDD.

Remark 3.15

As was mentioned at the end of the section *Row Reduction/Gaussian Elimination* naive Gausion elminaor alwo worek for *positive definite* matrices,amnd thus so does th Doolittle LU factirozation. However, there is another LU factorization that works even better in that case, the *Cholesky factorization*; this topic might be returned to later.

3.5 Solving $Ax = b$ With Both Pivoting and LU Factorization

References:

- Section 2.4 *The PA=LU Factorization* of [Sauer, 2019].
- Section 6.5 *Matrix Factorizations* of [Burden *et al.*, 2016].
- Section 8.1 *Matrix Factorizations* of [Chenney and Kincaid, 2012].

3.5.1 Introduction

The last step in producing an algorithm for solving the general case of n simultaneous linear equations in n variables that is *robust*, *efficient* and with good control of *rounding error* is to combine the ideas of **partial pivoting** from *Partial Pivoting* and **LU factorization** from *Solving Ax = b with LU factorization*.

This is sometimes described in three parts:

- *permute* (reorder) the rows of the matirx A by multiplying it at left by a suitable *permutation matrix* P; one with a single "1" in each row and each column and zeros elsewhere;
- Get the LU factorization of this matrix: $PA = LU$.
- To solve $Ax = b$
  - Express as $PAx = LUx = Pb$ (which just involves computing Pb, which reorders the elements of b)
  - Solve $Lc = Pb$ for c by forward substitution
  - Solve $Ux = c$ for x by backward substitution: as before, this gives $LUx = Lc = Pb$ and $LUx = PAx$, so $PAx = Pb$; since a permutation matrix P is invertible (just unravel the row swaps), this ensures that $Ax = b$.

This gives a nice formulas in terms of matrices; however we can describe it a bit more compactly and efficiently by just talking about the permutation of the rows, described by a *permutation vector* — an n component vector $\pi = [\pi_1, \pi_2, \ldots, \pi_n]$ whose elements are the integers from 1 to n in some order. So that is how the algorithm will be described below.

(Aside: I use the conventional name π for a permutation vector, partly to distinguish from the notation p_i used for *pivot* rows; however, feel free to use the name p instead, especially in Julia code.)

A number of details of this sketch will now be filled in, including the very useful fact that the permutation vector (or matrix) can be contsructed "on the fly", as rows are swapped in partial pivoting.

3.5.2 Row swapping is all you need

Let us look at maximal element partial pivoting, but described in terms of the entries of the factors L and U, and updating matrix A with a succession of row swaps.

(For now, I omit what happens to the right-hand side vector b; that is where the permutation vector p will come in, as addressed below.)

What happens if pivoting occurs at some stage k, with swapping of row k with a row $p_k > 5$?

One might fear that the process has to start again from the top using the modified version of matrix A, but in fact all previous work can be reused, just swapping those rows "everywhere".

Example: what happens at stage 5 ($k = 5$)?

To see this with a concrete example consider what happens if at stage $k = 5$ we swap rows 5 and 10 of A.

A) Firstly, what happens to matrix A?

The previous steps of the LU factorization process only involved entries of A in its first four rows and first four columns, and this row swap has no effect of them. Likewise, in row reduction, changes at and below row $k = 5$ have no effect on the first four rows of the row reduced form, U.

Thus, the only change here is to swap the entries of A between rows 5 and 10. What is more, the subsequent calculations only involve columns of index $j = 5$ upwards, so in fact we only need to update those entries. This can be written as

$$a_{5,j} \leftrightarrow a_{10,j}, \quad 5 \leq j \leq n$$

Thus if we are working in Julia with A stored in an array, the update is the slice operation

```
( A[5, 5:end], A[10, 5:end] ) = ( A[10, 5:end], A[5, 5:end] )
```

B) Next, look at the work done so far on U.

That just consists of the previous rows $1 \leq i \leq 4$, and the swapping of rows 5 with 10 has no effect up there:

Values already computed in U are unchanged.

C) Finally, look at the work done so far on the multipiers $l_{i,j}$; that is, matrix L.

The values computed so far are the first four columns of L; the multiples $l_{i,j}$, $1 \leq j \leq 4$ of row j subtracted from row $i > j$. These *do* change: for example, the multiple $l_{5,2}$ of row 2 is now subtracted from what was row 5 but is now row 10: thus, the new value of $l_{10,2}$ is the previous value of $l_{5,2}$.

Likewise, the same is true in reverse: the new value of $l_{5,2}$ is the previous value of $l_{10,2}$. This applies for all of the first four rows, so second index $1 \leq j \leq 4$:

The entries of L computed so far are swapped between rows 5 and 10, leaving the rest unchanged.

As this is again only for some columns — the first four — the swaps needed are:

$$l_{5,j} \leftrightarrow l_{10,j}, \quad 1 \leq j \leq 4$$

or in Julia'a slice notation for an array L:

```
( L[5, 1:4], L[10, 1:4] ) = ( L[10, 1:4], L[5, 1:4] )
```

The general pattern

The example above extends to all stages k of row reduction or computing the LU factorization of a row-permuted version of matrix A, where we adjust the pivot element at position (k, k) by first swapping row k with a row $p_k, \geq k$. (Allowing that sometimes no swap is needed, so that $p_k = k$.)

Gathering the key formulas above, this part of the algorithm is

---

Algorithm 3.6

for k from 1 to n-1

Find the pivot row $p_k, \geq k$.

if $p_k > k$

Swap $l_{k,j} \leftrightarrow l_{p_k,j}, \quad 1 \leq j < k$

Swap $a_{k,j} \leftrightarrow a_{p_k,j}, \quad k \leq j \leq n$

end

end

Pseudo-code for LU factorization with row swapping (first version)

Here I also adopt slice notation; for example, $a_{k,k:n}$ denotes the slice $[a_{k,k} \ldots a_{k,n}]$.

Algorithm 3.7 (LU factorization with row swapping, I)

for k from 1 to n

Find the pivot element:

$p = k$ *(p will be the index of the pivot row)*

for i from k+1 to n

if |u_{i, k}| > |u_{p, k}|

p ← i

end

end

if p > k *(Swap rows)*

$l_{k,1:k-1} \leftrightarrow l_{p,1:k-1}$

$a_{k,k:n} \leftrightarrow a_{p,k:n}$

end

for j from k to n *(Get the non-zero elements in row k of U)*

$u_{k,j} = a_{k,j} - \sum_{s=1}^{k-1} l_{k,s} u_{s,j}$

end

for i from k+1 to n *(Get the non-zero elements in column k of L — except the 1's on its diagonal)*

$$l_{i,k} = \frac{a_{i,k} - \sum_{s=1}^{k-1} l_{i,s} u_{s,k}}{u_{k,k}}$$

end

end

But what about the right-hand side, b?

One thing is missing from this strategy so far: if we are solving with a given right-hand-side column vector b, we would also swap its rows at each stage, with

$$b_k \leftrightarrow b_{p_k}$$

but with the LU factorization we need to keep track of these swaps for use later.

This turns out to mesh nicely with another detail: we can avoid actually copying array entries around by just keeping track of the order in which we use rows to get zeros in other rows. Our goal will be a permutation vector $\pi = [\pi_1, \pi_2, \ldots \pi_n]$ which says:

- First use row π_1 to get zeros in column 1 of the $n-1$ other rows.
- Then use row π_2 to get zeros in column 2 of the $n-2$ remaining rows.
- …

To do this:

- first, initialize an array $\pi = [1, 2, \ldots, n]$
- at stage k, if the pivot element is in row $p_k \neq k$, swap the corresponding elements in π (rather than swapping entire rows of arrays):

$$\pi_k \leftrightarrow \pi_{p_k}$$

Introducing the name A' for the new version of matrix A, its row k has entries $a'_{k,j} = a_{\pi_k, j}$.

This pattern persists through each row swap: instead of computing a succesion of updated versions of matrix A, we leave it alone and just change the row indices:

All references to entries of A are now done with permuted row index: $a_{\pi_i, j}$

The same applies to the array L of multipliers:

All references to entries of L are now done with $l_{\pi_i, j}$.

Finally, since these row swaps also apply to the right-hand side b, we do the same there:

All references to entries of b are now done with b_{π_i}.

Pseudo-code for LU factorization with a permutation vector

Algorithm 3.8 (LU factorization with row swapping, II)

Initialize the permutation vector, $\pi \leftarrow [1, 2, \dots, n]$

for k from 1 to n

Find the pivot element:

$p \leftarrow k$ *(p will be the index of the pivot row)*

for i from k+1 to n

if $|u_{i,k}| > |u_{p,k}|$:

$p \leftarrow i$

end

if p > k *(Just swap indices, not rows)*

$\pi_k \leftrightarrow \pi_p$

end

for j from k to n *(Get the non-zero elements in row k of U)*

$u_{k,j} \leftarrow a_{k,j} - \sum_{s=1}^{k-1} l_{k,s} u_{s,j}$

end

for i from k+1 to n *(Get the non-zero elements in column k of L — except the 1's on its diagonal)*

$l_{i,k} \leftarrow \dfrac{a_{i,k} - \sum_{s=1}^{k-1} l_{i,s} u_{s,k}}{u_{k,k}}$

end

end

Remark 3.16

For the version with a permutation matrix P, instead:

- start with an array P that is the identity matrix, and then
- swap its rows $k \leftrightarrow p_k$ at stage k instead of swapping the entries of π or the rows of A and L.

```
using LinearAlgebra: norm
using LinearAlgebra: ·  # For the dot product (the centered dot can be typed as \cdot␣
↪then tab)
```

```
include("NumericalMethods.jl")
using .NumericalMethods: printmatrix
```

```
function plu(A; demomode=false)
    # Compute the Doolittle PA=LU factorization of A —
    # but with the permutation recorded as permutation vector, not as the permutation matrix P.
    # Sums like $\sum_{s=1}^{k-1} l_{k,s} u_{s,j}$ are done as matrix products;
    # in the above case, row matrix L[k, 1:k-1] by column matrix U[1:k-1,j] gives the sum for a give j,
    # and row matrix L[k, 1:k-1] by matrix U[1:k-1,k:n] gives the relevant row vector.

    n = size(A)[1]  # gives the number of rows in the 2D array.
    # Julia can use Greek letters (and in fact, UNICODE):
    # to insert character π, type \pi, hit tab, and select "π" from the menu.
    # Or just call it "perm" or such.
    π = collect(1:n)
    # Julia language note: function "collect" converts the abstract entity "1:n" into an array of numbers.

    # Initialize U as the zero matrix;
    # correct below the main diagonal, with the other entries to be computed below.
    U = zeros(n,n)

    # Initialize L as zeros;
    # correct above the main diagonal, with the other entries to be computed below,
    # including the ones on the diagonal.
    L = zeros(n,n)

    for k in 1:n-1
        if demomode; println("k=$k"); end
        # Find the pivot element in column k:
        pivotrow = k
        abs_u_ik_max = abs(A[π[k],k])
        for row in k+1:n
            abs_u_ik = abs(A[π[row],k])
            if abs_u_ik > abs_u_ik_max
                pivotrow = row
                abs_u_ik_max = abs_u_ik
            end
        end
        if pivotrow > k # swap rows, virtually
            if demomode; println("Swap row $k with row $pivotrow"); end
            (π[k], π[pivotrow]) = (π[pivotrow], π[k])
        else
            if demomode; println("No row swap needed."); end
        end
        U[k,k:end] = A[[π[k]],k:end] - L[[π[k]],1:k] * U[1:k,k:end]
        L[π[k],k] = 1.
        for row in k+1:n
            L[π[row],k] = ( A[π[row],k] - L[π[row],1:k] ⋅ U[1:k,k] ) / U[k,k]
            # Julia note: To enter the centered dot notation for the dot product, type "\cdot" and then hit the tab key.
        end
        if demomode
            println("permuted A is:")
            for row in 1:n
                println(A[π[row],:])
            end
            println("Intermediate L is"); printmatrix(L)
```

(continues on next page)

(continued from previous page)

```
            println("Intermediate U is"); printmatrix(U)
        end
    end
    # The last row (index "end") is special: nothing to do for L except put in the 1
 ↪on the "permuted main diagonal"
    L[π[end],end] = 1.
    U[end,end] = A[π[end],end] - L[π[end],1:end-1] ⋅ U[1:end-1,end]
    if demomode
        println("After the final step, k=$(n-1)")
        println("L is"); printmatrix(L)
        println("U is"); printmatrix(U)
    end
    return (L, U, π)
end;
```

```
A = [ 1.0  -3.0  22.0 ; 3.0 5.0 -6.0 ; 4.0 235.0 7.0 ]
println("A is"); printmatrix(A)
(L, U, π) = plu(A, demomode=true)
println("\nFunction plu gives")
println("L="); printmatrix(L)
println("U="); printmatrix(U)
println("row permution $(π)")
println("The 'residual' or 'backward error' A-LU is"); printmatrix(A - L*U)
```

```
A is
[ 1.0 -3.0 22.0
  3.0 5.0 -6.0
  4.0 235.0 7.0 ]
k=1
Swap row 1 with row 3
permuted A is:
[4.0, 235.0, 7.0]
[3.0, 5.0, -6.0]
[1.0, -3.0, 22.0]
Intermediate L is
[ 0.25 0.0 0.0
  0.75 0.0 0.0
  1.0 0.0 0.0 ]
Intermediate U is
[ 4.0 235.0 7.0
  0.0 0.0 0.0
  0.0 0.0 0.0 ]
k=2
No row swap needed.
permuted A is:
[4.0, 235.0, 7.0]
[3.0, 5.0, -6.0]
[1.0, -3.0, 22.0]
Intermediate L is
[ 0.25 0.3605839416058394 0.0
  0.75 1.0 0.0
  1.0 0.0 0.0 ]
Intermediate U is
[ 4.0 235.0 7.0
  0.0 -171.25 -11.25
```

(continues on next page)

(continued from previous page)

```
  0.0 0.0 0.0 ]
After the final step, k=2
L is
[ 0.25 0.3605839416058394 1.0
  0.75 1.0 0.0
  1.0 0.0 0.0 ]
U is
[ 4.0 235.0 7.0
  0.0 -171.25 -11.25
  0.0 0.0 24.306569343065693 ]

Function plu gives
L=
[ 0.25 0.3605839416058394 1.0
  0.75 1.0 0.0
  1.0 0.0 0.0 ]
U=
[ 4.0 235.0 7.0
  0.0 -171.25 -11.25
  0.0 0.0 24.306569343065693 ]
row permution [3, 2, 1]
The 'residual' or 'backward error' A-LU is
[ 0.0 0.0 0.0
  0.0 0.0 0.0
  0.0 0.0 0.0 ]
```

Matrix L is not actually lower triangular, due to the permutation of its rows, but is still fine for a version of forward substitution, because

- row π_1 only involves x_1 (multiplied by 1) and so can be used to solve for x_1
- row π_2 only involves x_1 and x_2 (the latter multiplied by 1) and so can be used to solve for x_2
- …

Definition 3.6 (Psychologically [lower] triangular)

A matrix like this — one that is a row-permutation of a [lower] triangular matrix — is called **psychologically [lower] triangular**. (Maybe because it believes itself to be such?)

Forward and backward substitution with a permutation vector

To solve $Lc = b$, all one has to change from the formulas for forward substitution seen in the previous section *Solving Ax = b with LU factorization* is to put the permuted row index π_i in both L and b:

$$c_i = b_{\pi_i} - \sum_{j=1}^{i-1} l_{\pi_i,j} c_j,\ 1 \le i \le n$$

```
function forwardsubstitution(L, b, π)
    # Version 2: with permutation of rows
    # Solve L c = b for c, with permutation of the rows of L and of b.
    n = length(b)
    c = zeros(n)
```

(continues on next page)

(continued from previous page)

```
    c[1] = b[π[1]]
    for i in 2:n
        c[i] = b[π[i]] - L[π[i], 1:i] ⋅ c[1:i]
    end
    return c
end;
```

```
b = [2.0, 3.0, 4.0];
```

```
c = forwardsubstitution(L, b, π)
print("c = $c")
```

```
c = [4.0, 0.0, 1.0]
```

Then the final step, solving $Ux = b$ for x, needs no change, because U had no rows swapped, so we are done; we can import the function `backwardSubstitution` seen previously

```
using .NumericalMethods: backwardsubstitution
```

```
x = backwardsubstitution(U, c)
println("x = $x")
Ax = A*x
r = b - A*x
println("The residual r = b - Ax is \n$r\nwith maximum norm $(norm(r, Inf))")
```

```
x = [1.0867867867867869, -0.002702702702702703, 0.04114114114114114]
The residual r = b - Ax is
[0.0, 0.0, 0.0]
with maximum norm 0.0
```

3.6 Error bounds for linear algebra, condition numbers, matrix norms, etc.

References:

- Section 2.3.1 *Error Magnification and Condition Number* of [Sauer, 2019].
- Section 7.5 *Error Bounds and Iterative Refinement* of [Burden *et al.*, 2016] — but you may skip the last part, on *Iterative Refinement*; that is not relevant here.
- Section 8.4 of [Chenney and Kincaid, 2012].

3.6.1 Residuals, backward errors, forward errors, and condition numbers

For an approximation x_a of the solution x of $Ax = b$, the *residual* $r = Ax_a - b$ measures error as *backward error*, often measured by a single number, the *residual norm* $\|Ax_a - b\|$. Any norm could be used, but the maximum norm is usualt preferred, for reasons that we will see soon.

The corresponding (dimensionless) measure of relative error is defined as

$$\frac{\|r\|}{\|b\|}.$$

However, these can greatly underestimate the *forward* errors in the solution: the absolute error $\|x - x_a\|$ and relative error

$$Rel(x_a) = \frac{\|x - x_a\|}{\|x\|}$$

To relate these to the residual, we need the concepts of a *matrix norm* and the *condition number* of a matrix.

3.6.2 Matrix norms induced by vector norms

Given any vector norm $\|\cdot\|$ — such as the maximum ("infinity") norm $\|\cdot\|_\infty$ or the Euclidean norm (length) $\|\cdot\|_2$ — the correponding *induced matrix norm* is

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

This maximum exists for ethe rof these vector norms, and for the infinity norm there ia an explicit formula for it: for any $m \times n$ matrix,

$$\|A\|_\infty = \max_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|$$

(On the other hand, it is far harder to compute the Euclidean norm of a matrix: the formula requires computing eigenvalues.)

Note that when the matrix is a vector considered as a matrix with a single column — so $n = 1$ — the sum goes away, and this agrees with the infinity vector norm. This allows us to consider vectors as being just matrices with a single column, which we will often do from now on.

3.6.3 Properties of (induced) matrix norms

These induced matrix norms have many properties in common with Euclidean length and other vector norms, but there can also be products, and then one has to be careful.

1. $\|A\| \geq 0$ (positivity)
2. $\|A\| = 0$ if and only if $A = 0$ (definiteness)
3. $\|cA\| = |c| \, \|A\|$ for any constant c (absolute homogeneity)
4. $\|A + B\| \leq \|A\| + \|B\|$ (sub-additivity or the triangle inequality),

   and when the product of two matrices makes sense (including matrix-vector products),
5. $\|AB\| \leq \|A\| \, \|B\|$ (sub-multiplicativity)

Note the failure to always have equality with products. Indeed one can have $AB = 0$ with A and B both non-zero, such as when A is a singular matrix and B is a null-vector for it.

Remark 3.17 (Other matrix norms)

There are other matrix norms of use in some contexts, in particular the Frobenius norm. Then the above properties are often used to *define* what it is to be a matrix form, much as the first four define what it is to be a vector norm.

Remark 3.18 (Julia functions norm and opnorm)

Julia package `LinearAlgebra` provides the functions `norm` and `opnorm` for evaluating matrix norms, as seen in the examples in the previous section *Solving Ax = b With Both Pivoting and LU Factorization*, where `norm` computes the vector norms $\|v\|_p$ and `opnorm` computes the coresponding matrix norms ("operator norms") $\|A\|_p$.

- If p is omitted, it defaults to $p = 2$, so `norm(v)` is the familiar Euclidean vector norm.
- To get the "maximum" or "∞" norm, use value `Inf` for `p`.

Warning. Even if the argument of `norm` is a matrix, it is treated as a vector: for example, `norm(A, Inf)` returns the maximum of all the absolute values of the elements of an array `A`.

3.6.4 Relative error bound and condition number

It can be proven that, for any choice of norm,

$$Rel(x_a) = \frac{\|x - x_a\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|r\|}{\|b\|},$$

where the last factor is the relative backward error.

Since we can (though often with considerable effort, due to the inverse!) compute the right-hand side when the infinity norm is used, we can compute an upper bound on the relative error. From this, an upper bound on the absolute error can be computed if needed.

The *growth factor* between the relative backward error measured by the residual and the relative (forward) error is called the *condition number*, $K(A)$:

$$\kappa(A) := \|A\|\|A^{-1}\|$$

so that the above bound on the relative error can be restated as

$$\text{Rel}(x_a) = \frac{\|x - x_a\|}{\|x\|} \leq \kappa(A)\frac{\|r\|}{\|b\|}$$

Actually there is one condition number for each choice of norm, so we work with

$$\kappa_\infty(A) := \|A\|_\infty\|A^{-1}\|_\infty$$

Note that for a singular matrix, this is undefined: we can intuitively say that the condition number is then infinite.

At the other extreme, the identity matrix I has norm 1 and condition number 1 (using any norm), and this is the best possible because in general $\kappa(A) \geq 1$. (This follows from sub-multiplicativity.)

Aside: estimating $\|A^{-1}\|_\infty$ and thence the condition number

In Julia, good approximations of condition numbers are given by the function `cond` from package `LinearAlgebra`.

As with functions `norm` and `opnorm`, the simple form `cond(A)` defaults to $\kappa_2(A)$ based on the Euclidian length $\|\cdot\|_2$ for vectors; to get the infinity norm version $\kappa_\infty(A)$ use `cond(A, Inf)`.

This is not done exactly, since computing the inverse is a lot of work for large matrices and good estimates can be got far more quickly. The basic idea is start with the formula

$$\|A^{-1}\| = \max_{\|x\|=1} \|A^{-1}x\|$$

and instead compute the maximum over some finite selection of values for x: call them $x^{(k)}$. Then to evaluate $y^{(k)} = A^{-1}x^{(k)}$, express this through the equation $Ay^{(k)} = x^{(k)}$. Once we have an LU factorization for A (which one probably would have when exploring errors in a numerical solution of $Ax = b$) each of these systems can be solved relatively fast: Then

$$\|A^{-1}\| \approx \max_k \|y^{(k)}\|.$$

3.6.5 Well-conditioned and ill-conditioned problems and matrices

Condition numbers, giving upper limit on the ratio of forward error to backward error, measure the amplification of errors, and have counterparts in other contexts. For example, with an approximation r_a of a root r of the equation $f(x) = 0$, the ratio of forward error to backward error is bounded by $\max 1/|f'(x)| = \dfrac{1}{\min |f'(x)|}$, where the maximum only need be taken over an interval known to contain both the root and the approximation. This condition number becomes "infinite" for a multiple root, $f'(r) = 0$, related to the problems we have seen in that case.

Careful calculation of an approximate solution x_a of $Ax = b$ can often get a *residual* that is at the level of machine rounding error, so that roughly the relative backward error is of size comparable to the machine unit, u. The condition number then guarantees that the (forward) relative error is no greater than about $u\,\kappa(A)$.

In terms of significant bits, with p bit machine arithmetic, one can hope to get $p - \log_2(\kappa(A))$ significant bits in the result, but can not rely on more, so one loses $\log_2(\kappa(A))$ significant bits. Compare this to the observation that one can expect to lose at least $p/2$ significant bits when using the approximation $Df(x) \approx D_h f(x) - (f(x+h) = f(x))/h$.

A **well-conditioned problem** is one that is not too highly sensitive to errors in rounding or input data; for an eqution $Ax = b$, this corresponds to the condition number of A not being to large; the matrix A is then sometimes also called well-conditioned. This is of course vague, but might typically mean that $p - \log_2(\kappa(A))$ is a sufficient number of significant bits for a particular purpose.

A problem that is not deemed well-conditioned is called **ill-conditioned**, so that a matrix of uncomfortably large condition number is also sometimes called ill-conditioned. An ill-conditioned problem might still be well-posed, but just requiring careful and precise solution methods.

Example 3.6 (the Hilbert matrices)

The $n \times n$ Hilbert matrix H_n has elements

$$H_{i,j} = \frac{1}{i + j - 1}$$

For example

$$H_4 = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}$$

and for larger or smaller n, one simply adds or remove rows below and columns at right.

These matrices arise in important situations like finding the polynomial of degree $n-1$ that fits given data in the sense of minimizing the root-mean-square error — as we will discuss later in this course if there is time and interest.

Unfortunately as n increases the condition number grows rapidly, causing severe rounding error problems. To illustrate this, I will do something that one should usually avoid: compute the inverse of these matrices. This is also a case that shows the advatage of the LU factorization, since one computes the inverse by succesively computing each column, by solving n different systems of equations, each with the same matrix A on the left-hand side.

```
include("NumericalMethods.jl")
using .NumericalMethods: lu_factorize, forwardsubstitution, backwardsubstitution,␣
↪solvelinearsystem, printmatrix
```

```
using LinearAlgebra: norm, opnorm, cond
```

```
using Random: rand
```

```
function inverse(A, demomode=false)
    # Use sparingly; there is usually a way to avoid computing inverses that is␣
↪faster and with less rounding error!
    n = size(A)[1]  # First index of the size, which is (n, n)
    A_inverse = zeros(size(A))
    (L, U) = lu_factorize(A)
    for i in 1:n
        if demomode; println("i=$i"); end
        e_i = zeros(n)
        e_i[i] = 1.0
        if demomode; println("e_$i=$e_i"); end
        c = forwardsubstitution(L, e_i)
        A_inverse[:,i] = backwardsubstitution(U, c)
        #A_inverse[:,i] = solvelinearsystem(A, e_i)
    end
    return A_inverse
end;
```

```
function hilbert(n)
    H = zeros(n,n)
    for i in 1:n
        for j in 1:n
            H[i,j] = 1.0/(i + j - 1.0)
        end
    end
    return H
end;
```

```
for n in 2:5
    H_n = hilbert(n)
    println("H_$n is")
    printmatrix(round.(H_n, sigdigits=4))
    H_n_inverse = inverse(H_n)
    println("and its inverse is")
    printmatrix(round.(H_n_inverse, sigdigits=4))
```

(continues on next page)

(continued from previous page)

```
    println("to verify, their product is")
    printmatrix(round.(H_n * H_n_inverse, sigdigits=2))
    println()
end
```

```
H_2 is
[ 1.0 0.5
  0.5 0.3333 ]
and its inverse is
[ 4.0 -6.0
  -6.0 12.0 ]
to verify, their product is
[ 1.0 0.0
  0.0 1.0 ]

H_3 is
[ 1.0 0.5 0.3333
  0.5 0.3333 0.25
  0.3333 0.25 0.2 ]
and its inverse is
[ 9.0 -36.0 30.0
  -36.0 192.0 -180.0
  30.0 -180.0 180.0 ]
to verify, their product is
[ 1.0 0.0 0.0
  0.0 1.0 0.0
  0.0 0.0 1.0 ]

H_4 is
[ 1.0 0.5 0.3333 0.25
  0.5 0.3333 0.25 0.2
  0.3333 0.25 0.2 0.1667
  0.25 0.2 0.1667 0.1429 ]
and its inverse is
[ 16.0 -120.0 240.0 -140.0
  -120.0 1200.0 -2700.0 1680.0
  240.0 -2700.0 6480.0 -4200.0
  -140.0 1680.0 -4200.0 2800.0 ]
to verify, their product is
[ 1.0 0.0 2.3e-13 -1.1e-13
  -1.3e-16 1.0 1.1e-13 -1.5e-13
  -2.3e-15 2.2e-14 1.0 -4.5e-14
  -4.0e-15 6.8e-14 -6.4e-14 1.0 ]

H_5 is
[ 1.0 0.5 0.3333 0.25 0.2
  0.5 0.3333 0.25 0.2 0.1667
  0.3333 0.25 0.2 0.1667 0.1429
  0.25 0.2 0.1667 0.1429 0.125
  0.2 0.1667 0.1429 0.125 0.1111 ]
and its inverse is
[ 25.0 -300.0 1050.0 -1400.0 630.0
  -300.0 4800.0 -18900.0 26880.0 -12600.0
  1050.0 -18900.0 79380.0 -117600.0 56700.0
  -1400.0 26880.0 -117600.0 179200.0 -88200.0
```

(continues on next page)

(continued from previous page)

```
  630.0 -12600.0 56700.0 -88200.0 44100.0 ]
to verify, their product is
[ 1.0 5.9e-13 -8.3e-13 1.2e-12 8.5e-13
  2.3e-14 1.0 8.2e-14 -1.0e-12 2.0e-13
  -9.4e-16 3.6e-13 1.0 -6.0e-13 -8.7e-13
  -1.4e-14 6.8e-13 -2.7e-12 1.0 -1.8e-12
  -8.6e-15 2.5e-13 -1.8e-12 1.8e-12 1.0 ]
```

Note how the inverses have some surprisingly large elements; this is the matrix equivalent of a number being very close to zero and so with a very large reciprocal.

Since we have the inverses, we can compute the matrix norms of each H_n and its inverse, and thence their condition numbers.

```
for n in 2:5
    H_n = hilbert(n)
    println("H_$n is")
    printmatrix(round.(H_n, sigdigits=6))
    println("with infinity norm $(round(opnorm(H_n, Inf), sigdigits=4))")
    H_n_inverse = inverse(H_n)
    println("and its inverse is")
    printmatrix(round.(H_n_inverse, sigdigits=6))
    println("with infinity norm $(round(opnorm(H_n_inverse, Inf), sigdigits=4))")
    println("Thus the condition number of H_$n is $(round(opnorm(H_n, Inf) * opnorm(H_
↪n_inverse, Inf), sigdigits=4))")
    println()
end
```

```
H_2 is
[ 1.0 0.5
  0.5 0.333333 ]
with infinity norm 1.5
and its inverse is
[ 4.0 -6.0
  -6.0 12.0 ]
with infinity norm 18.0
Thus the condition number of H_2 is 27.0

H_3 is
[ 1.0 0.5 0.333333
  0.5 0.333333 0.25
  0.333333 0.25 0.2 ]
with infinity norm 1.833
and its inverse is
[ 9.0 -36.0 30.0
  -36.0 192.0 -180.0
  30.0 -180.0 180.0 ]
with infinity norm 408.0
Thus the condition number of H_3 is 748.0

H_4 is
[ 1.0 0.5 0.333333 0.25
  0.5 0.333333 0.25 0.2
  0.333333 0.25 0.2 0.166667
  0.25 0.2 0.166667 0.142857 ]
```

(continues on next page)

(continued from previous page)

```
with infinity norm 2.083
and its inverse is
[ 16.0 -120.0 240.0 -140.0
  -120.0 1200.0 -2700.0 1680.0
  240.0 -2700.0 6480.0 -4200.0
  -140.0 1680.0 -4200.0 2800.0 ]
with infinity norm 13620.0
Thus the condition number of H_4 is 28370.0

H_5 is
[ 1.0 0.5 0.333333 0.25 0.2
  0.5 0.333333 0.25 0.2 0.166667
  0.333333 0.25 0.2 0.166667 0.142857
  0.25 0.2 0.166667 0.142857 0.125
  0.2 0.166667 0.142857 0.125 0.111111 ]
with infinity norm 2.283
and its inverse is
[ 25.0 -300.0 1050.0 -1400.0 630.0
  -300.0 4800.0 -18900.0 26880.0 -12600.0
  1050.0 -18900.0 79380.0 -117600.0 56700.0
  -1400.0 26880.0 -117600.0 179200.0 -88200.0
  630.0 -12600.0 56700.0 -88200.0 44100.0 ]
with infinity norm 413300.0
Thus the condition number of H_5 is 943700.0
```

Next, experiment with solving equations, to compare residuals with actual errors.

I will use the testing strategy of starting with a known solution x, from which the right-hand side b is computed; then slight simulated error is introduced to b. Running this repeatedly with use of different random "errors" gives an idea of the actual error.

Remark 3.19 (Julia function collect)

The function `collect` converts the abstract "range" object given by function `range` into an ordinary 1D array.

```
for n in 2:5
    println("n=$n")
    H_n = hilbert(n)
    x = collect(range(1.0, n, n))
    println("x is $x")
    b = H_n * x
    println("b is $b")
    error_scale = 1e-8
    b_imperfect = b + 2.0 * error_scale * (rand(Float64, (n)) .- 0.5) # add random
↪"errors" between -error_scale and error_scale
    println("b has been slightly changed to $b_imperfect")
    x_computed = solvelinearsystem(H_n, b_imperfect)
    residual = b - H_n * x_computed
    relative_backward_error = norm(residual, Inf)/norm(b, Inf)
    println("The residual maximum norm is $(round(norm(residual, Inf), sigdigits=2))")
    println("and the relative backward error ||r||/||b|| is $(round(relative_backward_
↪error, sigdigits=2))")
    absolute_error = norm(x - x_computed, Inf)
    relative_error = absolute_error/norm(x, Inf)
    println("The absolute error is $(round(absolute_error, sigdigits=2))")
```

(continues on next page)

(continued from previous page)

```
    println("The relative error is $(round(relative_error, sigdigits=2))")
    error_bound = cond(H_n, Inf) * relative_backward_error
    println("For comparison, the relative error bound from the formula above is
↪$(round(error_bound, sigdigits=2))")
    println("\nBeware: the relative error is larger than the relative backward error␣
↪by a factor ",
        "$(round(relative_error/relative_backward_error, sigdigits=2))")
    println()
end
```

```
n=2
x is [1.0, 2.0]
b is [2.0, 1.1666666666666665]
b has been slightly changed to [2.0000000080803297, 1.1666666764748042]
The residual maximum norm is 9.8e-9
and the relative backward error ||r||/||b|| is 4.9e-9
The absolute error is 6.9e-8
The relative error is 3.5e-8
For comparison, the relative error bound from the formula above is 1.3e-7

Beware: the relative error is larger than the relative backward error by a factor␣
↪7.1

n=3
x is [1.0, 2.0, 3.0]
b is [3.0, 1.9166666666666665, 1.4333333333333333]
b has been slightly changed to [2.999999991652919, 1.9166666621675716, 1.
↪433333325752684]
The residual maximum norm is 8.3e-9
and the relative backward error ||r||/||b|| is 2.8e-9
The absolute error is 8.1e-7
The relative error is 2.7e-7
For comparison, the relative error bound from the formula above is 2.1e-6

Beware: the relative error is larger than the relative backward error by a factor␣
↪96.0

n=4
x is [1.0, 2.0, 3.0, 4.0]
b is [4.0, 2.716666666666667, 2.1, 1.7214285714285713]
b has been slightly changed to [4.000000006782543, 2.7166666759068043, 2.
↪099999995598059, 1.72142857008299]
The residual maximum norm is 9.2e-9
and the relative backward error ||r||/||b|| is 2.3e-9
The absolute error is 4.6e-5
The relative error is 1.2e-5
For comparison, the relative error bound from the formula above is 6.6e-5

Beware: the relative error is larger than the relative backward error by a factor␣
↪5000.0

n=5
x is [1.0, 2.0, 3.0, 4.0, 5.0]
b is [5.0, 3.5500000000000003, 2.8142857142857145, 2.3464285714285715, 2.
↪0174603174603174]
```

(continues on next page)

(continued from previous page)

```
b has been slightly changed to [5.000000006354475, 3.5500000045593687, 2.
↪814285713289512, 2.346428570499546, 2.0174603141398526]
The residual maximum norm is 6.4e-9
and the relative backward error ||r||/||b|| is 1.3e-9
The absolute error is 0.00036
The relative error is 7.1e-5
For comparison, the relative error bound from the formula above is 0.0012

Beware: the relative error is larger than the relative backward error by a factor␣
↪56000.0
```

We see in these experiments that:

- As the condition number increases, the relative error becomes increasingly larger than the backward error computed from the residual.
- It is less than the above bound $\text{Rel}(x_a) = \dfrac{\|x - x_a\|}{\|x\|} \leq \kappa(A)\dfrac{\|r\|}{\|b\|}$, and typically quite a bit less.

3.7 Iterative Methods for Simultaneous Linear Equations

References:

- Section 2.5 *Iterative Methods* in [Sauer, 2019], sub-sections 2.5.1 to 2.5.3.
- Chapter 7 *Iterative Techniques in Linear Algebra* in [Burden *et al.*, 2016], sections 7.1 to 7.3.
- Section 8.4 in [Chenney and Kincaid, 2012].

3.7.1 Introduction

This topic is a huge area, with lots of ongoing research; this section just explores the first few methods in the field:

1. The Jacobi Method.
2. The Gauss-Seidel Method.

The next three major topics for further study are:

1. The Method of Succesive Over-Relaxation ("SOR"). This is usually done as a modification of the Gauss-Seidel method, though the strategy of "over-relaxation" can also be applied to other iterative methods such as the Jacobi method.
2. The Conjugate Gradient Method ("CG"). This is beyond the scope of this course; I mention it because in the realm of solving linear systems that arise in the solution of differential equations, CG and SOR are the basis of many of the most modern, advanced methods.
3. Preconditioning.

3.7.2 The Jacobi method

The basis of the Jacobi method for solving $Ax = b$ is splitting A as $D + R$ where D is the diagonal of A:

$$d_{i,i} = a_{i,i}$$
$$d_{i,j} = 0, \quad i \neq j$$

so that $R = A - D$ has

$$r_{i,i} = 0$$
$$r_{i,j} = a_{i,j}, \quad i \neq j$$

Visually

$$D = \begin{bmatrix} a_{11} & 0 & 0 & \dots \\ 0 & a_{22} & 0 & \dots \\ 0 & 0 & a_{33} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

It is easy to solve $Dx = b$: the equations are just $a_{ii}x_i = b_i$ with solution $x_i = b_i/a_{ii}$.

Thus we rewrite the equation $Ax = Dx + Rx = b$ in the fixed point form

$$Dx = b - Rx$$

and then use the familiar fixed point iteration strategy of inserting the currect approximation at right and solving for the new approximation at left:

$$Dx^{(k)} = b - Rx^{(k-1)}$$

Note: We could make this look closer to the standard fixed-point iteration form $x_k = g(x_{k-1})$ by dividing out D to get

$$x^{(k)} = D^{-1}(b - Rx^{(k-1)}),$$

but — as is often the case — it will be better to avoid matrix inverses by instead solving this easy system. This "inverse avoidance" becomes far more important when we get to the Gauss-Seidel method!

Exercise 1: Implement and test the Jacobi method

Write and test Python functions for this.

A) As usual start with a most basic version that does a fixed number of iterations

```
x = jacobi_basic(A, b, n)
```

B) Then refine this to apply an error tolerance, but also avoiding infinite loops by imposing an upper limit on the number of iterations:

```
x = jacobi(A, b, errorTolerance, maxIterations)
```

Test this with the matrices of form T below for several values of n, increasingly geometrically. To be cautious initially, try $n = 2, 4, 8, 16, ...$

3.7.3 The underlying strategy

To analyse the Jacobi method — answering questions like for which matrices it works, and how quickly it converges — and also to improve on it, it helps to described a key strategy underlying it, which is this: approximate the matrix A by another one E one that is easier to solve with, chosen so that the discrepacy $R = A - E$ is small enough. Thus, repeatedly solving the new easier equations $Ex^{(k)} = b^{(k)}$ plays a similar role to repeatedly solving tangent line approximations in Newton's method.

Of course to be of any use, E must be somewhat close to A; the remainder R must be small enough. We can make this requirement precise with the use of matrix norms introduced in *Error bounds for linear algebra, condition numbers, matrix norms, etc.* and an upgrade of the contraction mapping theorem seen in *Solving Equations by Fixed Point Iteration (of Contraction Mappings)*.

Thus consider a general *splitting* of A as $A = E + R$. As above, we rewrite $Ax = Ex + Rx = b$ as $Ex = b - Rx$ and thence as $x = E^{-1}b - (E^{-1}R)x$. (It is alright to use the matrix inverse here, since we are not actually computing it; only using it for a theoretical argument!) The fixed point iteration form is thus

$$x^{(k)} = g(x^{(k-1)}) = c - Sx^{(k-1)}$$

where $c = E^{-1}b$ and $S = E^{-1}R$.

For vector-valued functions we extend the previous *Definition 2.2* in Section *Solving Equations by Fixed Point Iteration (of Contraction Mappings)* as:

Definition 3.7 (Vector-valued contraction mapping)

For a set D of vectors in $\mathbb{R}^n$, a mapping $g : D \to D$ is called a *contraction* or *contraction mapping* if there is a constant $C < 1$ such that

$$\|g(x) - g(y)\| \leq C\|x - y\|$$

for any x and y in D. We then call C a *contraction constant*.

Next, the contraction mapping theorem *Theorem 2.1* extends to

---

Theorem 3.6 (Contraction mapping theorem for vector-valued functions)

- Any contraction mapping g on a closed, bounded set $D \in \mathbb{R}^n$ has exactly one fixed point p in D.
- This can be calculated as the limit $p = \lim_{k\to\infty} x^{(k)}$ of the iteration sequence given by $x^{(k)} = g(x^{(k-1)})$ for *any* choice of the starting point $x^{(0)} \in D$.
- The errors decrease at a guaranteed minimum speed: $\|x^{(k)} - p\| \leq C\|x^{(k-1)} - p\|$, so $\|x^{(k)} - p\| \leq C^k\|x^{(0)} - p\|$.

With this, it turns out that the above iteration converges if S is "small enough" in the sense that $\|S\| = C < 1$ — and it is enough that this works for *any* choice of matrix norm!

Theorem 3.7

If $S := E^{-1}R = E^{-1}A - I$ has $\|S\| = C < 1$ for any choice of matrix norm, then the iterative scheme $x^{(k)} = c - Sx^{(k-1)}$ with $c = E^{-1}b$ converges to the solution of $Ax = b$ for any choice of the initial approximation $x^{(0)}$. (Aside: the zero vector is an obvious and popular choice for $x^{(0)}$.)

Incidentally, since this condition guarantees that there exists a unique solution to $Ax = b$, it also shows that A is non-singular.

Proof. (sketch)

The main idea is that for $g(x) = c - Sx$,

$$\|g(x) - g(y)\| = \|(c - Sx) - (c - Sy)\| = \|S(y - x)\| \leq \|S\|\|y - x\| \leq C\|x - y\|,$$

so with $C < 1$, it is a contraction.

(The omitted more "technical" detail is to find a suitable bounded domain D that all the iterates x^{(k)} stay inside it.)

What does this say about the Jacobi method?

For the Jacobi method, $E = D$ so E^{-1} is the diagonal matrix with elements $1/a_{i,i}$ on the main diagonal, zero elsewhere. The product $E^{-1}A$ then multiplies each row i of A by $1/a_{i,i}$, giving

$$E^{-1}A = \begin{bmatrix} 1 & a_{1,2}/a_{1,1} & a_{1,2}/a_{1,1} & \cdots \\ a_{2,1}/a_{2,2} & 1 & a_{2,3}/a_{2,2} & \cdots \\ a_{3,1}/a_{3,3} & a_{3,2}/a_{3,3} & 1 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

so that subtracting the identity matrix to get S cancels the ones on the main diagonal:

$$S = E^{-1}A - I = \begin{bmatrix} 0 & a_{1,2}/a_{1,1} & a_{1,2}/a_{1,1} & \cdots \\ a_{2,1}/a_{2,2} & 0 & a_{2,3}/a_{2,2} & \cdots \\ a_{3,1}/a_{3,3} & a_{3,2}/a_{3,3} & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Here is one of many places that using the maximum-norm, a.k.a. ∞-norm, makes life much easier! Recalling that this is given by

$$\|A\|_\infty = \max_{i=1}^{n} \left(\sum_{j=1}^{n} |a_{i,j}| \right),$$

- First, sum the absolute values of elements in each row i; with the common factor $1/|a_{i,i}|$, this gives $\left(|a_{i,1}| + |a_{i,2}| + \cdots |a_{i,i-1}| + |a_{i,i+1}| + \cdots |a_{i,n}|\right) / |a_{i,i}|$.

  Such a sum, skipping index $j = i$, can be abbreviated as

$$\left(\sum_{1 \leq j \leq n, j \neq i} |a_{i,j}| \right) \Big/ |a_{i,i}|$$

- Then get the norm as the maximum of these:

$$C = \|E^{-1}A\|_\infty = \max_{i=1}^{n} \left[\left(\sum_{1 \leq j \leq n, j \neq i} |a_{i,j}| \right) \Big/ |a_{i,i}| \right]$$

and the contraction condition $C < 1$ becomes the requirement that each of these n "row sums" is less than 1:

Multiplying each of the inequalities by the denominator $|a_{i,i}|$ gives n conditions

$$\left(\sum_{1 \leq j \leq n, j \neq i} |a_{i,j}| \right) < |a_{i,i}|$$

This is strict diagonal dominance, as in *Definition 3.1* in the section *Row Reduction/Gaussian Elimination*, and as discussed there, one way to think of this is that such a matrix A is close to its main diagonal D, which is the intuitive condition that the approximation of A by D as done in the Jacobi method is "good enough".

And indeed, combining this result with *Theorem 3.7* gives:

Theorem 3.8 (Convergence of the Jacobi method)

The Jacobi Method converges if A is strictly diagonally dominant, for any initial approximation $x^{(0)}$.

Further, the error goes down by at least a factor of $\|I - D^{-1}A\|$ at each iteration.

By the way, other matrix norms give other conditions guaranteeing convergence; perhaps the most useful of these others is that it is also sufficient for A to be column-wise strictly diagonally dominant as in *Definition 3.2*.

3.7.4 The Gauss-Seidel method

To recap, two key ingredients for a splitting $A = E + R$ to be useful are that

- the matrix E is "easy" to solve with, and
- it is not too far from A.

The Jacobi method choice of E being the main diagonal of A strongly emphasizes the "easy" part, but we have seen another larger class of matrices for which it is fairly quick and easy to solve $Ex = b$: *triangular matrices*, which can be solved with forward or backward substitution, not needing row reduction.

The Gauss-Seidel Method takes E be the lower triangular part of A, which intuitively leaves more of its entries closer to A and makes the remainder $R = A - E$ "smaller".

To discuss this and other splittings, we write the matrix as $A = L + D + U$ where:

- D is the diagonal of A, as for Jacobi
- L is the strictly lower diagonal part of A (just the elements with $i > j$)
- U is the strictly upper diagonal part of A (just the elements with $i < j$)

That is,

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \cdots \\ a_{2,1} & 0 & 0 & \cdots \\ a_{3,1} & a_{3,2} & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} + \begin{bmatrix} a_{1,1} & 0 & 0 & \cdots \\ 0 & a_{2,2} & 0 & \cdots \\ 0 & 0 & a_{3,3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} + \begin{bmatrix} 0 & a_{1,2} & a_{1,3} & \cdots \\ 0 & 0 & a_{2,3} & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = L + D + U$$

Thus $R = L + U$ for the Jacobi method.

So now we use $E = L + D$, which will be called A_L, the *lower triangular part of* A, and the remainder is $R = U$. The fixed point form becomes

$$A_L x = b - Ux$$

giving the fixed point iteration

$$A_L x^{(k)} = b - Ux^{(k-1)}$$

Here we definitely do not use the inverse of A_L when calculating! Instead, solve with forward substitution.

However to analyse convergence, the mathematical form

$$x^{(k)} = A_L^{-1}b - (A_L^{-1}U)x^{(k-1)}$$

is useful: the iteration map is now $g(x) = c - Sx$ with $c = (L + D)^{-1}b$ and $S = (L + D)^{-1}U$.

Arguing as above, we see that convergence is guaranteed if $\|(L+D)^{-1}U\| < 1$. However it is not so easy in general to get a formula for $\|(L+D)^{-1}U\|$; what one can get is slightly disappointing in that, despite the $R = U$ here being in some sense "smaller" than the $R = L + U$ for the Jacobi method, the general convergence guarantee looks no better:

Theorem 3.9 (Convergence of the Gasuss-Seidel method)

The Gauss-Seidel method converges if A is strictly diagonally dominant, for any initial approximation $x^{(0)}$.

However, in practice the convergence rate as given by $C = C_{GS} = \|(L+D)^{-1}U\|$ is often better than for the $C = C_J = \|D^{-1}(L+U)\|$ for the Jacobi method.

Sometimes this reduces the number of iterations enough to outweigh the extra computational effort involved in each iteration and make this faster overall than the Jacobi method — but not always.

Exercise 2: Implement and test the Gauss-Seidel method, and compare to Jacobi

Do the two versions as above and use the same test cases.

Then compare the speed/cost of the two methods: one way to do this is by using Julia's "wall clock time" function `time` or the macro `@time`; see the linked descriptions in the Julia manual.

3.7.5 A family of test cases, arising from boundary value problems for differential equations

The **tri-diagonal** matrices T of the form

$$\begin{aligned} t_{i,i} &= 1 + 2h^2 \\ t_{i,i+1} = t_{i,i+1} &= -h^2 \\ t_{i,j} &= 0, \quad |i-j| > 1 \end{aligned}$$

and variants of this arise in the solutions of boundary value problems for ODEs like

$$\begin{aligned} -u''(x) + Ku &= f(x), \quad a \le x \le b \\ u(a) = u(b) &= 0 \end{aligned}$$

and related problems for partial differential equations.

Thus these provide useful initial test cases — usually with $h = (b-a)/n$.

3.8 Faster Methods for Solving $Ax = b$ for Tridiagonal and Banded matrices, and Strict Diagonal Dominance

Reference:

Section 6.6 *Special Types of Matrices* in [Burden *et al.*, 2016], the sub-sections on *Band Matrices* and *Tridiagonal Matrices*.

3.8.1 Tridiagonal systems

Differential equations often lead to the need to solve systems of equations $Tx = b$ where the matrix T is **tri-diagonal**: the only non-zero elements are on the main diagonal and the diagonals adjacent to it on either side, so that $T_{i,j} = 0$ if $|i - j| > 1$. That is, the system looks like:

Definition 3.8 (Tridiagonal matrix)

A matrix T is **tridiagonal** if the only non-zero elements are on the main diagonal and the diagonals adjacent to it on either side, so that $T_{i,j} = 0$ if $|i - j| > 1$. That is, the system looks like:

$$Tx = \begin{bmatrix} d_1 & u_1 & & & & \\ l_1 & d_2 & u_2 & & & \\ & l_2 & d_3 & u_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & l_{n-2} & d_{n-1} & u_{n-1} \\ & & & & l_{n-1} & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

with all "missing" entries being zeros. The notation used here suggests one efficient way to store such a matrix: as three 1D arrays d, l and u.

(Such equations also arise in other important situations, such as *Spline Interpolation*.)

It can be verified that LU factorization preserves all the non-zero values, so that the Doolittle algorithm — if it succeeds without any division by zero — gives $T = LU$ with the form

$$L = \begin{bmatrix} 1 & & & & & \\ L_1 & 1 & & & & \\ & L_2 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & L_{n-2} & 1 & \\ & & & & L_{n-1} & 1 \end{bmatrix}, \; U = \begin{bmatrix} D_1 & u_1 & & & & \\ & D_2 & u_2 & & & \\ & & D_3 & u_3 & & \\ & & & \ddots & \ddots & \\ & & & & D_{n-1} & u_{n-1} \\ & & & & & D_n \end{bmatrix}$$

Note that the first non-zero element in each column is unchanged, as with a full matrix, but now it means that the upper diagonal elements u_i are unchanged.

Again, one way to describe and store this information is with just the two new 1D arrays L and D, along with the unchanged array u.

3.8.2 Algorithms

Algorithm 3.9 (LU factorization)

$D_1 = d_1$

```
for i from 2 to n
```

$L_{i-1} = l_{i-1}/D_{i-1}$

$D_i = d_i - L_{i-1}u_{i-1}$

```
end
```

Algorithm 3.10 (Forward substitution)

$c_1 = b_1$

```
for i from 2 to n
```

$c_i = b_i - L_{i-1}c_{i-1}$

```
end
```

Algorithm 3.11 (Backward substitution)

$x_n = c_n/D_n$

```
for i from n-1 down to 1
```

$x_i = (c_i - u_i x_{i+1})/D_i$

```
end
```

3.8.3 Generalizing to banded matrices

As we have seen, approximating derivatives to higher order of accuracy and approximating derivatives of order greater than two requires more than three nodes, but the locations needed are all close to the ones where the derivative is being approximated. For example, the simplest symmetric approximation of the fourth derivative $D^4 f(x)$ used values from $f(x-2h)$ to $f(x+2h)$. Then row i of the corresponding matrix has all its non-zero elements at locations $(i, i-2)$ to $(i, i+2)$: the non-zero elements lie in the narrow "band" where $|i-j| \leq 2$, and thus on five "central" diagonals.

This is a **penta-digonal matrix**, and an example of the larger class of **banded matrices**: ones in which all the non-zero elements have indices $-p \leq j - i \leq q$ for p and q smaller than n — usually far smaller; $p = q = 2$ for a penta-digonal matrix.

Let us recap the general Doolittle algorithm for computing an LU factorization:

Algorithm 3.12 (Doolittle algorithm for computing an LU factorization)

The top row is unchanged:

```
for j from 1 to n
```

$u_{1,j} = a_{1,j}$

```
end
```

The left column requires no sums:

```
for i from 2 to n
```

$l_{i,1} = a_{i,1}/u_{1,1}$

```
end
```

The main loop: `for k from 2 to n`

```
  for j from k to n
```

$u_{k,j} = a_{k,j} - \sum_{s=1}^{k-1} l_{k,s}u_{s,j}$

```
  end
  for i from k+1 to n
```

$l_{i,k} = \left(a_{i,k} - \sum_{s=1}^{k-1} l_{i,s}u_{s,k}\right)/u_{k,k}$

```
  end
end
```

Eliminating redundant calculation in the above

With a banded matrix, many of the entries at right are zero, particularly in the two sums, which is where most of the operations are. Thus we can rewrite, exploiting the fact that all elements with indices $j - i < -p$ or $j - i > q$ are zero. To start with, the top diagonal is not modified, as already noted for the tridiagonal case: $u_{k,k+q} = a_{k,k+q}$ for $1 \leq k \leq n - q$.

Algorithm 3.13 (LU factorization of a banded matrix)

The top row is unchanged:

```
for j from 1 to 1+q
```

$u_{1,j} = a_{1,j}$

```
end
```

The top non-zero diagonal is unchanged:

```
for k from 1 to n - q
```

$u_{k,k+q} = a_{k,k+q}$

```
end
```

The left column requires no sums:

```
for i from 2 to 1+p
```

$l_{i,1} = a_{i,1}/u_{1,1}$

```
end
```

The main loop:

```
for k from 2 to n
  for j from k to min(n, k+q-1)
```

$$u_{k,j} = a_{k,j} - \sum_{s=max(1,k-p,j-q)}^{k-1} l_{k,s}u_{s,j}$$

```
  end
  for i from k+1 to min(n,k+p-1)
```

$$l_{i,k} = \left(a_{i,k} - \sum_{s=max(1,i-p,k-q)}^{k-1} l_{i,s}u_{s,k} \right) / u_{k,k}$$

```
  end
end
```

It is common for a banded matrix to have equal band-width on either side, $p = q$, as with tridiagonal and pentadiagonal matrices. Then the algorithm is somewhat simpler:

Algorithm 3.14 (LU factorization of a banded matrix, $p = q$)

The top row is unchanged:

```
for j from 1 to 1+p
```

$u_{1,j} = a_{1,j}$

```
end
```

The top non-zero diagonal is unchanged:

```
for k from 1 to n - p
```

$u_{k,k+p} = a_{k,k+p}$

```
end
```

The left column requires no sums:

```
for i from 2 to 1+p
```

$l_{i,1} = a_{i,1}/u_{1,1}$

```
end
```

The main loop:

```
for k from 2 to n
  for j from k to min(n, k+p-1)
```

$$u_{k,j} = a_{k,j} - \sum_{s=max(1,j-p)}^{k-1} l_{k,s}u_{s,j}$$

```
  end
  for i from k+1 to min(n,k+p)
```

$$l_{i,k} = \left(a_{i,k} - \sum_{s=max(1,i-p)}^{k-1} l_{i,s}u_{s,k} \right) / u_{k,k}$$

```
  end
end
```

3.8.4 Strict diagonal dominance helps again

These algorithms for banded matrices do no pivoting, and that is highly desirable, because pivoting creates non-zero elements outside the "band" and so can force one back to the general algorithm. Fortunately, we have seen one case where this is fine: the matrix being either row-wise or column-wise strictly diagonally dominant.

3.9 Computing Eigenvalues and Eigenvectors: the Power Method, and a bit beyond

References:

- Section 12.1 *Power Iteration Methods* of [Sauer, 2019].
- Section 7.2 *Eigenvalues and Eigenvectors* of [Burden *et al.*, 2016].
- Chapter 8, *More on Linear Equations* of [Chenney and Kincaid, 2012], in particular section 3 *Power Method*, and also section 2 *Eigenvalues and Eigenvectors* as background reading.

The *eigenproblem* for a square $n \times n$ matrix A is to compute some or all non-trivial solutions of

$$A\vec{v} = \lambda\vec{v}.$$

(By non-trivial, I mean to exclude $\vec{v} = 0$, which gives a solution for any λ.) That is, to compute the eigenvalues λ (of which generically there are n, but sometimes less) and the eigenvectors $\vec{v}$ corresponding to each.

With eigenproblems, and particularly those arising from differential equations, one often needs only the few smallest and/or largest eigenvalues. For these, the *power method* described next can be adapted, leading to the *shifted inverse power method*.

Here we often restict our attention to the case of a real *symmetric* matrix ($A^T = A$, or $A_{ij} = A_{ji}$), or a Hermitian matrix ($A^T = A^*$), for which many things are a bit simpler:

- all eigenvalues are real,
- for symmetric matrices, all eigenvectors are also real,
- there is a complete set of orthogonal eigenvectors $\vec{v}_k$, $1 \leq i \leq n$ that form a basis for all vectors, and so on.

However, the methods described here can be used more generally, or can be made to work with minor adjustments.

The eigenvalue are roots of the characteristic polynomial, $\det(A - \lambda I)$; repeated roots are possible, and they will all be named, so there are always values λ_i, $1 \leq i \leq n$. Here, these eigenvalues will be enumerated in decreasing order of magnitude:

$$|\lambda_1| \geq |\lambda_2| \cdots \geq |\lambda_n|.$$

Generically, all the magnitudes are different, which makes things works more easily, so that will sometimes be assumed while developing the intuition of the method.

3.9.1 The Power Method

The basic tool is the *Power Method*, which will usually *but not always* succeed in computing the eigenvalue of largest magnitude, λ_1, and **a** corresponding eigenvector $\vec{v}_1$. Its success mainly involves assuming there being a unique largest eigenvalue: $\lambda_1 > \lambda_i$ for $i > 1$.

In its simplest form, one starts with a unit-length vector $\vec{x}^{\,0}$, so $\|\vec{x}^{\,0}\| = 1$, constructs the successive multiples $\vec{y}^{\,k} = A^k\vec{x}^{\,0}$ by successive multiplications, and rescales at each stage to the unit vectors $\vec{x}^{\,k} = \vec{y}^{\,k}/\|\vec{y}^{\,k}\|$.

Note that $\vec{y}^{\,k+1} = A\vec{x}^{\,k}$, so that once $\vec{x}^{\,k}$ is approximately an eigenvector for eigenvalue λ, $\vec{y}^{\,k+1} \approx \lambda\vec{x}^{\,k}$, leading to the eigenvalue approximation

$$r^{(k)} := \langle \vec{y}^{\,k+1}, \vec{x}^{\,k} \rangle \approx \langle \lambda\vec{x}^{\,k}, \vec{x}^{\,k} \rangle \approx \lambda$$

Remark 3.20 (dot products in Julia)

Here and below, I use $\langle \vec{a}, \vec{b} \rangle$ to denote the inner product (a.k.a. *dot product* or *scalar product*) of two vectors.

With Julia arrays this is given by function dot(a,b) from package LinearAlgebra, obtained with

```
using LinearAlgebra: dot
```

You can even type this as

```
a ⋅ b
```

(To get that centered dot in Julia, type \cdot and then tab.)

Algorithm 3.15 (A basic version of the power method)

Choose initial vector $\vec{y}_0$, maybe with a random number generator.

Normalize to $\vec{x}^{\,0} = \vec{y}^{\,0}/\|\vec{y}^{\,0}\|$.

for k from 0 to k_{max}

$\vec{y}^{\,k+1} = A\vec{x}^{\,k}$

$r^{(k)} = \langle \vec{y}^{\,k+1}, \vec{x}^{\,k} \rangle$

$\vec{x}^{\,k+1} = \vec{y}^{\,k+1}/\|\vec{y}^{\,k+1}\|$

end

The final values of $r^{(k)}$ and $\vec{x}^{\,k}$ approximate λ_1 and $\vec{v}_1$ respectively.

Exercise 1

Implement this algorithm and test it on the real, symmetric matrix

$$A = \begin{bmatrix} 3 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 4 \end{bmatrix}$$

This all real eigenvalues, all within 2 of the diagonal elements (this claim should be explained as part of the project write-up), so start with it.

As a debugging strategy, you could replace all those off-diagonal ones by a small value δ:

$$A_\delta = \begin{bmatrix} 3 & \delta & \delta \\ \delta & 8 & \delta \\ \delta & \delta & 4 \end{bmatrix}$$

Then the Gershgorin circle theorem ensures that each eigenvalue is within 2δ of an entry on the main diagonal. Furthermore, if δ is small enough that the circles of radius 2δ centered on the diagonal elements do not overlap, then there is one eigenvalue in each circle.

You could even start with $\delta = 0$, for which you know exactly the eigenvalues: they are the diagonal elements.

Here and below you could check your work with Julia, using function eigvals from package LinearAlgebra.

However, that is almost cheating, so note that there is also a backward error check: see how small $\|Av - \lambda v\|/\|v\|$ is.

```
using LinearAlgebra: eigvals
```

```
include("NumericalMethods.jl")
using .NumericalMethods: printmatrix
```

```
delta = 0.01
A = [3.0 delta delta ; delta 8.0 delta ; delta delta 4.0]
eigenvalues = eigvals(A);
```

```
println("With delta=$(delta), so that A is"); printmatrix(A)
println("the eigenvalues are $(eigenvalues)")
```

```
With delta=0.01, so that A is
[ 3.0 0.01 0.01
  0.01 8.0 0.01
  0.01 0.01 4.0 ]
the eigenvalues are [2.999880409987068, 4.000074490251736, 8.00004509976119]
```

Remark 3.21 (On Julia)

That package LinearAlgebra also has a function eigvecs for computing eignevevctors.

It can compute them "from scratch" with `eigenvectors = eigvecs(A)`, which returns them in the columns of that matrix `eigenvectors`.

However, if you already have the eigenvalues, the calculation is much quicker by using them: that is done with `eigenvectors = eigvecs(A, eigenvalues)`

Refinement: deciding the iteration count

Some details are omitted above; above all, how to decide the number of iterations.

One approach is to use the fact that an eigenvector-eigenvalue pair satisfies $A\vec{v} - \lambda\vec{v} = 0$, so the "residual norm"

$$\frac{\|A\vec{x}^{\,k} - r^{(k)}\vec{x}^{\,k}\|}{\|\vec{x}^{\,k}\|}, = \|\vec{y}^{\,k+1} - r^{(k)}\vec{x}^{\,k}\| \text{ since } \|\vec{x}^{\,k}\| = 1$$

is a measure of "relative backward error".

Thus one could repace the above `for` loop by a `while` loop based on a condition like stopping when

$$\|\vec{y}^{\,k+1} - r^{(k)}\vec{x}^{\,k}\| \leq \epsilon.$$

Alternatively, keep the `for` loop, but exit early (with `break`) if this condition is met.

I generally recommend this `for-if-break` form for implementing iterative methods, because it makes avoidance of infinite loops simpler, and avoids the common `while` loop issue that you do not yet have an error estimate when the loop starts.

Exercise 2

Modify your code from Exercise 1 to implement this accuracy control.

3.9.2 The Inverse Power Method

The next step is to note that if A is nonsingular, its inverse $B = A^{-1}$ has the same eigenvectors, but with eigenvalues $\mu_i = 1/\lambda_i$.

Thus we can apply the power method to B in order to compute its largest eigenvalue, which is $\mu_n = 1/\lambda_n$, along with the corresponding eigenvector $\vec{v}_n$.

The main change to the above is that

$$\vec{y}^{\,k+1} = A^{-1}\vec{x}_k.$$

However, as usual one can (and should) avoid actually computing the inverse. Instead, express the above as the sysem of equations.

$$A\vec{y}^{\,k+1} = \vec{x}_k.$$

Here is an important case where the LU factorization method can speed things up greatly: a single LU factorization is needed, after which for each k one only has to do the far quicker forward and backward substitution steps: $O(n^2)$ cost for each iteration instead of $O(n^3/3)$.

Algorithm 3.16 (A basic version of the inverse power method)

Choose initial vector $\vec{y}_0$, maybe with a random number generator.

Normalize to $\vec{x}^{\,0} = \vec{y}^{\,0}/\|\vec{y}^{\,0}\|$.

Compute an LU factorization $LU = A$.

for k from 0 to k_{max}

  Solve $L\vec{z}^{\,k+1} = \vec{x}^{\,k}$

  Solve $U\vec{y}^{\,k+1} = \vec{z}^{\,k+1}$

  $r^{(k)} = \langle \vec{y}^{\,k+1}, \vec{x}^{\,k} \rangle$

  $\vec{x}^{\,k+1} = \vec{y}^{\,k+1}/\|\vec{y}^{\,k+1}\|$

end

(If all goes well) the final values of $r^{(k)}$ and $\vec{x}^{\,k}$ approximate λ_n and $\vec{v}_n$ respectively.

Exercise 3

Implement this basic algorithm (with a fixed iteration count, as in Example 1), and then create a second version that imposes an accuracy target (as in Example 2).

3.9.3 Getting other eigenvalues with the Shifted Inverse Power Method

The inverse power method computes the eigenvalue closest to 0; by *shifting*, we can compute the eigenvalue closest to any chosen value s. Then by searching various values of s, we can hope to find all the eigenvectors. As a variant, once we have λ_1 and λ_n, we can search nearby for other large or small eigenvalues: often the few largest and/or the few smallest are most important.

With a symmetric (or Hermitian) matrix, once the eigenvalue of largest magnitude, λ_1 is known, the rest are known to be real values in the interval $[-|\lambda_1|, |\lambda_1|]$, so we know roughly where to seek them.

The main idea here is that for any number s, matrix $A - sI$ has eigenvalues $\lambda_i - s$, with the same eigenvectors as A:

$$(A - sI)\vec{v}_i = (\lambda_i - s)\vec{v}_i$$

Thus, applying the inverse power method to $A - sI$ computes its largest eigenvalue γ, and then $\lambda = 1/(\gamma + s)$ is the eigenvalue of A closest to s.

Exercise 4

As above, implement this, probably sarting with a fixed iteration count version.

For the test case above, some plausible initial choices for the shifts are each if the entries on the main diagonal, and as above, testing with A_s

3.9.4 Further topics: getting all the eigenvalues with the QR Method, etc.

The above methods are not ideal when many or all of the eigenvalues of a matrix are wanted; then a variety of more advanced methods have been developed, starting with the QR (factorization) method.

We will not address the details of that method in this course, but one way to think about it for a symmetric matrix is that:

- The eigenvectors are orthogonal.
- Thus, if after computing λ_1 and $\vec{v}_1$, one uses the power iteration starting with $\vec{x}^{\,0,2}$ orthogonal to $\vec{v}_1$, then all the new iterates $\vec{x}^{\,k,2}$ will stay orthogonal, and one will get the eigenvector corresponding to the largest remaining eigenvector: you get $\vec{v}_2$ and λ_2.
- Continuing likewise, one can get the eigenvalues in descending order of magnitude.
- As a modification, one can do all these almost in parallel: at iteration k, have an approximation $\vec{x}^{\,k,i}$ for each λ_i and at each stage, got by adjusting these new approximations so that $\vec{x}^{\,k,i}$ is orthogonal to all the approximations $\vec{x}^{\,k,j}$, $j < i$, for all the previous (larger) eigenvalues. This uses a variant of the Gram-Schmidt method for orthogonalizing a set of vectors.

3.10 Solving Nonlinear Systems of Equations by generalizations of Newton's Method — a brief introduction

References:

- Section 2.7 *Nonlinear Systems of Equations* of [Sauer, 2019]; in particular Sub-section 2.7.1 *Multivariate Newton's Method*.
- Chapter 10 *Numerical Solution of Nonlinear Systems of Equations* of [Burden *et al.*, 2016]; in particular Sections 10.1 and 10.2.

3.10.1 Background

A system of simultaneous nonlinear equations

$$F(x) = 0,\ F : \mathbb{R}^n \to \mathbb{R}^n$$

can be solved by a variant of Newton's method.

Remark (Some notation)

For the sake of emphasise analogies between results for vector-valued quantities and what we have already seen for real- or compex-valued quantities, several notational choices are made in thes notes:

- Using the same notation for vectors as for real or complex numbers (no over arrows or bold face).
- Often denoting the derivative of function f as Df rather than f', and higher derivatives as $D^p f$ rather than $f^{(p)}$. Partial derivatives of $f(x, y, \dots)$ are $D_x f$, $D_y f$, etc., and for vector arguments $x = (x_1, \dots, x_j, \dots x_n)$, they are $D_{x_1} f, \dots, D_{x_j} f, \dots, D_{x_n} f$, or more concisely, $D_1 f \dots, D_j f, \dots, D_n f$. (This also fits better with Julia code notation — even for partial derivatives.)
- Subscripts will mostly be reserved for components of vectors, labelling the terms of a sequence with superscripts, $x^{(k)}$ and such.
- Explicit division is avoided.

However, I use capital letters for vector-valued functions, for analogy to the use of capital letter for matrices.

Rewriting Newton's method according to this new style, \$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{Df(x^{(k)})}$\$

or to avoid explicit division and introducing the useful increment $\delta^{(k)} := x^{(k+1)} - x^{(k)}$,

$$Df(x^{(k)})(\delta^{(k)}) = f(x^{(k)}), \quad x^{(k+1)} = x^{(k)} + \delta^{(k)}.$$

3.10.2 Newton's method iteration formula for systems

For vector valued functions, we will see in a while that an analogous result is true:

$$(D\mathbf{F}(x^{(k)})(\delta^{(k)}) = \mathbf{F}(x^{(k)}), \quad x^{(k+1)} = x^{(k)} + \delta^{(k)}$$

where $DF(x)$ is the $n \times n$ matrix of all the partial derivatives $(D_{x_j} F_i)(x)$ or $(D_j F_i)(x)$, where $x = (x_1, x_2, \dots, x_n)$.

Justification: linearization for function of several variables

To justify the above result, we need at least a case of Taylor's Theorem for functions of several variables, for both $f : \mathbb{R}^n \to \mathbb{R}$ and $F : \mathbb{R}^n \to \mathbb{R}^n$; just for linear approximations. This material from multi-variable calculus will be reviewed when we need it.

Warning: although mathematically this can be written with matrix inverses as

$$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} - (D\mathbf{F}(\mathbf{X}^{(k)})^{-1}(\mathbf{F}(\mathbf{X}^{(k)}),$$

evaluation of the inverse is in general about three times slower than solving the linear system, so is best avoided. (We have seen a good compromise; *Solving Ax = b with LU factorization* the LU factorization of a matrix.)

Even avoiding matrix inversion, this involves repeatedly solving systems of n simultaneous linear equations in n unknowns, $Ax = b$, where the matrix A is $D\mathbf{F}(x^{(k)})$, and that will be seen to involve about $n^3/3$ arithmetic operations.

It also requires computing the new values of these n^2 partial derivatives at each iteration, also potentially with a cost proportional to n^3.

When n is large, as is common with differential equations problems, this factor of n^3 indicates a potentially very large cost per iteration, so various modifications have been developed to reduce the computational cost of each iteration (with the trade-off being that more iterations are typically needed): so-called *quasi-Newton methods.*

CHAPTER

FOUR

POLYNOMIAL COLLOCATION AND APPROXIMATION

References:

- Chapter 3 *Interpolation* of [Sauer, 2019].
- Chapter 3 *Interpolation and Polynomial Approximation* of [Burden *et al.*, 2016].
- Chapter 4 of [Kincaid and Chenney, 1990].

4.1 Polynomial Collocation (Interpolation/Extrapolation) and Approximation

References:

- Section 3.1 *Data and Interpolating Functions* in [Sauer, 2019].
- Section 3.1 *Interpolation and the Lagrange Polynomial* in [Burden *et al.*, 2016].
- Section 4.1 in [Chenney and Kincaid, 2012].

4.1.1 Introduction

Numerical methods for dealing with functions require describing them, at least approximately, using a finite list of numbers, and the most basic approach is to approximate by a polynomial. (Other important choices are rational functions and "trigonometric polynomials": sums of multiples of sines and cosines.) Such polynomials can then be used to aproximate derivatives and integrals.

The simplest idea for approximating $f(x)$ on domain $[a, b]$ is to start with a finite collection of **node** values $x_i \in [a, b]$, $0 \leq i \leq n$ and then seek a polynomial p which **collocates** with f at those values: $p(x_i) = f(x_i)$ for $0 \leq i \leq n$. Actually, we can put the function aside for now, and simply seek a polynomial that passes through a list of points (x_i, y_i); later we will achieve collocation with f by choosing $y_i = f(x_i)$.

In fact there are infinitely many such polynomials: given one, add to it any polynomial with zeros at all of the $n + 1$ notes. So to make the problem well-posed, we seek the collocating polynomial of *lowest degree*.

Theorem 4.1 (Existence and uniqueness of a collocating polynomial)

Given $n + 1$ distinct values x_i, $0 \leq i \leq n$, and corresponding y-values y_i, there is a unique polynomial P of degree at most n with $P(x_i) = y_i$ for $0 \leq i \leq n$.

(*Note:* although the degree is typically n, it can be less; as an extreme example, if all y_i are equal to c, then $P(x)$ is that constant c.)

Historically there are several methods for finding P_n and proving its uniqueness, in particular, the *divided difference* method introduced by Newton and the *Lagrange polynomial* method. However for our purposes, and for most modern needs, a different method is easiest, and it also introduces a strategy that will be of repeated use later in this course: the **Method of Undertermined Coefficients** or **MUC**.

In general, this method starts by assuming that the function wanted is a sum of unknown multiples of a collection of known functions. Here, $P(x) = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1} + c_n x^n = \sum_{j=0}^{n} c_j x^j$.

(*Note:* any of the c_i could be zero, including c_n, in which case the degree is less than n.)

The unknown factors $(c_0 \cdots c_n)$ are the *undetermined coefficients.*

Next one states the problem as a system of equations for these undetermined coefficients, and solves them.

Here, we have $n+1$ conditions to be met:

$$P(x_i) = \sum_{j=0}^{n} c_j x_i^j = y_i, \quad 0 \le i \le n$$

This is a system if $n+1$ simultaneous linear equations in $n+1$ unknowns, so the question of existence and uniqueness is exactly the question of whether the corresponding matrix is singular, and so is equivalent to the case of all $y_i = 0$ having *only* the solution with all $c_i = 0$.

Back in terms of polynomials, this is the claim that the only polynomial of degree at most n with zeros $x_0 \ldots x_n$. And this is true, because any non-trivial polynomial with those $n+1$ distinct roots is of degree at least $n+1$, so the only "degree n" polynomial fitting this data is $P(x) \equiv 0$. The theorem is proven.

The proof of this theorem is completely constructive, and it gives the only numerical method we need.

Briefly, the algorithm is this (indexing from 0 !)

- Create the $n+1 \times n+1$ matrix V with elements

$$v_{i,j} = x_i^j, \ 0 \le i \le n, \ 0 \le j \le n$$

and the $n+1$-element column vector y with elements y_i as above.

- Solve $Vc = y$ for the vector of coefficients c_j as above.

I use the name V because this is called the **Vandermonde Matrix.**

Enable graph plotting; PyPlot is an interface to the Python package `matplotlib.pyplot`

If this does not work in a downloaded notebook, see the instructions in the introduction.

```
using PyPlot
```

```
# A helper function, short-hand for rounding
import Base: round
round(x, n) = Base.round(x, sigdigits=n);
```

Example 4.1 (Exact fit)

As usual, I concoct a first example with known correct answer, by using a polynomial as f:

$$f(x) = 4 + 7x - 2x^2 - 5x^3 + 2x^4$$

using the nodes $x_0 = 1$, $x_1 = 2$, $x_2 = 0$, $x_3 = 3.3$ and $x_4 = 4$ (They do not need to be in order.)

```
f(x) = 4 + 7x -2x^2 - 5x^3 + 2x^4;
```

Remark 4.1 (A Julia shorthand for products)

Note the convenient short-hand of writing products by juxtapositon, without `*`. This works when the first factor is a literal number, not a variable name.

To avoid ambiguity with the use of dots for *vectorization* introduced below, avoid ending a floating point number with a period when it it an integer; for example you can use

```
f(x) = 4 + 7x ...
```

or

```
f(x) = 4 + 7.0x ...
```

but not

```
f(x) = 4 + 7.x + ...
```

```
xnodes = [1., 2., 0., 3.3, 4.]  # They do not need to be in order
println("The x nodes 'x_i' are $(xnodes)")
ynodes = f.(xnodes)
println("The y values at the nodes are $(round.(ynodes, 6))")  # Rounded for nicer␣
↪display
```

```
The x nodes 'x_i' are [1.0, 2.0, 0.0, 3.3, 4.0]
The y values at the nodes are [6.0, 2.0, 4.0, 62.8192, 192.0]
```

Remark 4.2 (Vectorizing a Julia function)

Appending the "point" to the function name (so it is `f.(...)` instead of `f(...)`) does **vectorization**: the function `f` is applied "pointwise", to each element of the input array `x_nodes` in turn, with the results returned in an aray of the same size and shape.

If there are several arguments, all are vectorized.

See *Vectorization of Functions* in the *Notes on the Julia Language*.

We will see this again soon.

The Vandermonde matrix:

```
nnodes = length(xnodes)
n = nnodes-1
V = zeros(nnodes, nnodes)
for i in 0:n
    for j in 0:n
        V[i+1,j+1] = xnodes[i+1]^j  # Shift the array indices up by one, since Julia␣
↪counts from 1, not 0.
    end
end
```

Solve, using our functions seen in earlier sections and gathered in *Module NumericalMethods*

```
include("NumericalMethods.jl")
using .NumericalMethods: solvelinearsystem, printmatrix
```

```
c_A = solvelinearsystem(V, ynodes)
# It helps the visual presentation to round off;
c_A = round.(c_A, 6)
println("The coefficients of P are $c_A")
```

```
The coefficients of P are [4.0, 7.0, -2.0, -5.0, 2.0]
```

These are correct!

To compare the computed values P[i] to what they shoudl be (y_nodes[i]) we use the convenient tool `zip` for looping over two or more "parallel" lists of values: `zip(x,y)` effectively produces a list of pairs `[(x[1], y[1]), (x[2], y[2]) ...)` up till one or both arrays runs out of elements, and it also works for three or more lists.

We can also check the resulting values of the polynomial:

```
P = c_A[1] .+ c_A[2]*xnodes + c_A[3]*xnodes.^2 + c_A[4]*xnodes.^3 + c_A[5]*xnodes.^4;
```

Remark 4.3 (Vectorizing operators and broadcasting values in Julia)

The period prefix in the exponential notation `.^` and the sum notation `.+` is **vectorization** and **broadcasting**:

- **vectorization**: `a.^b` means that if one of `a` and `b` are arrays (or both are arrays, of the same shape and size) then the exponentiation is applied to each element in turn, producing an array of the same size and shape;
- **brodcasting** of the addition: that first addition is "number plus array"; the notation `.+` indicates that this be done by promoting the number to an array matching the other addend.

Vectorization can also be applied to other binary operations such as multiplication and division.

See the notes on *Arithmetic operations on arrays: vectorization and broadcasting* in *Notes on the Julia Language*.

Now we can check the y values

Remark 4.4 (Zipping arrays together in Julia)

The function `zip` takes two or more arrays (preferably of the same size and shape) and turns them into an array with each element a tuple of values, one from the corresponding position in each input array.

For example, with 1D arrays `x` and `y` of length n, `zip(x, y)` is `[ ( x[1], y[1] ), ( x[2], y[2] ) ... ( x[n], y[n] ) ]`

```
for (x, y, P_i) in zip(xnodes, ynodes, P)
    println("P($x) should be $(round(y, 6)); it is $(round(P_i, 6))")
end
```

```
P(1.0) should be 6.0; it is 6.0
P(2.0) should be 2.0; it is 2.0
P(0.0) should be 4.0; it is 4.0
```

(continues on next page)

(continued from previous page)

```
P(3.3) should be 62.8192; it is 62.8192
P(4.0) should be 192.0; it is 192.0
```

4.1.2 Functions for computing the coefficients and evaluating the polynomials

We will use this procedure several times, so it time to put it into a functions. The names `polyfit` and `polyval` come from very similar functions that are standard in Python and Matlab.

We also add a pretty printer for polynomials, and import the pretty printer `printmatrix` for matrices.

```
function polyfit(x, y)
    # Compute the coeffients c_i of the polynomial of lowest degree that collocates␣
↪the points (x[i], y[i]).
    # These are returned in an array c of the same length as x and y, even if the␣
↪degree is less than the normal length(x)-1,
    # in which case the array has some trailing zeroes.
    # The polynomial is thus p(x) = c[1] + c[2]x + ... c[n+1] x^n where n=length(x)-1,
↪ the nominal degree.

    nnodes = length(x)
    n = nnodes - 1
    V = zeros(nnodes, nnodes)
    for i in 0:n
        for j in 0:n
            V[i+1,j+1] = x[i+1]^j  # Shift the array indices up by one, since Julia␣
↪counts from 1, not 0.
        end
    end
    c = solvelinearsystem(V, y)
    return c
end;
```

```
function polyval(x; coeffs)  # coeffs has to be a keyword argument in order that only␣
↪x gets vectorized
    # Evaluate the polynomial with coefficients in c (as given by polyfit, for␣
↪example).
    n = length(coeffs) - 1
    powers = collect(0:n)  # function collect turns the "AbstractRange" 0:n into an␣
↪array of numbers.
    y = sum(coeffs .* x.^powers)
    return y
end;
```

```
function displaypolynomial(c; sigdigits=6)
    # Note that the function "print" does not end by going to a new line as `println`␣
↪does;
    # this is useful when you want to create a line of output piece-by-piece, as done␣
↪here.

    c = round.(c, sigdigits) # Clean up by rounding to sigdigits significant digits;␣
↪yes, that's vectorization again.
    degree=length(c)-1
    print("P_$(degree)(x) = ")
```

(continues on next page)

(continued from previous page)

```
    print(c[1])
    if degree > 0
        c_1 = c[2]
        if c_1 > 0
            print(" + $(c_1)x")
        elseif coeff < 0
            print(" - $(-c_1) x")
        end
    end
    if degree > 1
        for j in 2:degree
            c_j = c[j+1]
            if c_j > 0
                print(" + $(c_j)x^$j")
            elseif c_j < 0
                print(" - $(-c_j)x^$j")
            end
            # Note: if c_j=0, there is nothing to say.
         end
    end
    println()
end;
```

Example 4.2 ($f(x)$ not a polynomial of degree $\leq n$)

Make an exact fit impossible by using the same function but using only four nodes and reducing the degree of the interpolating P to three: $x_0 = 1$, $x_1 = 2$, $x_2 = 3$ and $x_3 = 4$

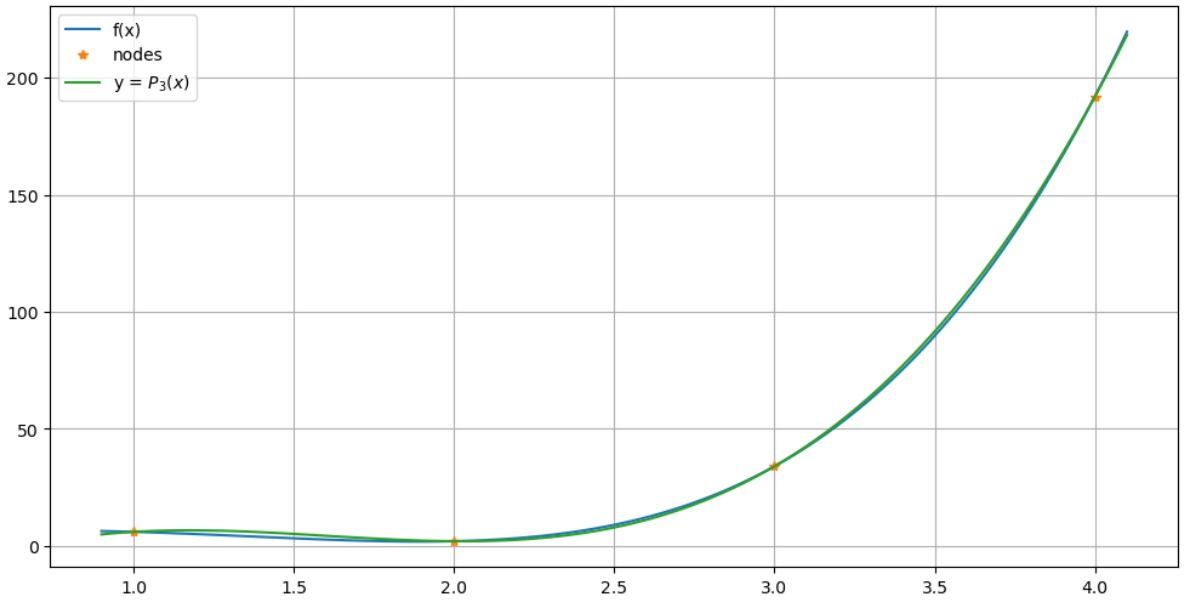
```
xnodes = [1., 2., 3., 4.];
println("The x nodes 'x_i' are $xnodes")
ynodes = f.(xnodes)
println("The y values at the nodes are $ynodes")
c_B = polyfit(xnodes, ynodes);
displaypolynomial(c_B)
```

```
The x nodes 'x_i' are [1.0, 2.0, 3.0, 4.0]
The y values at the nodes are [6.0, 2.0, 34.0, 192.0]
P_3(x) = -44.0 + 107.0x - 72.0x^2 + 15.0x^3
```

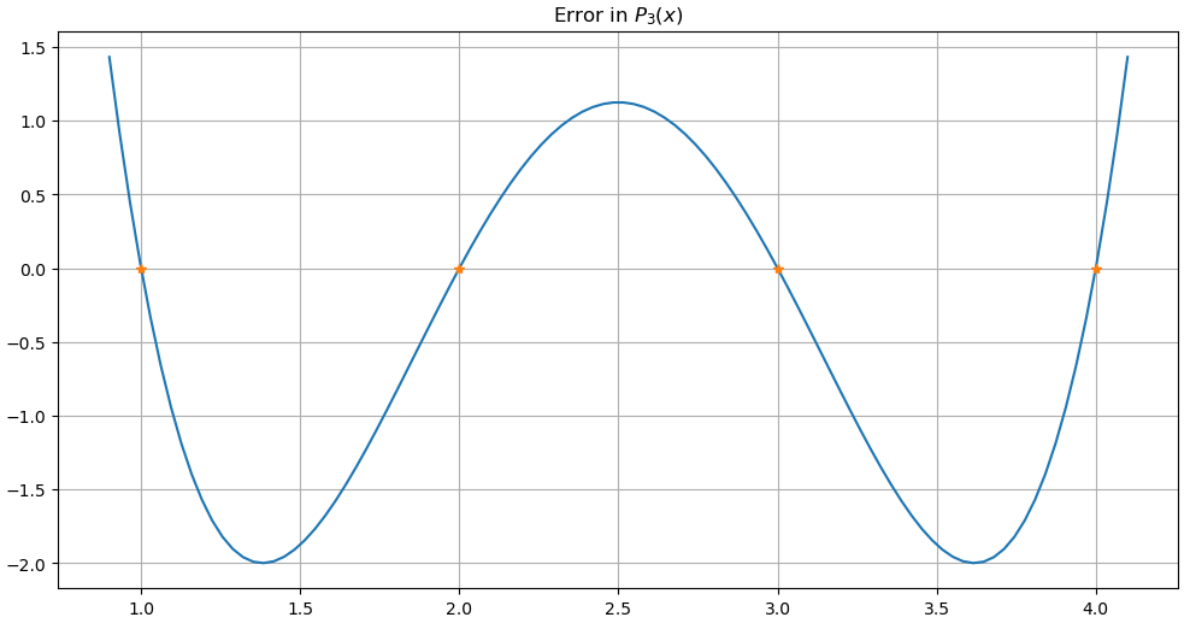
There are several ways to assess the accuracy of this fit; we start graphically, and later consider the maximum and root-mean-square (RMS) errors.

```
xplot = range(0.9, 4.1, 100)  # for graphing: go a bit beyond the nodes
fplot = f.(xplot);
Pplot = polyval.(xplot, coeffs=c_B);
```

```
figure(figsize=[12,6])
plot(xplot, fplot, label="f(x)")
plot(xnodes, ynodes, "*", label="nodes")
plot(xplot, Pplot, label=L"y = $P_3(x)$")
legend()
grid(true);
```




```
Perror = fplot - Pplot
figure(figsize=[12,6])
title(L"Error in $P_3(x)$")
plot(xplot, Perror)
plot(xnodes, zero(xnodes), "*")
grid(true)
```



Example 4.3 ($f(x)$ not a polynomial at all)

$f(x) = e^x$ with five nodes, equally spaced from -1 to 1

```
g(x) = exp(x)
a_g = -1.0
b_g = 1.0;
```

```
degree = 4
n_nodes = degree + 1

xnodes = range(a_g, b_g, n_nodes)
gnodes = g.(xnodes)
c_C = polyfit(xnodes, gnodes)
println("node x values $xnodes")
println("node y values $(round.(gnodes, 4))")
displaypolynomial(c_C, sigdigits=4)
```

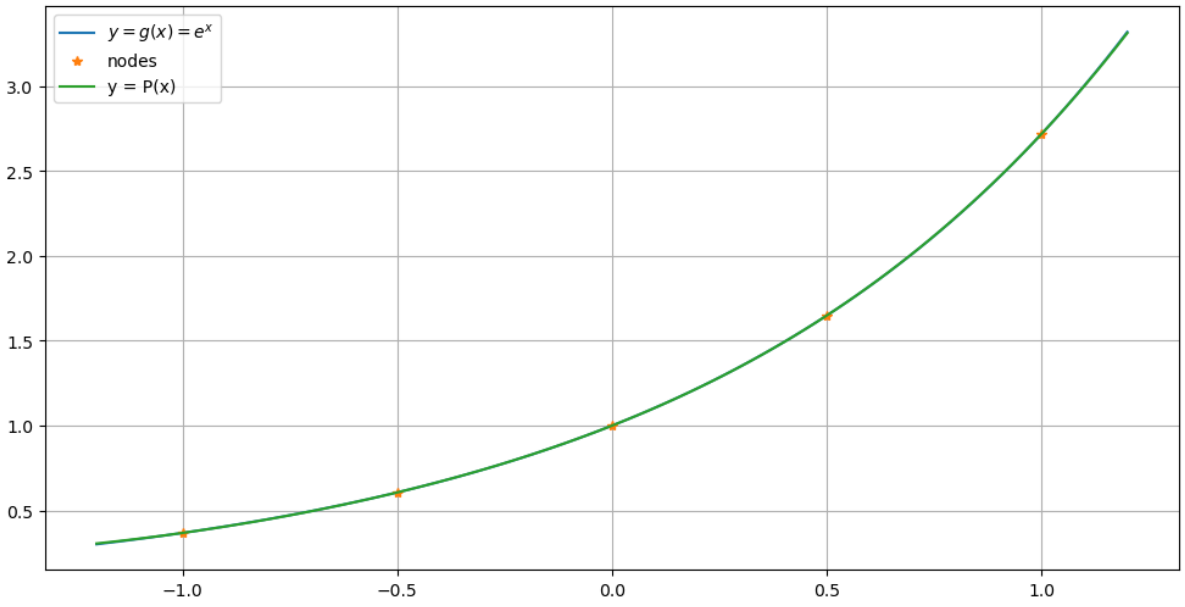
```
node x values -1.0:0.5:1.0
node y values [0.3679, 0.6065, 1.0, 1.649, 2.718]
P_4(x) = 1.0 + 0.9979x + 0.4996x^2 + 0.1773x^3 + 0.04344x^4
```

For comparison, the degree 4 Taylor polynomial for e^x with center 0 is (to the same 6 significant digits)

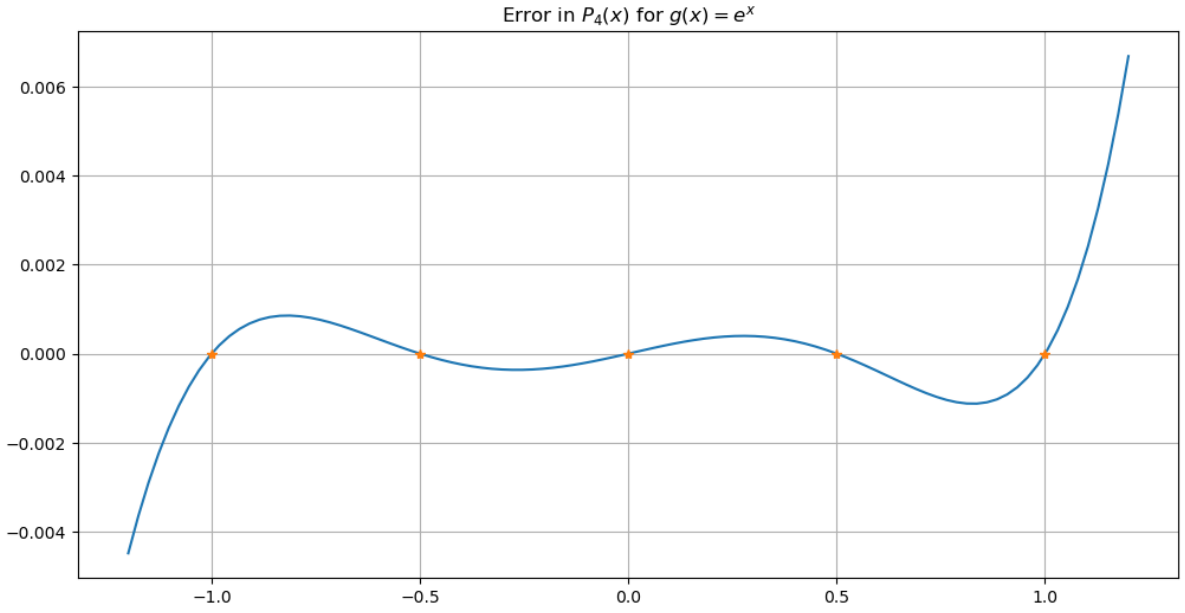
$$1 + x + 0.5x^2 + 0.1667x^3 + 0.04167x^4$$

```
xplot = range(a_g - 0.2, b_g + 0.2, 100)  # Go a bit beyond the nodes in each␣
↪direction
gplot = g.(xplot)
Pplot_g = polyval.(xplot, coeffs=c_C);
```

```
figure(figsize=[12,6])
plot(xplot, gplot, label=L"y = g(x) = e^x")
plot(xnodes, gnodes, "*", label="nodes")
plot(xplot, Pplot_g, label="y = P(x)")
legend()
grid(true)
```



```
Perror_g = gplot - Pplot_g
figure(figsize=[12,6])
degree = 8
title(L"Error in $P_4(x)$ for $g(x) = e^x$")
plot(xplot, Perror_g)
plot(xnodes, zero(xnodes), "*")
grid(true)
```



4.2 Error Formulas for Polynomial Collocation

References:

- Section 3.2.1 *Interpolation error formula* in [Sauer, 2019].
- Section 3.1 *Interpolation and the Lagrange Polynomial* in [Burden *et al.*, 2016].
- Section 4.2 *Errors in Polynomial Interpolation* in [Kincaid and Chenney, 1990]..

4.2.1 Introduction

When a polynomial P_n is given by collocation to $n+1$ points (x_i, y_i), $y_i = f(x_i)$ on the graph of a function f, one can ask how accurate it is as an approximation of f at points x other than the nodes: what is the error $E(x) = f(x) - P(x)$?

As is often the case, the result is motivated by considering the simplest "non-trival case": f a polynomial of degree one too high for an exact fit, so of degree $n+1$. The result is also analogous to the familiar error formula for Taylor polynonial approximations.

```
using PyPlot
```

```
include("NumericalMethods.jl")
using .NumericalMethods: polyfit, polyval
```

4.2.2 Error in $P_n(x)$ collocating a polynomial of degree $n+1$

With $f(x) = c_{n+1}x^{n+1} + c_n x^n + \cdots + c_0$ a polynomial of degree $n+1$ and $P_n(x)$ the unique polynomial of degree at most n that collocates with it at the $n+1$ nodes $x_0, \ldots x_n$, the error

$$E_n(x) = f(x) - P(x)$$

is a polynomial of degree $n+1$ with all its roots known: the nodes x_i. Thus it can be factorized as

$$\begin{aligned} E_n(x) &= C(x-x_0)\cdot(x-x_1)\cdots(x-x_n) && (4.1)\\ &= Cx^{n+1} + \text{lower powers of } x. && (4.2)\end{aligned}$$

On the other hand, the only term of degree x^{n+1} in $f(x)-P(x)$ is the leading term of $f(x)$, $c_{n+1}x^{n+1}$. Thus $C = c_{n+1}$.

It will be convenient to note that the order $n+1$ derivative of f is the constant $f^{(n+1)}D^{n+1}f(x) = (n+1)!c_{n+1}$, so the error can be written as

$$E_n(x) = f(x) - P_n(x) = \frac{D^{n+1}f(x)}{(n+1)!}(x-x_0)\cdots(x-x_n) = \frac{D^{n+1}f(x)}{(n+1)!}\prod_{i=0}^{n}(x-x_i). \tag{4.3}$$

4.2.3 Error in $P_n(x)$ when collocating with a sufficiently differentiable function

Theorem 4.2

For a function f with continuous derivative of order $n+1$ $D^{n+1}f$, the polynomial P_n of degree at most n that fits the points $(x_i, f(x_i))$ $0 \le i \le n$ differs from f by

$$E_n(x) = f(x) - P_n(x) = \frac{D^{n+1}f(\xi_x)}{(n+1)!}\prod_{i=0}^{n}(x-x_i) \tag{4.4}$$

for some value of ξ_x that is amongst the $n+2$ points $x_0, \ldots x_n$ and x.

In particular, when $a \le x_0 < x_1 \cdots < x_n \le b$ and $x \in [a,b]$, then also $\xi_x \in [a,b]$.

Observation 4.1

This is rather similar to the error formula for the Taylor polynomial p_n with center x_0:

$$e_n(x) = f(x) - p_n(x) = \frac{D^{n+1}f(\xi)}{(n+1)!}(x-x_0)^{n+1}, \text{ some } \xi \text{ between } x_0 \text{ and } x. \tag{4.5}$$

This is effectively the limit of Equation (4.4) when all the x_i congeal to x_0.

4.2.4 Error bound with equally spaced nodes is $O(h^{n+1})$, but ...

An important special case is when there is a single parameter h describing the spacing of the nodes; when they are the equally spaced values $x_i = a + ih$, $0 \le i \le n$, so that $x_0 = a$ and $x_n = b$ with $h = \dfrac{b-a}{n}$. Then there is a somewhat more practically usable error bound:

Theorem 4.3

For $x \in [a, b]$ and the above equaly spaced nodes in that interval $[a, b]$,

$$|E_n(x)| = |f(x) - P_n(x)| \leq \frac{M_{n+1}}{n+1} h^{n+1}, = O(h^{n+1}), \tag{4.6}$$

where $M_{n+1} = \max_{x \in [a,b]} |D^{n+1} f(x)|$.

4.2.5 Possible failure of convergence

A major practical problem with this error bound is that is does not in general guarantee convergence $P_n(x) \to f(x)$ as $n \to \infty$ with fixed interval $[a, b]$, because in some cases M_{n+1} grows too fast.

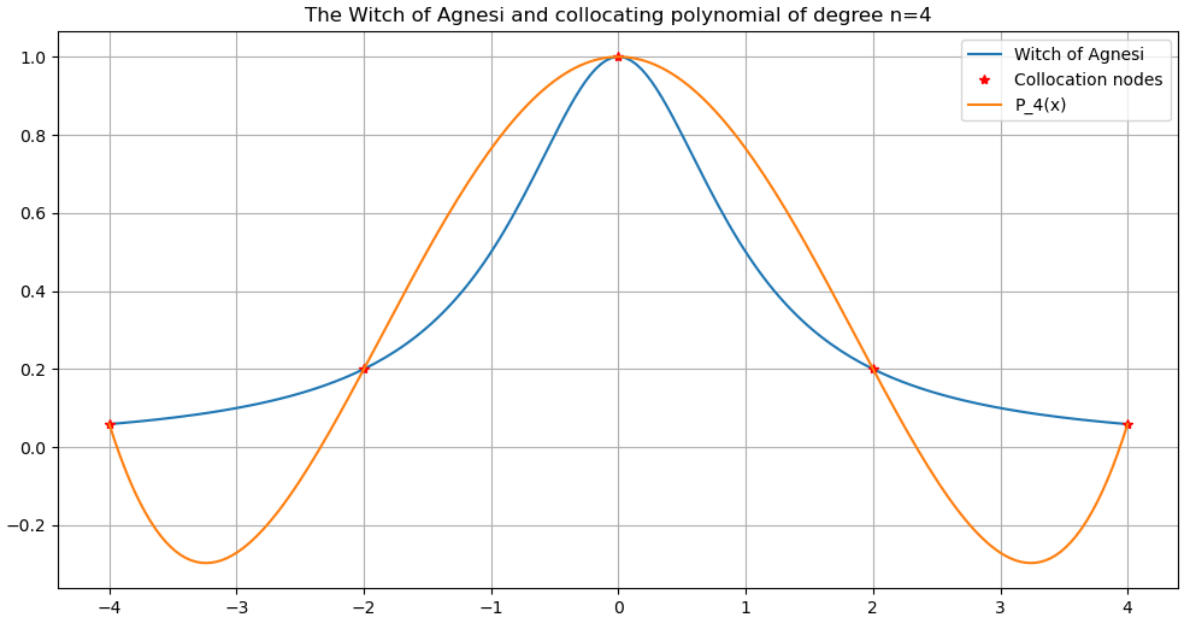
A famous example is the "Witch of Agnesi" (so-called because it was introduced by Maria Agnesi, author of the first textbook on differential and integral calculus).

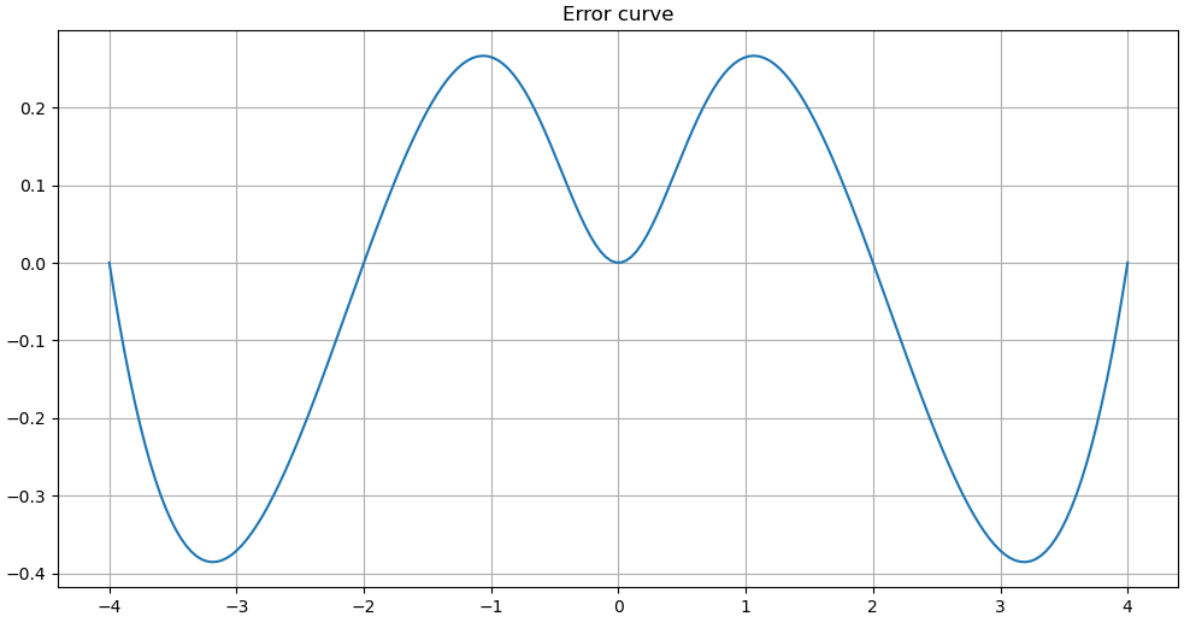
```
agnesi(x) = 1.0/(1 + x^2);
```

```
function graph_agnesi_collocation(a, b, n)
    figure(figsize=[12, 6])
    title("The Witch of Agnesi and collocating polynomial of degree n=$n")
    x = range(a, b, 200)  # Plot 200 points as some fine detail is needed
    y = agnesi.(x)
    plot(x, y, label="Witch of Agnesi")
    xnodes = range(a, b, n+1)
    ynodes = agnesi.(xnodes)
    c = polyfit(xnodes, ynodes)
    P_n = polyval.(x, coeffs=c)
    plot(xnodes, ynodes, "r*", label="Collocation nodes")
    plot(x, P_n, label="P_$n(x)")
    legend()
    grid(true)

    figure(figsize=[12, 6])
    title("Error curve")
    E_n = P_n - y
    plot(x, E_n)
    grid(true)
end;
```

Start with $n = 4$, which seems somewhat well-behaved:

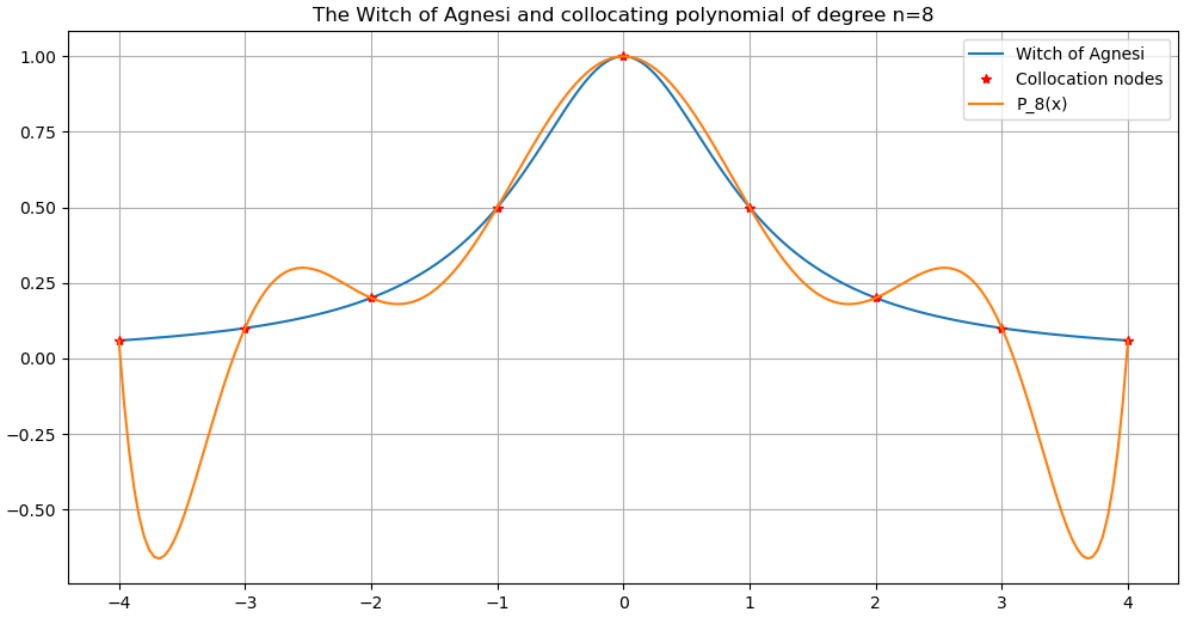
```
graph_agnesi_collocation(-4.0, 4.0, 4)
```

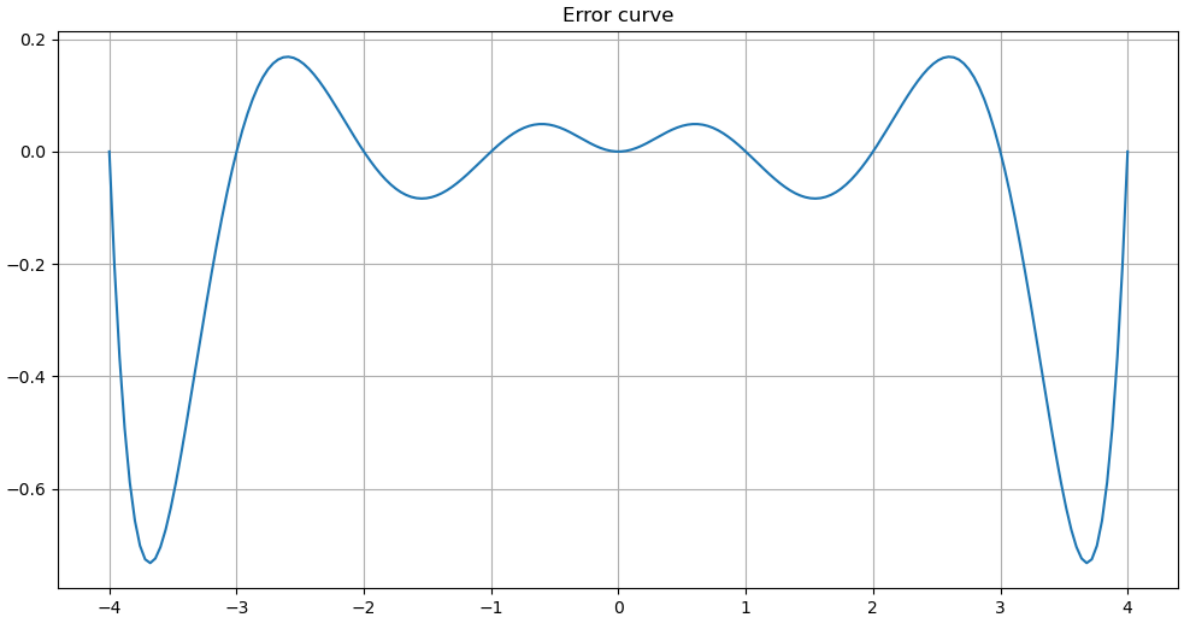




Now increase the number of inervals, doubling each time.

```
graph_agnesi_collocation(-4.0, 4.0, 8)
```





The curve fits better in the central part, but gets worse towards the ends!

One hint as to why is to plot the polynomial factor in the error formula above:

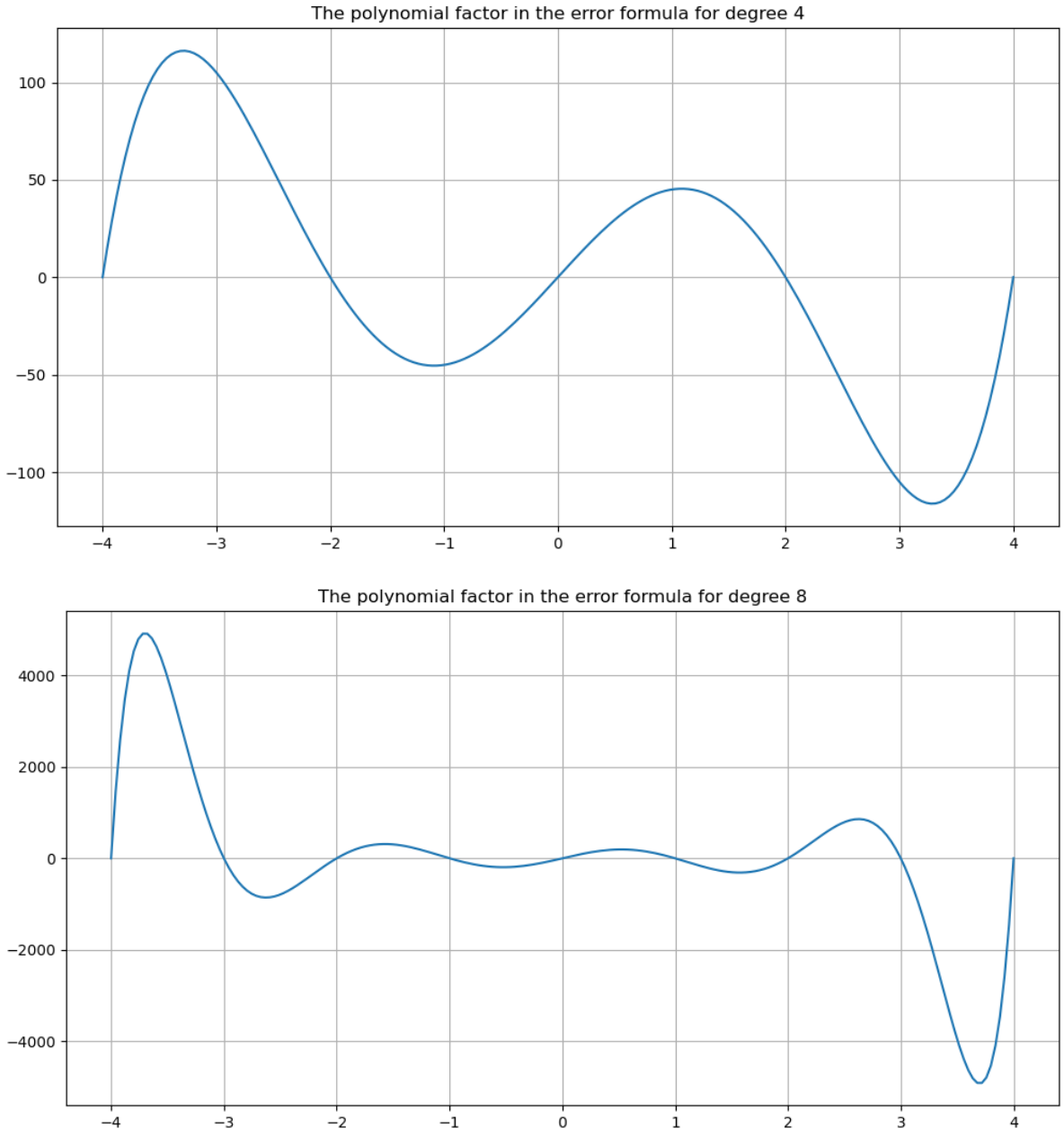
```
function graph_error_formula_polynomial(a, b, n)
    figure(figsize=[12, 6])
    title("The polynomial factor in the error formula for degree $n")
    n_plot_points = 200
    x = range(a, b, n_plot_points)
    x_nodes = range(a, b, n+1)
    polynomial_factor = ones(n_plot_points)
    for x_node in x_nodes
        polynomial_factor .*= (x .- x_node)
    end
```

(continues on next page)

(continued from previous page)

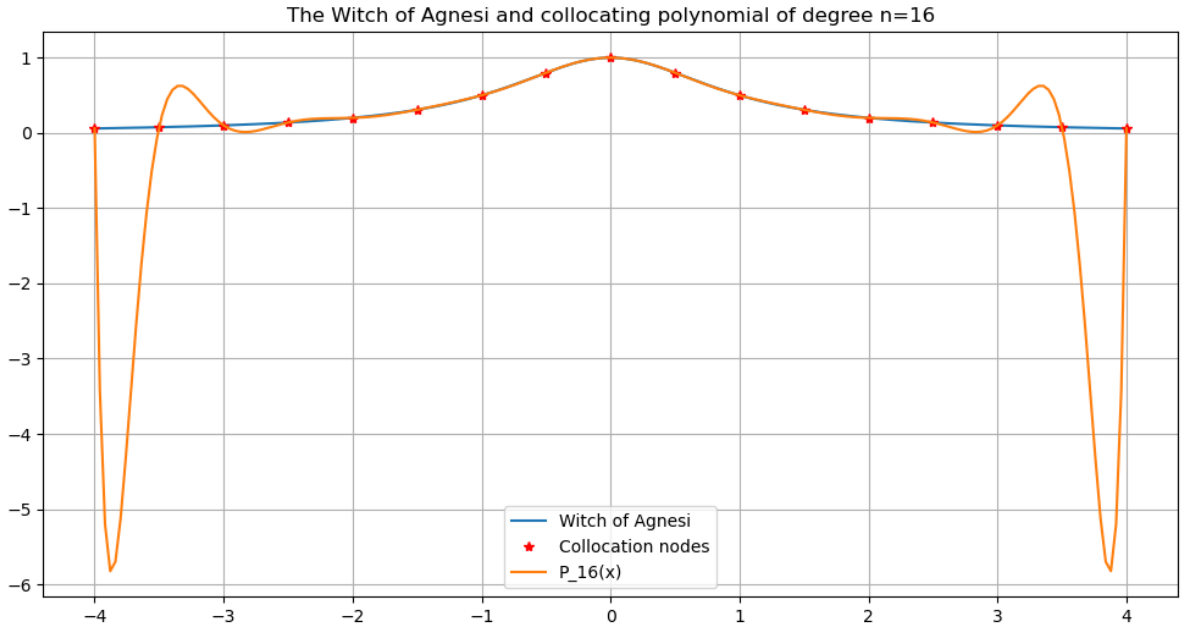
```
    plot(x, polynomial_factor)
    grid(true)
end;
```

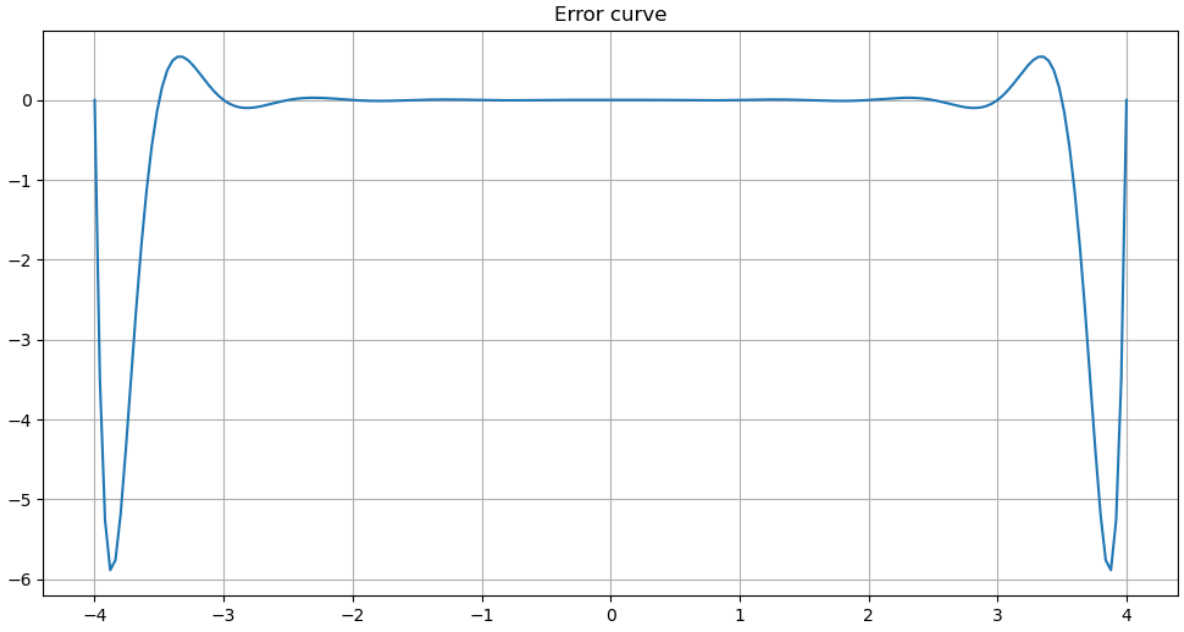
```
graph_error_formula_polynomial(-4.0, 4.0, 4)
graph_error_formula_polynomial(-4.0, 4.0, 8)
```

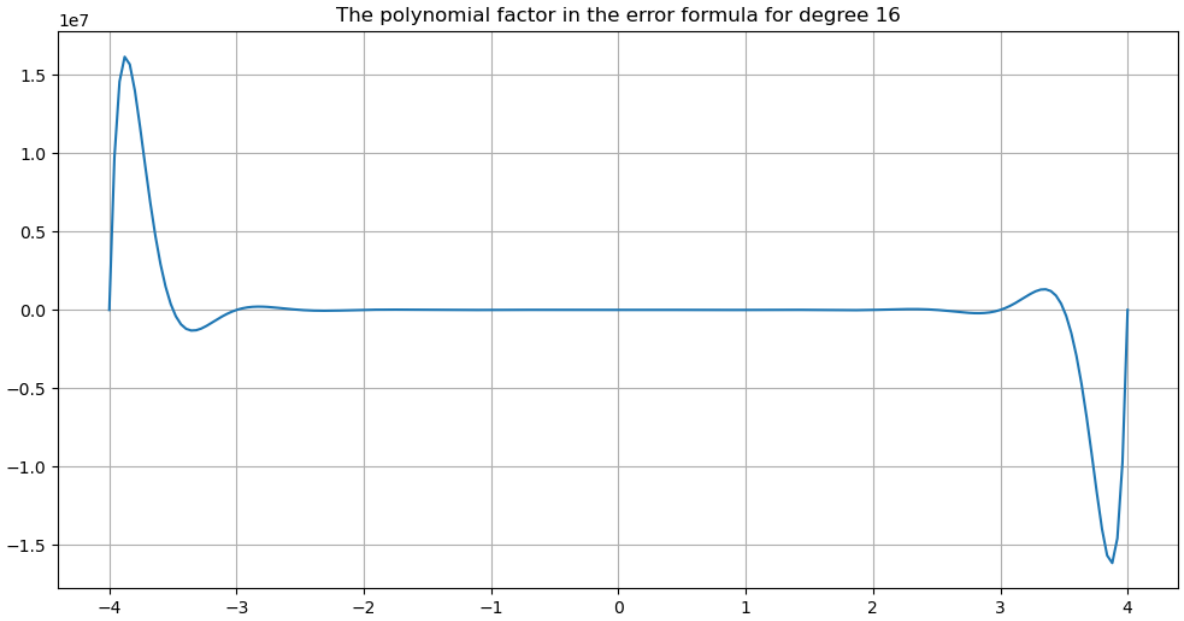


As n increases, it just gets worse:

```
graph_agnesi_collocation(-4.0, 4.0, 16)
graph_error_formula_polynomial(-4.0, 4.0, 16)
```


The Witch of Agnesi and collocating polynomial of degree n=16
Witch of Agnesi
Collocation nodes
P_16(x)


Error curve



4.2.6 Two solutions: piecewise interpolation and least squares approximation

The approach of least squares approximation is introduced in the next section *Least-Squares Fitting to Data*; that can be appropriate when the original data is not exact (due ot measureknet eror in an experiment, for example) so a good approximation at each node can be more appropriate than exact collocation at each but with implausable behavior between the nodes.

When instead exact collocation is sought, piecewise interpolation is typically used. This involves collocation with multiple polynomials of a fixed degree, ech on a prt of the domain. Then for each such polynomial M_{m+1} in the above error formula is independnt of the number N of nodes and with the nodes on interval $[a, b]$ at equal spacing $h = (b-a)/(N-1)$, one has the convergence result

$$|E_m(x) \leq \frac{M_{m+1}}{m+1} h^{m+1} = O(h^{m+1}) = O\left(\frac{1}{N^{m+1}}\right), \to 0 \text{ as } N \to \infty.$$

This only requires that f has a continuous derivatives up to order $m+1$.

The simplest case of this — quite often used in comuter graphics, including `PyPlot.plot` — is to divide the domain into $N-1$ sub-intervals of equal width separated by nodes $x_i = a + ih$, $0 \leq i \leq n$, and then approximate $f(x)$ *linearly* on each sub-interval by using the two surrounding nodes x_i and x_{i+1} determined by having $x_i \leq x \leq x_{i+1}$: this is **piecewise linear interpolation.**

This gives the approximating function $L_N(x)$, and the above error formula, now with $m = 1$, says that the worst absolute error anywhere in the interval $[a, b]$ is

$$|E_2(x)| = |f(x) - L_N(x)| \leq \frac{M_2}{2} h^2, \quad M_2 = \max_{x \in [a,b]} |f''(x)|.$$

Thus for any f that is is twice continuously differentiable the error at each x-value converges to zero as $N \to \infty$. Further, it is *uniform convergence*: the maximum error over all points in the domain goes to zero.

Preview: definite integrals (en route to solving differential equations)

Integrating this piecewise linear approximation over interval $[a, b]$ gives the Compound Trapezoid Rule approximation of $\int_a^b f(x)dx$. As we will soon see, this also has error at worst $O(h^2), = O(1/N^2)$: each doubling of effort reduces errors by a factor of about four.

Also, you might have heard of Simpson's Rule for approximating definite integrals (and anyway, you will soon!): that uses piecewise quadratic interpolation and we will see that this improves the errors to $O(h^4), = O(1/N^4)$: each doubling of effort reduces errors by a factor of about 16.

Remark 4.5 (Computer graphics and smoother approximating curves)

As mentioned, computer graphics often draws graphs from data points this way, most often with either piecewise linear or piecewise cubic ($m = 3$) approximation.

However, this can give sharp "corners" at the nodes, so many modes are needed to make this visually acceptable. That is unavoidable with piecewise linear curves, but for higher degrees there are modifications of this strategy that give smoother curves: piecewise cubics turn out to work very nicely for that, and these are inroduced in the section *Piecewise Polynomial Approximating Functions: Splines and Hermite Cubics*.

4.3 Choosing the collocation points: the Chebyshev method

Co-authored with Stephen Roberts of the Australian National University.

References:

- Section 3.1 *Data and Interpolating Functions* in [Sauer, 2019].
- Section 3.1 *Interpolation and the Lagrange Polynomial* in [Burden *et al.*, 2016].
- Section 4.2 of [Chenney and Kincaid, 2012].
- Section 6.1 of [Kincaid and Chenney, 1990].

In some situations, one can choose the points x_i to use in polynomial collocations (these points are also called the **nodes**) and a natural objective is to minimise the worst case error over some interval $[a, b]$ on which the approximation is to be used. As discussed previously, the best one can do in most cases is to minimise the maximum absolute value of the polynomial $w_{n+1}(x) := \prod_{i=0}^{n}(x - x_i)$ arising in the error formula.

The intuitive idea of using equally spaced points is not optimal as w_{n+1} reaches considerably larger values between the outermost pairs of nodes than elsewhere. Better intuition suggests that moving the nodes a bit closer in these regions of large error will reduce the maximum error there while not increasing it too much elsewhere,and reduce the maximum error. Further it would seem that this strategy is possible so long as the maximum amplitude in some of the intervals between the nodes is larger than others: the endpoints a and b need not be nodes so there are $n + 2$ such intervals.

This suggests the conjecture that the smallest possible maximum amplitude of $w_{n+1}(x)$ on an interval $[a, b]$ will be obtained for a set of nodes such that $|w_{n+1}(x)|$ takes it maximum value $n+2$ times, once in each of the interval separated by the nodes. Indeed this is true, and the nodes achieving this result are the so-called **Chebyshev points**, given by the simple formula

$$\frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{2i+1}{2n+2}\pi\right), \qquad 0 \le i \le n \tag{4.7}$$

To understand this result, consider the case where the interval of interest is $[-1, 1]$, so that these special nodes are $\cos\left(\frac{2i+1}{2n+2}\pi\right)$ The general case then follows by using the change of variables $x = (a + b)/2 + t(b - a)/2$. The reason that this works is that these are the roots of the function

$$T_{n+1}(x) := \cos((n + 1)\cos^{-1} x) \tag{4.8}$$

which turns out to be a polynomial of degree $n+1$ that takes its maximum absolute value of 1 at the $n+2$ points $\cos\left(\frac{i}{n+1}\pi\right), 0 \le i \le n+1$.

There are a number of claims here: most are simple consequences of the definition and what is known about the roots and extreme values of cosine. The one surprising fact is that $T_n(x)$ is a polynomial of degree n, known as a **Chebyshev polynomial**. The notation comes from an old transliteration, Tchebychev, of this Russian name.

This can be checked by induction. The first few cases are easy to check: $T_0(x) = 1$, $T_1(x) = x$ and $T_2(x) = \cos 2\theta = 2\cos^2\theta - 1 = 2x^2 - 1$. In general, let $\theta = \cos^{-1} x$ so that $\cos\theta = x$. Then trigonometric identities give

$$\begin{aligned} T_{n+1}(x) &= \cos(n+1)\theta \\ &= \cos n\theta\cos\theta - \sin n\theta\sin\theta \\ &= T_n(x)x - \sin n\theta\sin\theta \end{aligned}$$

and similarly

$$\begin{aligned} T_{n-1}(x) &= \cos(n-1)\theta \\ &= \cos n\theta\cos\theta + \sin n\theta\sin\theta \\ &= T_n(x)x + \sin n\theta\sin\theta \end{aligned}$$

Thus $T_{n+1}(x) + T_{n-1}(x) = 2xT_n(x)$ or

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \tag{4.9}$$

Since T_0 and T_1 are known to be polynomials, the same follows for each successive n from this formula. The induction also shows that

$$T_n(x) = 2^{n-1}x^n + \quad \text{terms involving lower powers of } x$$

so in particular the degree is n.

With this information, the error formula can be written in a special form. Firstly w_{n+1} is then a polynomial of degree $n+1$ with the same roots as T_{n+1}, so is a multiple of the latter function. Secondly, the leading coefficient of w_{n+1} is 1, compared to 2^{n+1} for the Chebyshev polynomial, so $w_{n+1} = T_{n+1}/2^n$. Finally, the maximum of w_{n+1} is seen to be $1/2^n$ and we have the result that

---

Theorem 4.4

When a polynomial approximation $p(x)$ to a function $f(x)$ on the interval $[-1,1]$ is constructed by collocation at the roots of T_{n+1}, the error is bounded by

$$|f(x) - p(x)| \le \frac{1}{2^n(n+1)!}\max_{-1\le t\le 1}|f^{(n+1)}(t)|$$

When the interval is $[a,b]$ and the collocation points are the appropriately rescaled Chebychev points as given in (4.7).

$$|f(x) - p(x)| \le \frac{(b-a)^{n+1}}{2^{2n+1}(n+1)!}\max_{a\le x\le b}|f^{(n+1)}(x)|$$

This method works well in many cases. Further, it is known that any continuous on any interval $[a,b]$ can be approximated arbitrarily well by polynomials, in the sense that the maximum error over the whole interval can be made as small as one likes [this is the *Weierstrass Approximation Theorem*]. However, collocation at these Chebyshev nodes will not work for all continuous functions: indeed no choice of points will work for all cases, as is made precise in theorem 6 on page 288 of Kincaid&Chenney. One way to understand the problem is that the error bound relies on derivatives of ever higher order, so does not even apply to some continuous functions.

This suggests a new strategy: break the interval $[a,b]$ into smaller interval, approximate on each interval by a polynomial of some small degree, and join these polynomials together. Hopefully, the errors will only depend on a few derivatives, and so will be more controllable, while using enough nodes and small enough intervals will allow the errors to be made as small as desired. This fruitful idea is dealt with next.

4.4 Piecewise Polynomial Approximating Functions: Splines and Hermite Cubics

Co-authored with Stephen Roberts of the Australian National University.

References:

- Sections 3.6, 6.2, 6.4 of [Kincaid and Chenney, 1990].
- Section 3.4 *Cubic Splines* in [Sauer, 2019].
- Sections 3.5 *Cubic Spline Interpolation* and 3.4 *Hermite Interpolation* of [Burden *et al.*, 2016].
- Sections 6.1 and 6.2 of Chapter 6 *Spline Functions* [Chenney and Kincaid, 2012].

The idea of approximating a function (or interpolating between a set of data points) with a function that is piecewise polynomial takes its simplest form using *continuous piecewise linear functions*. Indeed, this is the method most commonly used to produce a graph from a large set of data points: for example, the command `plot` from `matplotlib.pyplot` (for Python) or `PyPlot` (for Julia) does it.

The idea is simply to draw straight lines between each successive data point. It is worth analysing this simple method before considering more accurate approaches.

Consider a set of $n+1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ again, this time requiring the x values to be in increasing order. Then define the linear functions

$$L_i(x) = y_i + (x - x_i)\frac{y_{i+1} - y_i}{x_{i+1} - x_i}, \quad x_i \le x \le x_{i+1}, \quad 0 \le i < n$$

These can be joined together into a continuous function

$$L(x) = L_i(x) \text{ for } x_i \le x \le x_{i+1}$$

with the values $L(x_i) = y_i$ at all nodes, so that the definition is consistent at the points where the domains join, also guaranteeing continuity.

4.4.1 Spline Interpolation

Reference: Section 6.4 of [Kincaid and Chenney, 1990].

If a piecewise linear approximation is approximated that passes through a given set of $n+1$ points or **knots**

$$(t_0, y_0), \ldots, (t_n, y_n)$$

and is linear in each of the n interval between them, the "smoothest" curve that one can get is the continuous one given by using linear interpolation between each consecutive pair of points. Less smooth functions are possible, for example the piecewise constant approximation where $L(x) = y_i$ for $x_i \le x < x_{i+1}$.

The general strategy of spline interpolation is to approximate with a piecewise polynomial function, with some fixed degree k for the polynomials, and is as smooth as possible at the joins between different polynomials. Smoothness is measured by the number of continuous derivatives that the function has, which is only in question at the knots of course.

The traditional and most important case is that of **cubic splines interpolants**, which have the form

$$S(x) = S_i(x), \quad t_i \le x \le t_{i+1}, \quad 0 \le i < n$$

where each $S_i(x)$ is a cubic and the interpolation conditions are

$$S_i(t_i) = y_i, \quad S_i(t_{i+1}) = y_{i+1}, \qquad 0 \le i < n$$

These conditions automatically give continuity, but leave many degrees of freedom to impose more smoothness. Each cubic is described by four coefficients and so there are $4n$ in all, and the interpolation conditions give only $2n$ conditions. There are $n-1$ knots where different cubics join, so requiring S to have continuous first and second derivatives imposes $2(n-1)$ further conditions for a total of $4n-2$. This is the best smoothness possible without $S(x)$ becoming a single cubic, and leaves two degrees of freedom. These will be dealt with later, but one approach is imposing zero second derivatives at each end of the interval.

Thus we have the equations

$$S'_{i-1}(t_i) = S'(t_i)$$

and

$$S''_{i-1}(t_i) = S''(t_i),$$

$1 \le i \le n-1$.

The brute force method would be to write something like

$$S_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$$

which would leave to a set of $4n$ simultaneous linear equations for these $4n$ unknowns once the two missing conditions have been chosen.

This could then be solved numerically, but the size and cost of the problem can be considerably reduced, to a tridiagonal system of $n-1$ equations.

Start by considering the second derivative of $S(x)$, which must be continuous and piecewise linear. Its values at the knots can be called $x_i = S''_i(t_i)$ and the lengths of the interval called $h_i = x_{i+1} - x_i$ so that

$$S''_i(x) = \frac{z_i}{h_i}(t_{i+1} - x) + \frac{z_{i+1}}{h_i}(x - t_i)$$

Integrating twice,

$$S_i(x) = \frac{z_i}{6h_i}(t_{i+1} - x)^3 + \frac{z_{i+1}}{6h_i}(x - t_i)^3 + C_i(t_{i+1} - x) + D_i(x - t_i)$$

The interpolation conditions then determine C_i and D_i:

$$S_i(x) = \frac{z_i}{6h_i}(t_{i+1} - x)^3 + \frac{z_{i+1}}{6h_i}(x - t_i)^3 + \left(\frac{y_i}{h_i} - \frac{z_i h_i}{6}\right)(t_{i+1} - x) + \left(\frac{y_{i+1}}{h_i} - \frac{z_{i+1} h_i}{6}\right)(x - t_i) \tag{4.10}$$

In effect, three quarters of the equations have been solved explicitly, leaving only the z_i to be determined using the remaining condition of the continuity of $S'(x)$.

Differentiating the above expression and evaluating at the appropriate points gives the expressions

$$S'_i(t_i) = -\frac{h_i}{3} z_i - \frac{h_i}{6} z_{i+1} - \frac{y_i}{h_i} + \frac{y_{i+1}}{h_i} \tag{4.11}$$

$$S'_{i-1}(t_i) = -\frac{h_{i-1}}{6} z_{i-1} + \frac{h_{i-1}}{3} z_i - \frac{y_{i-1}}{h_{i-1}} + \frac{y_i}{h_{i-1}} \tag{4.12}$$

Equating these at the internal knots (and simplifying a bit) gives

$$h_{i-1} z_{i-1} + 2(h_i + h_{i-1}) z_i + h_i z_{i+1} = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}) \tag{4.13}$$

These are $n-1$ linear equations in the $n+1$ unknowns z_i, so various different cubic spline interpolants can be constructed by adding two extra conditions in the form of two more linear equations. The traditional way is the one mentioned above: require the second derivative to vanish at the two endpoints. That is

$$S''(t_0) = S''(t_n) = 0$$

which gives a **natural spline**.

In terms of the z_i this gives the trivial equations $z_0 = z_n = 0$. Thus these two unknowns can be eliminated from the equations in (4.13) giving the following tridiagonal system:

$$\begin{bmatrix} 2(h_0+h_1) & h_1 & & \\ h_1 & 2(h_1+h_2) & \ddots & \\ & \ddots & \ddots & h_{n-2} \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} 6((y_2-y_1)/h_1 - (y_1-y_0)/h_0) \\ 6((y_3-y_2)/h_2 - (y_2-y_1)/h_1) \\ \vdots \\ 6((y_n-y_{n-1})/h_{n-1} - (y_{n-1}-y_{n-2})/h_{n-2})) \end{bmatrix}$$

Solving tridiagonal systems is far more efficient if it can be done without pivoting by the method seen earlier, and this is a good method if the matrix is diagonally dominant.

That is true here: recalling that the t_i are in increasing order, each h_i is positive, so each diagonal element is at least twice the sum of the absolute values of all other elements in the same row. This result incidentally also shows that the equations have a unique solution, which means that the natural cubic spline exists and is determined uniquely by the data, requiring about $O(n)$ operations.

Evaluation of $S(x)$ is then done by finding the i such that $t_i \leq x < t_{i+1}$ and then evaluating the appropriate case in (4.10).

4.4.2 Clamped Splines and Error Bounds

Reference: Section 3.6 of [Kincaid and Chenney, 1990].

Though the algorithm for natural cubic spline interpolation is widely available in software [TO DO: add Numpy/Julia references] it is worth knowing the details. In particular, it is then easy to consider minor changes, like different conditions at the end points.

Recall that the **natural** or **free** spline has the boundary conditions

$$S''(t_0) = S''(t_n) = 0 \tag{4.14}$$

When the spline is to be used to approximate a function $f(x)$ one useful alternative choice of boundary conditions is to specify the derivative of the spline function to match that of f at the endpoints:

$$S'(t_0) = f'(t_0), \qquad S'(t_n) = f'(t_n) \tag{4.15}$$

This is called a **clamped spline**.

When the function f or its derivatives are not known, they can be approximated from the data itself. Thus a generalisation of the last condition is

$$S'(t_0) = d_0, \qquad S'(t_n) = d_n \tag{4.16}$$

for some approximations of the derivatives.

The subject of approximating a function's derivative using a finite collection of values of the function will be taken up soon in more detail, but the simplest approach is to use the difference quotient from the definition of the derivative. This gives

$$\begin{aligned} d_0 &:= \frac{y_1 - y_0}{h_0} = \frac{f(t_1) - f(t_0)}{t_1 - t_0} \\ d_n &:= \frac{y_n - y_{n-1}}{h_{n-1}} = \frac{f(t_n) - f(t_{n-1})}{t_n - t_{n-1}} \end{aligned}$$

as one choice for the approximate derivatives.

The cubic splines given by using some such approximate derivatives will be called **modified clamped spline**.

These new conditions require a revision of the previous algorithm, but one benefit is that there is a better result guaranteeing the accuracy of the approximation.

To derive the new equations and algorithm for [modified] clamped splines return to the equations (4.11) and (4.12) used to derive the equation (4.13) that defines the tridiagonal system of $n-1$ equations for the second derivatives $z_1, \ldots, z_{n-1}$.

Instead of eliminating the two unknowns z_0 and z_n, we can add two more linear equations by using those equations (4.11) and (4.12) respectively at t_0 and t_n [i.e. for $i = 0$ and $i = n$] and equating to the values to whatever d_0 and d_n we are using:

$$\begin{aligned}
S'(t_0) &= S_0'(t_0) \\
&= -\frac{h_0}{3}z_0 - \frac{h_0}{6}z_1 - \frac{y_0}{h_0} + \frac{y_1}{h_0} \\
&= d_0 \\
S'(t_n) &= S_{n-1}'(t_n) \\
&= \frac{h_{n-1}}{6}z_{n-1} + \frac{h_{n-1}}{3}z_n - \frac{y_{n-1}}{h_{n-1}} + \frac{y_n}{h_{n-1}} \\
&= d_n
\end{aligned}$$

In conjunction with equation (4.13), this gives the new tridiagonal system

$$\begin{bmatrix}
2h_0 & h_0 & & & & \\
h_0 & 2(h_0+h_1) & h_1 & & & \\
& \ddots & \ddots & \ddots & & \\
& & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\
& & & h_{n-1} & 2h_{n-1}
\end{bmatrix}
\begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n-1} \\ z_n \end{bmatrix}$$

$$= \begin{bmatrix}
6\left((y_1-y_0)/h_0 - d_0\right) \\
6((y_2-y_1)/h_1 - (y_1-y_0)/h_1) \\
\vdots \\
6((y_n-y_{n-1})/h_{n-1} - (y_{n-1}-y_{n-2})/h_{n-2}) \\
6\left(d_n - (y_n-y_{n-1})/h_{n-1}\right)
\end{bmatrix}$$

As in the case of the tridiagonal system for natural splines, the rows of the matrix also satisfy the condition of diagonal dominance, so again this system has a unique solution that can be computed accurately with only $O(n)$ operations and no pivoting.

4.4.3 Error Bounds for Approximation with Clamped Splines

If the exact derivatives mentioned in (4.15) are available, the errors are bounded as follows

---

Theorem 4.5

Suppose that $f(x)$ is four times continuously differentiable on the interval $[a, b]$, with $\max_{a\le x\le b} |f^{(4)}(x)| \le M$. Then the clamped cubic spline approximation $S(x)$ using the points $a = t_0 < t_1 < \cdots < t_n = b$ and $y_i = f(t_i)$ satisfies

$$|f(x) - S(x)| \le M\frac{5}{384}\left(\max_{0\le i\le n-1} h_i\right)^4$$

for every point $x \in [a, b]$.

There is also an error bound of the same "fourth order" form for the natural cubic spline- that is, one of the form of some constant depending on f times the fourth power of $\max_{0\le i\le n-1} h_i$. However it is far more complicated to describe: see page 138 of Burden and Faires for more comments on this.

When we have studied methods for approximating derivatives, it will be possible to establish error bounds for modified clamped splines with various approximations for the derivatives at the endpoints, so that they depend only on the values of f at the knots. With care, these more practical approximations can also be made fourth order accurate.

4.4.4 Hermite Cubic Approximation

Reference: Section 6.2 of [Kincaid and Chenney, 1990].

Hermite interpolation in general consists in finding a polynomial $H(x)$ to approximate a function $f(x)$ by giving a set of points $t_0, \dots, t_n$ and requiring that the value of the polynomial and its first few derivatives match that of the original function.

The simplest case that is not simply polynomial interpolation or Taylor polynomial approximation is where there are two points, and first derivatives are required to match. This gives four conditions

$$\begin{aligned} H(t_0) &= f(t_0) = y_0, H'(t_0) = f'(t_0) = y_0' \\ H(t_1) &= f(t_1) = y_1, H'(t_1) = f'(t_1) = y_0' \end{aligned}$$

and counting constants suggests that there should be a unique cubic h with these properties. From now on, I will use "cubic" to include the degenerate cases that are actually quadratics and so on.

To determine this cubic it is convenient to put it in the form

$$H(x) = a + b(x - t_0) + (x - t_0)^2[c + d(x - t_{i+1})]$$

and let $h = t_1 - t_0$: then applying the four conditions in turn gives

$$\begin{aligned} a &= y_0, \qquad b = y_0' \\ c &= \frac{y_1 - y_0}{h^2} - \frac{y_0'}{h}, \qquad d = \frac{y_1' - y_0'}{3h^2} - \frac{2(y_1 - y_0)}{3h^3} \end{aligned}$$

With more points, one could look for higher order polynomials, but it is useful in some cases to construct a piecewise cubic approximation, with the cubic between each consecutive pair of nodes determined only by the value of the function and its derivative at those nodes. Thus the piecewise Hermite cubic approximation to f on the interval $[a, b]$ for the points $a = t_0 < t_1 < \cdots < t_n$ is given by a set of n cubics

$$H(x) = H_i(x) = a_i + b_i(x - t_i) + (x - t_i)^2[c_i + d_i(x - t_{i+1})], \quad t_i \le x < t_{i+1}$$

with

$$\begin{aligned} a_i &= y_i, \qquad b_i = y_i' \\ c_i &= \frac{y_{i+1} - y_i}{h_i^2} - \frac{y_i'}{h_i} \\ d_i &= \frac{y_{i+1}' - y_i'}{3h_i^2} - \frac{2(y_{i+1} - y_i)}{3h_i^3} \end{aligned}$$

where $y_i := f(t_i)$, $y_i' := f'(t_i)$ and $h_i := t_{i+1} - t_i$. Most often, the points are equally spaced so that

$$h_i - h := (b - a)/n.$$

There is an error formula for this (which is also an error formula for a clamped spline in the case $n = 1$)

Theorem 4.6

For $x \in [t_t, t_{i+1}]$

$$f(x) - H(x) = \frac{f^{(4)}(\xi)}{4!}[(x - t_i)(x - t_{i+1})]^2$$

where $\xi \in [t_t, t_{i+1}]$ Thus if $|f^{(4)}(x)| \leq M_i$ for $x \in [t_t, t_{i+1}]$,

$$|f(x) - H(x)| \leq \frac{M_i}{384} h_i^4$$

Proof. See page 311 of [Kincaid and Chenney, 1990].

Thus the accuracy is about as good as for clamped splines: the trade off is that the Hermite approximation is less smooth (only one continuous derivative at the nodes), but the error is "localised". That is, if the fourth derivative of f is large or non-existent in one interval, the accuracy of the Hermite approximation only suffers in that interval, not over the whole domain.

However this comparison is a bit unfair, as the Hermite approximation uses the extra information about the derivatives of f. This is also often impractical: either the derivatives are not known, or there is no known function f but only a collection of values y_i.

To overcome this problem, the derivatives needed in the above formulas can be approximated from the y_i as was done for modified clamped splines. To do this properly, it is worth taking a thorough look at methods for approximating derivatives and bounding the accuracy of such approximations.

4.5 Least-Squares Fitting to Data

References:

- Chapter 4 *Least Squares* of [Sauer, 2019], sections 1 and 2.
- Section 8.1 *Discrete Least Squares Approximation* of [Burden *et al.*, 2016].

We have seen that when trying to fit a curve to a large collection of data points, fitting a single polynomial to all of them can be a bad approach. This is even more so if the data itself is inaccurate, due for example to measurement error.

Thus an important approach is to find a function of some simple form that is close to the given points but not necesarily fitting them exactly: given N points

$$(x_i, y_i),\ 1 \leq i \leq N$$

we seek a function $f(x)$ so that the errors at each point,

$$e_i = y_i - f(x_i),$$

are "small" overall, in some sense.

Two important choices for the function $f(x)$ are

(a) polynomials of low degree, and

(b) periodic sinusidal functions;

we will start with the simplest case of fitting a straight line.

4.5.1 Measuring "goodness of fit": several options

The first decision to be made is how to measure the overall error in the fit, since the error is now a vector of values $e = \{e_i\}$, not a single number. Two approaches are widely used:

- *Min-Max*: minimize the maximum of the absolute errors at each point, $\|e\|_{max}$ or $\|e\|_\infty, = \max_{1 \le i \le n} |e_i|$
- *Least Squares*: Minimize the sum of the squares of the errors, $\sum_1^n e_i^2$

4.5.2 What doesn't work

Another seemingly natural approach is:

- Minimize the sum of the absolute errors, $\|e\|_1 = \sum_1^n |e_i|$

but this often fails completely. In the following example, all three lines minimize this measure of error, along with infinitely many others: any line that passes below half of the points and above the other half.

```
using PyPlot
using LinearAlgebra
using Random
```

```
include("NumericalMethods.jl")
using .NumericalMethods: solvelinearsystem, polyval
```

Remark 4.6

This original version of `polyfit` seen at *Functions for computing the coefficients and evaluating the polynomials* in the section *Polynomial Collocation (Interpolation/Extrapolation) and Approximation* is not used here; instead a variant is defined below that is closer to the eponymous functions in Matlab and in Python package Numpy. Julia allows the reuse a function's name so long as the different functions are distinguished by different input parameters, a Julia feature known as *multiple dispatch*.

However, when one version comes from a module, we must use `import` rather than `using`. (All other cases of `using .NumericalMethods` in this book could also be `import .NumericalMethods`.)

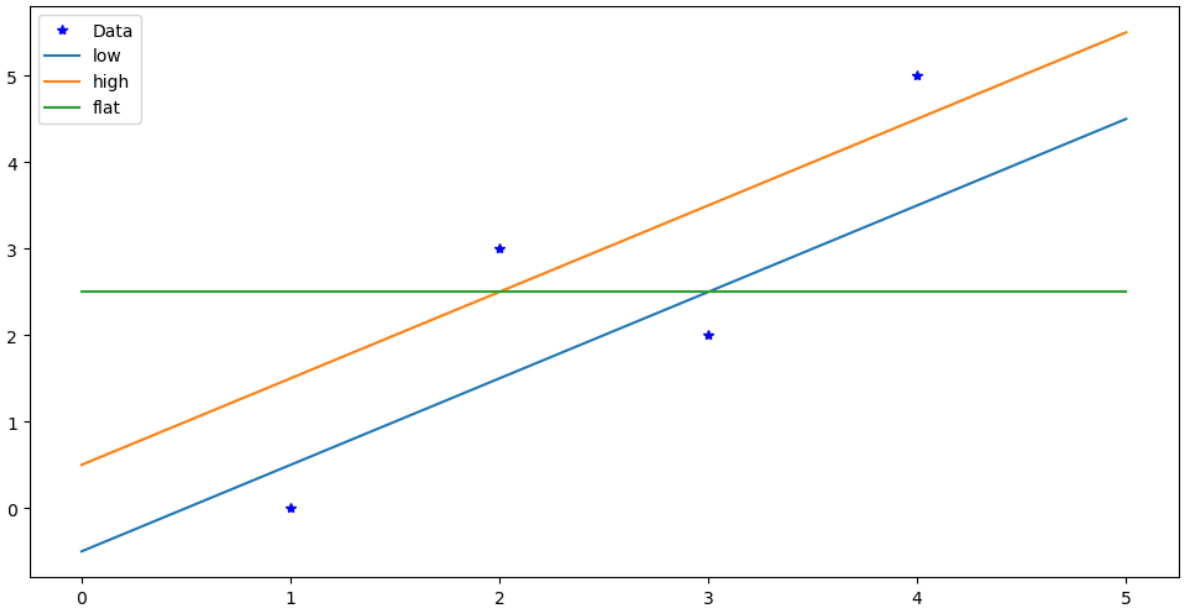
```
import .NumericalMethods: polyfit
```

```
xdata = [1., 2., 3., 4.]
ydata = [0., 3., 2., 5.]
figure(figsize=[12,6])

plot(xdata, ydata, "b*", label="Data")
nplotpoints = 100
xplot = range(0., 5.0, nplotpoints)
ylow = xplot .- 0.5
yhigh = xplot .+ 0.5
yflat = 2.5*ones(nplotpoints)
plot(xplot, ylow, label="low")
plot(xplot, yhigh, label="high")
```

(continues on next page)

(continued from previous page)

```
plot(xplot, yflat, label="flat")
legend(loc="best");
```



The Min-Max method is important and useful, but computationally difficult. One hint is the presence of absolute values in the formula, which get in the way of using calculus to get equations for the minimum.

Thus the easiest and most common approach is **Least Squares**, or equivalently, minimizing the root-mean-square error, which is just the Euclidean length $\|e\|_2$ of the error vector e. That "geometrical" interpretation of the goal can be useful. So we start with that.

4.5.3 Linear least squares

The simplest approach is to seek the straight line $y = f(x) = c_0 + c_1 x$ that minimizes the total square sum error,

$$E(c_0, c_1) = \sum_i e_i^2 = \sum_i (c_0 + c_1 x_i - y_i)^2.$$

Note well that the unknowns here are just the two values c_0 and c_1, and E is s fairly simple polynomial function of them. The minimum error must occur at a critical point of this function, where both partial derivatives are zero:

$$\frac{\partial E}{\partial c_0} = 2 \sum_i (c_0 + c_1 x_i - y_i) = 0,$$

$$\frac{\partial E}{\partial c_1} = 2 \sum_i (c_0 + c_1 x_i - y_i) x_i = 0.$$

These are just simultaneous linear equations, which is the secret of why the least squares approach is so much easier than any alternative. The equations are:

$$\begin{bmatrix} \sum_i 1 & \sum_i x_i \\ \sum_i x_i & \sum_i x_i^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} \sum_i y_i \\ \sum_i x_i y_i \end{bmatrix}$$

where of course $\sum_i 1$ is just N.

It will help later to introduce the notation

$$m_j = \sum_i x_i^j, \quad p_j = \sum_i x_i^j y_i$$

so that the equations are

$$Mc = p$$

with

$$M = \begin{bmatrix} m_0 & m_1 \\ m_1 & m_2 \end{bmatrix}, p = \begin{bmatrix} p_0 \\ p_1 \end{bmatrix}, c = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}.$$

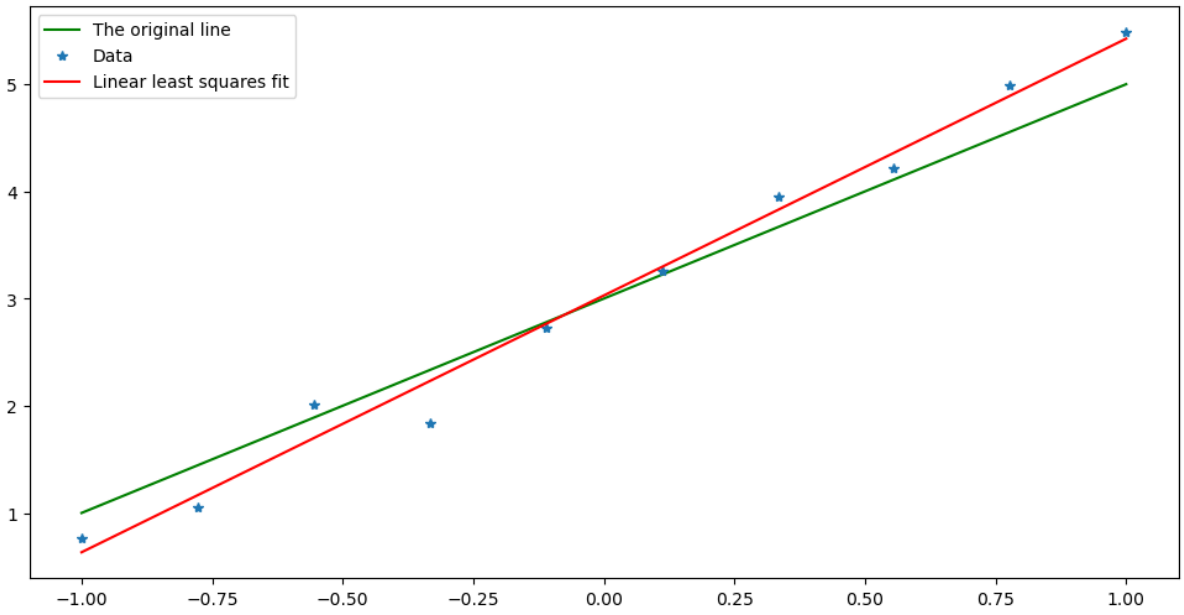
Remark 4.7 (Alternative geometrical derivation)

In the next section *Least-squares Fitting to Data: Appendix on The Geometrical Approach*, another way to derive this result is given, using geometry and linear algebra instead of calculus.

```
function linefit(x, y)
    m0 = length(x)
    m1 = sum(x)
    m2 = sum(x.^2)
    M = [ m0 m1 ; m1 m2 ]
    p = [ sum(y); sum(x.*y) ]
    return solvelinearsystem(M, p)
end;
```

```
N = 10
xmin = -1.0
xmax = 1.0
x = range(xmin, xmax, N)
# Emulate a straight line with measurement errors:
# random(N) gives N values uniformly distributed in the range [0,1], and so with mean␣
↪0.5.
# Thus subtracting 1/2 simulates more symmetric "errors", of mean zero.
yline = 3 .+ 2x
y = yline + (rand(N) .- 0.5)

figure(figsize=[12,6])
plot(x, yline, "g", label="The original line")
plot(x, y, "*", label="Data")
c = linefit(x, y)
print("The coefficients are $c")
xplot = range(xmin, xmax, 100)
plot(xplot, polyval.(xplot, coeffs=c), "r", label="Linear least squares fit")
legend(loc="best");
```

```
The coefficients are [3.028889515396512, 2.395702991142943]
```



4.5.4 Least squares fiting to higher degree polynomials

The method above extends to finding a polynomial

$$p(x) = c_0 + c_1 x + \cdots + c_n x^n$$

that gives the best least squares fit to data

$$(x_1, y_1), \ldots (x_N, y_N)$$

in that the coefficients c_k given the minimum of

$$E(c_0, \ldots c_n) = \sum_i (p(x_i) - y_i)^2 = \sum_i \left(y_i - \sum_k c_k x_i^k \right)^2$$

Note that when $N = n + 1$, the solution is the interpolating polynomial, with error zero.

The necessary conditions for a minimum are that all $n + 1$ partial derivatives of E are zero:

$$\frac{\partial E}{\partial c_j} = 2 \sum_i \left(y_i - \sum_k c_k x_i^k \right) x_i^j = 0, \; 0 \le j \le n.$$

This gives

$$\sum_i \sum_k \left(c_k x_i^{j+k} \right) = \sum_k \left(\sum_i x_i^{j+k} \right) c_k = \sum_i y_i x_i^j, \; 0 \le j \le n,$$

or with the notation $m_k = \sum_i x_i^k$, $p_k = \sum_i x_i^k y_i$ introduced above,

$$\sum_k m_{j+k} c_k = p_j, \; 0 \le j \le n.$$

That is, the equations are again $Mc = p$, but now with

$$M = \begin{bmatrix} m_0 & m_1 & \dots & m_n \\ m_1 & m_2 & \dots & m_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ m_n & m_{n+1} & \dots & m_{2n} \end{bmatrix}, \quad p = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{bmatrix}, \quad c = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}.$$

This is done by a variation of `polyfit` with a third argument `n` specfying the degree of the polynomial sought: `polyfit(x, y, n)`.

Remark 4.8 (Multiple dispatch in Julia)

As mentioned above, Julia allows the following version of `polyfit` to coexist with the version at *Functions for computing the coefficients and evaluating the polynomials*, which is imported above; they are distinguished by the input arguments being `(x, y)` in one case, `(x, y, n)` in the other.

```
function polyfit(x, y, n)
    # Version 2: least squares fitting.
    # Compute the coeffients c_i of the polynomial of degree n that give the best␣
↪least squares fit to data (x[i], y[i]).

    N = length(x)
    m = zeros(2n+1)
    for k in 0:2n
        m[k+1] = sum(x.^k)  # Here and below, shift the indices up by one, since␣
↪Julia counts from 1, not 0.
    end
    M = zeros(n+1,n+1)
    for i in 0:n
        for j in 0:n
            M[i+1, j+1] = m[i+j+1]
        end
    end
    p = zeros(n+1)
    for k in 0:n
        p[k+1] = sum(x.^k .* y)
    end
    c = solvelinearsystem(M, p)
    return c
end;
```

This time, let us look at extrapolation too: values beyond the interval containing the data, and try all degrees up to three:

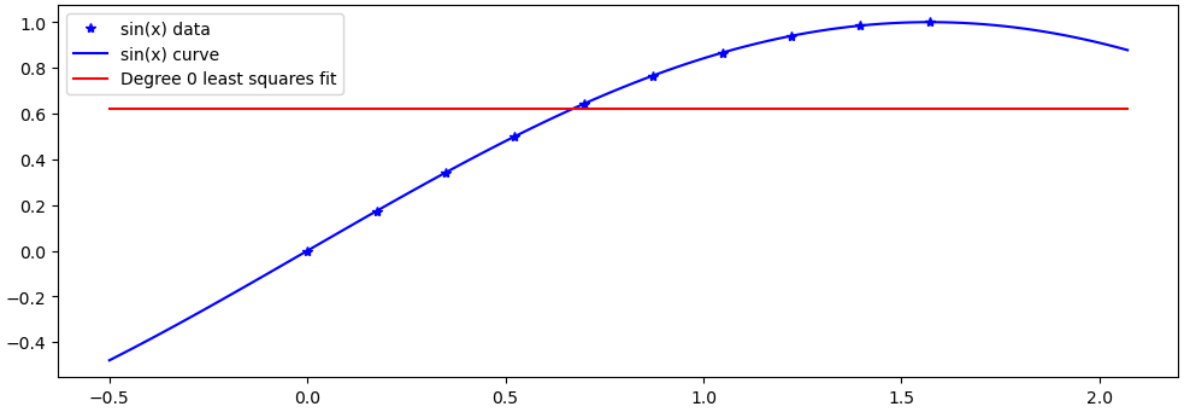
```
N = 10
n = 3
xdata = range(0.0, pi/2, N)
ydata = sin.(xdata)
for n in 0:3
    figure(figsize=[12,4])
    plot(xdata, ydata, "b*", label="sin(x) data")
    xplot = range(-0.5, pi/2 + 0.5, 100)
    plot(xplot, sin.(xplot), "b", label="sin(x) curve")
    c = polyfit(xdata, ydata, n)
    println("For degree $n the coefficients are ", c)
    plot(xplot, polyval.(xplot, coeffs=c), "r", label="Degree $n least squares fit")
    legend(loc="best")
```

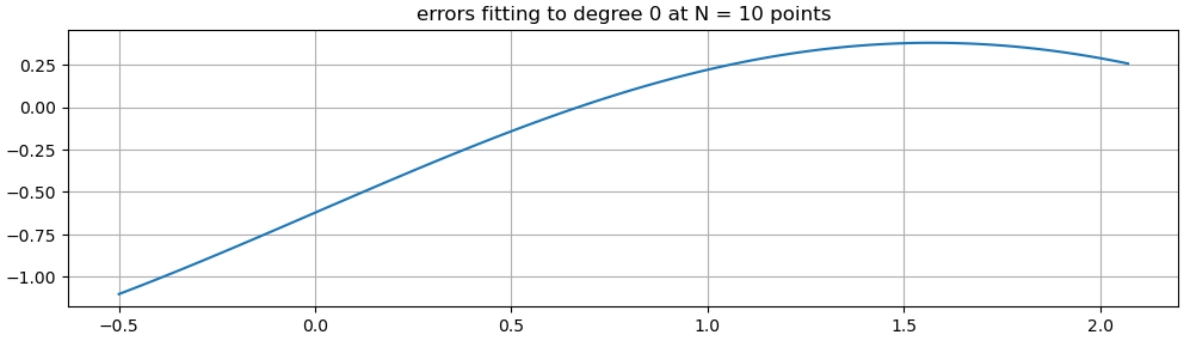
(continues on next page)

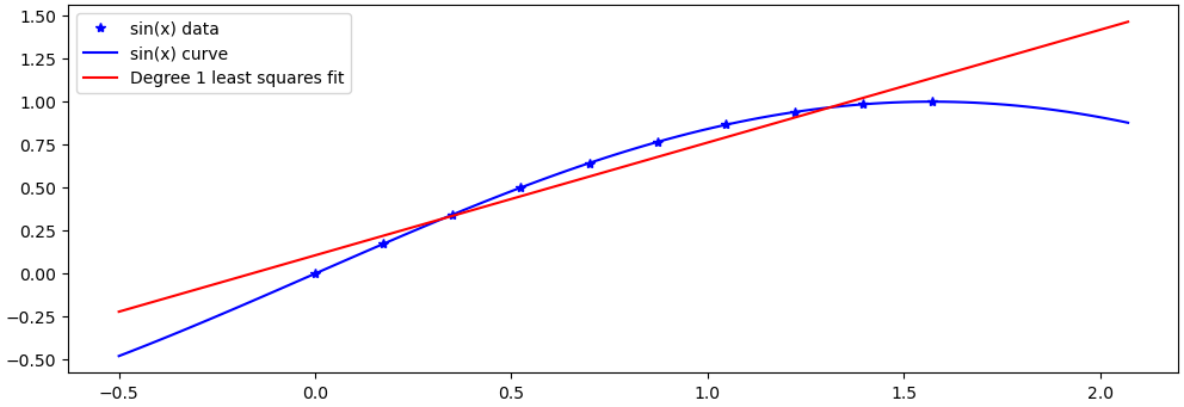
(continued from previous page)

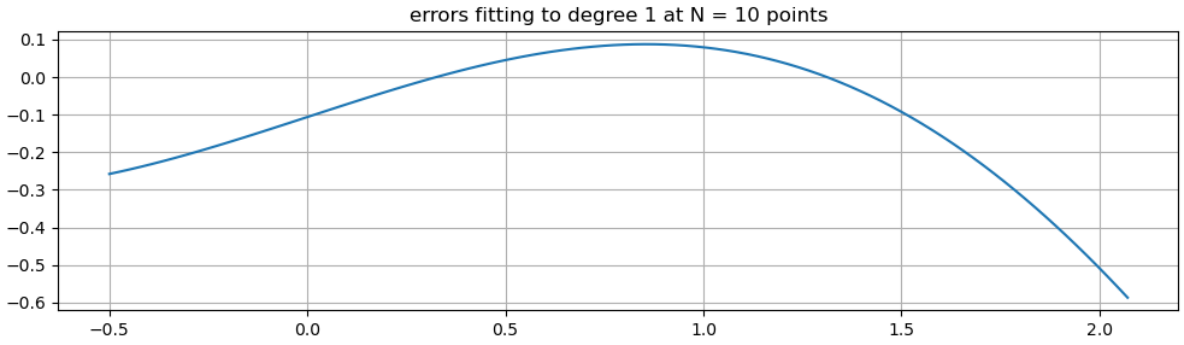
```
    # Errors:
    figure(figsize=[12,3])
    plot(xplot, sin.(xplot) - polyval.(xplot, coeffs=c))
    title("errors fitting to degree $n at N = 10 points")
    grid(true)
end
```

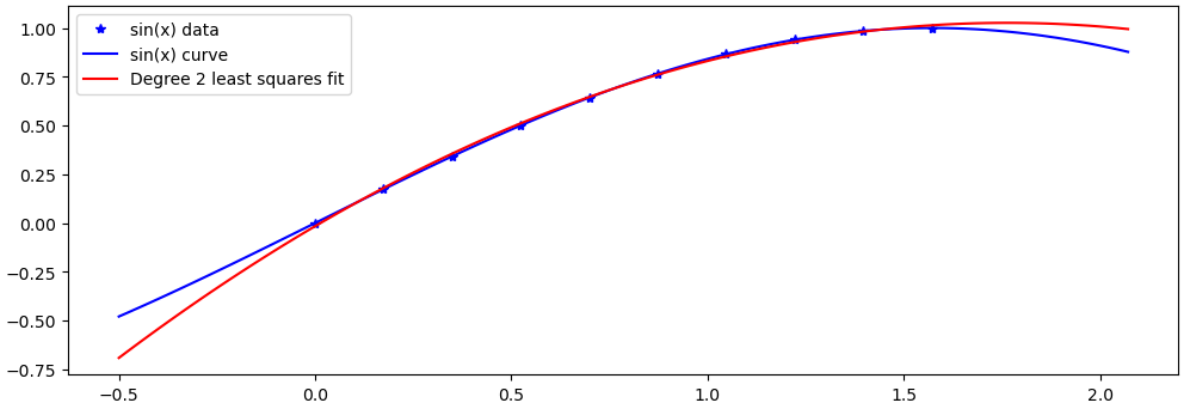
```
For degree 0 the coefficients are [0.621502615138067]
For degree 1 the coefficients are [0.10638045715159698, 0.6558739019176877]
For degree 2 the coefficients are [-0.016265239454253864, 1.182904961239479, -0.
↪335518393016084]
For degree 3 the coefficients are [-0.0010872800198948873, 1.023817333884883, -0.
↪06859177724109337, -0.11328717424901459]
```

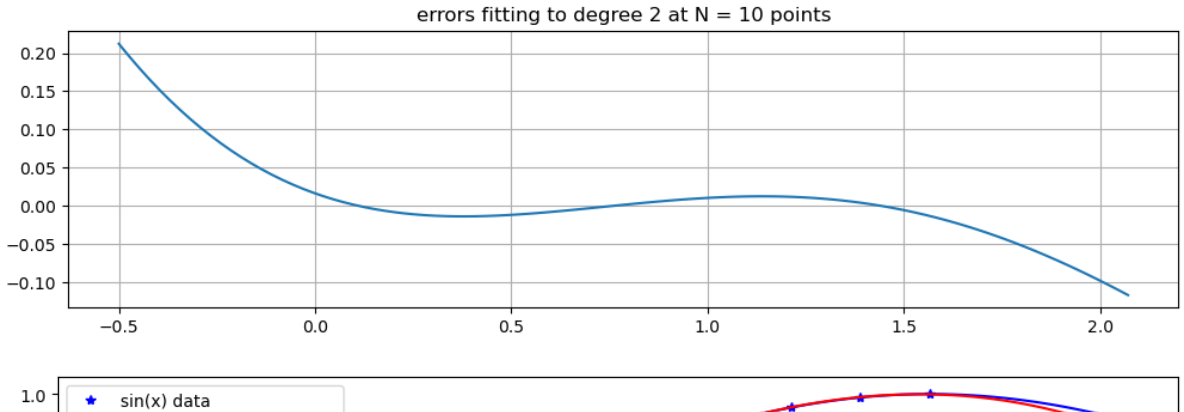


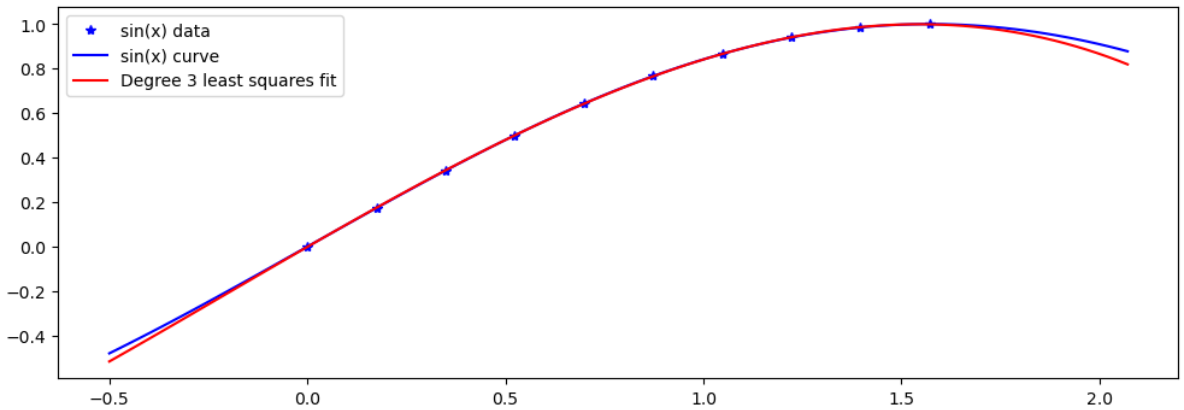


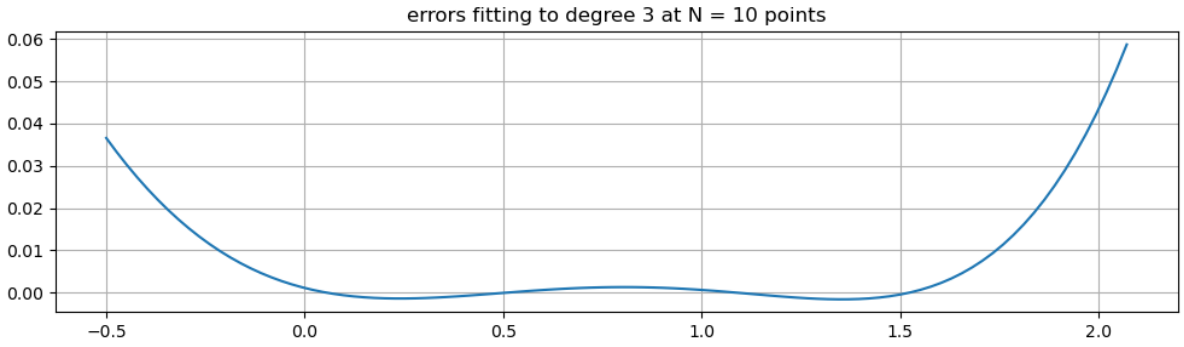



errors fitting to degree 1 at N = 10 points
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
−0.5
0.0
0.5
1.0
1.5
2.0


sin(x) data
sin(x) curve
Degree 2 least squares fit
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.5
0.0
0.5
1.0
1.5
2.0


errors fitting to degree 2 at N = 10 points
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.5
0.0
0.5
1.0
1.5
2.0

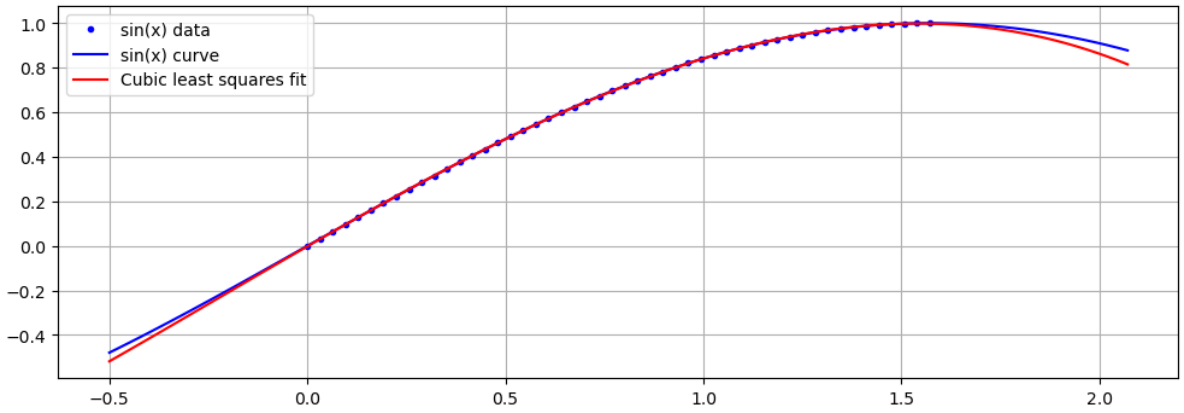

sin(x) data
sin(x) curve
Degree 3 least squares fit
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.5
0.0
0.5
1.0
1.5
2.0

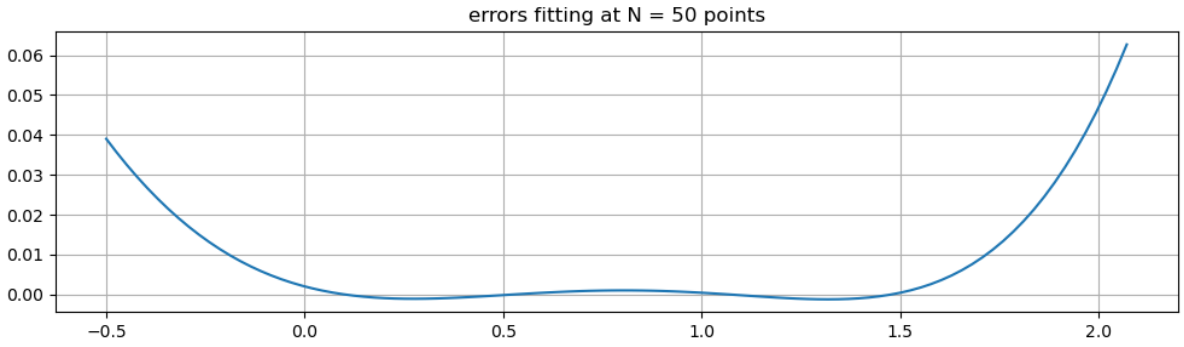


What if we fit at more points?

```
N = 50
xdata = range(0.0, pi/2, N)
ydata = sin.(xdata)

figure(figsize=[12,4])
plot(xdata, ydata, "b.", label="sin(x) data")
plot(xplot, sin.(xplot), "b", label="sin(x) curve")
c = polyfit(xdata, ydata, n)
print("The coefficients are ", c)
plot(xplot, polyval.(xplot, coeffs=c), "r", label="Cubic least squares fit")
legend(loc="best")
grid(true)
# Errors:
figure(figsize=[12,3])
plot(xplot, sin.(xplot) - polyval.(xplot, coeffs=c))
title("errors fitting at N = 50 points")
grid(true)
```

```
The coefficients are [-0.002007892581016861, 1.0264756845620808, -0.
↪06970463260725364, -0.11371819516982211]
```





Not much changes!

This hints at another use of least squares fitting: fitting a simpler curve (like a cubic) to a function (like $\sin(x)$), rather than to discrete data; this is essentialy given by the limit as the number of fitting points goes to infinity.

4.5.5 Nonlinear fitting: power-law relationships

When data (x_i, y_i) is inherently positive, it is often natural to seek an approximate power law relationship

$$y_i \approx cx_i^p$$

That is, one seeks the power p and scale factor c that minimizes error in some sense.

When the magnitudes and the data y_i vary greatly, it is often appropriate to look at the relative errors

$$e_i = \left| \frac{cx_i^p - y_i}{y_i} \right|$$

and this can be shown to be very close to looking at the absolute errors of the logarithms

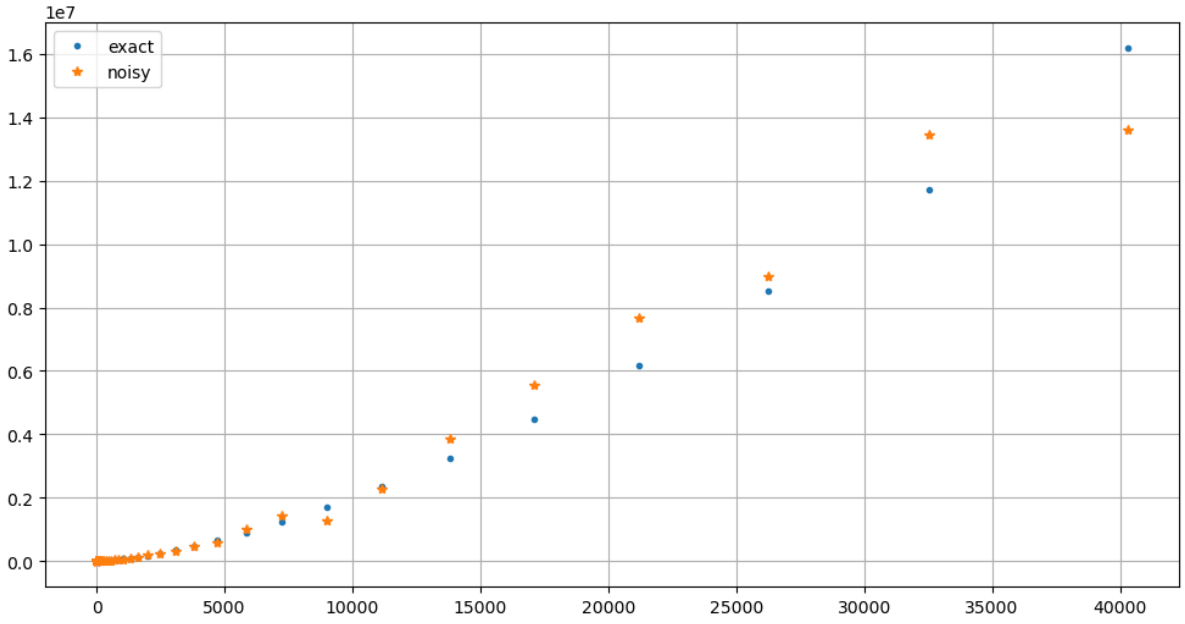
$$|\ln(cx_i^p) - \ln(y_i)| = |\ln(c) + p\ln(x_i) - \ln(y_i)|$$

Introducing the new variables $X_i = \ln(x_i)$, $Y_i = \ln(Y_i)$ and $C = \ln(c)$, this becomes the familiar problem of finding a linear approxation of the data Y_i by $C + pX_i$.

4.5.6 A simulation

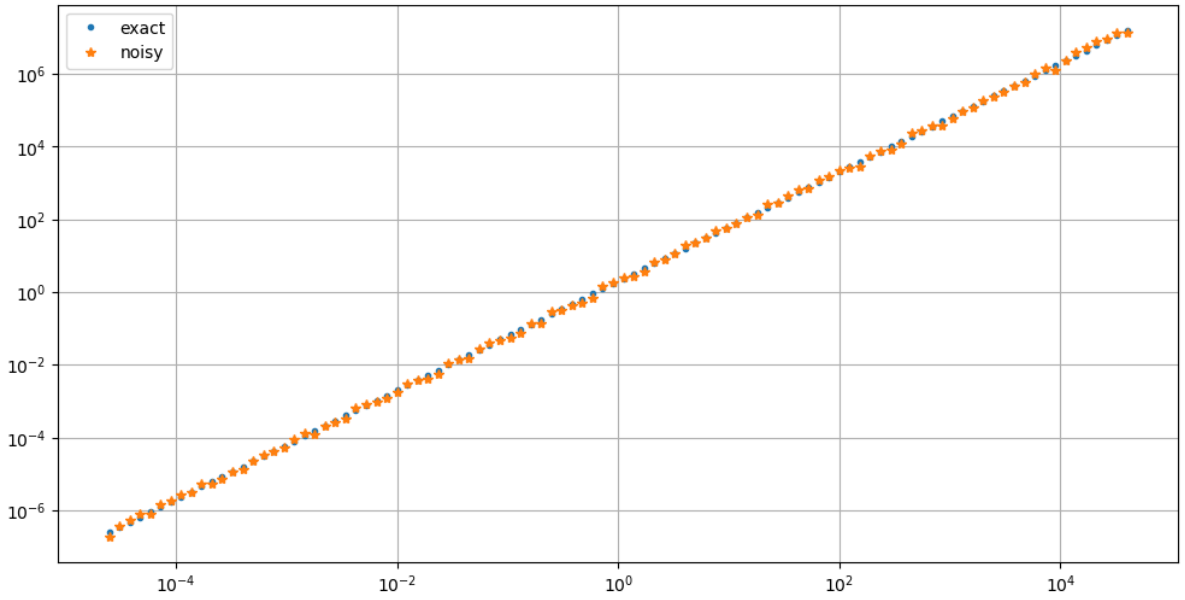
```
cexact = 2.0
pexact = 1.5
xmin = 0.01
xmax= 100.0
N = 100
x = (10.0).^range(log(xmin), log(xmax), N)
xplot = (10.0).^range(log(xmin), log(xmax), 100)  # For graphs later
yexact = cexact * x.^pexact
y = yexact .* (1.0 .+ (rand(N) .- 0.5)/2.0);
```

```
figure(figsize=[12,6])
plot(x, yexact, ".", label="exact")
plot(x, y, "*", label="noisy")
legend()
grid(true)
```



A log-log plot makes the situation far clearer:

```
figure(figsize=[12,6])
loglog(x, yexact, ".", label="exact")
loglog(x, y, "*", label="noisy")
legend()
grid(true)
```




```
X = log.(x)
Y = log.(y)
pC = polyfit(X, Y, 1)
c = exp(pC[1])
p = pC[2]
```

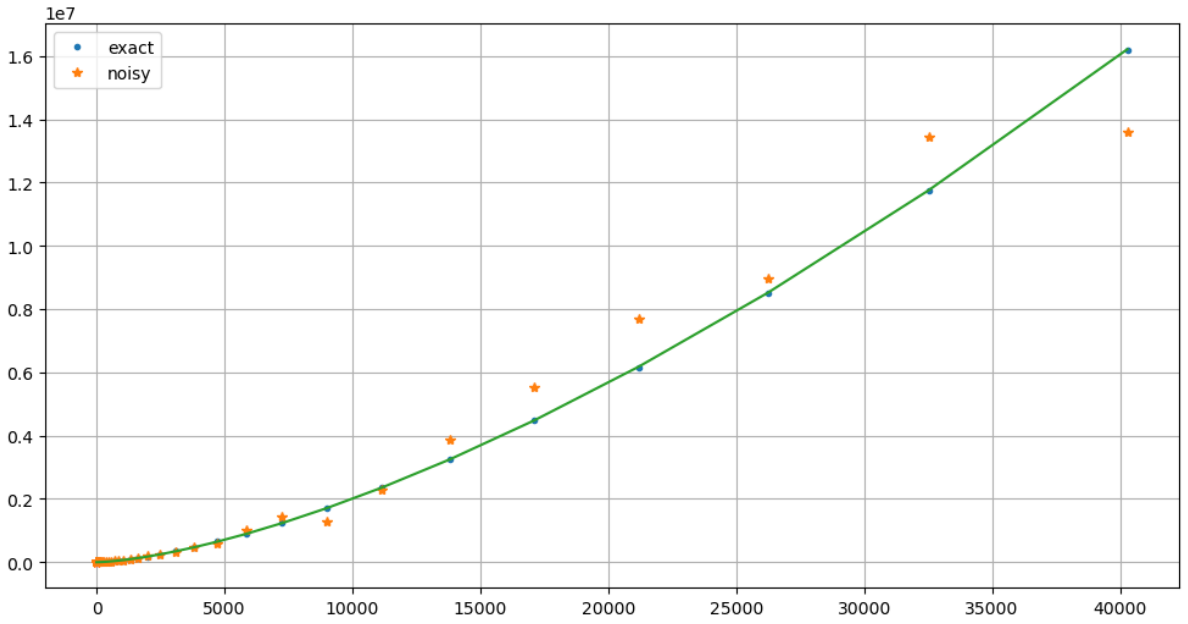
(continues on next page)

(continued from previous page)

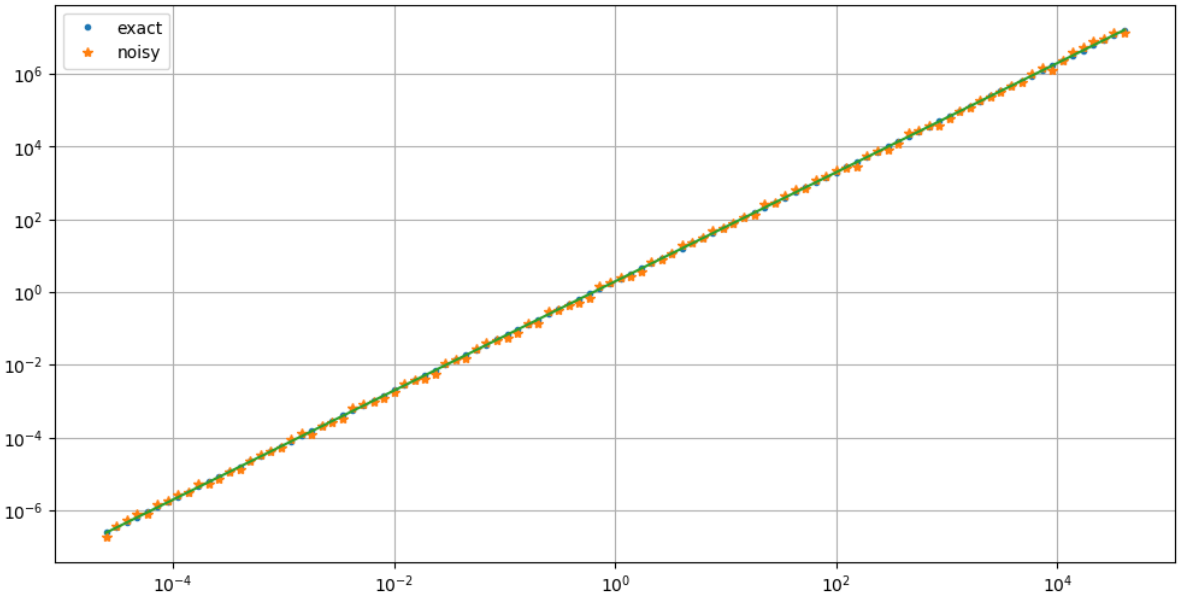
```
print("p=$p, c=$c")
```

```
p=1.50125904206987, c=1.9785543393777234
```

```
figure(figsize=[12,6])
plot(x, yexact, ".", label="exact")
plot(x, y, "*", label="noisy")
plot(xplot, c * xplot.^p)
legend()
grid(true)
```




```
figure(figsize=[12,6])
loglog(x, yexact, ".", label="exact")
loglog(x, y, "*", label="noisy")
loglog(xplot, c * xplot.^p)
legend()
grid(true)
```



4.6 Least-squares Fitting to Data: Appendix on The Geometrical Approach

References:

- Chapter 4 *Least Squares* of [Sauer, 2019], sections 1 and 2.
- Section 8.1 *Discrete Least Squares Approximation* of [Burden *et al.*, 2016].

4.6.1 Introduction

We have seen that one common and important approach to approximating data

$$(x_i, y_i),\ 1 \leq i \leq N$$

by a polynomial $y = p(x) = c_0 + \cdots c_n x^n$ of degree at most n is to minimize the "average" of the errors

$$e_i = y_i - f(x_i),$$

in the sense of the *root-mean-square error* $E_{RMS} = \sqrt{\sum_{i=1}^{N} e_i^2}$. Equivalently, we will avoid the square root and just minimize the sum of the squares of the errors:

$$E(c_0, c_1, \ldots, c_n) = \sum_{i=1}^{N} e_i^2$$

4.6.2 Linear least squares: minimizing RMS error using calculus

One way to derive the needed formulas is by seeking the critical point og th abev function via teh $n+1$ equations

$$\frac{\partial E}{\partial c_i} = 0, \quad 0 \le i \le n$$

Fortunately these gives a systems of linear equations, and it has a unique solution, thus giving the desired global minimum.

However, there is another "geometrical" approach, that is also relevant as an introduction to strategies also used for other minimization problems, for example with application to the numerical solutions of boundary value problems for differential equations.

4.6.3 Linear least squares: minimizing RMS error by minimizing "Euclidean" distance with geometry

For approximation by a polynomial $y = p(x) = c_0 + \cdots c_n x^n$, we can think of the data $y_i, 1 \le i \le N$ as giving a point in N-dimensional space $(R)^N$, and the approximations as giving another point with coordinates $\tilde{y}_i := p(x_i)$.

Then the least squares problem is to minimize the Euclidean distance $\|y - \tilde{y}\|_2$.

One way to think of this is that we attempt unsuccessfully to solve the collocation equations $p(x_i) = y_i$ as an *over-determined* sytem of N equations in $n+1$ unknowns $Ac = y$, where

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_i & x_i^2 & \dots & x_i^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^n \end{bmatrix}$$

so that Ac evaluates the polynomial at all the x_i values.

Now we introduce a key geometrical idea: the possible values of $\tilde{y} = Ac$ lie in an $(n+1)$-dimensional sub-space or "hyperplane" within $\mathbb{R}^N$, and the point in this hyper-plane closest to $y \in \mathbb{R}^N$ is the perpendular projection of the latter point onto this hyper-plane: that is, the error vector $e = y - \tilde{y}$ is perpendicular to every vector in the subspace of vectors Ac'. Thus, $e \perp Ac'$ for every $c' \in \mathbb{R}^{n+1}$.

Writing this in terms of inner products,

$$(y - \tilde{y}, Ac') = 0 \quad \text{for every } c' \in \mathbb{R}^{n+1}.$$

Recall that $(x, Ay) = (A^T x, y)$ where A^T is the transpose of A: the mirror image with $a^T_{i,j} = a_{j,i}$.

Using this gives

$$(A^T(y - \tilde{y}), c') = 0 \quad \text{for every } c' \in (R)^{n+1}.$$

and so the vector at left must be zero: $A^T(y - \tilde{y}) = 0$.

Inserting $\tilde{y} = Ac$ gives $A^T y = A^T A c$, so

$$Mc = A^T y$$

where $M := A^T A$

Since here A is $N \times (n+1)$, A^T is $(n+1) \times N$, and the product M is an $(n+1) \times (n+1)$ square matrix.

Further calculation shows that in fact

$$M = \begin{bmatrix} m_0 & m_1 & \dots & m_n \\ m_1 & m_2 & \dots & m_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ m_n & m_{n+1} & \dots & m_{2n} \end{bmatrix}, \quad m_k = \sum_{i=1}^{N} x_i^k$$

and the right-hand side is

$$A^T y = p = [p_0, p_1, \dots, p_n]^T, \quad p_k = \sum_{i=1}^{N} x_i^k y_i$$

so these equations are the same ones $Mc = p$ given by the previous calculus derivation.

CHAPTER

FIVE

DERIVATIVES AND DEFINITE INTEGRALS

5.1 Approximating Derivatives by the Method of Undetermined Coefficients

References:

- Section 5.1 *Numerical Differentiation* of [Sauer, 2019].
- Section 4.1 *Numerical Differentiation* of [Burden *et al.*, 2016].
- Section 4.2 *Estimating Derivatives and Richardson Extrapolation* of [Chenney and Kincaid, 2012].

We have seen several formulas for approximating a derivative $Df(x)$ or higher derivative $D^k f(x)$ in terms of several values of the function f, such as

$$Df(x) \approx D_h f(x) := \frac{f(x+h) - f(x)}{h} \tag{5.1}$$

and

$$D^2 f(x) \approx \delta^2 f(x) := \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}. \tag{5.2}$$

In the first case we get an error formula

$$D_h f(x) = \frac{f(x+h) - f(x)}{h} = Df(x) + \frac{D^2 f(\xi)}{2} h, \quad \xi \text{ between } x \text{ and } x+h$$

by inserting the Taylor formula $f(x+h) = f(x) + Df(x) + \frac{1}{2} D^2 f(\xi)$, and thus an "order of accuracy formula"

$$D_h f(x) = \frac{f(x+h) - f(x)}{h} = Df(x) + O(h)$$

so that the error is

$$D_h f(x) - Df(x) = \frac{1}{2} D^2 f(\xi) = O(h).$$

These are linear combinations of values of f at various points, with the denominator scaling with the k-th power of the mode spacing scale h, which makes sense given the linearity of derivatives and the way that the k-th derivative scales when one rescales $f(x)$ to $f(ck)$.

Thus we will make the *Ansatz* that the k-th derivative $D^k f(x)$ can be approximated using values at the $r - l + 1$ equally spaced points

$$x + lh, x + (l+1)h, \ldots x + rh$$

where the integers l and r can be negative, positive or zero. The assumed form then is

$$D^k f(x) \approx D_h^k f(x) = \frac{C_l f(x + lh) + C_{l+1} f(x + (l+1)h) + \cdots + C_r f(x + rh)}{h^k} + O(h^p)$$

(The reason for the power k in the denominator will be seen soon.)

So we seek to determine the values of the initially undetermined coefficients C_i, by the criterion of giving an error $O(h^p)$ with the highest possible order p. With $r - l + 1$ coefficients to choose, we generally get $p = r - l + 1 - k$, but with symmetry $l = -r$ and k even we get one better, $p = r - l + 2 - k$, because the order p must then be even. Thus we need the number of points $r - l + 1$ to be more than k: for example, at least two for a first derivative as already seen.

Example 5.1 (The basic forward difference approximation)

$$Df(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

has $k = 1$, $l = 0$, $r = 1$, $p = 1$.

Example 5.2 (A three-point one-sided difference approximation of the first derivative)

This is the case $k = 1$ and can be sought with $l = 0$, $r = 2$, as

$$Df(x) = \frac{C_0 f(x) + C_1 f(x+h) + C_2 f(x+2h)}{h} + O(h^p)$$

and the most accurate choice is $C_0 = -3/2, C_1 = 2, C_2 = -1/2$, again of second order, which is exactly $p = r - l + 1 - k$, with no "symmetry bonus":

$$Df(x) \approx \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} + O(h^2).$$

One can use Taylor's Theorem to *check* an approximation like this, and also get information about its accuracy. To do this, insert a Taylor series formula with center x, like

$$f(x+h) = f(x) + Df(x)h + \frac{D^2 f(x)}{2}h^2 + \frac{D^3 f(x)}{6}h^3 + \cdots$$

If you are not sure how accurate the result is, you might need to initially be vague about how may terms are needed, so I will do it that way and then go back and be more specific once we know more.

A series for $f(x + 2h)$ is also needed:

$$\begin{aligned} f(x+2h) &= f(x) + Df(x)(2h) + \frac{D^2 f(x)}{2}(2h)^2 + \frac{D^3 f(x)}{6}(2h)^3 + \cdots \\ &= f(x) + 2Df(x)h + \frac{D^2 f(x)}{2}4h^2 + \frac{D^3 f(x)}{6}8h^3 + \cdots \\ &= f(x) + 2Df(x)h + 2D^2 f(x)h^2 + \frac{4D^3 f(x)}{3}h^3 + \cdots \end{aligned}$$

Insert these into the above three-point formula, and see how close it is to the exact derivative:

$$\begin{aligned} &\frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} \\ &= \frac{-3f(x) + 4[f(x) + Df(x)h + \frac{D^2 f(x)}{2}h^2 + \frac{D^3 f(x)}{6}h^3 + \cdots] - [f(x) + 2Df(x)h + 2D^2 f(x)h^2 + \frac{4D^3 f(x)}{3}h^3 + \cdots]}{2h} \end{aligned}$$

Now gather terms with the same power of h (which is also gathering terms with the same order of derivative):

$$\begin{aligned} \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} &= f(x)\frac{-3+4-1}{2h} + Df(x)\frac{4-2}{2} + D^2 f(x)\frac{4/4 - 2/2}{h} + D^3 f(x)\frac{4/12 - 4/6}{h} + \cdots \\ &= Df(x) - \frac{D^3 f(x)}{3}h^2 + \cdots \end{aligned}$$

and it is clear that the omitted terms have higher power of h: h^3 and up. That is, they are $O(h^3)$, or more conveniently $o(h^2)$.

Thus we have confirmed that the error in this approximation is

$$Df(x) - \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} = \frac{D^3 f(x)}{3} h^2 + o(h^2) = O(h^2).$$

Example 5.3 (A three-point centered difference approximation of $D^2 f(x)$)

This has $k = 2$, $l = -1$, $r = 1$ and so

$$D^2 f(x) \approx \frac{C_{-1} f(x-h) + C_0 f(x) + C_1 f(x+h)}{h^2}$$

and it can be found (as discussed below) that the coefficients $C_{-1} = C_1 = 1$, $C_0 = -2$ give the highest order error: $p = 2$; one better than $p = r - l + 1 - k = 1$ due to symmetry:

$$D^2 f(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + O(h^2).$$

5.1.1 Method 1: use Taylor polynomials in h of degree p+k-1

(so with error terms $O(h^{p+k})$.)

Each of the terms $f(x + ih)$ in the above formula for the approximation $D_h^k f(x)$ of the k-th derivative $D_k f(x)$ can be expanded with the Taylor Formula up to order $p + k$,

$$f(x+ih) = f(x) + (ih)Df(x) + (ih)^2/2 D^2 f(x) + \cdots + (ih)^j/j! D^j f(x) + \cdots + (ih)^{p+k}/(p+k)! D^{p+k} f(x) + o(h^{p+k})$$

Then these can be rearranged, putting the terms with the same derivative $D^j f(x)$ together — all of which have the same factor h^j in the numeriator, and so the same factor h^{j-p} overall:

$$\begin{aligned}
D_h^k f(x) = &(C_l + \cdots + C_r) f(x) h^{-k} \\
&+ (lC_l + \cdots + rC_r) Df(x) h^{1-k} \\
&+ (l^2 C_l + \cdots + r^2 C_r) D^2 f(x) h^{2-k} \\
&\vdots \\
&+ (l^j C_l + \cdots + r^j C_r) D^j f(x) h^{j-k} \\
&\vdots \\
&+ (l^{p+k} C_l + \cdots + p + k^j C_r) D^{p+k} f(x) h^p \\
&+ o(h_p)
\end{aligned}$$

The final "small" term $o(h^p)$ comes from the terms $o(h^{p+k})$ in each Taylor's formula term, each divided by h^k.

We want this whole thing to be approximately $D^k f(x)$, and the strategy is to match the coefficients of the derivatives:

- Matching the coefficients of $D_h^k f(x)$,

$$(l^k C_l + \cdots + r^k C_r) D^k f(x) h^{k-k} = (l^k C_l + \cdots + r^k C_r) D^k f(x) = D^k f(x)$$

so

$$l^k C_l + \cdots + r^k C_r = 1 =$$

- On the other hand, there should be no term with factor $f(x)h^{-k}$, so

$$C_l + \cdots + C_r = 0$$

- More generally, for any j other than k the coefficients should vanish, so

$$l^j C_l + \cdots + r^j C_r = 0, \quad 0 \le j \le p+k \text{ except for } j = k$$

This last line gives $p+k$ linear equations in the $p+k+1$ coefficients $C_1, \ldots, C_{p+k}$, and then the previous equation gives us a total of $p+k+1$ equations — as needed for the existence of a unique solution.

$$C_l + \cdots + C_r = 0 \tag{5.3}$$
$$l^j C_l + \cdots + r^j C_r = 0, j \neq k \tag{5.4}$$
$$l^k C_l + \cdots + r^k C_r = 1 \tag{5.5}$$
$$\tag{5.6}$$

And indeed it can be verified that the resulting matrix for this system of equations is non-singular, and so there is a unique solution for the coefficients $C_l \ldots C_r$.

See *Exercise 1*.

5.1.2 Degree of Precision and testing with monomials

This concept relates to a simpler way of determining the coefficients.

The **degree of precision** of an approximation formula (of a derivative or integral) is the highest degree d such that the formula is exact for all polynomials of degree up to d. For example it can be checked that in the examples above, the degrees of precision are 1, 2, and 3 respectively. All three conform to a general pattern:

Theorem 5.1

The degree of precision is $d = p + k - 1$, so in the typical case with no "symmetry bonus" $d = r - l$

This is confirmed by the above derivation: for f any polynomial of degree $p + k - 1$ or less, the Taylor polynomials of degree at most $p + k - 1$ used there have no error.

Thus for example, the minimal symmetric aproximation of a fourth derivative, which must have even order $p = 2$, will have degree of precision 5.

5.1.3 Method 2: use monomials of degree up to p+k-1

From the above degree of precision result, one can determine the coefficients by requiring degree of precision $p + k - 1$, and for this it is enough to require exactness for each of the simple monomial functions 1, x, x^2, and so on up to x^{p+k-1}.

Also, this only needs to be tested at $x = 0$, since "translating" the variables does not effect the result.

This is probably the simplest method in practice.

Example 5.4

Let us revisit *Example 5.2*. The goal is to get exactness in

$$\frac{C_0 f(x) + C_1 f(x+h) + C_2 f(x+2h)}{h} = Df(x)$$

for the monomials $f(x) = 1$, $f(x) = x$, and so on, to the highest power possible, and this only needs to be checked at $x = 0$.

First, $f(x) = 1$, so $Df(0) = 0$:

$$\frac{C_0 \times 1 + C_1 \times 1 + C_2 \times 1}{h} = 0,$$

so

$$C_0 + C_1 + C_2 = 0$$

Next, $f(x) = x$, so $Df(0) = 1$:

$$\frac{C_0 f(0) + C_1 f(h) + C_2 f(2h)}{h} = \frac{C_0 0 + C_1 h + C_2 2h}{h} = C_1 + 2C_2 = 1$$

so

$$C_1 + 2C_2 = 1$$

We need at least three equations for the three unknown coefficients, so continue with $f(x) = x^2$, $Df(0) = 0$:

$$\frac{C_0 f(0) + C_1 f(h) + C_2 f(2h)}{h} = \frac{C_0 0 + C_1 h^2 + C_2 (2h)^2}{h} = (C_1 + 4C_2)h = 0$$

so

$$C_1 + 4C_2 = 0$$

We can solve these by elimination; for example:

- The last equation gives $C_1 = -4C_2$
- The previous one then gives $-4C_2 + 2C_2 = 1$, so $C_2 = -1/2$ and thus $C_1 = -4C_2 = 2$.
- The first equation then gives $C_0 = -C_1 - C_2 = -3/2$ all as claimed above.

So far the degree of precision has been shown to be at least 2. In some cases it is better, so let us check by looking at $f(x) = x^3$:

$Df(x) = 0$, whereas

$$\frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} = \frac{-30 + 4h^3 - (2h)^3}{2h} = \frac{-2h^3}{2h} = -4h^2, \neq 0$$

So, no luck this time (that typically requires some symmetry), but this calculation does indicate in a relatively simple way that the error is $O(h^2)$.

Remark 5.1

If you want to verify more rigorously the order of accuracy of a formula devised by this method, one can use the "checking" procedure with Taylor polynomials and their error terms as done in *Example 5.2* above.

5.1.4 Exercises

Exercise 1

A) Derive the formula in *Example 5.1*.

Do this by setting up the three equations as above for the coefficients C_0, C_1 and C_2, and solving them. Do this "by hand", to get exact fractions as the answers; use the two Taylor series formulas, but now take advantage of what we saw above: that the error starts at the terms in $D^3 f(x)$. So use the forms

$$f(x+h) = f(x) + Df(x)h + \frac{D^2 f(x)}{2}h^2 + \frac{D^3 f(x)}{6}h^3 + O(h^4)$$

and

$$f(x+2h) = f(x) + 2Df(x)h + 2D^2 f(x)h^2 + \frac{4D^3 f(x)}{3}h^3 + O(h^4)$$

B) Verify the result in *Example 5.3*.

Again, do this by hand, and exploit the symmetry. Note that it works a bit better than expected, due to the symmetry.

Exercise 2: like Exercise 1, but using Method 2

A) Verify the result in *Example 5.1*, this time by Method 2.

That is, impose the condition of giving the exact value for the derivative at $x = 0$ for the monomial $f(x) = 1$, then the same for $f(x) = x$, and so on until there are enough equations to determine a unique solution for the coefficients.

B) Verify the result in *Example 5.3*, by Method 2.

5.2 Richardson Extrapolation

References:

- Section 5.1.3 *Extrapolation* in [Sauer, 2019].
- Section 4.2 *Richardson Extrapolation* im [Burden *et al.*, 2016].
- Section 4.2 *Estimating Derivatives and Richardson Extrapolation* in [Chenney and Kincaid, 2012].

5.2.1 Motivation

With derivative approximations like

$$\Delta_h f(x) := \frac{f(x+h) - f(x)}{h} = Df(x) + \frac{D^2 f(x)}{2}h + O(h^2) = Df(x) + O(h)$$

and

$$\delta_h^2 f(x) := \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} = D^2 f(x) + \frac{D^4 f(x)}{12}h^2 + O(h^4) = D^2 f(x) + O(h^2)$$

there are limits on the ability to improve accuracy by simply using a smaller value of h: one is that rounding error become problematic.

Another is that when we are approximating the derivative at a collection of points in an interval $[a, b]$, $x_i = a + ih$, $0 \le i \le n$, $h = \dfrac{b-a}{n}$, reducing h requires increasing the number of points $n+1$, and so increases the "cost" (time and other resources needed) of the calculation.

Thus we would like to produce new approximation formulas of higher order p; that is, with error $O(h^p)$ for p greater than the values $p = 1$ for $\Delta_h f(x)$ or $p = 2$ for $\delta_h^2 f(x)$.

5.2.2 Procedure

The general framework for this is an exact quantity Q_0 for which we have an approximation formula $Q(h)$ with

$$Q(h) = Q_0 + c_p h^p + O(h^q), = Q_0 + O(h^p), \quad q > p$$

and we wish to achieve adequate accuracy while keeping h as large as possible.

The kernel of the idea is to initially ignore the smaller part of the error, $O(h^q)$ and just consider

$$Q(h) \approx Q_0 + c_p h^p,$$

and evaluate for two values of h; most often either h and $2h$ (or h and $h/2$, which ismore or less equivalent.)

That gives

$$Q(2h) \approx Q_0 + c_p (2h)^p = Q_0 + c_p 2^p h^p,$$

and with only Q_0 and c_p unknown, this is two (approximate) linear equations in two unknowns, so we can solve for the desired quantity Q_0 by basic Gaussian elimination. This gives

$$Q_0 \approx \frac{2^p Q(h) - Q(2h)}{2^p - 1} =: Q_q(h).$$

But is this new approximation any better than the original? Using the more complete error formula above for $Q(h)$ and its version with h replaced by $2h$,

$$Q(2h) = Q_0 + c_p (2h)^p + O((2h)^q), = Q_0 + 2^p c_p h^p + O(h^q),$$

one gets

$$Q_q(h) = \frac{2^p \phi(h) - \phi(2h)}{2^p - 1} = Q_0 + O(h^q),$$

so indeed an improvement, since $q > p$.

Rewriting to get an error estimate

We can get a useful practical error estimate by rewriting the above result as

$$Q_0 \approx Q(h) + \frac{Q(h) - Q(2h)}{2^p - 1} \tag{5.7}$$

so that the quantity

$$E_h := \frac{Q(h) - Q(2h)}{2^p - 1} \approx Q_0 - Q(h) \tag{5.8}$$

is approximately the error in $Q(h)$. Thus,

1. Richardson extrapolation can be viewed as "correcting" Q_h by subtracting of this estimated error:

$$Q_0 \approx Q_q(h) = Q_h + E_h$$

1. This magnitude $|E_h|$ of this error estimate can be used as a (typically pessimistic!) estimate of the error in the correted result Q_q. Sometimes makes sens to use an even more cautious error estimate, by discarding the denominator $2^p - 1$: using $|Q(h) - Q(2h)|$ as an estimate of the error in the extrapolated value Q_q.

Either way, these follow the pervasive pattern of using the change between the two most recent approximations as an error estimate.

Note the analogy to Newton's method for solving $f(x) = 0$, which can be broken into the two steps

- estimate the error in approximate root x_n as $E_n := -f(x_n)/f'(x_n)$
- update the approximation to $x_{n+1} = x_n + E_n$.

Finally, note that this is always *extrapolation*, in the sense of "going beyond": the new approximation is on the opposite side of the better of the original approximations from the less accurate of them.

Example 5.5

For the basic forward difference approximation above, this process give a three-point method of second order accuracy ($q = 2$):

$$\begin{aligned}\frac{2\Delta_h f(x) - \Delta_{2h} f(x)}{2-1} &= 2\frac{f(x+h) - f(x)}{h} - \frac{f(x+2h) - f(x)}{2h} \\ &= \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} \\ &= Df(x) + O(h^2).\end{aligned}$$

Exercise 1(a)

Apply Richardson extrapolation to the standard three-point, second order accurate approximation $Q(h) := \delta_h^2 f(x)$ of the second derivative $Q_0 := D^2 f(x)$ as given above, and verify that it gives a fourth-order accurate five-point approximation formula.

Exercise 1(b)

As a supplementary exercise, one could verify the order of accuracy directly with Taylor polynomials, or verify that the new formula has degree of precision $d = 5$, and hence is of order $p = 4$ due to the formula $d = p + k - 1$ for approximations of k-th derivatives, given in the notes for Day 11.

One could also derive the same formula "from scratch" using the Method of Undetermined Coefficients.

Exercise 2

Apply Richardson extrapolation to the above one-sided three-point, second order accurate approximation of the derivative $Df(x)$, and verify that it gives a third-order accurate four-point approximation formula.

But note something strange about this new formula: it skips $f(x + 3h)$.

Here, instead of extrapolating, one is probably better off applying the Method of Undetermined Coefficients directly with data $f(x)$, $f(x + h)$, $f(x + 2h)$, $f(x + 3h)$ and $f(x + 4h)$: what order of accuracy does that give?

5.2.3 A variant, more useful for integration and ODE boundary value problems: parameter n

A slight variant of the above is approximation with an integer parameter n, such as approximations of integrals by the (composite) trapezoid rule with n intervals, T_n, or the approximate solution of an ordinary differential equation at the above-described collection of $n+1$ equally spaced values in domain $[a, b]$. Then a more natural ntito of teh approxatio formula is Q_n instead of $Q(h)$.

The errors of the form $c_p h^p + O(h^q)$ become

$$Q_n = Q_0 + O\left(\frac{1}{n^p}\right) = Q_0 + \frac{c_p}{n^p} + O\left(\frac{1}{n^q}\right).$$

The main difference is that to work with integer values of n, it must be the quantity that is doubled, whereas doubling of h would correspond to halving of n.

The extrapolation formula becomes

$$Q_0 = \frac{2^p Q_{2n} - Q_n}{2^p - 1} + O\left(\frac{1}{n^q}\right). \tag{5.9}$$

Remark 5.2

For the slightly more general case of increasing from n to kn, one gets

$$Q_0 = \frac{k^p Q_{kn} - Q_n}{k^p - 1} + O\left(\frac{1}{n^q}\right).$$

A common verbal description for both forms

This can be summarized with the same verbal form as the original formula:

- 2^p *times the more accurate approximation,*
- *minus the less accurate approximation,*
- *all divided by* $(2^p - 1)$

Also

The error in the more accurate approximation is approximated by the difference between the two approximations, divided by $(2^p - 1)$

Rewriting to get an error estimate, again

As with the "h" form above, this extrapolation can be broken into two steps

$$E_{2n} := \frac{Q_{2n} - Q_n}{2^p - 1},$$
$$Q_0 = Q_{2n} + E_{2n} + O\left(\frac{1}{n^q}\right).$$

so E_{2n} estimates the error in Q_{2n}, and the improved approxmation can be expressed as

$$Q_{2n} + E_{2n}.$$

5.2.4 Repeated Richardson extrapolation

The new improved approximation formulas have the same sort of error formula, but for order q instead of order p, so we could extrapolate again to get an even higher order method, and this can be done numerous times if there is a suitable power series in h or $1/n$ for the errors.

That is not so useful for derivative approximations, where one can get the same or better results with the method of underermined coefficients, but can be very useful for integration methods, and for the related task of solving boundary value problems for ordinary differential equations.

For example, it can be applied to the composite trapezoid rule, giving the composite Simpson's rule at the first step, and then a succession of approximations of ever higher order – this is known as the **Romberg method.**

Repeated Richardson extrapolation can also be applied to the approximate solution of dofferential equations; we might explore that later.

5.3 Definite Integrals, Part 1: The Building Blocks

References:

- Sections 5.2.1 and 5.2.4 of Chapter 5 *Numerical Differentiation and Integration* in [Sauer, 2019].
- Sections 4.3 *Elmenets of Numerical Integration* of [Burden *et al.*, 2016].

5.3.1 Introduction

The objective of this and several subsequent sections is to develop methods for approxmating a definite integral

$$I = \int_a^b f(x)\, dx$$

This is arguably even more important than approximating derivatives, for several reasons; in particular, because there are many functions for which antiderivative formulas cannot be found, so that the result of the Fundamental Theorem of Calculus, that

$$\int_a^b f(x)\, dx = F(b) - F(a), \text{ for } F \text{ any antiderivative of } f$$

does not help us.

One core idea is to approximate the function f by a polynomial (or several), and use its integral as an approximation. The two simplest possibilities here are approximating by a constant and by a straight line; here we explore the latter; the former will be visited soon.

```
using PyPlot
```

5.3.2 Approximating with a single linear function: the Trapezoid Rule

The idea is to approximate $f : [a, b] \to \mathbb{R}$ by collocation at the end points of this interval:

$$f(x) \approx L(x) := \frac{f(a)(b-x) + f(b)(x-a)}{b-a}, = f_{ave}(b-a)$$

Then the approximation — which will be called T_1, for reasons that will becom clear soon — is

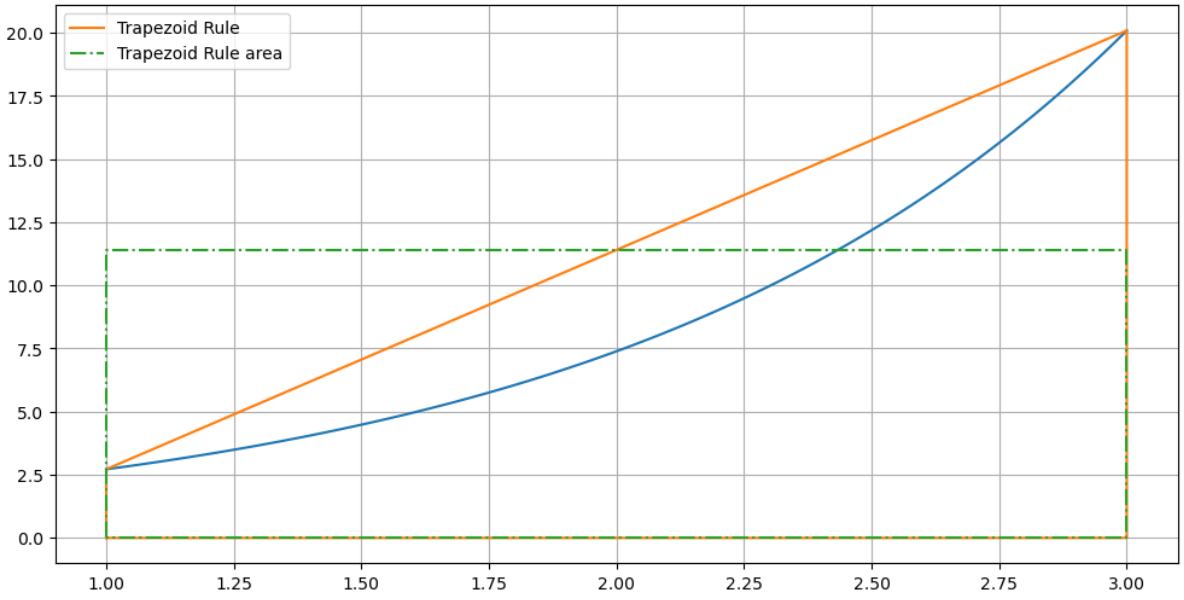
$$I \approx T_1 = \int_a^b L(x)dx = \frac{f(a)+f(b)}{2}(b-a)$$

This can be interpreted as replacing $f(x)$ by f_{ave} the average of the value at the end points, and inegrting that simple function.

For the example $f(x) = e^x$ on $[-1, 3]$

```
a = 1.0
b = 3.0
f(x) = exp(x);
```

```
f_average = (f(a) + f(b))/2
x = range(a, b, 100)
figure(figsize=[12, 6])
plot(x, f.(x))
plot([a, a, b, b, a], [0, f(a), f(b), 0, 0], label="Trapezoid Rule")
plot([a, a, b, b, a], [0, f_average, f_average, 0, 0], "-.", label="Trapezoid Rule␣
↪area")
legend()
grid(true)
```



The approximation T_1 is the area of the orange trapezoid (hence the name!) which is also the area of the green rectangle.

5.3.3 Approximating with a constant: the Midpoint Rule

The idea here is to approximate $f : [a, b] \to \mathbb{R}$ by its value at the midpoint of the interval, like the building blocks in a Riemann sum with the middel being the intuitive best choice of where to put the rectangle.

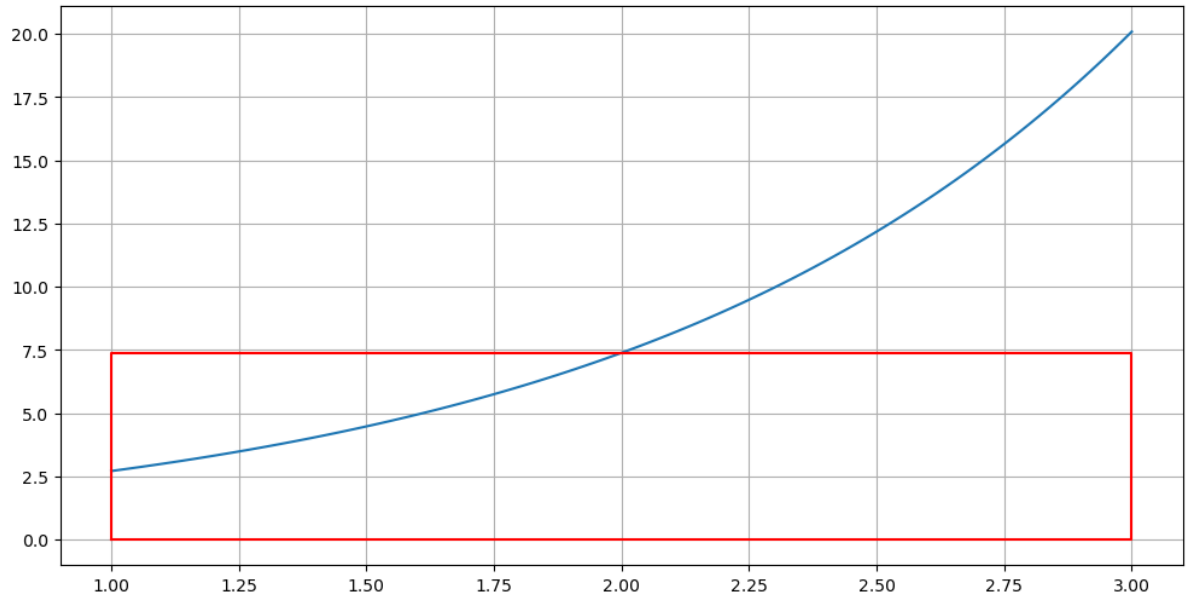
$$f(x) \approx f_{mid} := f\left(\frac{a+b}{2}\right)$$

Then the approximation — which will be called M_1 — is

$$I \approx M_1 = \int_a^b f_{mid}\, dx = f\left(\frac{a+b}{2}\right)(b-a)$$

For the same example $f(x) = e^x$ on $[-1, 3]$

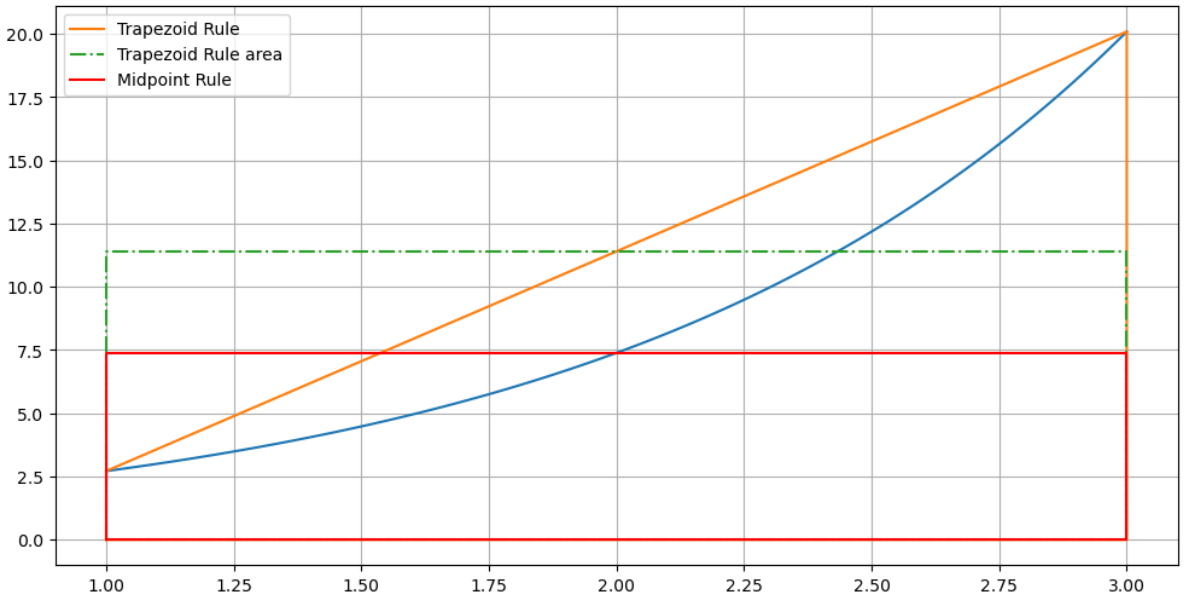
```
f_midpoint = f((a+b)/2)
figure(figsize=[12, 6])
plot(x, f.(x))
plot([a, a, b, b, a], [0, f_midpoint, f_midpoint, 0, 0], "r", label="Midpoint Rule")
grid(true)
```



The approximation M_1 is the area of the red rectangle.

The two methods can be compared my combining these graphs:

```
f_midpoint = f((a+b)/2)
figure(figsize=[12, 6])
plot(x, f.(x))
plot([a, a, b, b, a], [0, f(a), f(b), 0, 0], label="Trapezoid Rule")
plot([a, a, b, b, a], [0, f_average, f_average, 0, 0], "-.", label="Trapezoid Rule␣
↪area")
plot([a, a, b, b, a], [0, f_midpoint, f_midpoint, 0, 0], "r", label="Midpoint Rule")
legend()
grid(true)
```



5.3.4 Error Formulas

These graphs indicate that the trapezoid rule will over-estimate the error for this and any function that is convex up on the interval $[a, b]$. With closer examination it can perhaps be seen that the Midpoint Rule will instead underestimate in this situation, because its "overshoot" at left is less than its "undershoot" at right.

We can derive error formulas that confirm this, and which are the basis for both practical error estimates and for deriving more accurate approximation methods.

The first such method will be to use multiple small intervals instead of a single bigger one (using piecewise polynomial approximation) and for that, it is convenient to define $h = b - a$ which will become the parameter that we reduce in order to improve accuracy.

Theorem 5.2 (Error in the Trapezoid Rule, T_1)

For a function f that is twice differentiable on interval $[a, b]$, the error in the Trapezoid Rule is

$$\int_a^b f(x)dx - T_1 = -\frac{(b-a)^3}{12} f''(\xi) \quad \text{for some } \xi \in [a, b]$$

It will be convenient to define $h := b - a$ so that this becomes

$$\int_a^b f(x)dx - T_1 = -\frac{h^3}{12} f''(\xi) \quad \text{for some } \xi \in [a, b].$$

Theorem 5.3 (Error in the Midpoint Rule, M_1)

For a function f that is twice differentiable on interval $[a, b]$ and again with $h = b - a$, the error in the Midpoint Rule is

$$\int_a^b f(x)dx - M_1 = \frac{h^3}{24} f''(\xi) \quad \text{for some } \xi \in [a, b]$$

These will be verified below, using the error formulas for Taylor polynomials and collocation polynomials.

For now, note that:

- The results confirm that for a function that is convex up, the Trapezoid Rule overestimates and the Midpoint Rule underestimates.
- The ratio of the errors is approximately -2. This will be used to get a better result by using a weighted average: *Simpson's Rule.*
- The errors are $O(h^3)$. This opens the door to Richardson Extrapolation, as will be seen soon in the method of *Romberg Integration.*

Proofs of these error results

One side benefit of the following verifications is that they also offer illustrations of how the two fundamental error formulas help us: Taylor's Formula and its cousin the error formula for polynomial collocation.

To help prove the above formulas, we introduce a result that also helps in various places later:

Theorem 5.4 (The Integral Mean Value Theorem)

In an integral

$$\int_a^b f(x)w(x)\,dx$$

with f continuous and the "weight function" $w(x)$ positive valued (actually, it is enough that $w(x) \geq 0$ and it is not zero everyhere), there is a point $\xi \in [a, b]$ that gives a "weighted average value" for $f(x)$ in the sense that

$$\int_a^b f(x)w(x)\,dx = \int_a^b f(\xi)w(x)\,dx, = f(\xi)\int_a^b w(x)\,dx$$

Proof. As f is continuous on the closed, bounded interval $[a, b]$, the **Extreme Value Theorem** from calculus says that f has a minimum L and a maximum H on this interval: $L \leq f(x) \leq H$. Since $w(x) \geq 0$, this gives

$$Lw(x) \leq f(x)w(x) \leq Hw(x)$$

and by integrating,

$$L\int_a^b w(x)\,dx \leq \int_a^b f(x)w(x)\,dx \leq H\int_a^b w(x)\,dx$$

Dividing by $\int_a^b w(x)\,dx$ (which is positive),

$$L \leq \frac{\int_a^b f(x)w(x)\,dx}{\int_a^b w(x)\,dx} \leq H$$

and the Mean Value Theorem says that f attains this value for some $\xi \in [L, H]$:

$$f(\xi) = \frac{\int_a^b f(x)w(x)\,dx}{\int_a^b w(x)\,dx} \tag{5.10}$$

Clearing the denominator gives the claimed result.

Proof. (of *Theorem 5.2*, *the trapezoid rule error formula)*

The function integrated to get the Trapezoid Rule is the linear collocating polynomial $L(x)$, and from the section *Error Formulas for Polynomial Collocation*, we have

$$f(x) - L(x) = \frac{f''(\xi_x)}{2}(x-a)(x-b)$$

Integrating each side gives

$$\int_a^b (f(x) - L(x)) = I - T_1 = \int_a^b \frac{f''(\xi_x)}{2}(x-a)(x-b)\,dx$$

To get around the complication that ξ_x depends on x in an unknown way, use the Ingtgral Measn Vlaue Theorem with weight function $w(x) = (x-a)(b-x), \geq 0$ for $a \leq x \leq b$. Then with $-f''$ as the function f in (5.10):

$$I - T_1 = -\int_a^b \frac{f''(\xi_x)}{2}(x-a)(b-x)\,dx = -\frac{f''(\xi)}{2}\int_a^b (x-a)(b-x)\,dx$$

A bit of calculus gives $\displaystyle\int_a^b (x-a)(b-x)\,dx = \frac{(b-a)^3}{6}$, so

$$I - T_1 = -\frac{f''(\xi)}{2}\frac{(b-a)^3}{6} = -\frac{f''(\xi)}{12}(b-a)^3 = -\frac{f''(\xi)}{12}h^3,$$

as advertised.

Proof. (of *Theorem 5.3*, *the midpoint rule error formula)*

For this, we can use Taylor's Theorem for the linear approximation

$$f(x) = f(c) + f'(c)(x-c) + \frac{f''(\xi_x)}{2}(x-c)^2$$

with $c = (a+b)/2$, the midpoint. That is,

$$f(x) - f(c) = f'(c)(x-c) + \frac{f''(\xi_x)}{2}(x-c)^2$$

and integrating each side gives

$$\int_a^b f(x) - f(c)\,dx = I - M_1 = \int_a^b \left[f'(c)(x-c) + \frac{f''(\xi_x)}{2}(x-c)^2\right]dx$$

Here symmetry helps, by eliminating the first (potentialy biggest) term in the error: we use the fact that $a = c - h/2$ and $b = c + h/2$

$$\int_a^b f'(c)(x-c)\,dx = f'(c)\int_{c-h/2}^{c+h/2} x - c\,dx = \left[(x-c)^2/2\right]_{c-h/2}^{c+h/2} = (h/2)^2 - (h/2)^2 = 0$$

Thus the error simplifies to

$$I - M_1 = \int_a^b \frac{f''(\xi_x)}{2}(x-c)^2\,dx$$

and much as above, the Integral Mean Value Theorem can be used, this time with weight function $w(x) = (x-c)^2, \geq 0$:

$$I - M_1 = \frac{f''(\xi)}{2} \int_a^b (x-c)^2 \, dx$$

Another caluclus exercise: $\int_a^b (x-c)^2 dx = \int_{-h/2}^{h/2} x^2 dx = [x^3/3]_{-h/2}^{h/2} = h^3/12$, so indeed,

$$I - M_1 = \frac{f''(\xi)}{24} h^3$$

5.3.5 Appendix: Approximating a Definite Integral With the Left-hand Endpoint Rule

An even simpler approximation of $\int_a^b f(x) \, dx$ is the *Left-hand Endpoint Rule*, probably seen in a calculus course. For a single interval, this uses the approximation

$$f(x) \approx f(a)$$

leading to

$$I := \int_a^b f(x) \, dx \approx L_1 := \int_a^b f(a) \, dx = f(a)(b-a)$$

The correpsonding composite rule with n sub-intervals of equal width $h = (b-a)/n$ is

$$L_n = \sum_{i=0}^{n-1} f(x_i)h, \; = \sum_{i=0}^{n-1} f(a+ih)h$$

with $x_i = a + ih$ as before.

Theorem 5.5 (Error in the Left-hand Endpoint Rule, L_1)

For a function f that is differentiable on interval $[a, b]$, the error in the Left-hand Endpoint Rule is

$$\int_a^b f(x)dx - L_1 = \frac{(b-a)^2}{2} f'(\xi), = \frac{h^2}{2} f'(\xi) \quad \text{for some } \xi \in [a,b]n$$

Proof. This time use Taylor's Theorem just for the constant approximation with center a:

$$f(x) = f(c) + f'(\xi_x)(x-a)$$

That is,

$$f(x) - f(a) = f'(\xi_x)(x-a)$$

so integrating each side gives

$$\int_a^b f(x) - f(a) \, dx = I - L_1 = \int_a^b f'(\xi_x)(x-a)dx$$

Using the Integral Mean Value Theorem again, now with weight $w(x) = x - a$ gives

$$\int_a^b f'(\xi_x)(x-a)dx = f'(\xi) \int_a^b (x-a)dx = f'(\xi)\frac{(b-a)^2}{2} = \frac{h^2}{2} f'(\xi) \text{ for some } \xi \in [a,b]$$

and inserting this into the previous formula gives the result.

5.4 Definite Integrals, Part 2: The Composite Trapezoid and Midpoint Rules

References:

- Section 5.2.3 and 5.2.4 of Chapter 5 *Numerical Differentiation and Integration* in [Sauer, 2019].
- Section 4.4 *Composite Numerical Integration* of [Burden *et al.*, 2016].

5.4.1 Introduction

The "elementary" integral approximations of the definite integral

$$I = \int_a^b f(x)\,dx$$

seen in the previous section the Trapzoid Rule

$$T_1 = \int_a^b L(x)\,dx = \frac{f(a)+f(b)}{2}(b-a)$$

and the Midpoint Rule

$$M_1 = f\left(\frac{a+b}{2}\right)(b-a)$$

are of course of very low accuracy in themselves. They are however central building blocks for various more accurate methods and also for some good methods for numerical solution of differential equations.

The basic strategy for improving accuracy is to derive the domain of integration $[a, b]$ into numerous smaller intervals, and use these rules on each such sub-interval: the *composite* rules.

In turn, the most straightforward way to do this is to use n sub-intervals of equal width $h = (b-a)/n$, so that the sub-interval endpoints are $x_0 = a + ih$, $0 \le i \le n$: that is

sub-intervals $[x_{i-1}, x_i]$, $1 \le i \le n$ separated by the nodes.

$$a, a+h, a+2h, \ldots, b-h, b$$

The Composite Midpoint Rule

Using the Midpoint Rule on each interval and summing gives a formula that could be familiar:

$$\begin{aligned} M_n &:= f\left(\frac{x_0+x_1}{2}\right)h + f\left(\frac{x_1+x_2}{2}\right)h + \cdots + f\left(\frac{x_{n-1}+x_n}{2}\right)h \\ &= f\left(\frac{a+(a+h)}{2}\right)h + f\left(\frac{(a+h)+(a+2h)}{2}\right)h + \cdots + f\left(\frac{(b-h)+b}{2}\right)h \\ &= \left[f(a+h/2) + f(a+3h/2) + \cdots + f(b-h/2)\right]h \end{aligned}$$

This is a *Riemann Sum* as used in the definition of the defnite integral; possibly the best and natural one in most situations, by using the midpoints of each interval. The theory of definite integrals also guarantees that $M_n \to I$ as $n \to \infty$ so long as the function f is continuous — the next question for us will be "how fast?*

The Composite Trapezoid Rule

Using the Tapezoid Rule on each interval instead gives

$$\begin{aligned} T_n &:= \frac{f(x_0)+f(x_1)}{2}h + \frac{f(x_1)+f(x_2)}{2}h + \cdots + \frac{f(x_{n-1})+f(x_n)}{2}h \\ &:= \frac{f(a)+f(a+h)}{2}h + \frac{f(a+h)+f(a+2h)}{2}h + \cdots + \frac{f(b-h)+f(b)}{2}h \\ &= \left[\frac{f(a)}{2} + f(a+h) + f(a+2h) + \cdots + f(b-h) + \frac{f(b)}{2}\right]h \end{aligned}$$

This is also a Riemann sum, with intervals if length $h/2$ at each end, using value at teh ends of thos intervals, and the rest of width h, with the Midpoint Rule used. So again, we know that $T_n \to I$ as $n \to \infty$ and next want to know "how fast?*

Accuracy and Error Formulas

In brief, the errors for ech of rhese rules is the sum of the errors for each of the pieces; I will just state them for now.

Firstly,

$$I - M_n = \sum_{i=1}^{n} \frac{h^3}{24} f''(\xi_i), \quad \text{for some } \xi_i \in [x_{i-1}, x_i]$$

This can be rewitten as

$$I - M_n = \frac{h^3}{24} \sum_{i=1}^{n} f''(\xi_i)$$

and as we will see, this sum can have each $f''(\xi_i)$ replace by an "average value" $f''(\xi_i)$, $\xi \in [a,b]$:

$$I - M_n = \frac{h^3}{24} \sum_{i=1}^{n} f''(\xi) = \frac{h^3}{24} n f''(\xi) = \frac{h^2}{24}(b-a) f''(\xi)$$

and the most important conclusion for now is that

$$I - M_n = O(h^2)$$

Similarly,

$$I - T_n = -\frac{h^2}{12}(b-a) f''(\xi) = O(h^2)$$

again with $\xi \in [a,b]$, but note well: these two ξ values are probably not the same!

5.4.2 Cancelling Some Error Terms: The Composite Simpson's Rule

Ignoring the ξ values being different, this suggests again that we can cancel some of the errors wi ha weighted average:

$$S_{2n} := \frac{2M_n + T_n}{3}$$

Indeed we will see that the main, $O(h^2)$, errors cancel out, and also due to symmetry, the error is even in h, so that

$$I - S_{2n} = O(h^4)$$

The name is because this is the *Composite Simpson's Rule*, and the interleaving of the different x values used by M_n and T_n means that is uses $2n+1$ nodes, and so $2n$ sub-intervals.

The Missing Step: A Generalized Mean Value Theorem

A key step in getting more useful error formulas for approximations of integrals is the following result:

Theorem 5.6 (Generalized Mean Value Theorem)

For any continuous function f on an interval $[a, b]$ and any collection of points $x_i \in [a, b]$, $1 \leq i \leq n$, there is a point $\xi \in [a, b]$ for which

$$f(c) = \frac{\sum_{i=1}^{n} f(x_i)}{n}, \text{ so } \sum_{i=1}^{n} f(x_i) = nf(c)$$

That is, the value of the function at c is the average of its values at those other points.

Proof. The proof is rather similar to that of *The Integral Mean Value Theorem* in the previous section; essentially replacing the integral there by a sum:

As f is continuous on the closed, bounded interval $[a, b]$, the **Extreme Value Theorem** from calculus says that f has a minimum L and a maximum H on this interval. Each of the values $f(x_i)$ is in interval $[L, H]$ so their average is also:

$$f(x_i) \in [L, H] \quad \text{and thus} \quad \frac{\sum_{i=1}^{n} f(x_i)}{n} \in [L, H]$$

The **Mean Value Theorem** then says that f attains this mean value for some $\xi \in [L, H]$.

Completing the derivation of the error formulas for these composite rules

I will spell this out for the Composite Trapezoid Rule; it works very similarly for the "midpoint" case.

First, break the exact integral up as

$$I = \int_a^b f(x)\,dx = \sum_{i=1}^{n} I^{(i)}, \quad \text{where} \quad I^{(i)} = \int_{x_{i-1}}^{x_i} f(x)\,dx$$

Similarly,

$$T_n = \sum_{i=1}^{n} T^{(i)}$$

where each $T^{(i)}$ is the Trapezoid Rule approximation of $I^{(i)}$:

$$T^{(i)} = \frac{f(x_{i-1}) + f(x_i)}{2} h$$

The error in T_n is the sum of the errors in each piece:

$$\begin{aligned} I - T_n &= \sum_{i=1}^{n} I^{(i)} - \sum_{i=1}^{n} T^{(i)} \\ &= \sum_{i=1}^{n} (I^{(i)} - T^{(i)}) \\ &= \sum_{i=1}^{n} -\frac{h^3}{12} f''(\xi_i), \quad x_i \in [x_{i-1}, x_i] \\ &= -\frac{h^3}{12} \sum_{i=1}^{n} f''(\xi_i) \end{aligned}$$

Now we can use the above mean value result (with f'' in place of f) to replace the last sum above by $nf''(\xi)$, some $\xi \in [a, b]$, so that as claimed,

$$I - T_n = -\frac{h^3}{12} n f''(\xi), = -\frac{h^2}{12}(b-a)f''(\xi) = O(h^2),$$

using $hn = b - a$.

Another error formula, useful for Richardson Extrapolation

Starting from

$$I - T_n = -\frac{h^3}{12}\sum_{i=1}^{n} f''(\xi_i), = -\frac{h^2}{12}\sum_{i=1}^{n}(f''(\xi_i)h)$$

note that the sum in the second version is a Riemann sum for approximating the integral

$$I'' := \int_a^b f''(x)\,dx, = [f'(x)]_a^b = f'(b) - f'(a),$$

so it seems that

$$I - T_n \approx -\frac{f'(b) - f'(a)}{12} h^2, = O(h^2)$$

A virtue of this form is that now we have a good chance of evaluating the coefficient of h^2, so this given a "practical error formula" when $f'(x)$ is known.

Another useful fact (not proven in these notes) is that the error for the basic Trapezoid rule can be computed with the help of Taylor's Theorem in a series:

$$T_1 = \int_a^b f(x)\,dx = B_2 D^2 f(\xi_2)h^3 + B_4 D^4(\xi_4)h^5 + \cdots$$

(where $B_2 = 1/12$ as seen above).

Putting the higher power terms into the above argument one can get

$$\begin{aligned} T_n &= \int_a^b f(x)\,dx + B_2[Df(b) - Df(a)]h^2 + B_4[D^3 f(b) - D^3 f(a)]h^4 + \cdots + B_{2k}[D^{2k-1}f(b) - D^{2k-1}f(a)]h^{2k} + \cdots \\ &= O(h^2) + O(h^4) + \cdots + O(h^{2k}) \end{aligned}$$

so that

$$T_n = \int_a^b f(x)\,dx + \frac{Df(b) - Df(a)}{12} h^2 + O(h^4)$$

The last form is the setup for Richardson extrapolation — and the previous one with a succession of "big-O" terms is the setup for *repeated* Richardson extrapolation, to get a succession of approximations with errors $O(h^2)$, then $O(h^4)$, then $O(h^6)$, and so on: *Definite Integrals, Part 4: Romberg Integration*

There are similar formulas for the Composite Midpoint Rule, like

$$I - M_n = \frac{h^2}{24}(b-a)f''(\xi) = \frac{Df(b) - Df(a)}{24} h^2 + O(h^4)$$

but we will see why the Composite Trapezoid Rule is far more useful for Richardson extrapolation.

5.4.3 Appendix: The Composite Left-hand Endpoint Rule, and its Error

The Composite Left-hand Endpoint Rule with n sub-intervals of equal width $h = (b-a)/n$ is

$$L_n = \sum_{i=0}^{n-1} f(x_i)h, \; = \sum_{i=0}^{n-1} f(a+ih)h$$

To study its errors, start as with the Compound Trapezoid Rule: break the integral up as

$$I = \int_a^b f(x)\,dx = \sum_{i=1}^{n} I^{(i)}, \quad \text{where} \quad I^{(i)} = \int_{x_{i-1}}^{x_i} f(x)\,dx$$

and the approximation as

$$L_n = \sum_{i=1}^{n} L^{(i)}$$

where each $L^{(i)}$ is the Left-hand Endpoint Rule approximation of $I^{(i)}$:

$$L^{(i)} = f(x_{i-1})h$$

Then the error in L_n is again the sum of the errors in each piece:

$$\begin{aligned} I - L_n &= \sum_{i=1}^{n} I^{(i)} - \sum_{i=1}^{n} L^{(i)} \\ &= \sum_{i=1}^{n} (I^{(i)} - L^{(i)}) \\ &= \sum_{i=1}^{n} \frac{h^2}{2} f'(\xi_i), \quad x_i \in [x_{i-1}, x_i] \\ &= \frac{h^2}{2} \sum_{i=1}^{n} f'(\xi_i) \end{aligned}$$

The Generalized Mean Value Theorem — now with f' in place of f — allows us to replace the last sum above by $nf'(\xi)$, some $\xi \in [a,b]$, so that as claimed,

$$I - L_n = \frac{h^2}{2} n f'(\xi), = \frac{h}{2}(b-a)f'(\xi) = O(h)$$

Remark 5.3

As with the Composite Trapezoid Rule, one can also get

$$L_n = \int_a^b f(x)\,dx + \frac{f(b)-f(a)}{2}h + O(h^2)$$

5.5 Definite Integrals, Part 3: The (Composite) Simpson's Rule and Richardson Extrapolation

References:

- Sections 5.2.2 and 5.2.3 of Chapter 5 *Numerical Differentiation and Integration* in [Sauer, 2019].
- Sections 4.3 and 4.4 of Chapter 5 *Numerical Differentiation and Integration* in [Burden *et al.*, 2016].

5.5.1 Introduction

The Composite Simpson's Rule can be be derived in several ways. The traditional approach is to devise Simpson's Rule by approximating the integrand function with a colocating quadratic (using three equally spaced nodes) and then "compounding", as seen with the Trapezoid and Midpoint Rules.

We have already seen another approach: using a 2:1 weighted average of the Trapezoid and Midpoint Rules with th goal of cancelling their $O(h^2)$ error terms.

This section will show a third approach, based on Richardson extrapolation: this will set us up for Romberg Integration.

5.5.2 The Basic Simpson's Rule by Richardson Extrapolation

From the section on *The Composite Trapezoid and Midpoint Rules*, we have

$$T_n = \int_a^b f(x)\,dx + \frac{Df(b) - Df(a)}{12}h^2 + O(h^4), = I + c_2h^2 + O(h^4)$$

where I is the integral to be approximated (the "Q" in the section on *Richardson Extrapolation*, and $c_2 = (Df(b) - Df(a))/12$.

Thus the "n form" of Richardson Extrapolation with $p = 2$ gives a new approximation that I will call S_{2n}:

$$S_{2n} = \frac{4T_{2n} - T_n}{4 - 1}$$

To start, look at the simplest case of this:

$$S_2 = \frac{4T_2 - T_1}{3}$$

Definfing $h = (b - a)/2$, the ingredients are

$$T_1 = \frac{f(a) + f(b)}{2}(b - a) = \frac{f(a) + f(b)}{2}2h = (f(a) + f(b))h$$

and

$$T_2 = \left[\frac{f(a)}{2} + f(a + h) + \frac{f(b)}{2}\right]h$$

so

$$S_2 = \frac{[2f(a) + 4f(a + h) + 2f(b)] - [f(a) + f(b)]}{3}h, = \frac{f(a) + 4f(a + h) + f(b)}{3}h$$

which is the basic Simpson's Rule. The subscript "2" is because this uses two intervals, with $h = (b - a)/2$

5.5.3 Accuracy and Order of Precision of Simpson's Rule

Rather than derive this the traditional way — by fitting a quadratic to the function values at $x = a$, $a + h$ and b — this can be confirmed "a postiori" by showing that the degree of precision is at least 2, so that it is exact for all quadratics. And actually we get a bonus, thanks to some symmetry.

For $f(x) = 1$, the exact integral is $I = b - a, = 2h$, and also

$$S_2 = \frac{1 + 4 \times 1 + 1}{3} h, = 2h$$

For $f(x) = x$, the exact integral is $I = \int_a^b x\,dx = [x^2/2]_a^b = (b^2 - a^2)/2 = (b - a)(b + a)/2 = (a + b)h$

and

$$S_2 = \frac{a + 4(a + b)/2 + b}{3} h = \frac{a + 2(a + b) + b}{3} h = (a + b)h$$

However, it is sufficient to traslate the domain to the symmetric interval $[-h, h]$, so redo the $f(x) = x$ case this easier way:

The exact integral is $\int_{-h}^{h} x\,dx = 0$ (because the function is odd)

$$S_2 = \frac{-h + 4 \times 0 + h}{3} h = 0$$

For $f(x) = x^2$, again do it just on the symmetric interval $[-h, h]$: the exact integral is $\int_{-h}^{h} x^2\,dx = [x^3/3]_{-h}^{h} = 2h^3/3$ and

$$S_2 = \frac{(-h)^2 + 4 \times 0^2 + h^2}{3} h = 2h^3/3$$

So the degree of precision is *at least* 2, as expected.

What about cubics? Check with $f(x) = x^3$, again on interval $[-h, h]$.

Almost no calculation is needed: symmetry does it all for us:

- on one hand, the exact integral is zero due to the function being odd on a symmetric interval: $\int_{-h}^{h} x^3\,dx = [x^4/4]_{-h}^{h} = 0$
- on the other hand,

$$S_2 = \frac{(-h)^3 + 4 \times 0^3 + h^3}{3} h = 0$$

The degree of precision is at least 3.

Our luck ends here, but looking at $f(x) = x^4$ is informative:

For $f(x) = x^4$,

- the exact integral is $\int_{-h}^{h} x^4\,dx = [x^5/5]_{-h}^{h} = 2h^5/5$
- on the other hand

$$S_2 = \frac{(-h)^4 + 4 \times 0^4 + h^4}{3} h = 2h^5/3$$

So there is a discrepancy of $(4/15)h^5, = O(h^5)$.

This Simpson's Rule has degree of precision 3: it is exact for all cubics, but not for all quartics.

The last result also indicate the order of error:

$$S_2 - I = O(h^5)$$

Just as for the composite Trapezoid and Midpoint Rules, when we combine multiple simple Simpson's Rule approximations with $2n$ intervals each of width $h = (b-a)/(2n)$, the error is roughly multiplied by n, so h^5 goes to $nh^5, = (b-a)h^4$, leading to

$$S_{2n} - I = O(h^4)$$

5.5.4 Appendix: Deriving Simpson's Rule by the Method of Undetermined Coefficients

We wish the determine the most accurate approximation of the form

$$\int_a^b f(x)\,dx \approx [C_1 f(a) + C_2 f(c) + C_3 f(b)]\,h$$

where c is the midpoint, $c = (a+b)/2$

This wilk be done by the first, "hardest" method: inserting Taylor polynomial and error terms, but to make it a bit less hard, we can consider just the symmetric case $a = -h$, $b = h$, $h = (b-a)/2$ by making the change of variables $x \to x - c$.

As we now know that this will be exact for cubics, use third order Tayloe polynomials:

$$f(\pm h) = f(0) \pm f'(0)h + \frac{f''(0)}{2}h^2 \pm \frac{f'''(0)}{6}h^3 + \frac{f''''(\xi_\pm)}{24}h^4$$

(Note that the special values $\xi_\pm$ are in general different for the "$+h$" and "$-h$" cases.

As usual, gather terms with the same power of h:

$$\begin{aligned} S_2 &= hf(0)(C_1 + C_2 + C_3) \\ &+ h^2 f^{(1)}(0)(-C_1 + C_3) \\ &+ h^3 f^{(2)}(0)(C_1/2 + C_3/2) \\ &+ h^4 f^{(3)}(0)(-C_1/6 + C_3/6) \\ &+ h^5 (C_1 f^{(4)}(\xi_-) + C_3 f^{(4)}(\xi_+))/24 \end{aligned}$$

The exact integral can also be computed with Taylor's formula:

$$\begin{aligned} I = \int_{-h}^{h} f(x)\,dx &= \int_{-h}^{h} \left[f(0) + Df(0)x + \frac{D^2 f(0)}{2}x^2 + \frac{D^3 f(0)}{6}x^3 + \frac{D^4 f(24)}{2}x^4 + \frac{D^5 f(\xi_x)}{120}x^5 \right] dx \\ &= 2hf(0) + \frac{D^2 f(0)}{3}h^3 + \frac{D^3 f(0)}{12}h^5 + O(h^6) \end{aligned}$$

(Symmetry causes all the odd power integrals to valish.)

so the error is

$$\begin{aligned} S_2 - I &= hf(0)(C_1 + C_2 + C_3 - 2) \\ &+ h^2 Df(0)(-C_1 + C_3) \\ &+ h^3 D^2 f(0)(C_1/2 + C_3/2 - 1/3) \\ &+ O(h^5) \end{aligned}$$

The best possibility is setting the coeficients of h, h^2 and h^3 to zero:

$$\begin{aligned} C_1 + C_2 + C_3 &= 2 \\ -C_1 + C_3 &= 0 \\ C_1/2 + C_3/2 &= 1/3 \end{aligned}$$

Symmetry helps, as the "h^2" equation $-C_1 + C_3 = 0$ gives $C_3 = C_1$, leaving

$$C_1 = 1/3, \quad 2C_1 + C_2 = 2$$

and thus

$$C_1 = C_3 = 1/3, \quad C_2 = 4/3$$

as claimed above.

5.6 Definite Integrals, Part 4: Romberg Integration

References:

- Section 5.3 *Romberg Integration* of [Sauer, 2019].
- Section 4.5 *Romberg Integration* of [Burden *et al.*, 2016].

5.6.1 Introduction

Romberg Integration is based on repeated Richardson extrapolalation from the composite trapezoidal rule, starting with one interval and repeatedly doubling. Our notation starts with

$$R_{i,0} = T_{2^i}, \quad i = 0, 1, 2, ...$$

where

$$T_n = \left(\frac{f(a)}{2} + \sum_{k=1}^{n-1} f(a + kh) + \frac{f(b)}{2} \right) h, \quad h = \frac{b-a}{n}$$

and the second index will indicate the number of extapolation steps done (none so far!)

Actually we only need this T_n formula for the single trapezoidal rule, to get

$$R_{0,0} = T_1 = \frac{f(a) + f(b)}{2}(b - a),$$

because the most efficient way to get the other values is recursively, with

$$T_{2n} = \frac{T_n + M_n}{2}$$

where M_n is the composite midpoint rule,

$$M_n = h \sum_{k=1}^{n} f(a + (k - 1/2)h), \quad h = \frac{b-a}{n}$$

Extrapolation is then done with the formula

$$R_{i,j} = \frac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1}, \quad j = 1, 2, ..., i$$

which can also be expressed as

$$R_{i,j} = R_{i,j-1} + E_{i,j-1}, \text{ where } E_{i,j-1} = \frac{R_{i,j-1} - R_{i-1,j-1}}{4^j - 1} \quad \text{is an error estimate.}$$

$$R_{i,1} = S_{2n} = \frac{4T_{2n} - T_n}{4 - 1}, \; n = 2^{i-1}$$

5.6.2 An algorithm, in pseudocode

The above can now be arranged into a basic algorithm. It does a fixed number M of levels of extrapolation so using 2^M intervals; a refinement would be to use the above error estimate $E_{i,j-1}$ as the basis for a stopping condition.

Algorithm 5.1 (Romberg Integration)

$n \leftarrow 1$

$h \leftarrow b - a$

$R_{0,0} = \dfrac{f(a) + f(b)}{2} h$

`for i from 1 to M:`

  $R_{i,0} = \left(R_{i-1,0} + h \sum_{k=1}^{n} f(a + (i - 1/2)h)\right)/2$

  `for j from 1 to i:`

    $R_{i,j} = \dfrac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1}$

  `end for`

  $n \leftarrow 2n$

  $h \leftarrow h/2$

`end for`

CHAPTER

SIX

MINIMIZATION

6.1 Finding the Minimum of a Function of One Variable Without Using Derivatives – under construction

References:

- Section 13.1 *Unconstrained Optimization Without Derivatives* of [Sauer, 2019], in particular sub-section 13.1.1 *Golden Section Search*.
- Section 11.1, *One-Variable Case* in Chapter 11 *Optimization* of [Chenney and Kincaid, 2012].

6.1.1 Introduction

The goal of this section is to find the minimum of a function $f(x)$ and more specifically to find its location: the argument p such that $f(p) \leq f(x)$ for all x in the domain of f.

Several features are similar to what we have seen with zero-finding:

- Some restictions on the function f are needed:
  - with zero-finding, to guarantee *existence* of a solution, we needed at least an interval $[a, b]$ on which the function is continuous and with a sign change between the endpoints;
  - for minimization, the criterion for existence is simply an interval $[a, b]$ on which the function is continuous.
- With zero-finding, we needed to compare the values of the function at three points $a < c < b$ to determine a new, smaller interval containing the root; with minimzation, we instead need to compare the values of the function at four points $a < c < d < b$ to determine a new, smaller interval containing the minimum.
- There are often good reasons to be able to do this without using derivatives.

As is often the case, a guarantee of a *unique* solution helps to devise a robust algorithm:

- to guarantee uniqueness of a zero in interval $[a, b]$, we needed an extra condition like the function being *monotonic*;
- to guarantee uniqueness of a minimum in interval $[a, b]$, the condition we use is being *monomodal*: The function is decreasing near a, increasing near b, and changes between decreasing and increasing only once (which must therefore happen at the minimum.)

So we assume from now on that the function is *monomodal* on the interval $[a, b]$.

6.1.2 Step 1: finding a smaller interval within $[a, b]$ that contains the minimum

As claimed above, three points are not enough: even if for $a < c < b$ we have $f(a) > f(c)$ and $f(c) < f(b)$, the minimum could be either to the left or the right of c.

So instead, choose two internal points c and d, $a < c < d < b$.

- if $f(c) < f(d)$, the function is increasing on at least part of the interval $[c, d]$, so the transition from decreasing to increasing is to the left of d: the minimum is in $[a, d]$;
- if instead $f(c) > f(d)$, the "mirror image" argument shows that the minimum is in $[c, b]$.

What about the borderline case when $f(c) = f(d)$? The monomodal function cannot be either increasing or decreasing on all of $[c, d]$ so must first decrease and then increase: the minimum is in $[c, d]$, and so is in either of the above intervals.

So we almost have a first algorithm, except for the isue of *choosing*; given an interval $[a, b]$ on which function f is monomodal:

1. *Choose* two internal points c and d, with $a < c < d < b$
2. Evaluate $f(c)$ and $f(d)$.
3. If $f(c) < f(d)$, replace the interval $[a, b]$ by $[a, d]$; else replace it by $[c, b]$.
4. If the new interval is short enough to locate the minimum with sufficient accuracy (e.g. its length is less that twice the error tolerance) stop; its midpoint is a sufficiently accurate approximate answer); othewise, repaeat from step (1).

6.1.3 Step 2: choosing the internal points so that the method is guaranteed to converge

There are a couple of details that need to be resolved:

(A) Deciding how to choose the internal points c and d.

(B) Verifying that the interval does indeed shrink to arbitrarily small length after enough iterations, so that the algorithm succeeds.

Once we have done that and got a working algorithm, there will be the issue of speed:

(C) Amongst the many ways that we could choose the internal points, finding one that (typically at least) is fastest, in the sense of minimizing the number of functions evaluations needed.

For now, I will just describe one "naive" approach that works, but is not optimal for speed; **Trisection:**

Take c and d to divide the interval $[a, b]$ into three equal-width sub-intervals: $c = (2a + b)/3$, $d = (a + 2b)/3$, so that each of $[a, c]$, $[c, d]$ and $[d, b]$ are of length $(b - a)/3$.

Then the new interval is $2/3$ as long as the previous one, and the errors shrink by a factor of $(2/3)^k$ after k steps, eventually getting as small as one wishes.

6.1.4 Step 3: choosing the internal points so that the method converges as fast as possible

Coming soon: this leads to the Golden Section Search …

6.2 Finding the Minimum of a Function of Several Variables — Coming Soon

References:

- Chapter 13 *Optimization* of [Sauer, 2019], in particular sub-sections 13.2.2 *Stepest Descent* and 13.1.3 *Nelder-Mead*.
- Chapter 11 *Optimization* of [Chenney and Kincaid, 2012].

6.2.1 Introduction

This future section will focus on two methods for computing the minimum (and its location) of a function $f(x, y, \dots)$ of several variables:

- *Steepest Descent* where the gradient is used iteratively to find the direction in which to search for a nw approxiate lovaitoi wher f has a lower value.
- The method of Nelder and Mead, which does not use derivatives.

CHAPTER

SEVEN

INITIAL VALUE PROBLEMS FOR ORDINARY DIFFERENTIAL EQUATIONS

7.1 Basic Concepts and Euler's Method

References:

- Sections 6.1.1 *Euler's Method* in [Sauer, 2019].
- Section 5.2 *Euler's Method* in [Burden *et al.*, 2016].
- Sections 7.1 and 7.2 in [Chenney and Kincaid, 2012].

```
using PyPlot
```

7.1.1 The Basic ODE Initial Value Problem

We consider the problem of solving (approx4imately) the ordinary differential equation

$$\frac{du}{dt} = f(t, u(t)), a \leq t \leq b \tag{7.1}$$

with the *initial condition*

$$u(a) = u_0 \tag{7.2}$$

I will follow the common custom of referring to the independent variable as "time".

For now, $u(t)$ is real-valued, but little will change when we later let it be vector-valued (and/or complex-valued).

Notation for the solution of an initial value problem

Sometimes, we need to be more careful and explicit in describing the function that solves the above initial value problem; then the input *parameters* a and $u_0 = u(a)$ will be included of the function's formula:

$$u(t) = u(t; a, u_0)$$

(It is standard mathematical convention to separate *parameters* like a and u_0 from *variables* like t by putting the former after a semicolon.

7.1.2 Examples

A lot of useful intuition comes from these four fairly simple examples.

Example (Integration)

If the derivative depends only on the independent variable t, so that

$$\frac{du}{dt} = f(t), a \le t \le b \tag{7.3}$$

the solution is given by integration:

$$u(t) = u_0 + \int_a^t f(s)\, ds.$$

In particular, with $u_0 = 0$ the value at b is

$$u(t) = \int_a^b f(t)\, dt,$$

and this gives us a back-door way to use numerical methods for solving ODEs to evaluate definite integrals.

Example (The simplest "real" ODE)

The simplest case with u present in f is $f(t, u) = f(u) = u$. But it does not hurt to add a constant, so:

$$\frac{du}{dt} = ku,\ k \text{ a constant.} \tag{7.4}$$

The solution is

$$u(t) = u_0 e^{k(t-a)}$$

We will see that this simple example contains the essence of ideas relevant far more generally.

Example (A nonlinear equation, with solutions that blow-up in a finite time)

In the previous examples, $f(t, u)$ is linear in u (consider t as fixed); nonlinearities can lead to more difficult behavior. The equation

$$\frac{du}{dt} = u^2,\ u(a) = u_0 \tag{7.5}$$

can be solved by separation of variables — or for now you can just verify the solution

$$u(t) = \frac{1}{T - t},\ T = a + 1/u_0.$$

Note that if $u_0 > 0$, the only exists for $t < T$. (The solution is also valid for $T > 0$, but that part has no connection to the initial data at $t = a$.)

Example 7.3 warns us that the IVP might not be **well-posed** when we set the interval $[a, b]$ in advance: all we can guarantee in general is that a solution exists up to some time b, $b > a$.

Example (A "stiff" equation with disparate time scales)

One common problem in practical situations is differential equations where some phenomena happen on a very fast time scale, but only ever at very small amplitudes, so they have very little relevance to the overall solution. One example is decriptions of some chemical reactions, where some reaction products (like free radicals) are producd in tiny quantities and break down very rapidly, so they change on a very fast time scale but are scarcely relevant to the overall solution.

This disparity of time-sales is called *stiffness*, from the analogy of a mechanical system in which some components are very stiff and so vibrate at very high frequencies, but typically only at very small amplitudes, or very quicky damped away, so that they can often be safely described by assuming that those stiff parts are completely rigid — do not move at all.

One equation that illustrates this feature is

$$\frac{du}{dt} = -\sin t - k(u - \cos t) \tag{7.6}$$

where k is large and positive. Its family of solutions is

$$u(t) = \cos t + ce^{-k(t-a)}$$

with $c = u_0 - \cos(a)$ for the initial value problem $u(a) = u_0$.

These all get close to $\cos t$ quickly and then stay nearby, but with a rapid and rapidly decaying "transient" ce^{-kt}.

Many of the most basic and widely use numerical methods (including Euler's Method thet we meet soon) need to use very small time steps to handle that fast transient, even when it is very small because $u_0 \approx 1$.

On the other hand there are methods that "supress" these transients, allowing use of larger time steps while still getting an accurate description of the main, slower, phenomena. The simplest of these is the *Backward Euler Method* that we will see in a later section.

7.1.3 The Tangent Line Method, a.k.a. Euler's Method

Once we know $u(t)$ (or a good approx4imation) at some time t, we also know the value of $u'(t) = f(t, u(t))$ there; in particular, we know that $u(a) = u_0$ and so $u'(a) = f(a, u_0)$.

This allows us to approx4imate u for slightly larger values of the argument (which I will call "time") using its tangent line:

$$u(a + h) \approx 4u(a) + u'(a)h = u_0 + f(a, u_0)h \text{ for "small" } h$$

and more generally

$$u(t + h) \approx 4u(t) + f(t, u(t))h \text{ for "small" } h$$

This leads to the simplest approx4imation: choose a step size h determining equally spaced times $t_i = a + ih$ and define — recursively — a sequence of approx4imations $U_i \approx 4u(t_i)$ with

$$\begin{aligned} U_0 &= u_0 \\ U_{i+1} &= U_i + hf(t_i, U_i)) \end{aligned}$$

If we choose a number of time steps n and set $h = (b-a)/n$ for $0 \le i \le n$, the second equation is needed for $0 \le i < n$, ending with $U_n \approx 4u(t_n) = u(b)$.

This "two-liner" does not need a pseudo-code description; instead, we can go directly to a rudimentary Julia function for Euler's Method:

```
function eulermethod(f, a, b, u_0, n)
    # Solve du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0

    h = (b-a)/n
    t = range(a, b, n+1)  # Note: "n" counts steps, so there are n+1 values for t.
    u = zeros(n+1)
    u[1] = u_0
    for i in 1:n
        u[i+1] = u[i] + f(t[i], u[i])*h
    end
    return (t, u)
end;
```

Exercise A

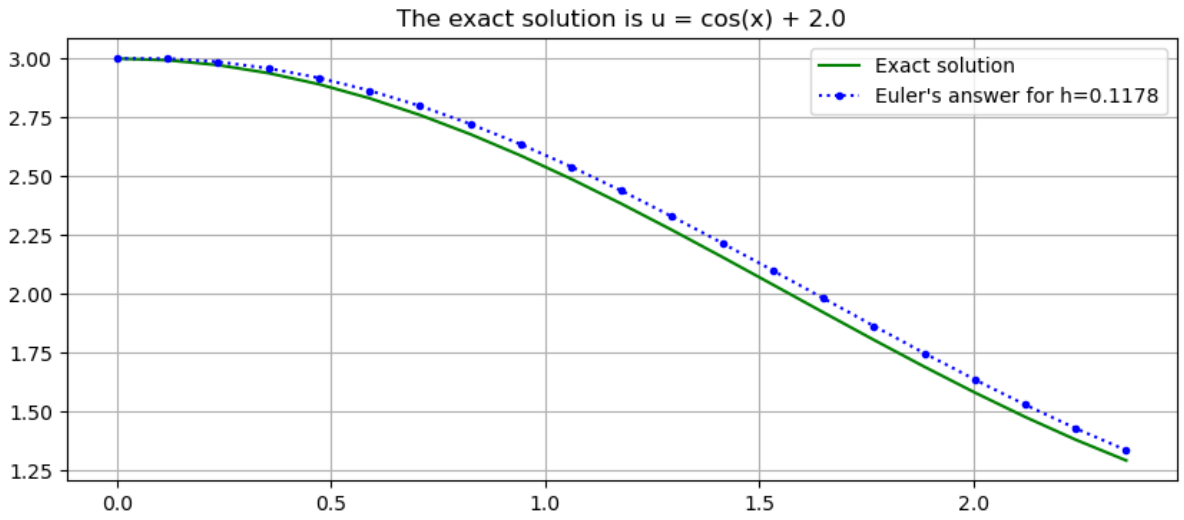
Show that for the integration case $f(t, u) = f(t)$, Euler's Method is the same as the Composite Left-hand Endpoint Rule, as in the section *Definite Integrals, Part 2: The Composite Trapezoid and Midpoint Rules*

Solving for Example 7.1, an integration

```
# For integration of -sin(t): The exact solution is cos(t) + u_0
f1(t, u) = -sin(t);
u1(t, a, u_0) = cos(t) + (u_0 - cos(a));
```

```
# A helper function for rounding some output to four significant digits
approx4(x) = round(x, sigdigits=4);
```

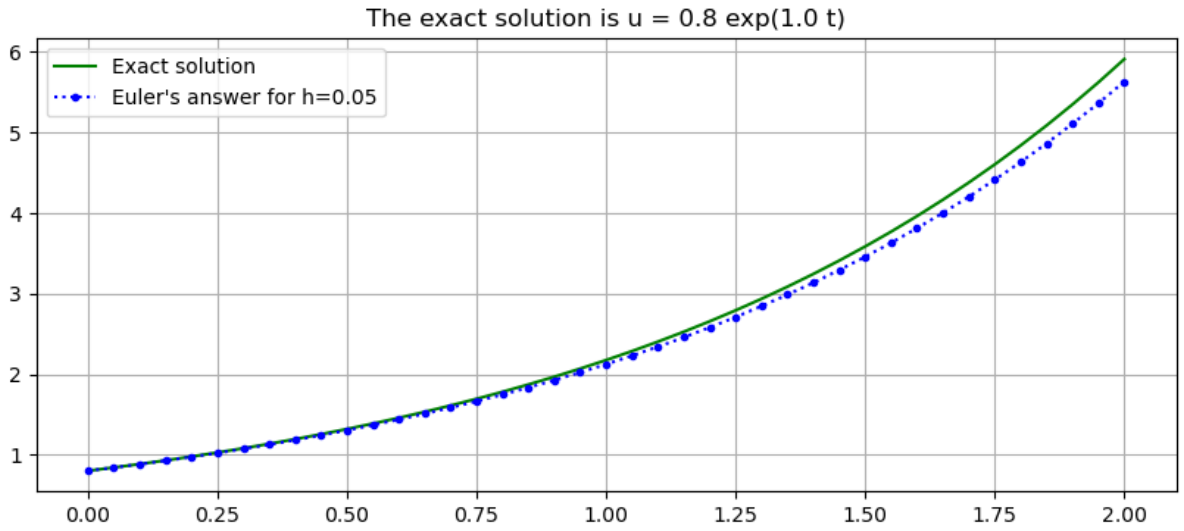
```
a = 0.0
b = 3/4*pi
u_0 = 3.0
n = 20

(t, U) = eulermethod(f1, a, b, u_0, n)
u = u1.(t, a, u_0)

figure(figsize=[10,4])
title("The exact solution is u = cos(x) + $(u_0 - 1)")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Euler's answer for h=$(approx4((b-a)/n))")
legend()
grid(true)
```



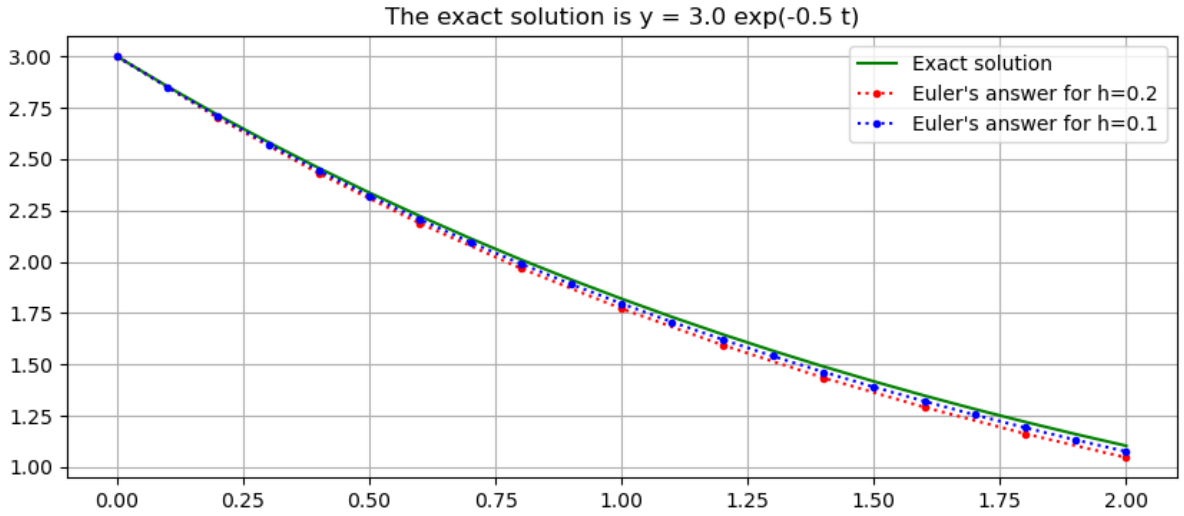
Solving for Example 7.2, some exponential functions

```
# For solving du/dt = k u: The exact solution is u_0 exp(k t).
f2(t, u) = k*u;
# The parameter k may be defined later, so long as that is done before this functions␣
 ↪are used.
u2(t, a, u_0, k) = u_0 * exp(k*(t-a));
```

```
# You could experiment by changing these values here;
# for now I instead redefine them below.
k = 1.0
u_0 = 0.8
a = 0.0
b = 2.0
n = 40

(t, U) = eulermethod(f2, a, b, u_0, n)
u = u2.(t, a, u_0, k)

figure(figsize=[10,4])
title("The exact solution is u = $u_0 exp($k t)")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Euler's answer for h=$(approx4((b-a)/n))")
legend()
grid(true)
```



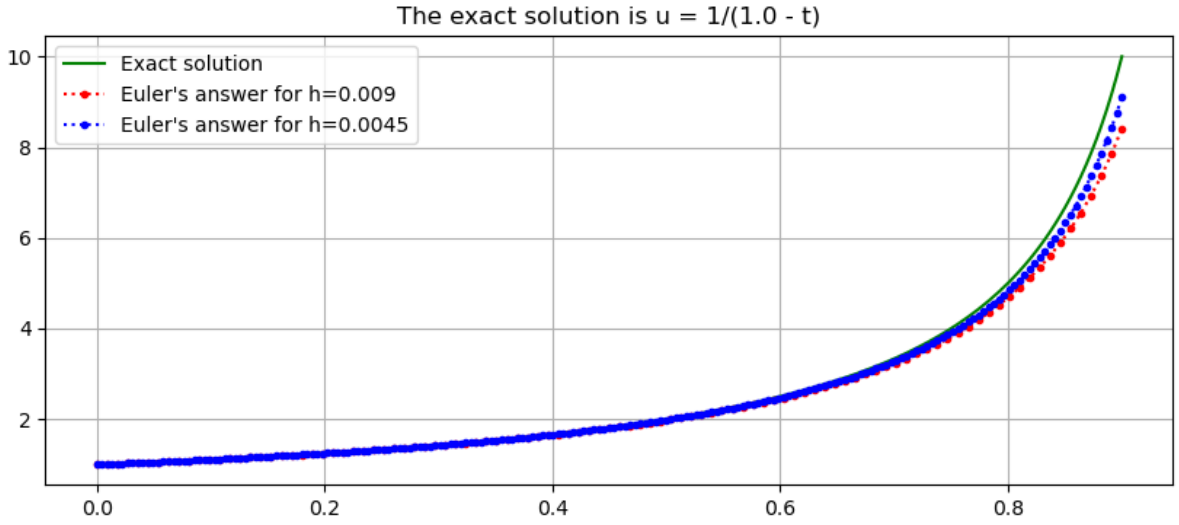

```
# You could experiment by changing these values here.
k = -0.5
u_0 = 3.0
a = 0.0
b = 2.0

(t, U) = eulermethod(f2, a, b, u_0, n)
(t10, U10) = eulermethod(f2, a, b, u_0, 10)
(t20, U20) = eulermethod(f2, a, b, u_0, 20)
t = t20
u = u2.(t, a, u_0, k)

figure(figsize=[10,4])
title("The exact solution is y = $u_0 exp($k t)")
plot(t, u, "g", label="Exact solution")
plot(t10, U10, ".:r", label="Euler's answer for h=$(approx4((b-a)/10))")
plot(t20, U20, ".:b", label="Euler's answer for h=$(approx4((b-a)/20))")
legend()
grid(true)
```



Solving for Example 7.3: solutions that blow up

```
f3(t, u) = u^2
# The solution is u(t) = 1/((a + 1/u_0) - t), = 1/(T-t) with T = a + 1/u_0
u3(t, a, u_0) = 1.0 / ((a + 1.0/u_0) - t);
```

```
a = 0.0
b = 0.9
u_0 = 1.0

(t100, U100) = eulermethod(f3, a, b, u_0, 100)
(t200, U200) = eulermethod(f3, a, b, u_0, 200)
t = t200
u = u3.(t, a, u_0)

T = a + 1/u_0

figure(figsize=[10,4])
title("The exact solution is u = 1/($T - t)")
plot(t, u, "g", label="Exact solution")
plot(t100, U100, ".:r", label="Euler's answer for h=$(approx4((b-a)/100))")
plot(t200, U200, ".:b", label="Euler's answer for h=$(approx4((b-a)/200))")
legend()
grid(true)
```



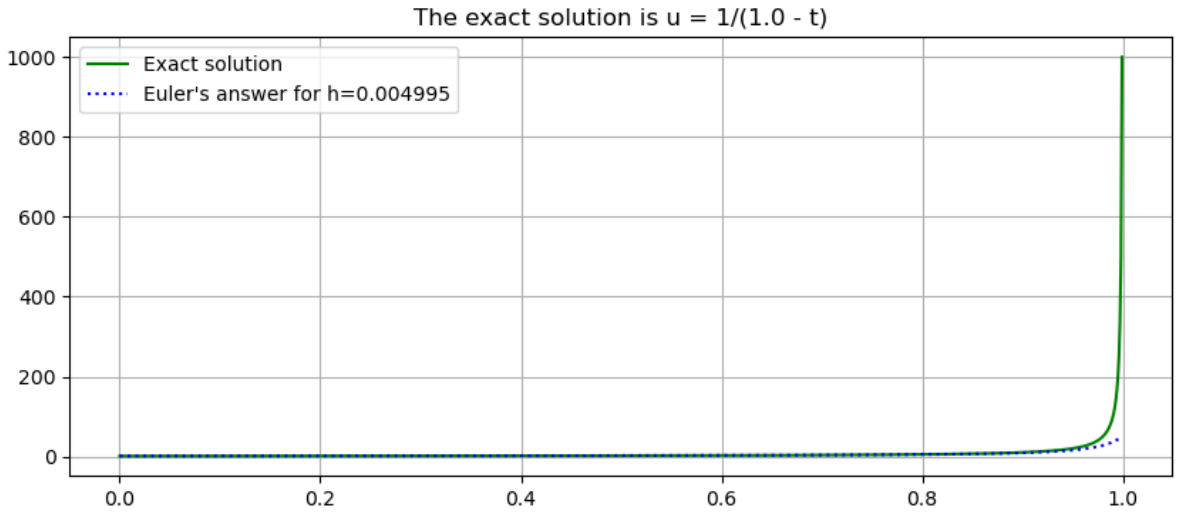
There is clearly a problem when t reaches 1; let us explore that:

```
a = 0.0
b = 0.999
u_0 = 1.0
n = 200

(t, U) = eulermethod(f3, a, b, u_0, n)
# More t values are needed to get a good graph of the exact solution near the␣
 ↪vertical asymptote:
tplot = range(a, b, 1000)
u = u3.(tplot, a, u_0)
```

(continues on next page)

(continued from previous page)

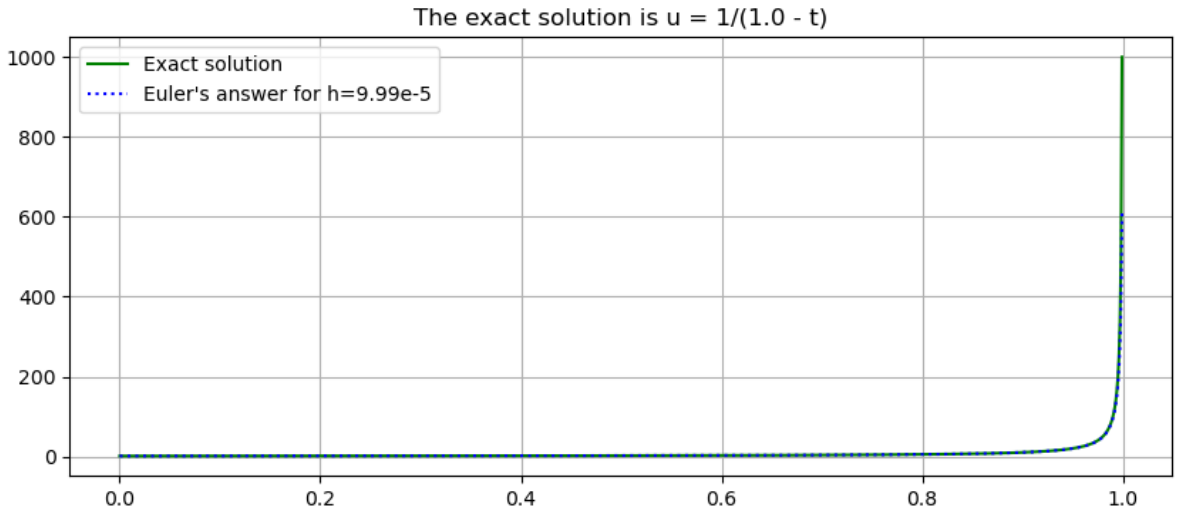
```
T = a + 1/u_0

figure(figsize=[10,4])
title("The exact solution is u = 1/($T - t)")
plot(tplot, u, "g", label="Exact solution")
plot(t, U, ":b", label="Euler's answer for h=$(approx4((b-a)/n))")
legend()
grid(true)
```



Clearly Euler's method can never produce the vertical asymptote.

The best we can do is improve accuracy by using more, smaller time steps:

```
b= 0.999
n = 10_000;  # Julia note: underscores can be used in numbers for readability, like␣
 ↪commas (or spaces in some countries)

(t, U) = eulermethod(f3, a, b, u_0, n)
tplot = range(a, b, 1000)
u = u3.(tplot, a, u_0)
T = a + 1/u_0

figure(figsize=[10,4])
title("The exact solution is u = 1/($T - t)")
plot(tplot, u, "g", label="Exact solution")
plot(t, U, ":b", label="Euler's answer for h=$(approx4((b-a)/n))")
legend()
grid(true)
```

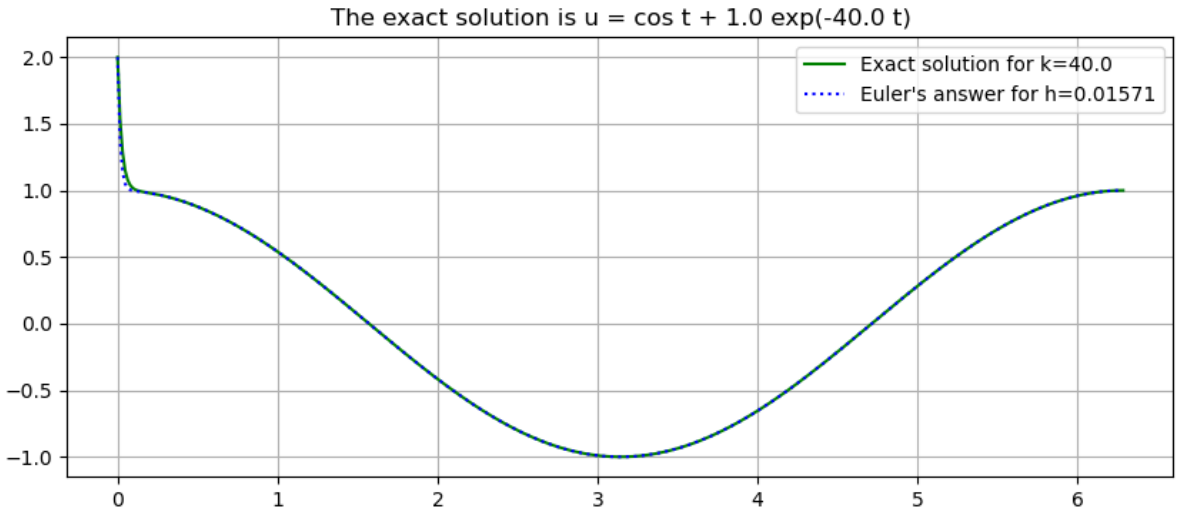


Solving for Solving for Example 7.4, a stiff ODE

```
f4(t, u) = -sin(t) - k*(u - cos(t));
# The parameter k may be defined later, so long as that is done before this function␣
↪is used.
# The general solution is u(t) = u(t; a, u_0) = cos t + (u_0 - cos(a)) e^(k (a-t))
u4(t, a, u_0, k) = cos(t) + (u_0 - cos(a)) * exp(k*(a-t));
```

With enough steps (small enough step size h), all is well:

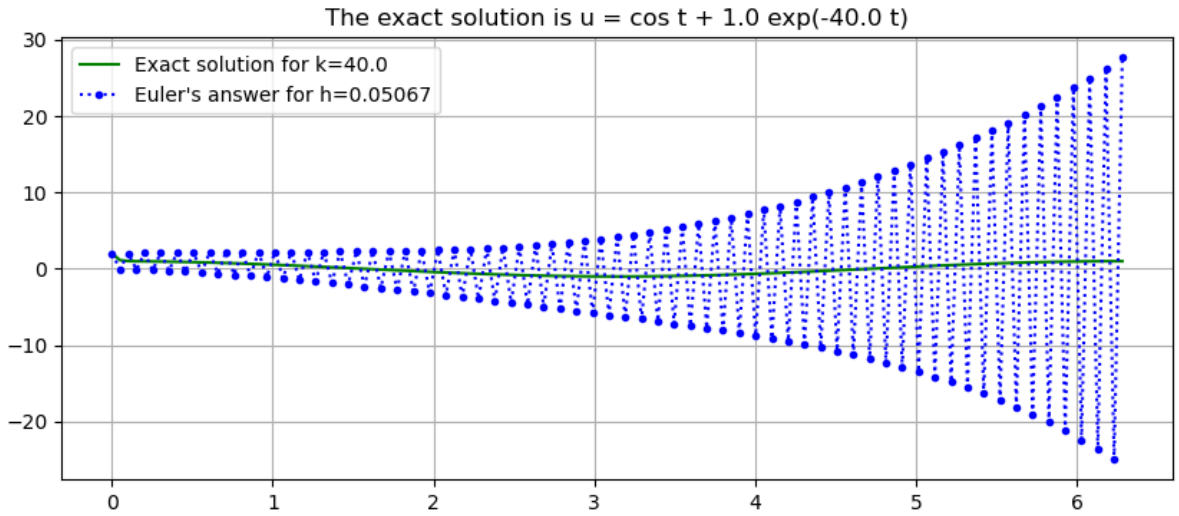
```
a = 0.0
b = 2pi  # One period
u_0 = 2.0
k = 40.0
n = 400

(t, U) = eulermethod(f4, a, b, u_0, n)
u = u4.(t, a, u_0, k)

figure(figsize=[10,4])
title("The exact solution is u = cos t + $(u_0-1) exp(-$k t)")
plot(t, u, "g", label="Exact solution for k=$k")
plot(t, U, ":b", label="Euler's answer for h=$(approx4((b-a)/n))")
legend()
grid(true);
```



However, with large steps (still small enough to handle the $\cos t$ part), there is a catastrophic failure, with growing oscillations.

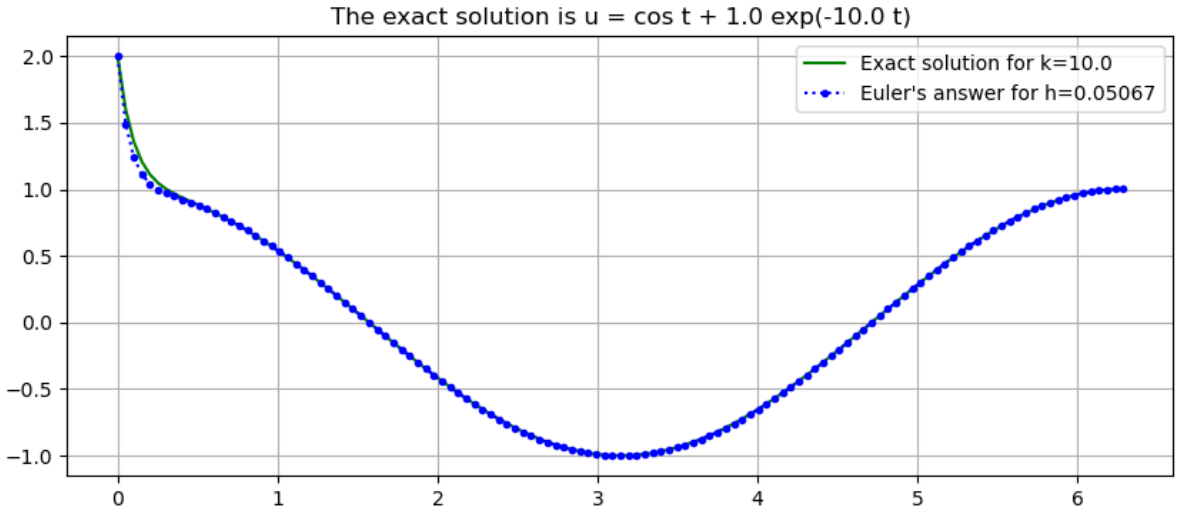
As we will see, these are a characteristic feature of *instability*.

```
n = 124

(t, U) = eulermethod(f4, a, b, u_0, n)
u = u4.(t, a, u_0, k)

figure(figsize=[10,4])
title("The exact solution is u = cos t + $(u_0-1) exp(-$k t)")
plot(t, u, "g", label="Exact solution for k=$k")
plot(t, U, ".:b", label="Euler's answer for h=$(approx4((b-a)/n))")
legend()
grid(true);
```



To show that the k part is the problem, reduce k while leaving the rest unchanged:

```
k = 10.0

(t, U) = eulermethod(f4, a, b, u_0, n)
u = u4.(t, a, u_0, k)

figure(figsize=[10,4])
title("The exact solution is u = cos t + $(u_0-1) exp(-$k t)")
plot(t, u, "g", label="Exact solution for k=$k")
plot(t, U, ".:b", label="Euler's answer for h=$(approx4((b-a)/n))")
legend()
grid(true);
```



Variable Time Step Sizes h_i (just a preview)

It is sometime useful to adjust the time step size; for example reducing it when the derivative is larger, (as happens in Example 3 above). This gives a slight variant, now expressed in pseudo-code:

Input: f, a, b, n $t_0 = a$ $U_0 = u_0$ for i from 1 to n Choose h_i somehow $t_i = t_{i-1} + h_i$ $U_i = U_{i-1} + h_i f(t_{i-1}, U_{i-1})$ end

In a later section, we will see how to estimate errors within an algorithm, and then how to use such error estimates to guide the choice of step size.

Error Analysis for the Canonical Test Case, $u' = ku$.

A great amount of intuition about numerical methods for solving ODE IVPs comes from that "simplest nontrivial example", number 2 above. We can solve it with constant step size h, and thus study its errors and accuracy. The recursion relation is now

$$U_i = U_{i-1} + hkU_{i-1} = U_{i-1}(1 + hk),$$

with solution

$$U_i = u_0(1 + hk)^i$$

For comparison, the exact solution of this ODE IVP is

$$u(t_i) = u_0 e^{k(t_i - a)} = u_0 e^{kih} = u_0 (e^{kh})^i$$

So each is a geometric series: the difference is that the *growth factor* is $G = (1 + hk)$ for Euler's method, vs $g = e^{kh} = 1 + hk + (hk)^2/2 + \cdots = 1 + hk + O(h^2)$ for the ODE.

Ths deviation at each time step is $O(h^2)$, suggesting that by the end $t = b$, at step n, the error will be $O(nh^2) = O\left(\frac{b-a}{h}h^2\right) = O(h)$.

This is in fact what happens, but to verify that, we must deal with the challenge that once an error enters at one step, it is potentially amplified at each subsequent step, so the errors introduced at each step do not simply get summed like they did with definite integrals.

Global Error and Local (Truncation) Error

Ultimately, the error we need to understand is the **global error**: at step i,

$$E_i = u(t_i) - U_i$$

We will approach this by first considering the new error added at each step, the **local truncation error** (or **discretization error**).

At the first step this is the same as above:

$$e_1 = u(t_1) - U_1 = u(a + h) - U_1$$

However at later steps we compare the results U_{i+1} to what the solution would be if it were exact at the start of that step: that is, if U_i were exact.

Using the notation $u(t; t_i, U_i)$ introduced above for the solution of the ODE with initial condition $u(t_i) = U_i$, the **location truncation error** at step i is the discrepancy at time t_{i+1} between what Euler's method and the exact solution give when both start at that point (t_i, U_i):

$$e_i = u(t; t_i, U_i) - U_{i+1}$$

Error propagation in $u' = ku$, $k \geq 0$.

After one step, $E_1 = u(t_1) - U_1 = e_1$.

At step 2,

$$E_2 = u(t_2) - U_2 = (u(t_2) - u(t_2, t_1, U_1) + (u(t_2, t_1, U_1) - U_2) = (u(t_2) - u(t_2, t_1, U_1) + e_2$$

The first term is the difference at $t = t_2$ of two solutions with values at $t = t_1$ being $u(t_1)$ and U_1 respectively. As the ODE is linear and homogeneous, this is the solution of the same ODE with value at $t = t_1$ being $u(t_1) - U_1$, which is e_1: that solution is $e_1 e^{y(t-t_1}$, so at $t = t_2$ it is $e_1 e^{kh}$. Thus the global error after two steps is

$$E_2 = e_2 + (e^{kh})e_1 :$$

the error from the previous step has been amplified by the growth factor $g = e^{kh}$:

$$E_2 = e_2 + ge_1 :$$

This continues, so that

$$E_3 = e_3 + gE_1 = e_3 + g(e_2 + ge_1) = e_3 + ge_2 + g^2 e_1$$

and so on, leading to

$$E_i = e_i + ge_{i-1} + g^2 e_{i-2} + \cdots + g^{i-1} e_1.$$

Bounding the local truncation errors ...

To get a bound on the global error from the formula above, we first need a bound on the local truncation errors e_i.

Taylor's theorem gives $e^{kh} = 1 + kh + e^{k\xi}(kh)^2/2$, $0 < \xi < kh$, so

$$e_i = U_i e^{kh} - U_i(1 + kh) = U_i(e^{k\xi}h^2/2)$$

and thus

$$|e_i| \le |U_i|\frac{e^{kh}}{2}h^2$$

Also, since $1+kh < e^{kh}$, $|U_i| < |u(t_i)| = |u_0|e^{k(t_i-a)}$, and we only need this to the beginning of the last step, $i \le n-1$, for which

$$|U_i| < |u_0|e^{k(b-h-a)}$$

Thus

$$|e_i| \le \frac{|u_0|e^{k(b-h-a)}e^{kh}}{2}h^2 = \frac{|u_0 e^{k(b-a)}|}{2}h^2$$

That is,

$$|e_i| \le Ch^2 \text{ where } C := \frac{|u_0 e^{k(b-a)}|}{2}$$

... and using this to complete the bound on the global truncation error

Using this bound on the local errors e_i in the above sum for the global error E_i,

$$|E_i| \le Ch^2(1 + g + \cdots g^{i-1}) = C\frac{g^i - 1}{g - 1}h^2$$

Since $g^i = e^{khi} = e^{k(t_i-a)}$ and the denominator $g - 1 = e^{kh} - 1 > kh$, we get

$$|E_i| \le C\frac{e^{k(t_i-a)} - 1}{kh}h^2 \le \frac{|u_0 e^{k(b-a)}|}{2}\frac{e^{k(t_i-a)} - 1}{k}h, = O(h)$$

Note that this global error formula is bulilt from three factors:

- The first is the constant $\frac{|u_0 e^{k(b-a)}|}{2}$ which is roughly half of the maximum value of the exact solution over the interval $[a, b]$.
- The second $\frac{e^{k(t_i-a)} - 1}{k}$ depends on t, and
- The third is h, showing the overall order of accuracy: the overall absolute error is $O(h)$, so first order.

A more general error bound

A very similar result applies to the solution $u(t; a, u_0)$ of the more general initial value problem

$$\frac{du}{dt} = f(t, u), \quad u(a) = u_0$$

so long as the function f is "somewhat well-behaved" in that it satisfies a so-called *Lipschitz Condition*: that there is some constant K such that

$$\left|\frac{\partial F}{\partial u}(t,u)\right| \leq K$$

for the relevant time values $a \leq t \leq b$.

(**Aside:** As you might have seen in a course on differential equations, such a Lipschitz condition is necessary to even guarantee that the initial value problem has a unique solution, so it is a quite reasonable requirement.)

Then this constant K plays the part of the exponential growth factor k above:

first one shows that the local trunction error is bounded by

$$|e_i| \leq Ch^2 \text{ where now } C := \frac{|u_0 e^{K(b-a)}|}{2};$$

then calculating as above bounds the global truncation error with

$$|E_i| \leq \frac{|u_0 e^{K(b-a)}|}{2} \frac{e^{K(t_i - a)} - 1}{k} h, = O(h)$$

There is much room for improvement

As with definite integrals, this is not very impressive, so in the next section on *Runge-Kutta Methods* we will explore several widely used methods that improve to second order and then fourth order accuracy. Later, we will see how to get even higher orders.

But first, we can illustrate how this exponential growth of errors looks in some examples, and coapr the the better behaved errors in definite integrals.

This will be done by looking at the effect of a small change in the initial value, to simulate an error that arises there.

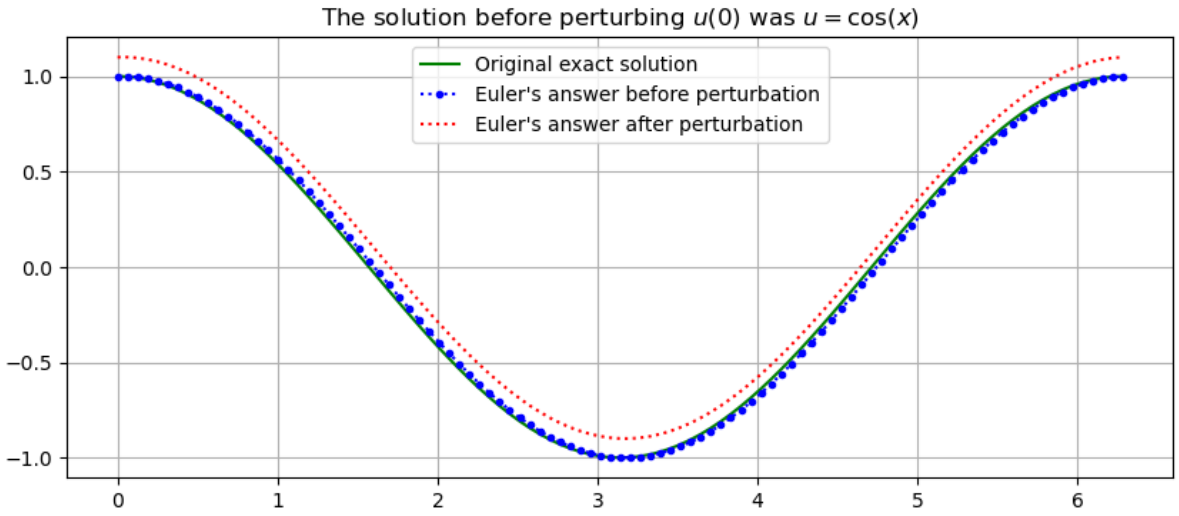
Error propagation for Example 7.1

```
a = 0.0
b = 2pi
u_0 = 1.0  # Original value
n = 100

(t, U) = eulermethod(f1, a, b, u_0, n);
```

But now "perturb" the initial value in all cases by this much:

```
delta_u_0 = 0.1
(t, U_perturbed) = eulermethod(f1, a, b, u_0+delta_u_0, n)
u = u1.(t, a, u_0);
```

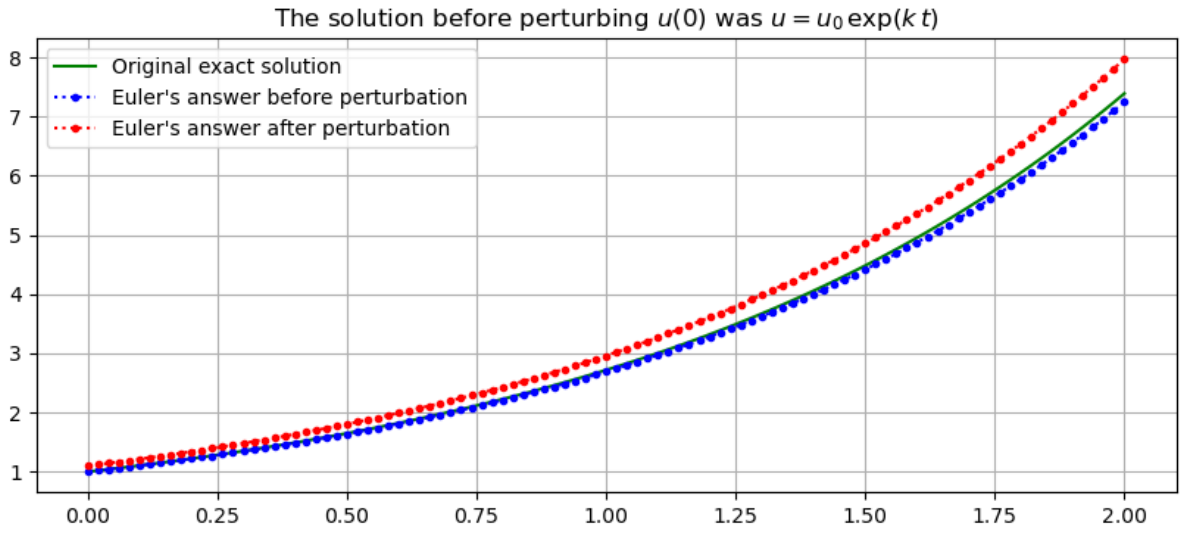
```
figure(figsize=[10,4])
title(L"The solution before perturbing $u(0)$ was $u = \cos(x)$")
plot(t, u, "g", label="Original exact solution")
plot(t, U, ".:b", label="Euler's answer before perturbation")
plot(t, U_perturbed, "r:", label="Euler's answer after perturbation")
legend()
grid(true)
```



This just shifts all the u values up by the perturbation of u_0.

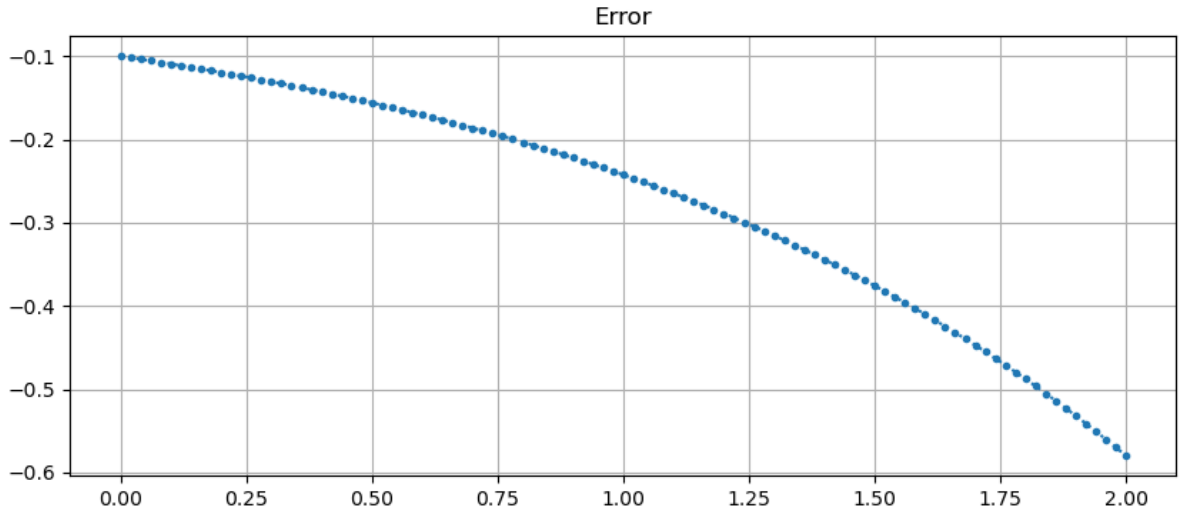
Error propagation for Example 7.2

```
k = 1.0
a = 0.0
b = 2.0
u_0 = 1.0  # Original value
delta_u_0 = 0.1
n = 100

(t, U) = eulermethod(f2, a, b, u_0, n)
(t, U_perturbed) = eulermethod(f2, a, b, u_0 + delta_u_0, n)
u = u2.(t, a, u_0, k)

figure(figsize=[10,4])
title(L"The solution before perturbing $u(0)$ was $u = {u_0} \, \exp({k} \, t)$")
plot(t, u, "g", label="Original exact solution")
plot(t, U, ".:b", label="Euler's answer before perturbation")
plot(t, U_perturbed, ".:r", label="Euler's answer after perturbation")
legend()
grid(true)
```



Graphing the error shows its exponential growth:

```
figure(figsize=[10,4])
title("Error")
plot(t, u - U_perturbed, ".:")
grid(true)
```



7.2 Runge-Kutta Methods

References:

- Sections 6.4 *Runge-Kutta Methods and Applicaitons* in [Sauer, 2019].
- Section 5.4 *Runge-Kutta Methods* in [Burden *et al.*, 2016].
- Sections 7.1 and 7.2 in [Chenney and Kincaid, 2012].

```
using PyPlot
```

7.2.1 Introduction

The original Runge-Kutta method is the fourth order accurate one to be described below, which is still used a lot, though with some modifications.

However, the name is now applied to a variety of methods based on a similar strategy, so first, here are a few simpler methods, all of some value, at least for small, low precision calculations.

7.2.2 Euler's Method as a Runge-Kutta method

The simplest of all methods of this general form is Euler's method. To set up the notation to be used below, rephrase it this way:

To get from (t, u) to an approximation of $(t + h, u(t + h))$, use the approximation

$$\begin{aligned} K_1 &= hf(t, u) \\ u(t + h) &\approx u + K_1 \end{aligned}$$

7.2.3 Second order Runge-Kutta methods

We have seen that the global error of Euler's method is $O(h)$: it is only first order accurate. This is often insufficient, so it is more common even for small, low precision calculation to use one of several second order methods:

The Explicit Trapezoid Method (a.k.a. the Improved Euler method or Huen's method)

One could try to adapt the trapezoid method for integrating $f(t)$ to solve $du/dt = f(t)$

$$u(t + h) = u(t) + \int_t^{t+h} f(s)ds \approx u(t) + h\frac{f(t) + f(t + h)}{2}$$

to solving the ODE $du/dt = f(t, u)$ but there is a problem that needs to be overcome:

we get

$$u(t + h) \approx u(t) + h\frac{f(t, u(t)) + hf(t + h, u(t + h))}{2}$$

and inserting the values $U_i \approx u(t_i)$ and so on gives

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

This is known as the **Implicit Trapezoid Method**, because the value U_{i+1} that we seek appears at the right-hand side too: we only have an *implicit* formula for it.

On one hand, one can in fact use this formula, by solving the equation at each time step for the unknown U_{i+1}; for example, one can use methods seen in earlier sections such as fixed point iteration or the secant method.

We will return to this in a later section; however, for now we get around this more simply by inserting an approximation at right — the only one we know so far, given by Euler's Method. That is:

- replace $u(t + h)$ at right by the tangent line approximation $u(t + h) \approx u(t) + hf(t, u(t))$, giving

$$u(t+h) \approx u(t) + h\frac{f(t,u(t)) + f(t+h, u(t) + hf(t,u(t)))}{2}$$

and for the formulas in terms of the U_i, replace U_{i+1} at right by $U_{i+1} \approx U_i + hf(t_i, U_i)$, giving

$$U_{i+1} = U_i + h\frac{f(t_i,U_i) + f(t_{i+1}, U_i + hf(t_i,U_i))}{2}$$

This is the **Explicit Trapezoid Method**.

It is convenient to break this down into two stages, one for each evaluation of $f(t,u)$:

$$\begin{aligned} K_1 &= hf(t,u) \\ K_2 &= hf(t+h, u+K_1) \\ u(t+h) &\approx u + \frac{1}{2}(K_1 + K_2) \end{aligned}$$

For equal sized time steps, this leads to

Algorithm 7.1 (The Explicit Trapezoid Method)

$$\begin{aligned} U_0 &= u_0 \\ U_{i+1} &= U_i + \frac{1}{2}(K_1 + K_2), \\ &\text{where} \\ K_1 &= hf(t_i, U_i) \\ K_2 &= hf(t_{i+1}, U_i + K_1) \end{aligned}$$

We will see that, despite the mere first order accuracy of the Euler approximation used in getting K_2, this method is second order accurate; the key is the fact that any error in the approximation used for $f(t+h, u(t+h))$ gets multiplied by h.

See Exercise 1

```
function explicittrapezoid(f, a, b, u_0, n; demomode=false)
    # Use the Explict Trapezoid Method (a.k.a Improved Euler) to solve
    #     du/dt = f(t, u)
    # for t in [a, b], with initial value u(a) = u_0

    h = (b-a)/n
    t = range(a, b, n+1)  # Note: "n" counts steps, so there are n+1 values for t.
    u = zeros(n+1)
    u[1] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i])*h
        K_2 = f(t[i]+h, u[i]+K_1)*h
        u[i+1] = u[i] + (K_1 + K_2)/2.0
    end
    return (t, u)
end;
```

As always, this function can now also be imported from module `NumericalMethods` with

```
include("NumericalMethods.jl")
using .NumericalMethods: explicitTrapezoid
```

Examples

For all methods in this section, we will solve for versions of *Example 7.2* and *Example 7.4* in *Basic Concepts and Euler's Method*.

$$\frac{du}{dt} = f_1(t,u) = ku \tag{7.7}$$

with general solution

$$u(t) = u_1(t; a, u_0, k) = u_0 e^{k(t-a)} \tag{7.8}$$

and

$$\frac{du}{dt} = f_2(t,u) = k(\cos(t) - u) - \sin(t) \tag{7.9}$$

with general solution

$$u(t) = u_2(t; a, u_0, k) = \cos t + ce^{-k(t-a)},\ c = u_0 - \cos(a) \tag{7.10}$$

For comparison to Euler's Method, the same examples are done with it *below*.

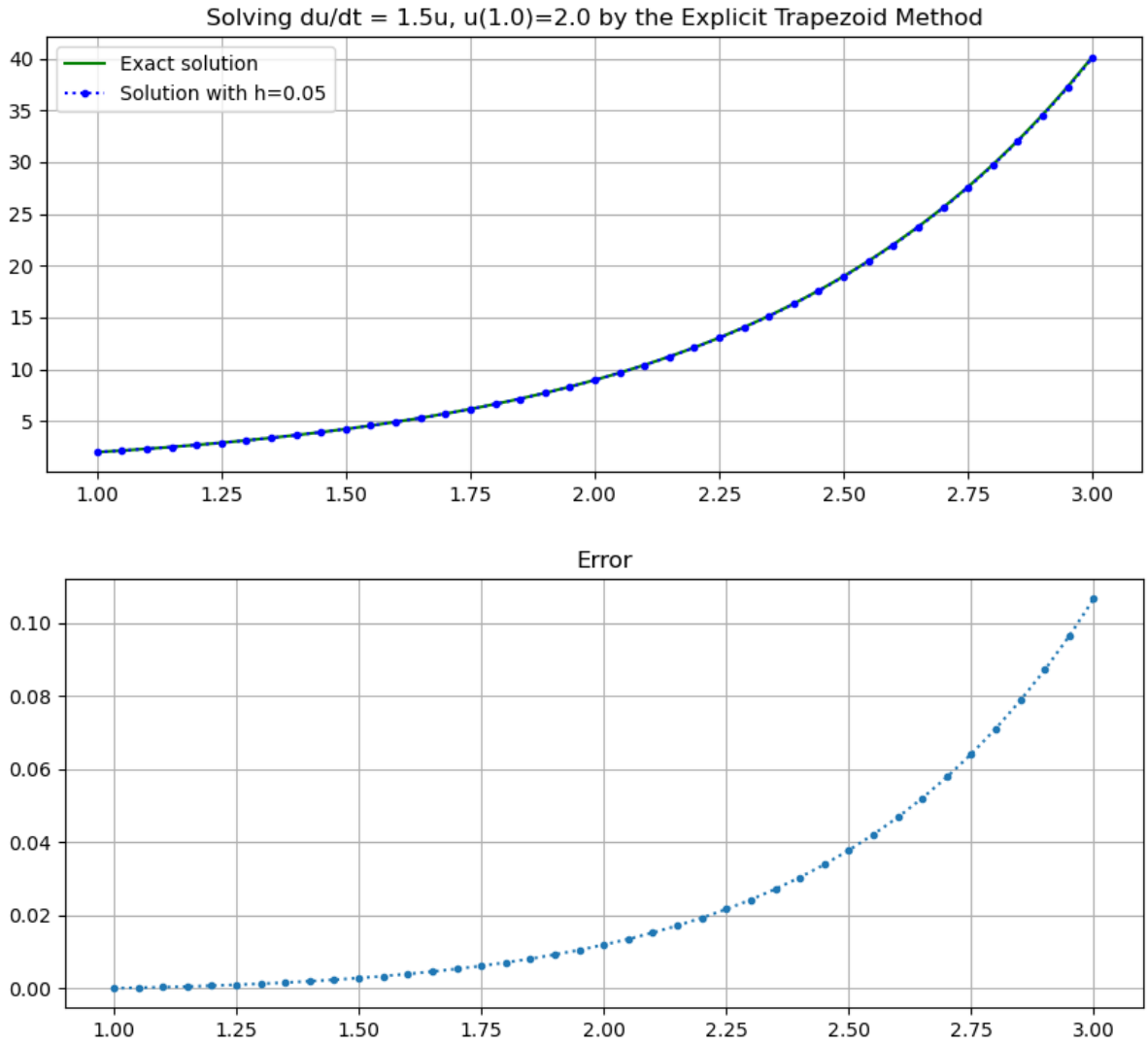
```
# A helper function for rounding some output to four significant digits
approx(x) = round(x, sigdigits=4);
```

Example 7.5

Let us first solve the simple ODE (7.7) from *Example 7.2*.

```
f1(t, u) = k*u
# The simplest "genuine" ODE, (not just integration):
# The solution is u(t) = u(t; a, u_0) = u_0 exp(t-a)
u1(t, u_0, k) = u_0 * exp(k*(t-a));
```

```
a = 1.0
b = 3.0
u_0 = 2.0
k = 1.5
n = 40

(t, U) = explicittrapezoid(f1, a, b, u_0, n; demomode=true)
u = u1.(t, u_0, k)
figure(figsize=[10,4])
title("Solving du/dt = $(k)u, u($a)=$u_0 by the Explicit Trapezoid Method")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Solution with h=$(approx(b-a)/n)")
legend()
grid(true)

figure(figsize=[10,4])
title("Error")
plot(t, u - U, ".:")
grid(true)
```



Example 7.6

Solve the stiff ODE (7.9) from *Example 7.4*.

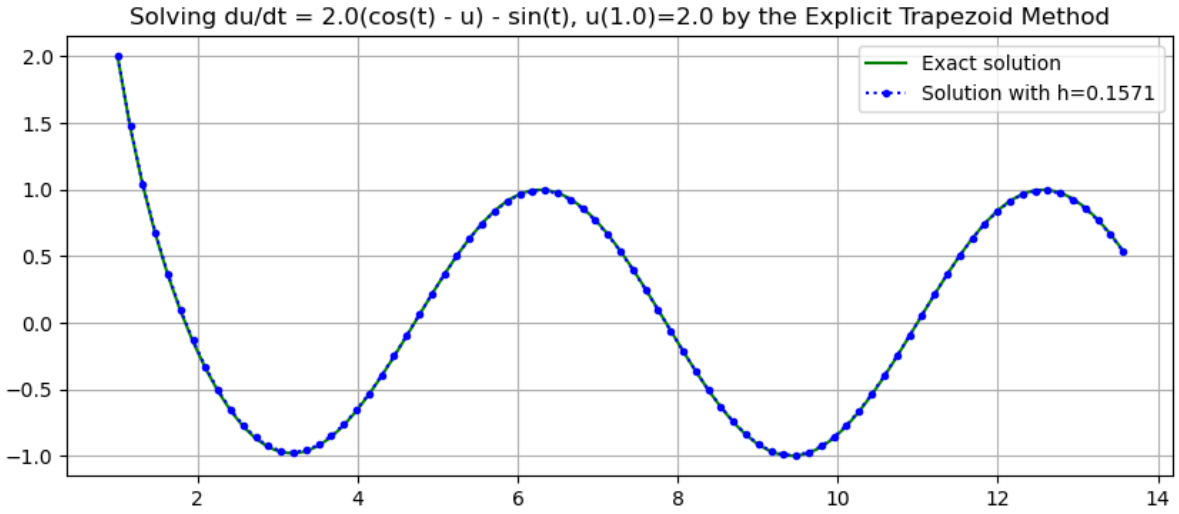
```
f2(t, u) = k*(cos(t) - u) - sin(t)
# A simple more "generic" test case, with f(t, u) depending on both variables.
# The general solution is u(t) = u(t; a, u_0) = cos t + (u_0 - cos(a)) e^(k (a-t))
u2(t, a, u_0, k) = cos(t) + (u_0 - cos(a)) * exp(k*(a-t));
```

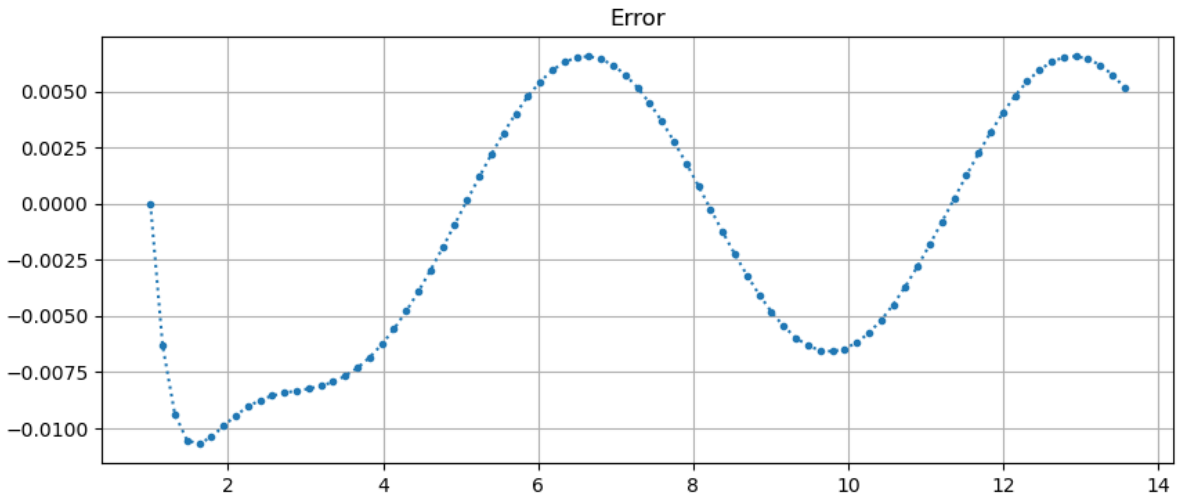
```
a = 1.0
b = a + 4pi  # Two periods
u_0 = 2.0
k = 2.0
n = 80

(t, U) = explicittrapezoid(f2, a, b, u_0, n)
u = u2.(t, a, u_0, k)
```

(continues on next page)

(continued from previous page)

```
figure(figsize=[10,4])
title("Solving du/dt = $k(cos(t) - u) - sin(t), u($a)=$u_0 by the Explicit Trapezoid␣
 ↪Method")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Solution with h=$(approx((b-a)/n))")
legend()
grid(true)

figure(figsize=[10,4])
title("Error")
plot(t, u - U, ".:")
grid(true)
```





The Explicit Midpoint Method (a.k.a. Modified Euler)

If we start with the Midpoint Rule for integration in place of the Trapezoid Rule, we similarly get an approximation

$$u(t+h) \approx u(t) + hf(t+h/2, u(t+h/2))$$

This has the slight extra complicatio that it involves three values of u including $u(t+h/2)$ which we are not trying to evaluate. We deal with that by making yet another approximation, using an average of u values:

$$u(t+h/2) \approx \frac{u(t)+u(t+h)}{2}$$

leading to

$$u(t+h) \approx u(t) + hf\left(t+h/2, \frac{u(t)+u(t+h)}{2}\right)$$

and in terms of $U_i \approx u(t_i)$, the **Implicit Midpoint Method**

$$U_{i+1} = U_i + hf\left(t+h/2, \frac{U_i+U_{i+1}}{2}\right)$$

We will see in a later section that this is a particularly useful method in some situations, such as long-time solutions of ODEs that describe the motion of physical systems with conservation of momentum, angular momentum and kinetic energy.

However, for now we again seek a more straightforward *explicit* method; using the same tangent line approximation strategy as above gives

$$\begin{aligned} K_1 &= hf(t,u) \\ K_2 &= hf(t+h/2, u+K_1/2) \\ u(t+h) &\approx u + K_2 \end{aligned}$$

and thus for equal-sized time steps

Algorithm 7.2 (The Explicit Midpoint Method)

$$\begin{aligned} U_0 &= u_0 \\ U_{i+1} &= U_i + K_2 \\ &\text{where} \\ K_1 &= hf(t_i, U_i) \\ K_2 &= hf(t_i + h/2, U_i + K_1/2) \end{aligned}$$

See Exercise 2

See Exercise 3

```
function explicitmidpoint(f, a, b, u_0, n; demomode=false)
    # Use the Explicit Midpoint Method (a.k.a Modified Euler) to solve
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0

    h = (b-a)/n
    t = range(a, b, n+1)  # Note: "n" counts steps, so there are n+1 values for t.
```

(continues on next page)

(continued from previous page)

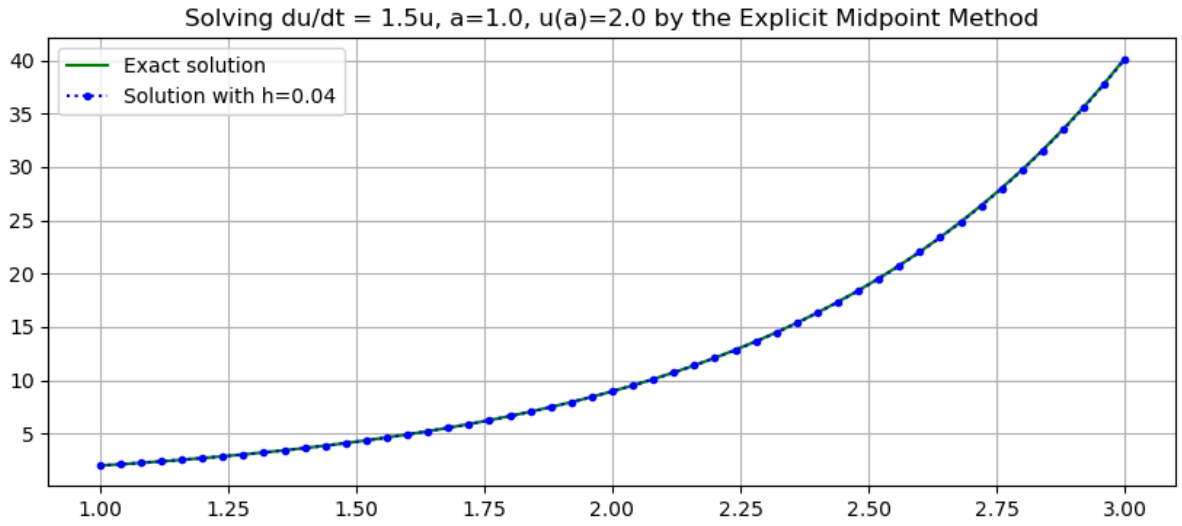
```
    u = zeros(length(t))
    u[1] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i])*h
        K_2 = f(t[i]+h/2, u[i]+K_1/2)*h
        u[i+1] = u[i] + K_2
    end
    return (t, u)
end;
```

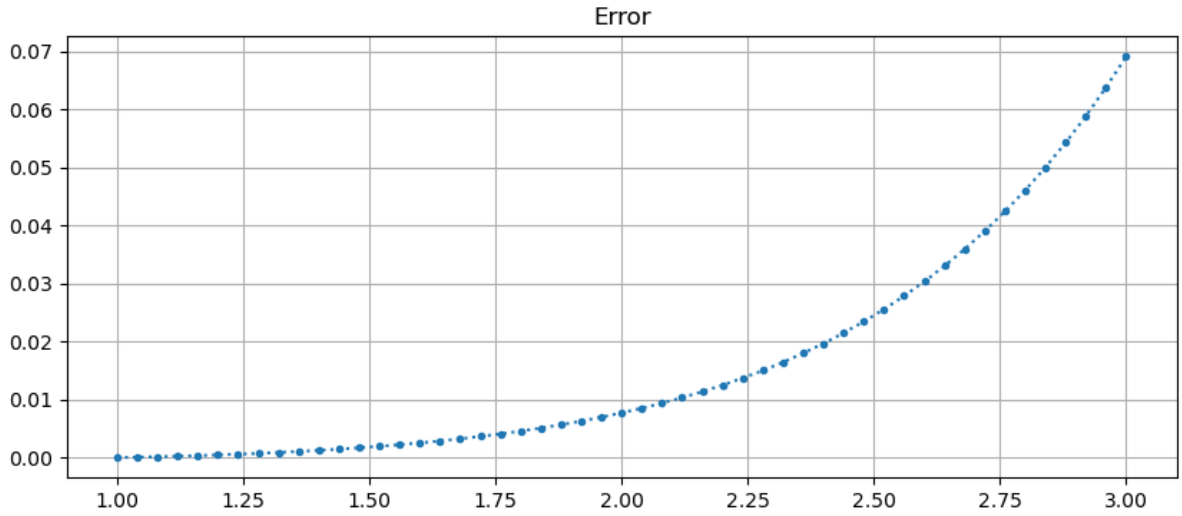
Again, available for import with

```
include("NumericalMethods.jl")
import .NumericalMethods: explicitmidpoint
```

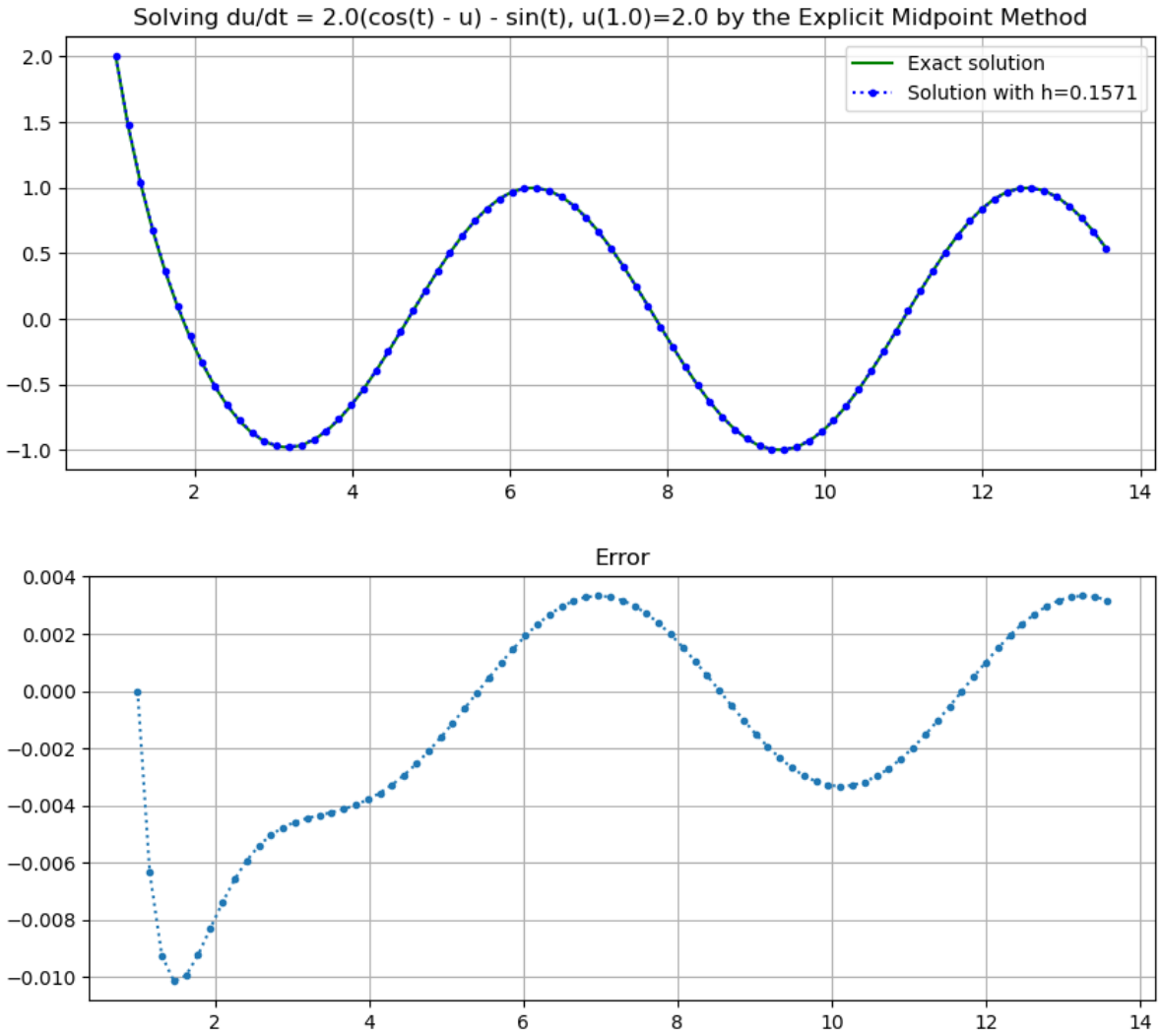
Examples

```
a = 1.0
b = 3.0
u_0 = 2.0
k = 1.5
n = 50

(t, U) = explicitmidpoint(f1, a, b, u_0, n; demomode=true)
u = u1.(t, u_0, k)
figure(figsize=[10,4])
title("Solving du/dt = $(k)u, a=$a, u(a)=$u_0 by the Explicit Midpoint Method")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Solution with h=$(approx((b-a)/n))")
legend()
grid(true)

figure(figsize=[10,4])
title("Error")
plot(t, u - U, ".:")
grid(true)
```






```
a = 1.0
b = a + 4pi  # Two periods
u_0 = 2.0
k = 2.0
n = 80

(t, U) = explicitmidpoint(f2, a, b, u_0, n)
u = u2.(t, a, u_0, k)
figure(figsize=[10,4])
title("Solving du/dt = $k(cos(t) - u) - sin(t), u($a)=$u_0 by the Explicit Midpoint␣
↪Method")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Solution with h=$(approx((b-a)/n))")
legend()
grid(true)

figure(figsize=[10,4])
title("Error")
plot(t, u - U, ".:")
grid(true)
```



7.2.4 The "Classical", Fourth Order Accurate, Runge-Kutta Method

This is the original Runge-Kutta method:

Algorithm 7.3 (The Runge-Kutta Method)

$$
\begin{aligned}
K_1 &= hf(t, u) \\
K_2 &= hf(t + h/2, u + K_1/2) \\
K_3 &= hf(t + h/2, u + K_2/2) \\
K_4 &= hf(t + h, u + K_3) \\
u(t + h) &\approx u + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)
\end{aligned}
$$

The derivation of this is far more complicated than those above, and is omitted. For now, we will instead assess its accuracy "*a postiori*", through the next exercise and some examples.

See *Exercise 4*.

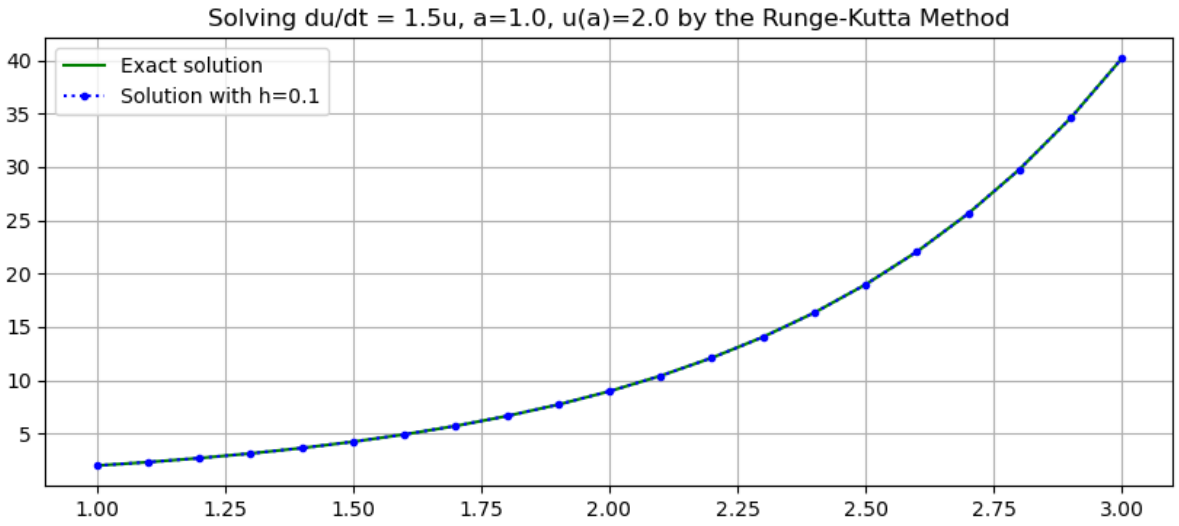
```
function rungekutta(f, a, b, u_0, n; demomode=false)
    # Use the (classical) Runge-Kutta Method to solve
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    h = (b-a)/n
    t = range(a, b, n+1)  # Note: "n" counts steps, so there are n+1 values for t.
    u = zeros(length(t))
    u[1] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i])*h
        K_2 = f(t[i]+h/2, u[i]+K_1/2)*h
        K_3 = f(t[i]+h/2, u[i]+K_2/2)*h
        K_4 = f(t[i]+h, u[i]+K_3)*h
        u[i+1] = u[i] + (K_1 + 2*K_2 + 2*K_3 + K_4)/6
    end
    return (t, u)
end;
```

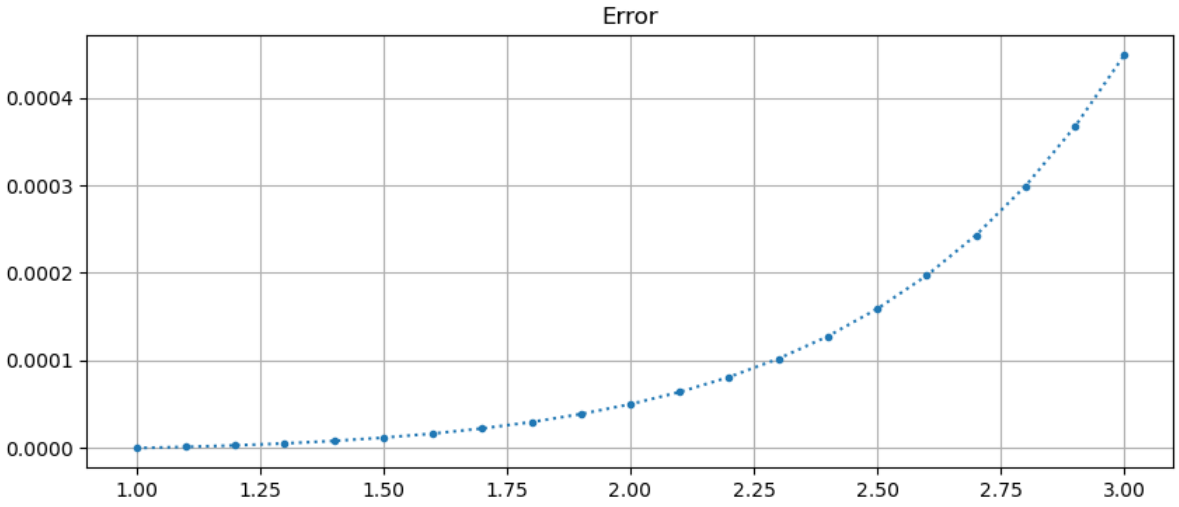
Yet again, available for import with

```
include("NumericalMethods.jl")
import .NumericalMethods: rungekutta
```

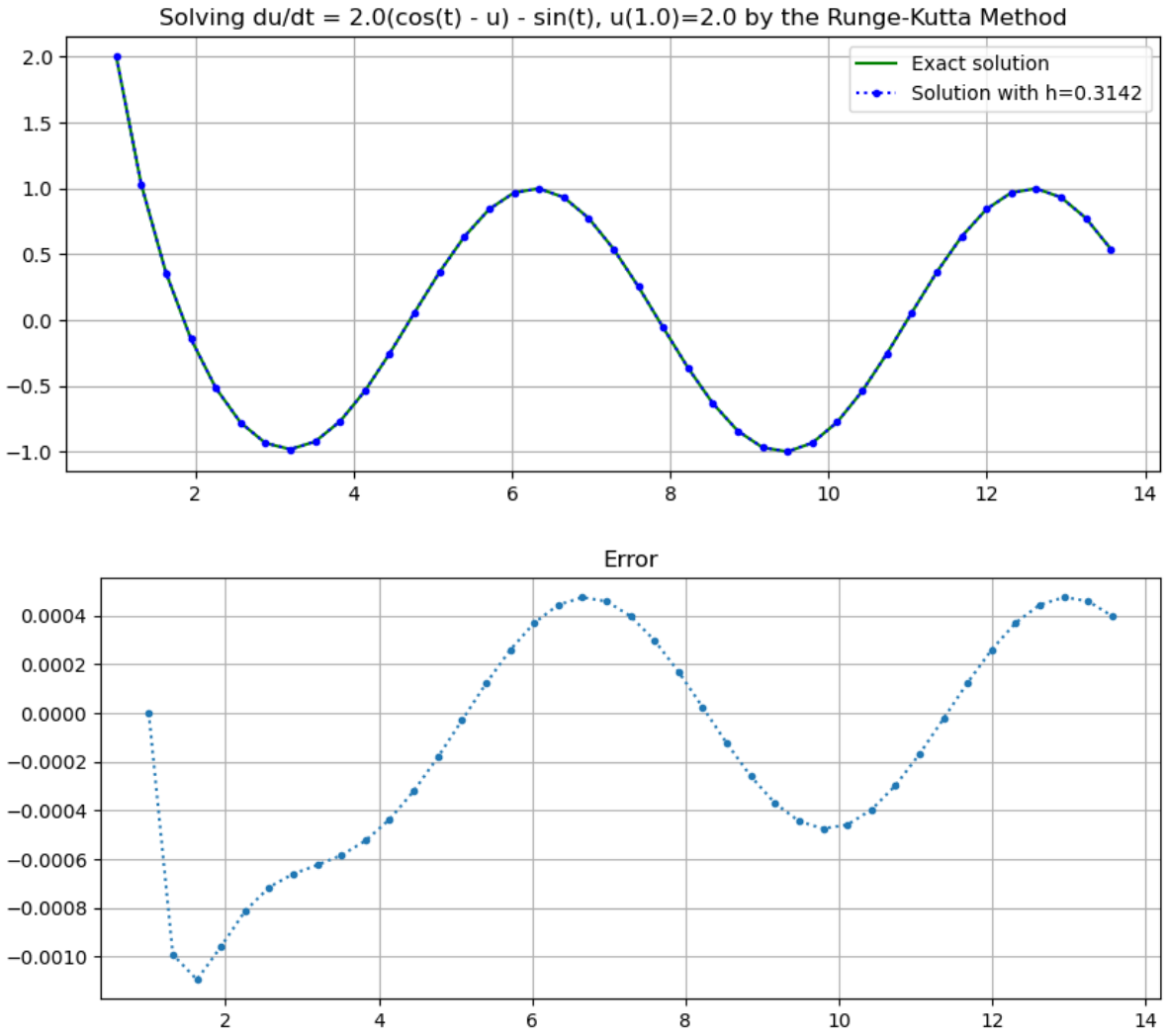
Examples

```
a = 1.0
b = 3.0
u_0 = 2.0
k = 1.5
n = 20

(t, U) = rungekutta(f1, a, b, u_0, n; demomode=true)
u = u1.(t, u_0, k)
h = round((b-a)/n, sigdigits=4)
figure(figsize=[10,4])
title("Solving du/dt = $(k)u, a=$a, u(a)=$u_0 by the Runge-Kutta Method")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Solution with h=$(approx((b-a)/n))")
legend()
grid(true)

figure(figsize=[10,4])
title("Error")
plot(t, u - U, ".:")
grid(true)
```






```
a = 1.0
b = a + 4pi  # Two periods
u_0 = 2.0
k = 2.0
n = 40

(t, U) = rungekutta(f2, a, b, u_0, n)
u = u2.(t, a, u_0, k)
figure(figsize=[10,4])
title("Solving du/dt = $k(cos(t) - u) - sin(t), u($a)=$u_0 by the Runge-Kutta Method")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Solution with h=$(approx((b-a)/n))")
legend()
grid(true)

figure(figsize=[10,4])
title("Error")
plot(t, u - U, ".:")
grid(true)
```



7.2.5 For comparison: the above examples done with Euler's Method

Since the (Explicit) Trapezoid and Midpoint methods do about twice as much work per step as Euler's method and the clasical Runge-Kutta method four time as much, a fair roughly equal cost comparions is done with

- 40 steps of Euler's method
- 20 steps of the Trapezoid and Midpoint methods
- 10 steps of the Runge-Kutta method

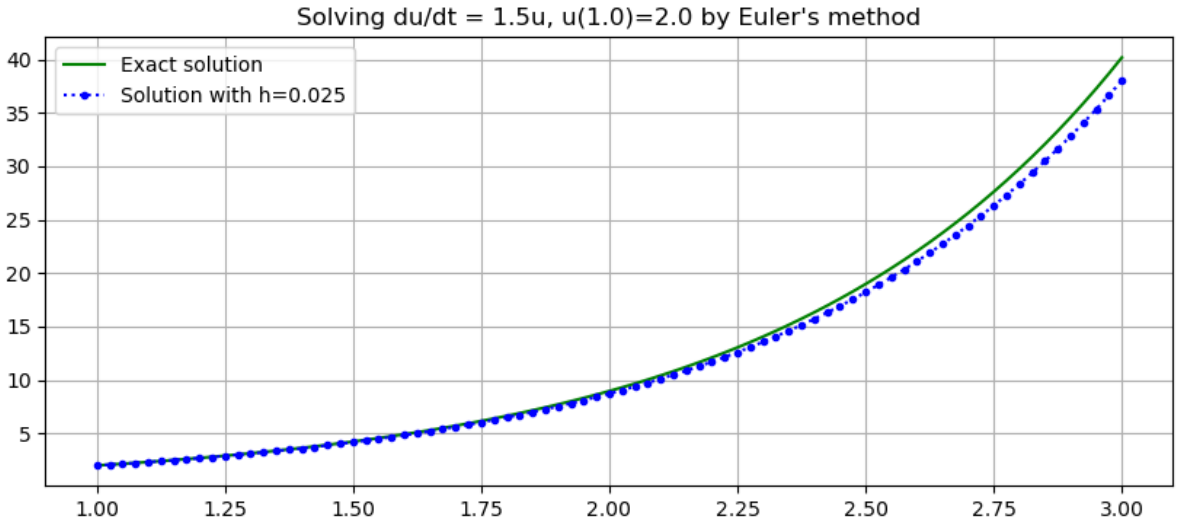
```
include("NumericalMethods.jl")
using .NumericalMethods: eulermethod
```

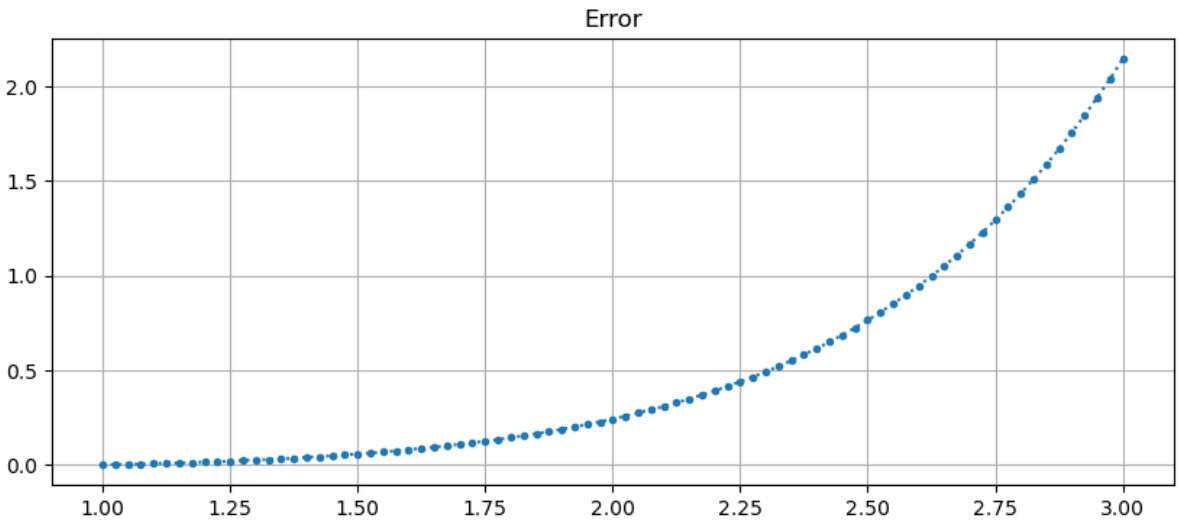
```
a = 1.0
b = 3.0
u_0 = 2.0
```

(continues on next page)

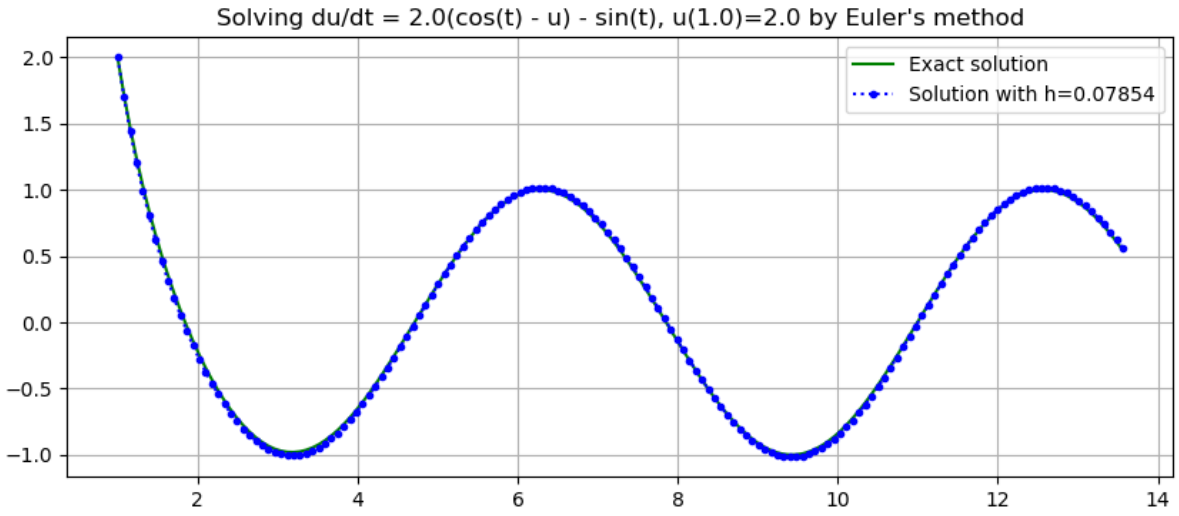
(continued from previous page)

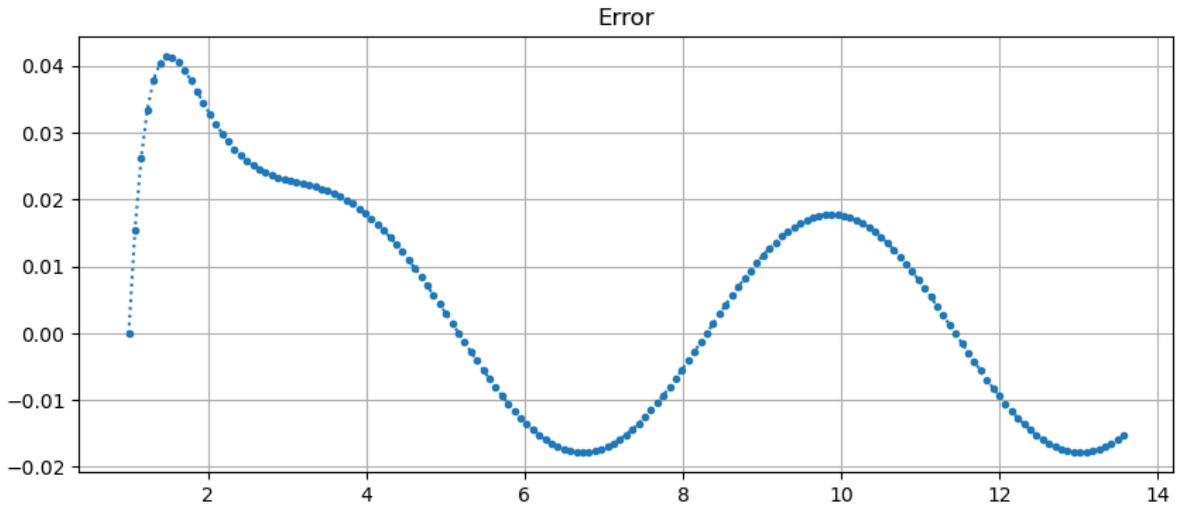
```
k = 1.5
n = 80

(t, U) = eulermethod(f1, a, b, u_0, n=n)
u = u1.(t, u_0, k)

figure(figsize=[10,4])
title("Solving du/dt = $(k)u, u($a)=$u_0 by Euler's method")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Solution with h=$(approx((b-a)/n))")
legend()
grid(true)

figure(figsize=[10,4])
title("Error")
plot(t, u - U, ".:")
grid(true)
```





```
a = 1.0
b = a + 4pi  # Two periods
u_0 = 2.0
k = 2.0
n = 160

(t, U) = eulermethod(f2, a, b, u_0, n=n)
u = u2.(t, a, u_0, k)
h = round((b-a)/n, sigdigits=4)
figure(figsize=[10,4])
title("Solving du/dt = $k(cos(t) - u) - sin(t), u($a)=$u_0 by Euler's method")
plot(t, u, "g", label="Exact solution")
plot(t, U, ".:b", label="Solution with h=$(approx((b-a)/n))")
legend()
grid(true)

figure(figsize=[10,4])
title("Error")
plot(t, u - U, ".:")
grid(true)
```





7.2.6 Exercises

Exercise 1

A) Verify that for the simple case where $f(t,u) = f(t)$, this gives the same result as the Composite Trapezoid Rule for integration.

B) Do one step of this method for the canonical example $du/dt = ku$, $u(t_0) = u_0$. It will have the form $U_1 = GU_0$ where the growth factor G approximates the factor $g = e^{kh}$ for the exact solution $u(t_1) = gu(t_0)$ of the ODE.

C) Compare to $G = 1 + kh$ seen for Euler's Method.

D) Use the previous result to express U_i in terms of $U_0 = u_0$, as done for Euler's Method.

Exercise 2 (a lot like the previous)

A) Verify that for the simple case where $f(t,u) = f(t)$, this give the same result as the Composite Midpoint rule for integration (same cooment as above).

B) Do one step of this method for the canonical example $du/dt = ku$, $u(t_0) = u_0$. It will have the form $U_1 = GU_0$ where the growth factor G approximates the factor $g = e^{kh}$ for the exact solution $u(t_1) = gu(t_0)$ of the ODE.

C) Compare to the growth factors G seen for previous methods, and to the growth factor g for the exact solution.

Exercise 3

A) Apply Richardson extrapolation to one step of Euler's method, using the values given by step sizes h and $h/2$.

B) This should give a second order accurate method, so compare it to the above two methods.

Exercise 4

A) Verify that for the simple case where $f(t,u) = f(t)$, this gives the same result as the Composite Simpson's Rule for integration.

B) Do one step of this method for the canonical example $du/dt = ku$, $u(t_0) = u_0$. It will have the form $U_1 = GU_0$ where the growth factor G approximates the factor $g = e^{kh}$ for the exact solution $u(t_1) = gu(t_0)$ of the ODE.

C) Compare to the growth factors G seen for previous methods, and to the growth factor g for the exact solution.

7.3 A Global Error Bound for One Step Methods

References:

- Subection 6.2.1 *Local and global truncation error* in [Sauer, 2019].
- Section 5.2 *Euler's Method* in [Burden *et al.*, 2016].
- Section 8.5 of [Kincaid and Chenney, 1990]

All the methods seen so far for solving ODE IVP's are *one-step methods*: they fit the general form

$$U_{i+1} = F(t_i, U_i, h)$$

For example, Euler's Method has

$$F(t, U, h) = U + hf(t, U),$$

the Explicit Midpoint Method (Modified Euler) has

$$F(t, U, h) = U + hf(t + h/2, U + hf(t, U)/2)$$

and even the Runge-Kutta method has a similar form, but it is long and ugly.

For these, there is a general result that gives a bound on the globl truncation error ("GTE") once one has a suitable bound on the local truncation error ("LTE"). This is very useful, because bounds on the LTE are usually far easier to derive.

Theorem 7.1

When solving the ODE IVP

$$du/dt = f(t, u), \quad u(a) = u_0$$

on interval $t \in [a, b]$ by a one step method, one has a bound on the local truncation error

$$|e_i| = |U_{i+1} - u(t_i + h; t_i, U_i) = |F(t_i, U_i, h) - u(t_i + h; t_i, U_i)| \leq Ch^{p+1} = O(h^{p+1})$$

and the ODE itself satisfies the *Lipschitz Condition* that for some constant K,

$$\left|\frac{\partial F}{\partial u}(t, u)\right| \leq K$$

then there is a bound on the global truncation error:

$$|E_i| = |U_i - u(t_i; a, u_0)| \leq C\frac{e^{K(t_i - a)} - 1}{k}h^p, = O(h^p)$$

So yet again, there is a loss of one factor of h in going from local to global error, as first seen with the composite rules for definite integrals.

We saw a glimpse of this for Euler's method, in the section *Basic Concepts and Euler's Method*, where the Taylor's Theorem error formula canbe used to get the LTE bound

$$|e_i| \leq Ch^2 \text{ where } C = \frac{|u_0 e^{K(b-a)}|}{2}$$

and this leads to to GTE bound

$$|E_i| \leq \frac{|u_0 e^{K(b-a)}|}{2}\frac{e^{K(t_i - a)} - 1}{k}h.$$

7.3.1 Order of accuracy for the basic Runge-Kutta type methods

- For Euler's method, it was stated in section *Basic Concepts and Euler's Method*, (and verified for the test case of $du/dt = ku$) that the global truncation error is of first order n step-size h:
- The Explicit (and Implicit) Trapezoid and Midpoint rules, the local truncation error is $O(h^3)$ and so their global truncation error is $O(h^2)$ — they are second order accurate, just as for the corresponding approximate integration rules.
- The classical Runge-Kutta method, has local truncation error $O(h^5)$ and so its global truncation error is $O(h^4)$ — just as for the composite Simpson's Rule, to which it corresponds for the "integration" case $dy/dt = f(t)$.

7.4 Systems of ODEs and Higher Order ODEs

References:

- Section 6.3 *Systems of Ordinary Differential Equations* in [Sauer, 2019], to Sub-section 6.3.1 *Higher order equations*.
- Section 5.9 *Higher Order Equations and Systems of Differential Equations* in [Burden *et al.*, 2016].

The short version of this section is that the numerical methods and algorithms developed so for for the initial value problem

$$\begin{aligned} \frac{du}{dt} &= f(t, u(t)), \quad a \le t \le b \\ u(a) &= u_0 \end{aligned}$$

all also work for system of first order ODEs by simply letting u and f be vector-valued, and for that, the Python code requires only one small change.

Also, higher order ODE's (and systems of them) can be converted into systems of first order ODEs.

7.4.1 Converting a second order ODE to a first order system

To convert

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

with initial conditions

$$y(a) = y_0,\ y'(a) = v_0$$

to a first order system, introduction the two functions

$$\begin{aligned} u_1(t) &= y(t) \\ u_2(t) &= \frac{dy}{dt}, = u_1'(t) \end{aligned}$$

Then

$$y'' = u_1' = f(t, u_0, u_1)$$

and combining with the definition of u_1 gives the system

$$\begin{aligned} u_0' &= u_1 \\ u_1' &= f(t, u_0, u_1) \end{aligned}$$

with initial conditions

$$\begin{aligned} u_0(a) &= y_0 \\ u_1(a) &= v_0 \end{aligned}$$

Next this can be put into vector form. Defining the vector-valued functions

$$\tilde{u}(t) = \langle u_1(t), u_2(t) \rangle$$
$$\tilde{f}(t, \tilde{u}(t)) = \langle u_1(t), f(t, u_2(t), u_2(t)) \rangle$$

and initial data vector

$$\tilde{u}_0 = \langle u_{0,1}, u_{0,2} \rangle = \langle y_0, v_0 \rangle$$

puts the equation into the form

$$\frac{d\tilde{u}}{dt} = \tilde{f}(t, \tilde{u}(t)), \quad a \le t \le b$$
$$\tilde{u}(a) = \tilde{u}_0$$

$$\frac{d\tilde{u}}{dt} = \tilde{f}(t, \tilde{u}(t)), \quad a \le t \le b$$
$$\tilde{u}(a) = \tilde{u}_0$$

7.4.2 Test Cases

In this and subsequent sections, numerical methods for higher order equations and systems will be compared using several test cases:

Test Case A: Motion of a (Damped) Mass-Spring System in One Dimension

A simple mathematical model of a damped mass-spring system is

$$M\frac{d^2y}{dt^2} = -Ky - D\frac{dy}{dt}$$
with initial conditions
$$y(a) = y_0$$
$$\left.\frac{dy}{dt}\right|_{t=a} = v_0$$

where K is the spring constant and D is the coefficient of friction, or drag.

The first order system form can be left in terms of y and y' as

$$\frac{d}{dt}\begin{bmatrix} y \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -K & -D \end{bmatrix}\begin{bmatrix} y \\ y' \end{bmatrix}$$

Exact solutions

For testing of numerical methods in this and subsequent sections, here are the exact solutions.

They depend on whether

- $D < D_0 := 2\sqrt{KM}$: *underdamped*,
- $D > D_0$: *overdamped*, or
- $D = D_0$: *critically damped*.

We will mostly explore the first two more "generic" cases.

For the underdamped case, the general solution is

$$y(t) = e^{-(D/(2M))(t-a)}[A\cos(\omega(t-a)) + B\sin(\omega(t-a))], \quad \omega = \frac{\sqrt{4KM - D^2}}{2M}$$

For the above initial conditions, $A = y_0$ and $B = (v_0 + y_0 D/(2M)/\omega$.

An important special case of this is the undamped system $M\frac{d^2y}{dt^2} = -Ky$ for which the solutions become

$$y(t) = A\cos(\omega(t-a)) + B\sin(\omega(t-a)), \quad \omega = \sqrt{K/M}$$

and it can be verified that the "energy"

$$E(t) = \frac{M}{2}(y'(t))^2 + \frac{K}{2}(y(t))^2 = \frac{1}{2}(Ku_1^2 + Mu_2^2)$$

is conserved: $dE/dt = 0$. Conserved quantities can provide a useful check of the acccuracy of numerical method, so we will look at this below.

For the overdamped case, the general solution is

$$y(t) = Ae^{\lambda_+(t-a)} + Be^{\lambda_-(t-a)}, \quad \lambda_\pm = \frac{-D \pm \Delta}{2M}, \ \Delta = \sqrt{D^2 - 4KM}$$

For the above initial conditions, $A = M(v_0 - \lambda_- y_0)/\Delta$ and $B = y_0 - A$.

Remark 7.1 (Stiffness)

Fixing M and scaling $K = D \to \infty$, $\Delta = D\sqrt{1 - 4M/D} \approx D - 2M$ so

$$\lambda_- \approx -\frac{D}{M} + 1 \to -\infty, \quad \lambda_+ \approx -1.$$

Thus the time scales of the two exponential decays become hugely different, with the fast term $Be^{\lambda_-(t-a)}$ becoming negligible compared to the slower decaying $Ae^{\lambda_+(t-a)}$.

This is a simple example of **stiffness**, and influences the choice of a good numerical method for solving such equations.

The variable can be rescaled to the case $K = M = 1$, so that will be done from now on, but of course you can easily experiment with other parameter values by editing copies of the Jupyter notebooks.

Test Case B: A "Fast-Slow" Equation

The equation

$$y'' + (K+1)y' + Ky = 0, \quad y(0) = y_0, y'(0) = v_0$$

has first order system form

$$\frac{d}{dt}\begin{bmatrix} y \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -K & -(K+1) \end{bmatrix}\begin{bmatrix} y \\ y' \end{bmatrix}$$

and the general solution

$$y(t) = Ae^{-t} + Be^{-Kt}$$

so for large K, it has two very disparate time scales, with only the slower scale of much significance after an initial transient.

This is a convenient "toy" example for testing two refinements to algorithms:

- Variable time step sizes, so that they can be short during the initial transient and longer later, when only the e^{-t} behavior is significant.
- Implicit methods that can effectively suppress the fast but extremely small e^{-kt} terms while hanling the larger, slower terms accurately.

The examples below will use $K = 100$, but as usual, you are free to experiment with other values.

Test Case C: The Freely Rotating Pendulum

Both the above equations are constant coefficient linear, which is convenient for the sake of having exact solution to compare with, but one famous nonlinear example is worth exporing too.

A pendulum with mass m concentrated at a distnace L from the axis of rotation and that can rotate freely in a vertical plane about that axis and with possible friction proportional to D, can be modeled in terms of its angular position θ and angular velocity $\omega = \theta'$ by

$$ML\theta'' = -Mg\sin\theta - DL\theta', \quad \theta(0) = \theta_0, \theta'(0) = \omega_0$$

or in system form

$$\frac{d}{dt}\begin{bmatrix} \theta \\ \omega \end{bmatrix} = \begin{bmatrix} \omega \\ -\frac{g}{L}\sin\theta - \frac{D}{M}\omega \end{bmatrix}$$

These notes will mostly look at the frictionelss case $D = 0$, which has conserved energy

$$E(\theta, \omega) = \frac{ML}{2}\omega^2 - Mg\cos\theta$$

For this, the solution fall into three qualitatively different cases depending on whether the energy is less than, equal to, or greater than the "critical energy" Mg, which is the energy of the unstable stationary solutions $\theta(t) = \pi(\mod 2\pi)$, $\omega(t) = 0$: "balancing at the top":

- For $E < Mg$, a solution can never reach the top, so the pendulum rocks back and forth, reach maximum height at $\theta = \pm\arccos(-E/(Mg))$
- For $E > Mg$, solutions have angular speed $|\omega| \geq \sqrt{E - Mg} > 0$ so it never drops to zero, and so the direction of rotation can never reverse: solutions rotate in one direction for ever.
- For $E = Mg$, one special type of solution is those up-side down stationary ones. Any other solution always has $|\omega| = \sqrt{E - Mg\cos\theta} > 0$, and so never stops or reverses direction but instead approaches the above stationary point asymptotically both as $t \to \infty$ and $t \to \infty$. To visualize concretely, the solution starting at the bottom with $\theta(0) = 0$, $\omega(0) = \sqrt{2g/L}$ has $\theta(t) \to \pm\pi$ and $\omega(t) \to 0$ as $t \to \pm\infty$.

Remark 7.2 (Separatrices)

This last kind of special solution is known as a **separatrix** due to separating the other two qualitatively different sorts of solution. They are also known as **heteroclinic orbits**, for "asymptotically" starting and ending at different stationary solutions in each time direction — or **homoclinic** if you consider the angle as a "mod 2π" value describing a position, so that $\theta = \pm\pi$ are the same location and the solutions start and end at the same stationary point.

```
using PyPlot
include("NumericalMethods.jl")
using .NumericalMethods: approx4
```

The Euler's method code from before does not quite work, but only slight modification is needed; that "scalar" version

```
function eulermethod(f, a, b, u_0, n)
    h = (b-a)/n
    t = range(a, b, n+1)
    U = zeros(n+1)
    U[1] = u_0
    for i in 1:n
        U[i+1] = U[i] + f(t[i], U[i])*h
    end
    return (t, U)
end;
```

becomes

```
function eulermethod_system(f, a, b, u_0, n)
    # TO DO: one could use multiple dispatch to keep the name "eulermethod".
    # The conversion for the system version is mainly "U[i] -> U[i,:]"

    h = (b-a)/n
    t = range(a, b, n+1)

    # The following three lines and the one in the for loop below change for the␣
 ↪system version
    n_unknowns = length(u_0)
    U = zeros(n+1, n_unknowns)
    U[1,:] = u_0  # Only for system version

    for i in 1:n
        U[i+1,:] = U[i,:] + f(t[i], U[i,:])*h  # For the system version
    end
    return (t, U)
end;
```

Note. Here and below, these notes follow the convention of using lowercase letters for exact solutions; uppercase for numerical approximations.

7.4.3 Solving the Mass-Spring System

```
f_mass_spring(t, u) = [ u[2], -(K/M)*u[1] - (D/M)*u[2] ];
```

```
E_mass_spring(y, Dy) = (K * y^2 + M * Dy^2)/2;
```

```
function y_mass_spring(t; t_0, u_0, K, M, D)
    (y_0, v_0) = u_0
    discriminant = D^2 - 4K*M
    if discriminant < 0  # underdamped
        omega = sqrt(4K*M - D^2)/(2M)
        A = y_0
        B = (v_0 + y_0*D/(2M))/omega
        return exp(-D/(2M)*(t-t_0)) * ( A*cos(omega*(t-t_0)) + B*sin(omega*(t-t_0)))
    elseif discriminant > 0  # overdamped
        Delta = sqrt(discriminant)
        lambda_plus = (-D + Delta)/(2M)
        lambda_minus = (-D - Delta)/(2M)
```

(continues on next page)

(continued from previous page)

```
        A = M*(v_0 - lambda_minus * y_0)/Delta
        B = y_0 - A
        return A*exp(lambda_plus*(t-t_0)) + B*exp(lambda_minus*(t-t_0))
    else
        lambda = -D/(2M)
        A = y_0
        B = v_0 - A * lambda
        return (A + B*t)*exp(lambda*(t-t_0))
    end
end;
```

```
function damping(K, M, D)
    if D == 0
        println("Undamped")
    else
        discriminant = D^2 - 4K*M
        if discriminant < 0
            println("Underdamped")
        elseif discriminant > 0
            println("Overdamped")
        else
            println("Critically damped")
        end
    end
end;
```

The above functions are available in module `NumericalMethods`; they will be used in later sections.

First solve without damping, so the solutions have sinusoidal solutions

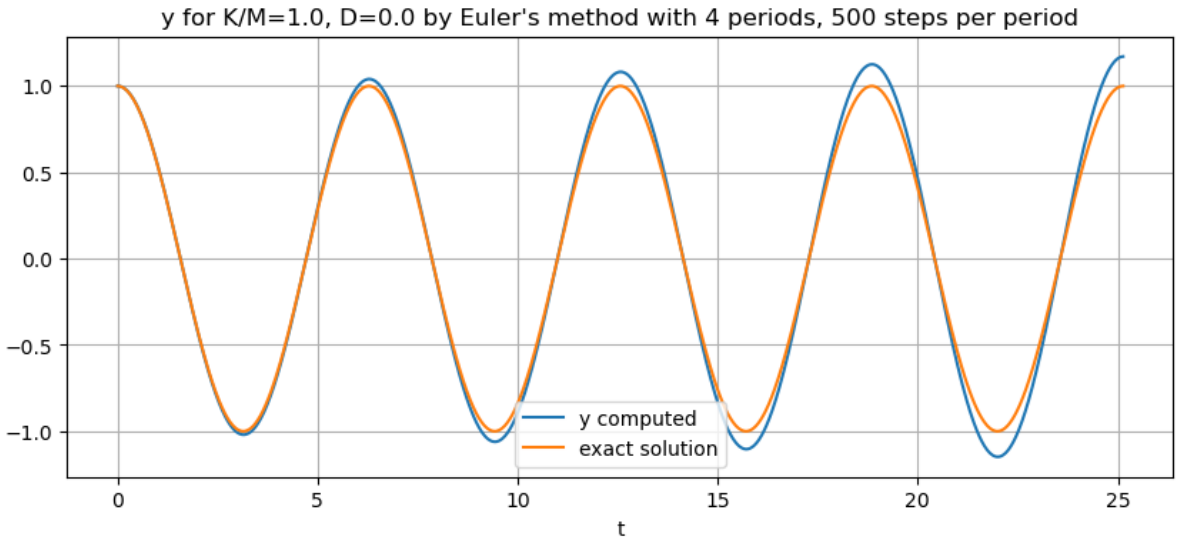
Note: the orbits go clockwise for undamped (and underdamped) systems.

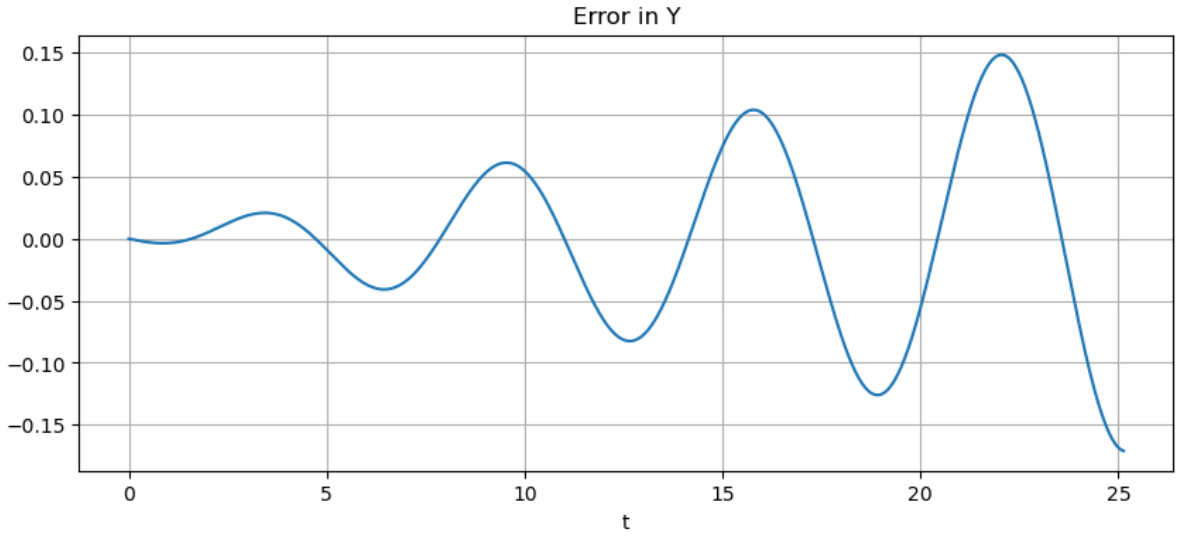
```
M = 1.0
K = 1.0
D = 0.0
y_0 = 1.0
Dy_0 = 0.0
u_0 = [y_0, Dy_0]
a = 0.0
periods = 4
b = 2pi * periods

stepsperperiod = 500
n = Int(stepsperperiod * periods)

(t, U) = eulermethod_system(f_mass_spring, a, b, u_0, n)
Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=u_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("y for K/M=$(K/M), D=$D by Euler's method with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
```

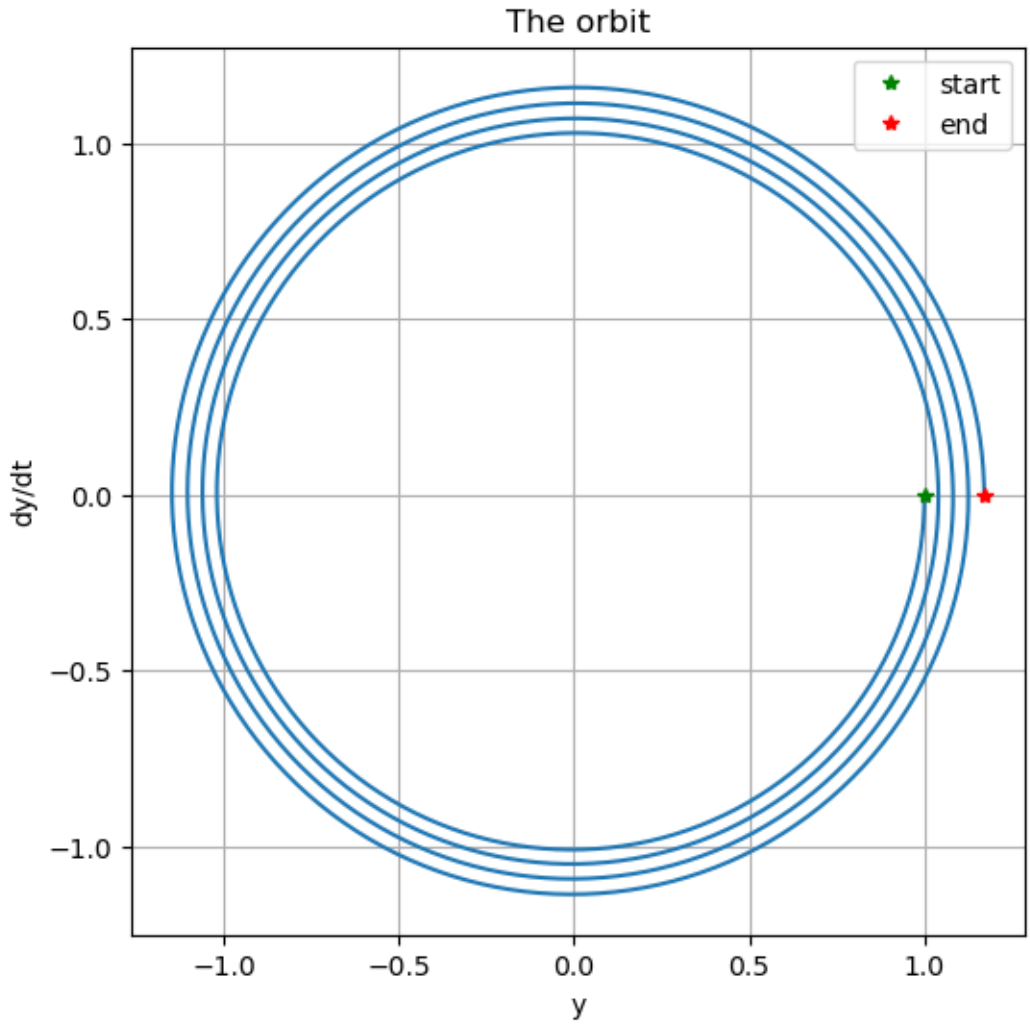
(continues on next page)

(continued from previous page)

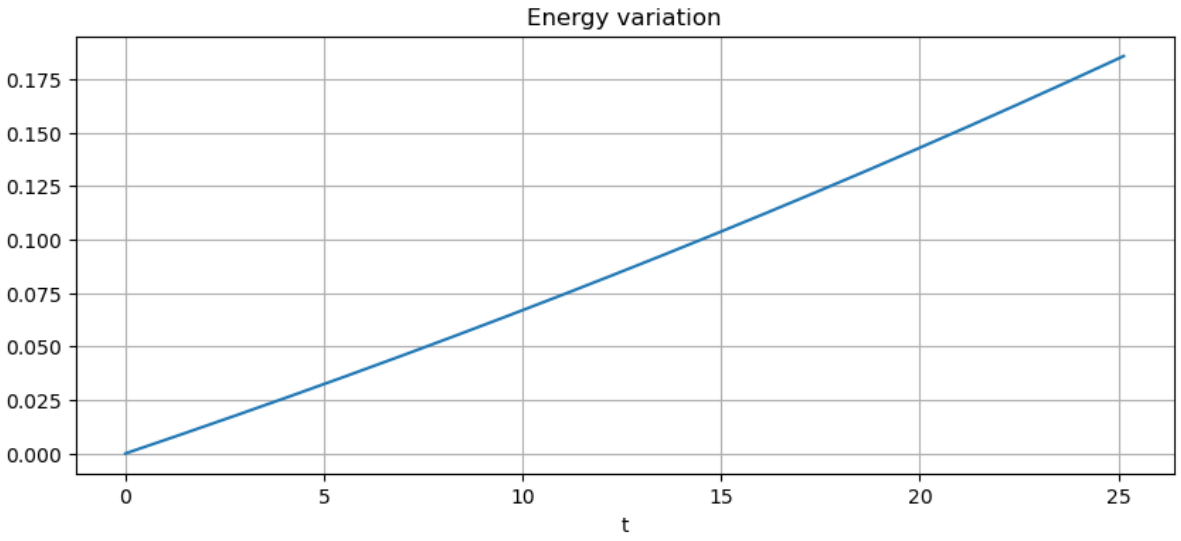
```
plot(t, y, label="exact solution")
xlabel("t")
legend()
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

# Phase plane diagram; for D=0 the exact solutions are ellipses (circles if M = k)
figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
 ↪"circular spirals"
title("The orbit")
plot(Y, DY)
xlabel("y")
ylabel("dy/dt")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```




Error in Y
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
5
10
15
20
25
t


The orbit
start
end
1.0
0.5
0.0
−0.5
−1.0
dy/dt
−1.0
−0.5
0.0
0.5
1.0
y

```
figure(figsize=[10,4])
E_0 = E_mass_spring(y_0, Dy_0)
E = E_mass_spring.(Y, DY)
title("Energy variation")
plot(t, E .- E_0)
xlabel("t")
grid(true)
```



Next solve with damping

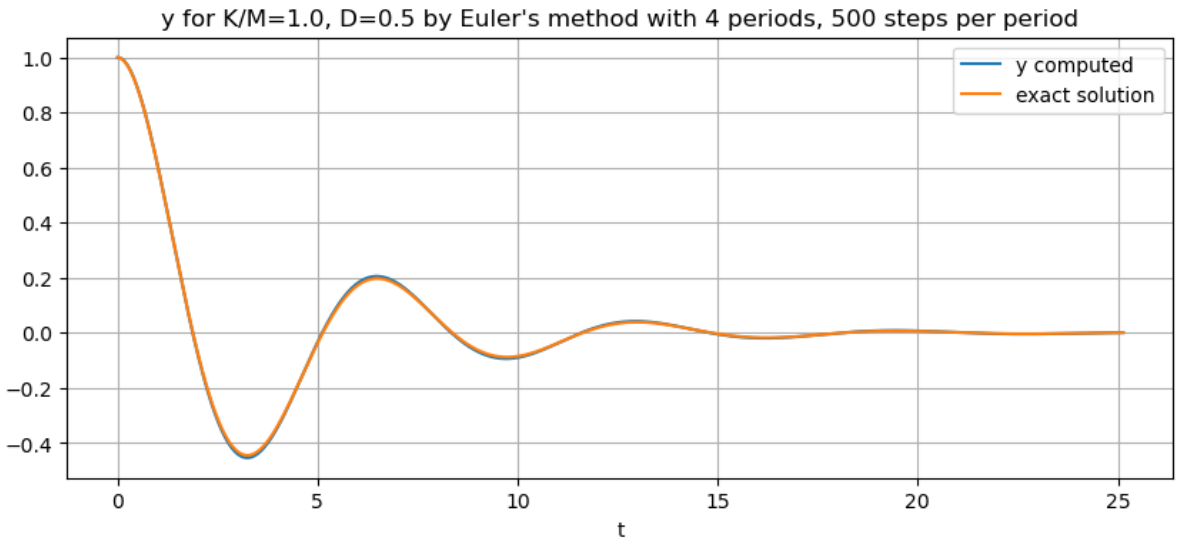
```
D = 0.5 # Underdamped: decaying oscillations
#D = 2 # Critically damped
#D = 2.1 # Overdamped: exponential decay

periods = 4
b = 2pi * periods

stepsperperiod = 500
n = Int(stepsperperiod * periods)

(t, U) = eulermethod_system(f_mass_spring, a, b, u_0, n)
Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=u_0, K=K, M=M, D=D)  # Exact solution

damping(K, M, D)

figure(figsize=[10,4])
title("y for K/M=$(K/M), D=$D by Euler's method with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
xlabel("t")
legend()
grid(true)
```

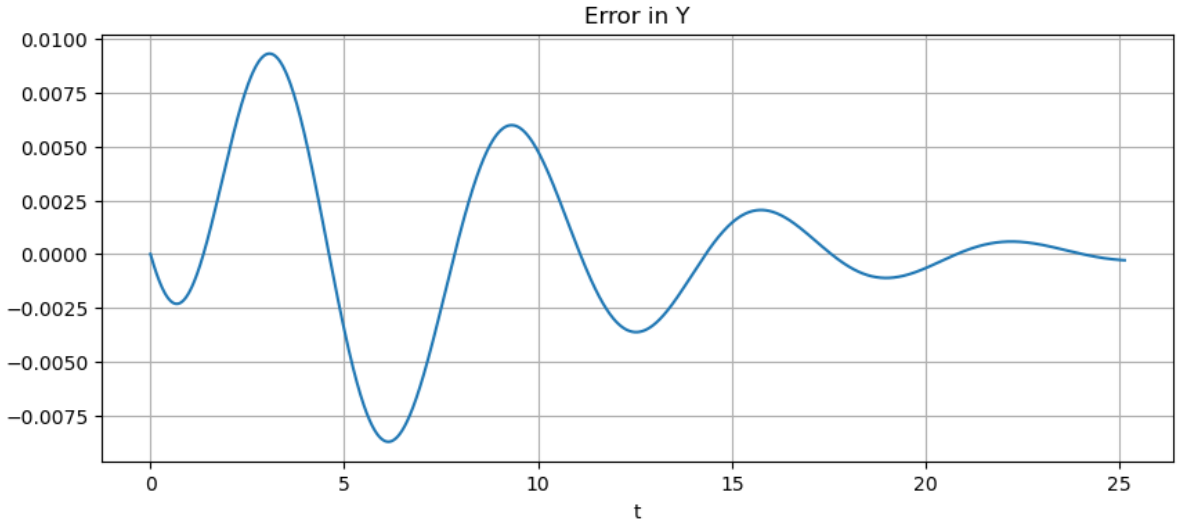
(continues on next page)

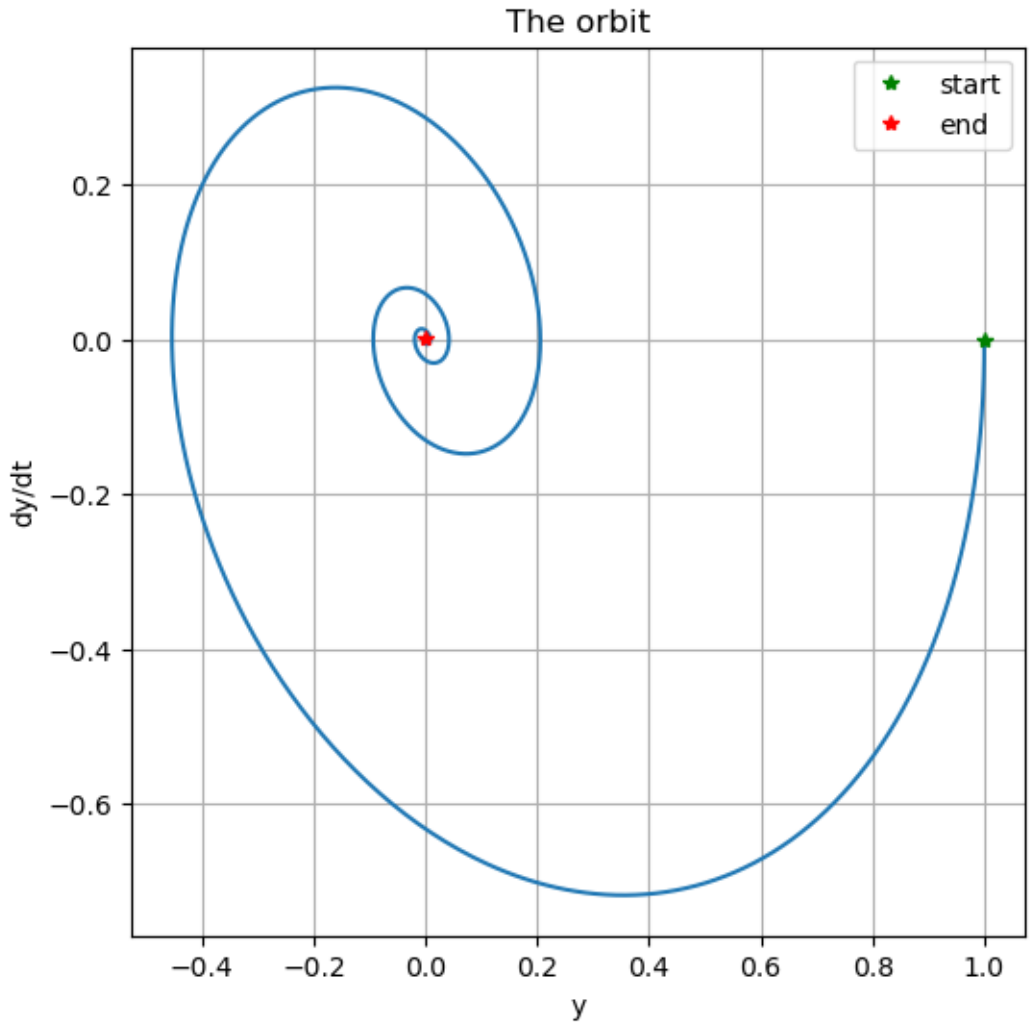
(continued from previous page)

```
figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

# Phase plane diagram; for D=0 the exact solutions are ellipses (circles if M = K)
figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
↪"circular spirals"
title("The orbit")
plot(Y, DY)
xlabel("y")
ylabel("dy/dt")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```

```
Underdamped
```




Error in Y
0.0100
0.0075
0.0050
0.0025
0.0000
−0.0025
−0.0050
−0.0075
0
5
10
15
20
25
t


The orbit
start
end
0.2
0.0
−0.2
−0.4
−0.6
dy/dt
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
y

7.4.4 The "Classical" Runge-Kutta Method, Extended to Systems of Equations

As above, the previous "scalar" function for this method needs just three lines of code modified.

Before:

```
function rungekutta(f, a, b, u_0, n)
    # Use the (classical) Runge-Kutta Method to solve
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0

    h = (b-a)/n
    t = range(a, b, n+1)
    u = zeros(length(t))
    u[1] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i])*h
        K_2 = f(t[i]+h/2, u[i]+K_1/2)*h
        K_3 = f(t[i]+h/2, u[i]+K_2/2)*h
        K_4 = f(t[i]+h, u[i]+K_3)*h
        u[i+1] = u[i] + (K_1 + 2*K_2 + 2*K_3 + K_4)/6
    end
    return (t, u)
end;
```

After:

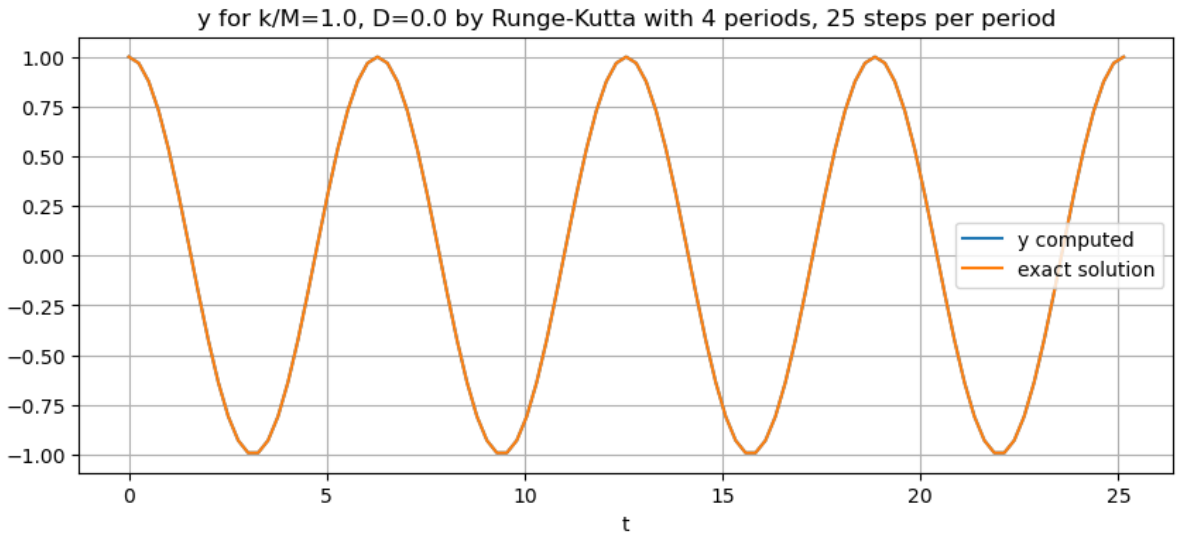
```
function rungekutta_system(f, a, b, u_0, n)
    # Use the (classical) Runge-Kutta Method to solve
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    # The conversion for the system version is mainly "u[i] -> u[i,:]"

    h = (b-a)/n
    t = range(a, b, n+1)
    n_unknowns = length(u_0)
    u = zeros(n+1, n_unknowns)
    u[1,:] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i,:])*h
        K_2 = f(t[i]+h/2, u[i,:]+K_1/2)*h
        K_3 = f(t[i]+h/2, u[i,:]+K_2/2)*h
        K_4 = f(t[i]+h, u[i,:]+K_3)*h
        u[i+1,:] = u[i,:] + (K_1 + 2*K_2 + 2*K_3 + K_4)/6
    end
    return (t, u)
end;
```

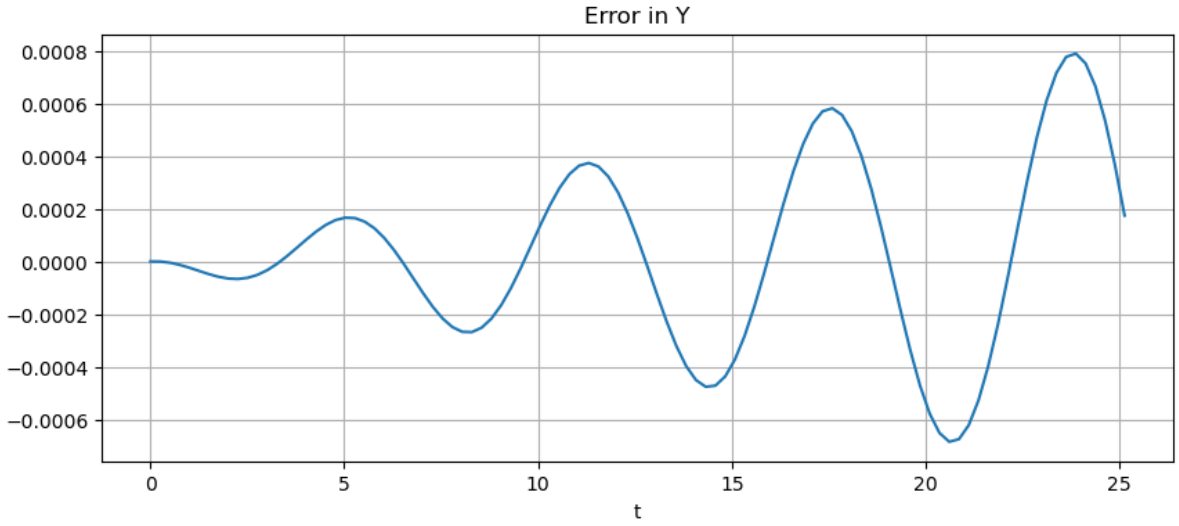
```
M = 1.0
k = 1.0
D = 0.0
u_0 = [1.0, 0.0]
a = 0.0
periods = 4
b = 2pi * periods

stepsperperiod = 25
n = stepsperperiod * periods
```

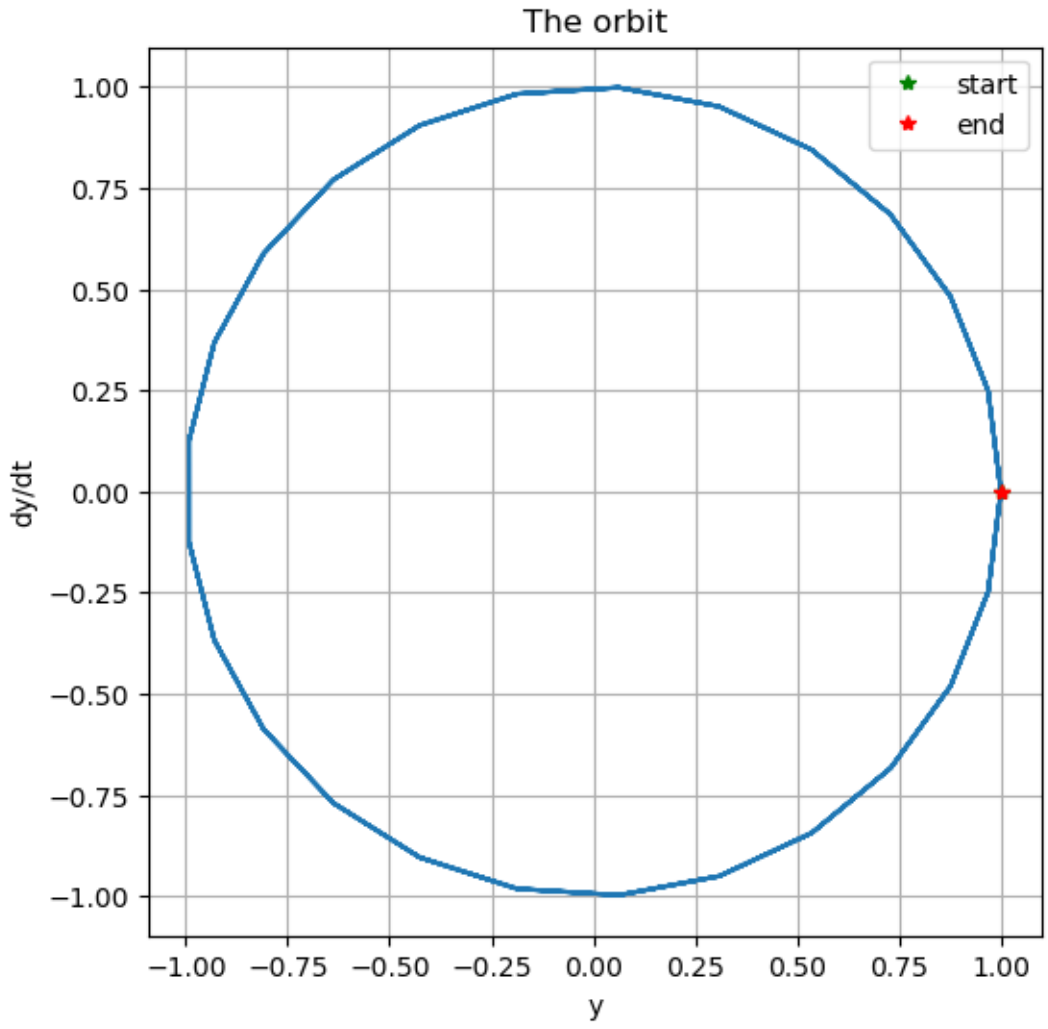
(continues on next page)

(continued from previous page)

```
(t, U) = rungekutta_system(f_mass_spring, a, b, u_0, n)
Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=u_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("y for k/M=$(k/M), D=$D by Runge-Kutta with $periods periods, $stepsperperiod
↪steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
xlabel("t")
legend()
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

# Phase plane diagram; for D=0 the exact solutions are ellipses (circles if M = k)
figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
↪"circular spirals"
title("The orbit")
plot(Y, DY)
xlabel("y")
ylabel("dy/dt")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```







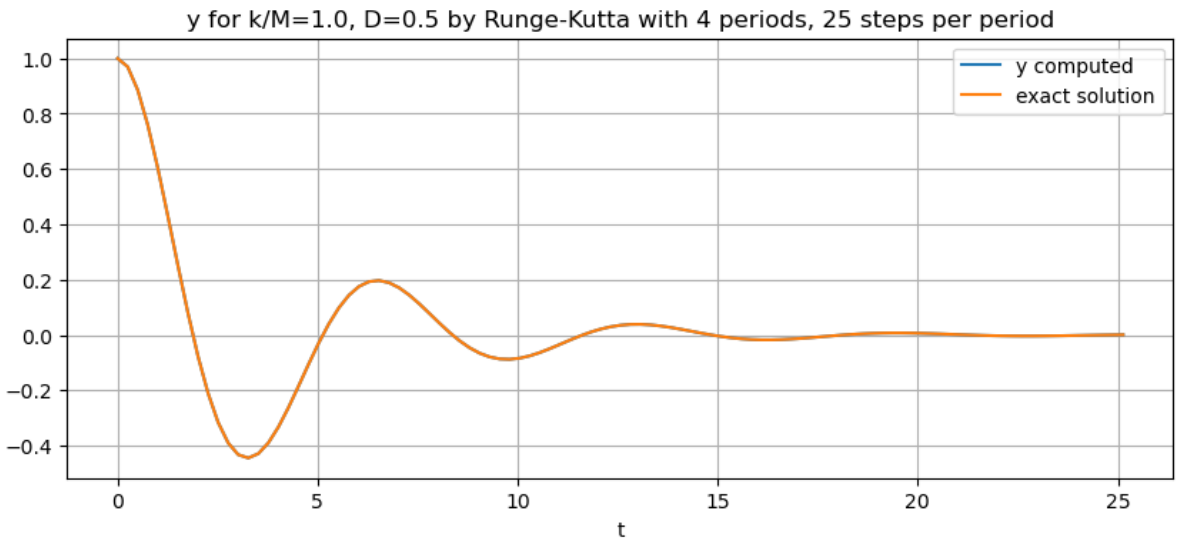

```
D = 0.5 # Underdamped: decaying oscillations
#D = 2 # Critically damped
```

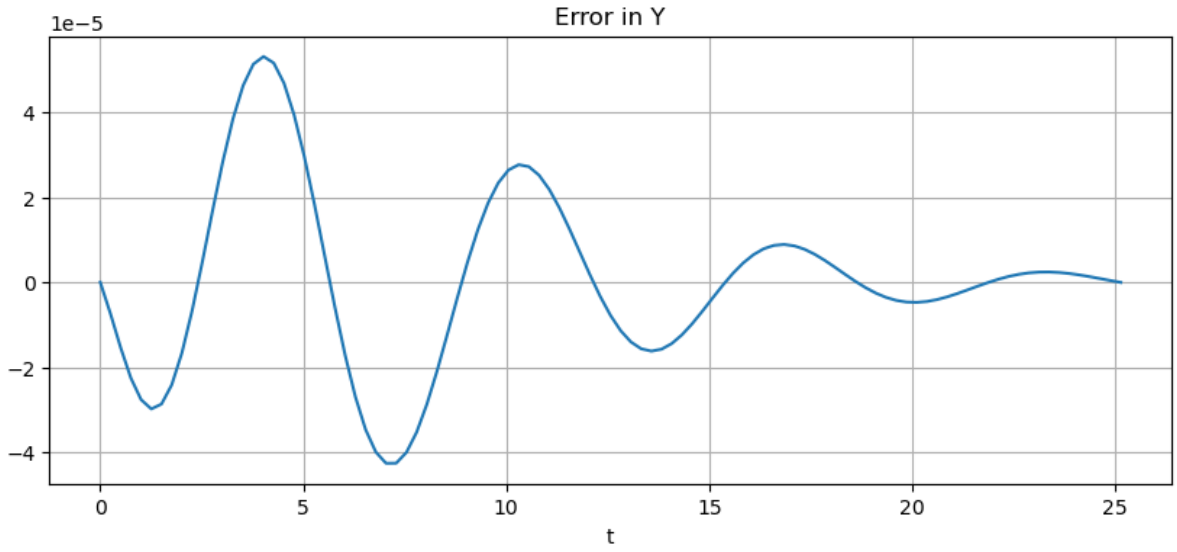
(continues on next page)

(continued from previous page)

```
#D = 2.1 # Overdamped: exponential decay

periods = 4
b = 2pi * periods

stepsperperiod = 25
n = Int(stepsperperiod * periods)

(t, U) = rungekutta_system(f_mass_spring, a, b, u_0, n)
Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=u_0, K=K, M=M, D=D)  # Exact solution

damping(k, M, D)

figure(figsize=[10,4])
title("y for k/M=$(k/M), D=$D by Runge-Kutta with $periods periods, $stepsperperiod␣
↪steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
xlabel("t")
legend()
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)
```

```
Underdamped
```





7.4.5 Solving the Freely Rotating Pendulum Equations

For now, this is just briefly explored as a cautionary tail of what can happen when slight changes in the system lead to qualitatively very different solution behavior. So we will look at a few examples for the conservative case $D = 0$, close to the separatrix solutions noted above.

Parameters can all be scaled away to $M = L = g = 1$ so the critical energy is $Mg = 1$.

```
f_pendulum(t, u) = [ u[2], -(g/L)*sin(u[1]) ];
```

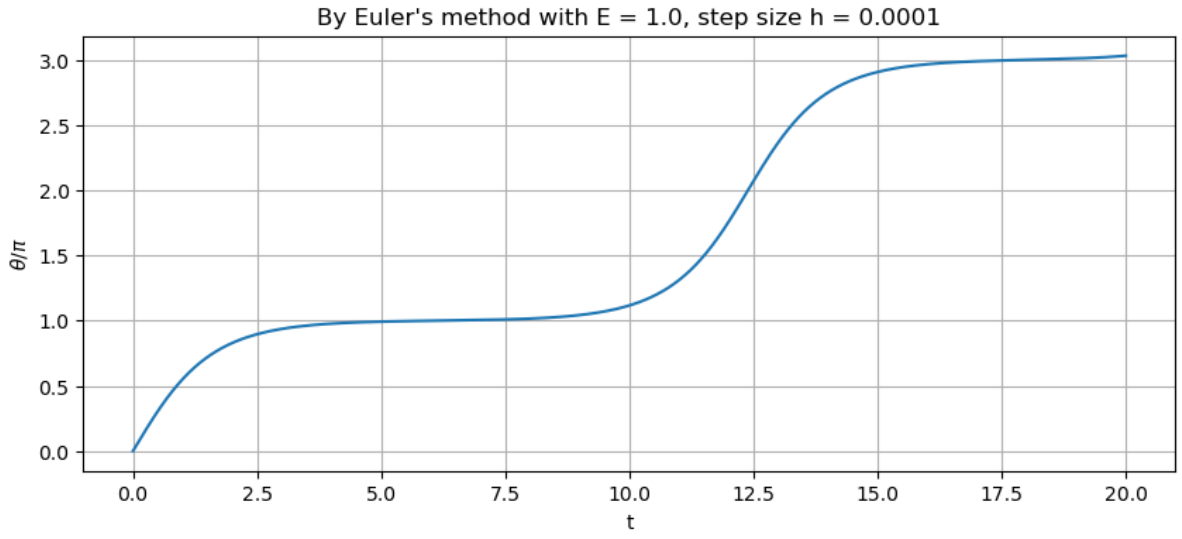
```
M = g = L = 1.0;
E_0 = 1.0  # Separatrix
#E_0 = 0.999
#E_0 = 1.001

theta_0 = 0.0
omega_0 = sqrt(2(E_0 + M*g*cos(theta_0))/(M*L));
u_0 = [theta_0, omega_0]

a = 0.0

#periods = 8  # periods of the linear approximation, "sin(theta) = theta"
#b = 2pi * sqrt(L/g) * periods

b = 80.0
b = 20.;
```

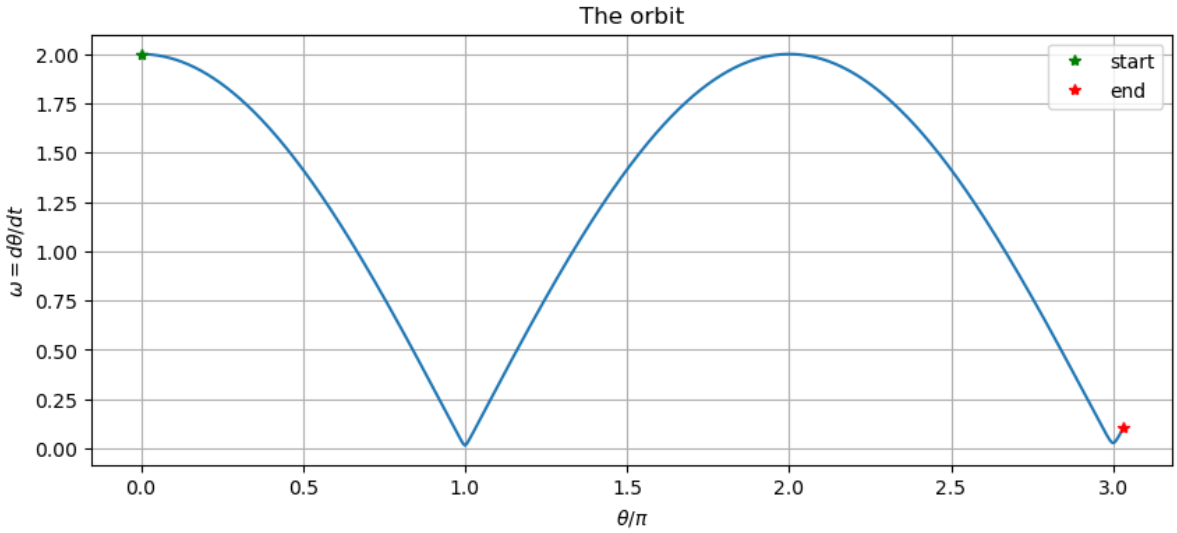
```
#stepsperperiod = 1_000
#n = Int(stepsperperiod * periods)
#h = (b-a)/n

stepsperunittime = 10_000
h = 1/stepsperunittime
n = Int(round((b-a)/h))
```

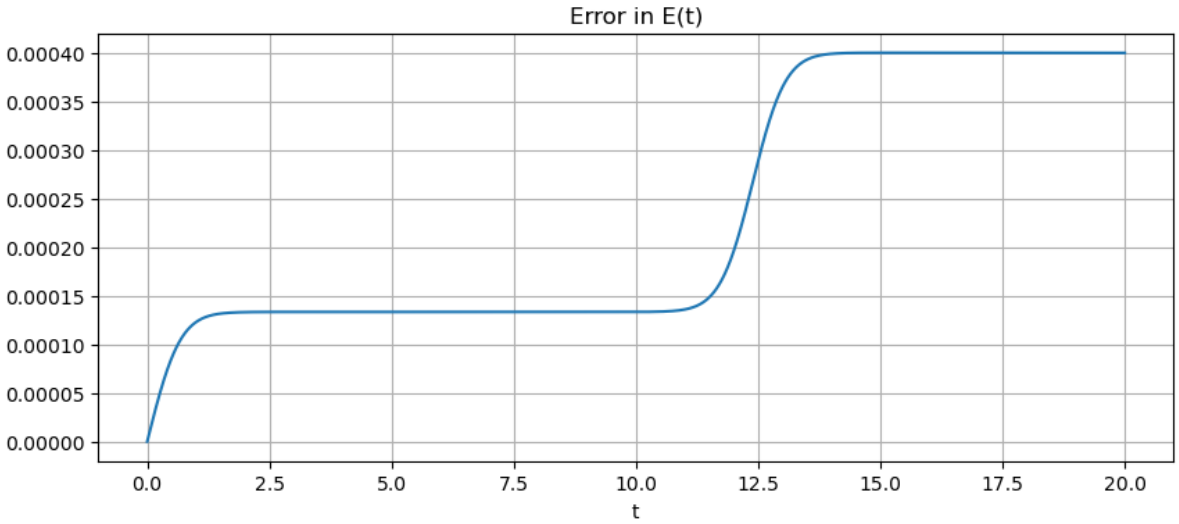
(continues on next page)

(continued from previous page)

```
(t, U) = eulermethod_system(f_pendulum, a, b, u_0, n)
theta = U[:,1]
omega = U[:,2]

figure(figsize=[10,4])
title("By Euler's method with E = $E_0, step size h = $(approx4(h))")
plot(t, theta/pi, label="theta")
xlabel("t")
ylabel(L"\theta/\pi")
grid(true)

# Phase plane diagram
figure(figsize=[10,4])
title("The orbit")
plot(theta/pi, omega)
xlabel(L"\theta/\pi")
ylabel(L"\omega = d\theta/dt")
plot(theta[1]/pi, omega[1], "g*", label="start")
plot(theta[end]/pi, omega[end], "r*", label="end")
legend()
grid(true)

# Error in the (conserved) energy E
figure(figsize=[10,4])
E = (M*L/2) * omega.^2 - M*g*cos.(theta)
E_error = E .- E_0
title("Error in E(t)")
plot(t, E_error, label="theta")
xlabel("t")
#ylabel(L"\theta/\pi")
grid(true)
```





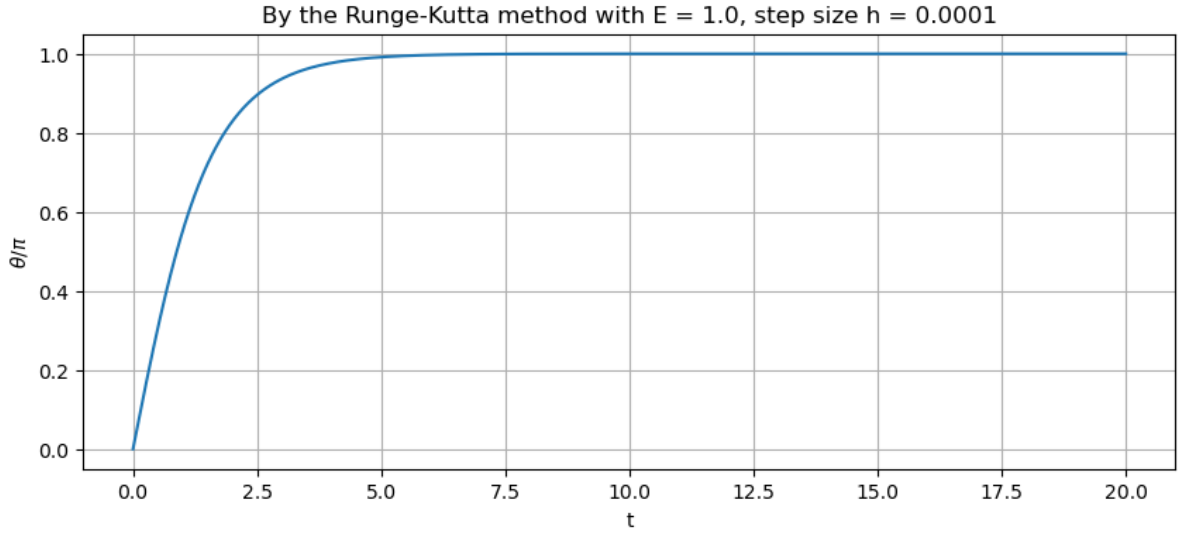


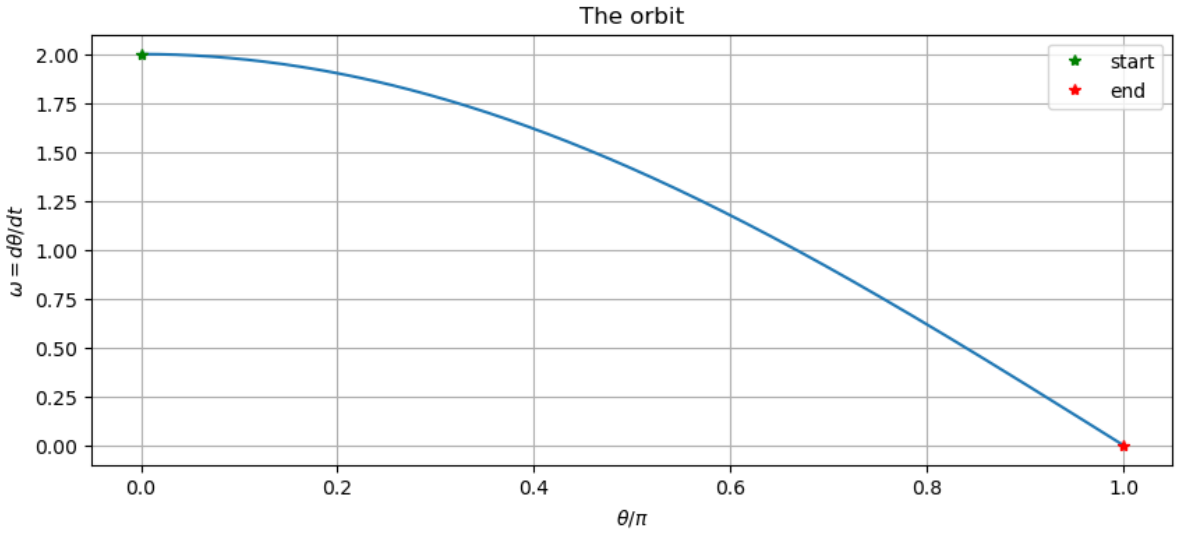

```
#stepsperperiod = 10_000
#n = Int(stepsperperiod * periods)
#h = (b-a)/n

stepsperunittime = 25
stepsperunittime = 10_000
h = 1/stepsperunittime
n = Int(round((b-a)/h))

(t, U) = rungekutta_system(f_pendulum, a, b, u_0, n)
theta = U[:,1]
omega = U[:,2]

figure(figsize=[10,4])
title("By the Runge-Kutta method with E = $E_0, step size h = $(approx4(h))")
plot(t, theta/pi, label="theta")
xlabel("t")
ylabel(L"\theta/\pi")
grid(true)
```

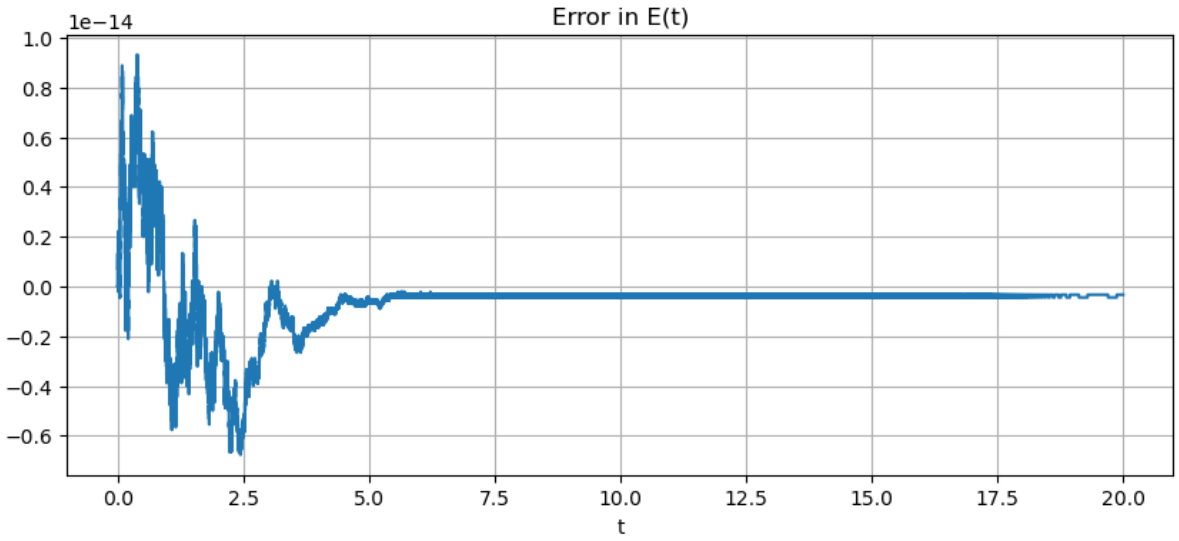
(continues on next page)

(continued from previous page)

```
# Phase plane diagram
figure(figsize=[10,4])
title("The orbit")
plot(theta/pi, omega)
xlabel(L"\theta/\pi")
ylabel(L"\omega = d\theta/dt")
plot(theta[1]/pi, omega[1], "g*", label="start")
plot(theta[end]/pi, omega[end], "r*", label="end")
legend()
grid(true)

# Error in the (conserved) energy E
E = (M*L/2) * omega.^2 - M*g*cos.(theta)
E_error = E .- E_0
figure(figsize=[10,4])
title("Error in E(t)")
plot(t, E_error, label="theta")
xlabel("t")
#ylabel(L"\theta/\pi")
grid(true);
```







7.4.6 Appendix: the Explicit Trapezoid and Midpoint Methods for systems

Yet again, the previous functions for these methods need just three lines of code modified.

The demos are just for the non-dissipative case, where the solution is known to be $y = \cos t$, $dt/dt = -\sin t$.

For a fairer comparison of "accuracy vs computational effort" to the Runge-Kutta method, twice as many time steps are used so that the same number of function evaluations are used for these three methods.

```
function explicittrapezoid_system(f, a, b, u_0, n)
    # Use the Explict Trapezoid Method (a.k.a Improved Euler) to solve the system
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    # The conversion for the system version is mainly "u[i] -> u[i,:]"

    h = (b-a)/n
    t = range(a, b, n+1)
    n_unknowns = length(u_0)
```

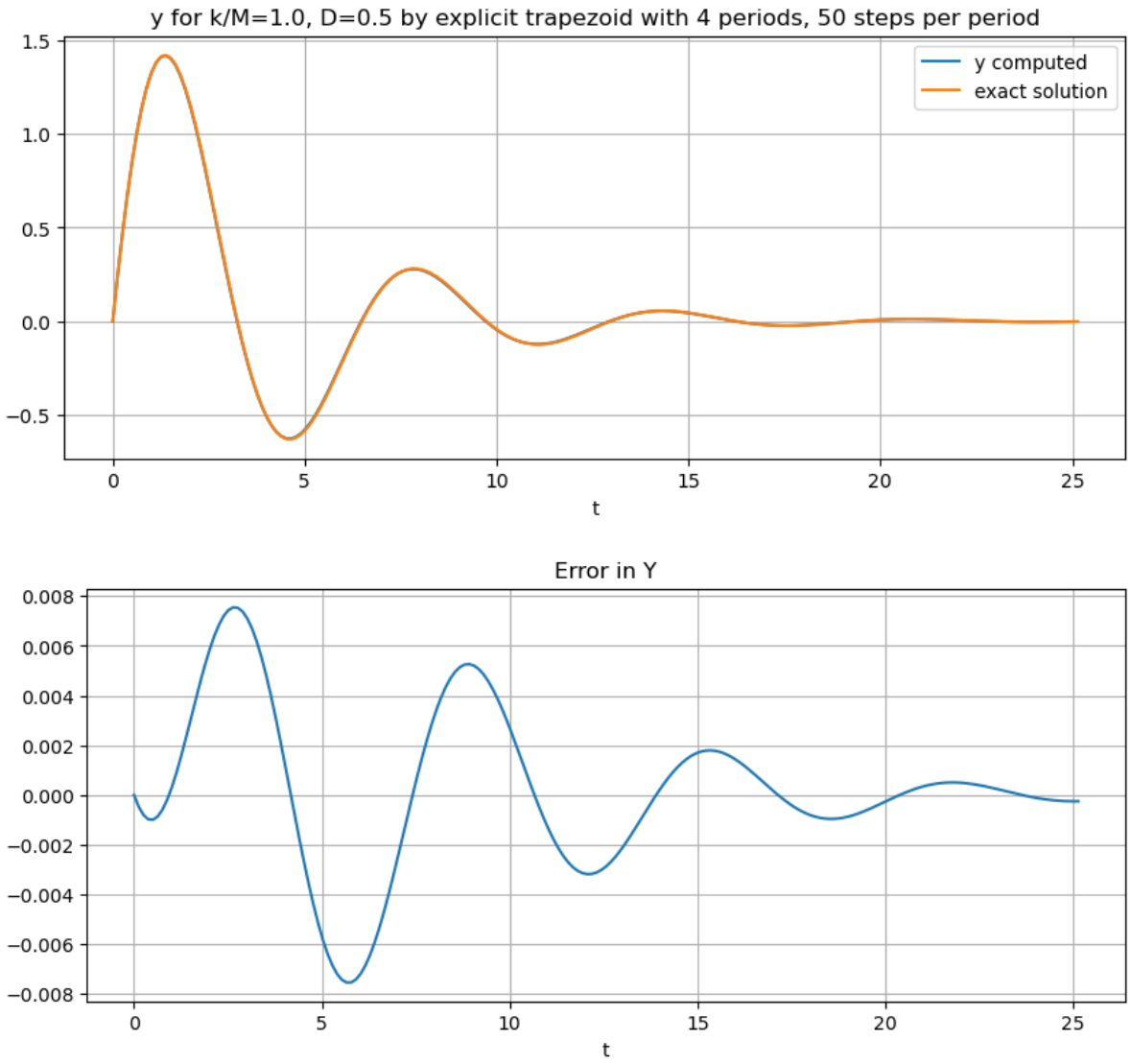
(continues on next page)

(continued from previous page)

```
    u = zeros(n+1, n_unknowns)
    u[1,:] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i,:])*h
        K_2 = f(t[i]+h, u[i,:]+K_1)*h
        u[i+1,:] = u[i,:] + (K_1 + K_2)/2.0
    end
    return (t, u)
end;
```

```
D = 0.5 # Underdamped: decaying oscillations
#D = 2 # Critically damped
#D = 2.1 # Overdamped: exponential decay

periods = 4
b = 2pi * periods

stepsperperiod = 50
n = Int(stepsperperiod * periods)

damping(k, M, D)

(t, U) = explicittrapezoid_system(f_mass_spring, a, b, u_0, n)
Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=u_0, K=K, M=M, D=D)  # Exact solution

damping(k, M, D)

figure(figsize=[10,4])
title("y for k/M=$(k/M), D=$D by explicit trapezoid with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
xlabel("t")
legend()
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)
```

```
Underdamped
Underdamped
```



At first glance this is doing well, keeping the orbits circular. However, note the discrepancy between the start and end points: these should be the same, as they are (visually) with the Runge-Kutta method.

```
function explicitmidpoint_system(f, a, b, u_0, n)
    # Use the Explict Midpoint Method (a.k.a Modified Euler) to solve the system
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    # The conversion for the system version is mainly "u[i] -> u[i,:]"

    h = (b-a)/n
    t = range(a, b, n+1)
    n_unknowns = length(u_0)
    u = zeros(n+1, n_unknowns)
    u[1,:] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i,:])*h
        K_2 = f(t[i]+h/2, u[i,:]+K_1/2)*h
        u[i+1,:] = u[i,:] + K_2
    end
```

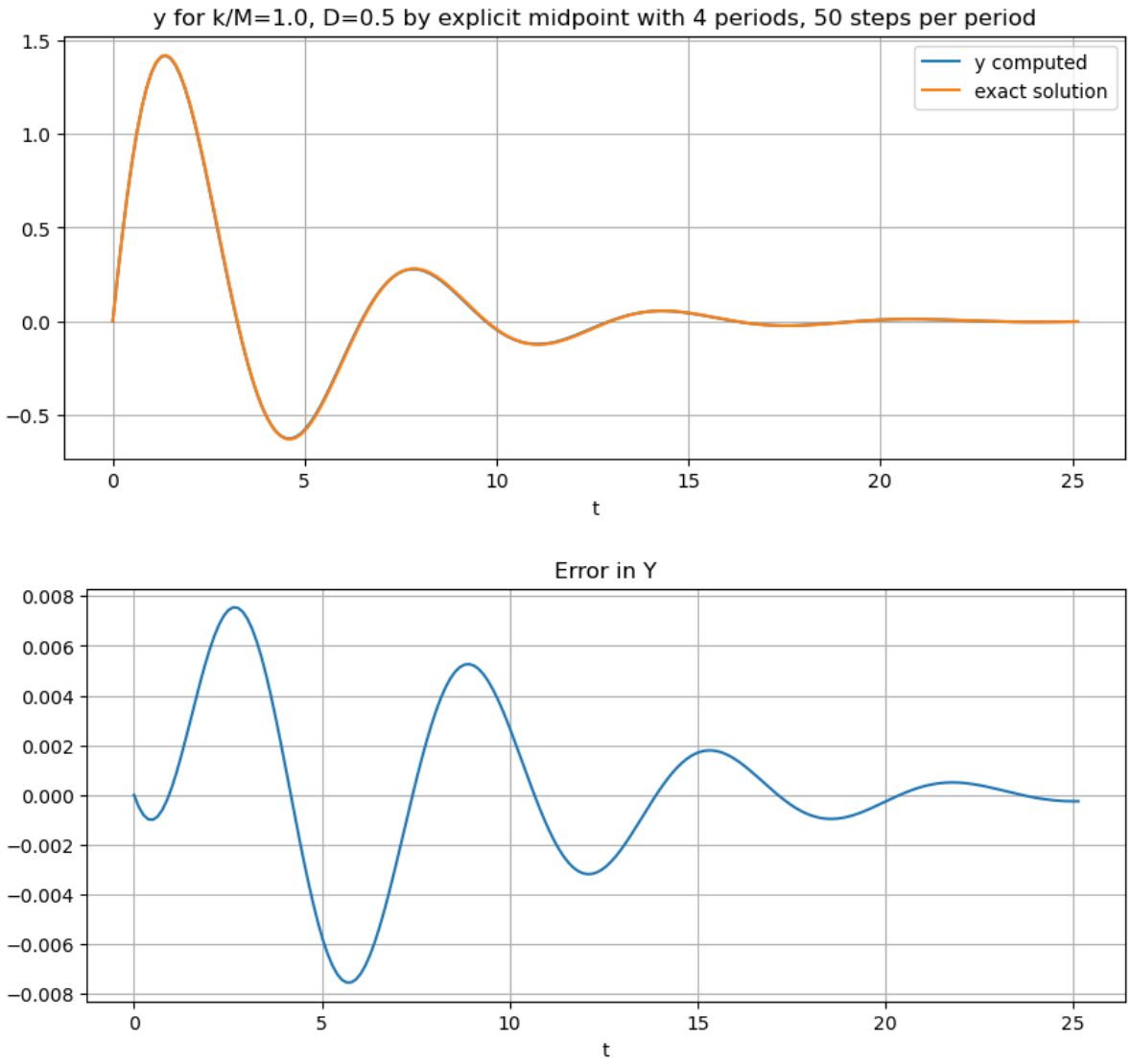
(continues on next page)

(continued from previous page)

```
    return (t, u)
end;
```

```
D = 0.5 # Underdamped: decaying oscillations
#D = 2 # Critically damped
#D = 2.1 # Overdamped: exponential decay

periods = 4
b = 2pi * periods

stepsperperiod = 50
n = Int(stepsperperiod * periods)

damping(k, M, D)

(t, U) = explicitmidpoint_system(f_mass_spring, a, b, u_0, n)
Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=u_0, K=K, M=M, D=D)  # Exact solution

damping(k, M, D)

figure(figsize=[10,4])
title("y for k/M=$(k/M), D=$D by explicit midpoint with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
xlabel("t")
legend()
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)
```

```
Underdamped
Underdamped
```



7.5 Error Control and Variable Step Sizes

References:

- Section 6.5 *Variable Step-Size Methods* in [Sauer, 2019].
- Section 5.5 *Error Control and the Runge-Kutta-Fehlberg Method* in [Burden *et al.*, 2016].
- Section 7.3 in [Chenney and Kincaid, 2012].

7.5.1 The Basic ODE Initial Value Problem

We consider again the initial value problem

$$\frac{du}{dt} = f(t,u) \quad a \le t \le b, \quad u(a) = u_0$$

We now allow the possibility that u and f are vector-valued as in the section *Systems of ODEs and Higher Order ODEs*, but omitting the tilde notation $\tilde{u}$, $\tilde{f}$.

7.5.2 Error Control by Varying the Time Step Size h_i

Recall the variable step-size version of Euler's method:

Algorithm 7.4

Input: f, a, b, n

$t_0 = a$

$U_0 = u_0$

$h = (b-a)/n$

for i in $[0, n)$:

Choose step size h_i somehow!

$t_{i+1} = t_i + h_i$

$U_{i+1} = U_i + h_i f(t_i, U_i)$

end

We now consider how to choose each step size, by estimating the error in each step, and aiming to have error per unit time below some limit like $\epsilon/(b-a)$, so that the global error is no more than about ϵ.

As usual, the theoretical error bounds like $O(h_i^2)$ for a single step of Euler's method are not enough for quantitative tasks like choosing h_i, but they do motivate more practical estimates.

7.5.3 A crude error estimate for Euler's Method: Richardson Extrapolation

Starting at a point (t, u(t)), we can estimate the error in Euler's method approximato at a slightly later time $t_i + h$ by using two approximations of $U(t+h)$:

- The value given by a step of Euler's method with step size h: call this U^h
- The value given by taking two steps of Euler's method each with step size $h/2$: call this $U_2^{h/2}$, because it involves 2 steps of size $h/2$.

The first order accuracy of Euler's method gives $e_h = u(t+h) - U^h \approx 2(u(t+h) - U_2^{h/2})$, so that

$$e_h \approx \frac{U_2^{h/2} - U^h}{2}$$

Step size choice

What do we do with this error information?

The first obvious ideas are:

- Accept this step if e_h is small enough, taking $h_i = h$, $t_{i+1} = t_i + h_i$, and $U_{i+1} = U^h$, but
- reject it and try again with a smaller h value otherwise; maybe halving h; but there are more sophisticated options too.

Exercise A

Write a formula for U_h and e_h if one starts from the point (t_i, U_i), so that $(t_i + h, U^h)$ is the proposed value for the next point (t_{i+1}, U_{i+1}) in the approximate solution — but only if e_h is small enough!

Error tolerance

One simple criterion for accuracy is that the estimated error in this step be no more than some overall upper limit on the error in each time step, T. That is, accept the step size h if

$$|e_h| \leq T$$

A crude approach to reducing the step size when needed

If this error tolerance is not met, we must choose a new step size h', and we can predict roughly its error behavior using the known order natue of the error in Euler's method: scaling dowen to $h' = sh$, the error in a single step scales with h^2 (in general it scales with h^{p+1} for a method of order p), and so to reduce the error by the needed factor $\frac{e_h}{T}$ one needs approximately

$$s^2 = \frac{T}{|e_h|}$$

and so using $e_h \approx \tilde{e}_h = |U^{h/2} - U^h|$ suggests using

$$s = \left(\frac{T}{|U^{h/2} - U^h|} \right)^{1/2}$$

However this new step size might have error that is still slightly too large, leading to a second failure. Another is that one might get into an infinite loop of step size reduction.

So refinements of this choice must be considered.

Increasing the step size when desirable

If we simply follow the above aproach, the step size, once reduced, will never be increased. This could lead to great inefficiency, through using an unecessarily small step size just because at an earlier part of the time domain, accuracy required very small steps.

Thus, after a successful time step, one might consider increasing h for the next step. This could be done using exactly the above formula, but again there are risks, so again refinement of this choice must be considered.

One problem is that if the step size gets too large, the error estimate can become unreliable; another is that one might need some minimum "temporal resolution", for nice graphs and such.

Both suggest imposing an upper limit on the step size h.

7.5.4 Another strategy for getting error estimates: two (related) Runge-Kutta methods

The recurring strategy of estimating errors by the difference of two different approximations — one expected to be far better than the other — can be used in a nice way here. I will first illustrate with the simplest version, using Euler's Mathod and the Explicit Trapezoid Method.

Recall that the increment in Euler's Method from time t to time $t + h$ is

$$K_1 = hf(t, U)$$

whereas for the Explict Trapezoid Method it is $(K_1 + K_2)/2$, as given by

$$K_1 = hf(t, U)$$
$$K_2 = hf(t + h, U + K_1)$$

Thus we can use the difference, $|K_1 - (K_1 + K_2)/2| = |(K_1 - K_2)/2|$ as an error estimate. In fact to be cautious, one often drops the factor of $1/2$, so using approximation $\tilde{e}_h = |K_1 - K_2|$.

One has to be careful: this estimates the error in Euler's Method, and one has to use it that way: using the less accurate value K_1 as the update.

A basic algorithm for the time step starting with t_i, U_i is

---

Algorithm 7.5

$K_1 \leftarrow hf(t_i, U_i)$

$K_2 \leftarrow hf(t_i + h, U_i + K_1)$

$e_h \leftarrow |K_1 - K_2|$

$s \leftarrow \sqrt{T/e_h}$

if $e_h < T$

$U_{i+1} = U_i + K_1$

$t_{i+1} = t_i + h$

Increase h for the *next* time step:

$h \leftarrow sh$

else: (not good enough: reduce h and try again)

$h \leftarrow sh$

Start again from $K_1 = ...$

end

However, in practice one needs:

- An upper limit h_{max} on the step size h, partly because error estimates become unreliable if h gets too large, and also becuase subsequent use of the results (like graphs) might need sufficiently "fine" data.
- A lower limit h_{max} on h, to avoid infinite loops and such.
- Since we are using only an approximation $\tilde{e}_h$ of e_h, and out of general caution, it is typical to include a "safety factor" of about 0.8 or 0.9, when computing the *next* time step: reducing the step size scale factor to $S = 0.9\sqrt{T/e_h}$.

Incorporating these refinements:

Algorithm 7.6

$K_1 = hf(t_i, U_i)$

$K_2 = hf(t_i + h, U_i + K_1)$

$e_h = |K_1 - K_2|$

$s = 0.9\sqrt{T/e_h}$

if $e_h < T$

$U_{i+1} = U_i + K_1$

$t_{i+1} = t_i + h$

Increase h for the *next* time step:

$h \leftarrow \min(0.9sh, h_{max})$

else: (not good enough; reduce h and try again)

$h \leftarrow \max(0.9sh, h_{min})$

Start again from $K_1 = ...$

end

Exercise B

Implement the above, and test on the two familiar examples

$$du/dt = Ku$$

and

$$du/dt = K(\cos(t) - u) - \sin(t)$$

($K = 1$ is enough.)

Partial Solution to Exercise B

```
using PyPlot
using LinearAlgebra: norm
```

```
import Base: round
round(x,n) = round(x,sigdigits=n);
```

```
function eulermethod_errorcontrol(f, a, b, u_0; errortolerance=1e-3, h_min=1e-6, h_
↪max=0.1, steps_max=1000, demomode=false)
    steps = 0
    t_i = a
    U_i = u_0
    t = [t_i]
    U = [U_i]
```

(continues on next page)

(continued from previous page)

```
    h = h_max  # Start optimistically!
    while t_i < b && steps < steps_max
        K_1 = h*f(t_i, U_i)
        K_2 = h*f(t_i + h/2, U_i + K_1/2)
        errorestimate = abs(K_1 - K_2)
        s = 0.9 * sqrt(errortolerance/errorestimate)
        if errorestimate <= errortolerance  # Success!
            t_i += h
            U_i += K_1
            append!(t, t_i)
            append!(U, U_i)
            # Adjust step size up, but not too big
            h = min(s*h, h_max)
        else  # Innacurate; reduce step size and try again
            h = max(s*h, h_min)
            if demomode
                println("t_i=$t_i: Decreasing step size to $(round(h,4)) and trying␣
 ↪again.")
            end
        end
        # A refinement not mentioned above; the next step should not overshoot t=b:
        if t_i + h > b
            h = b - t_i
        end
        steps += 1
    end
    return (t, U)
    # Note: if the step count ran out, this does not reach t=b, but at least it is␣
 ↪correct as far as it goes
end;
```

```
f(t, u) = K*u;
```

```
a = 1.0
b = 3.0
u_0 = 2.0
K = 1.0
u(t) = u_0*exp(K*(t-a));
```

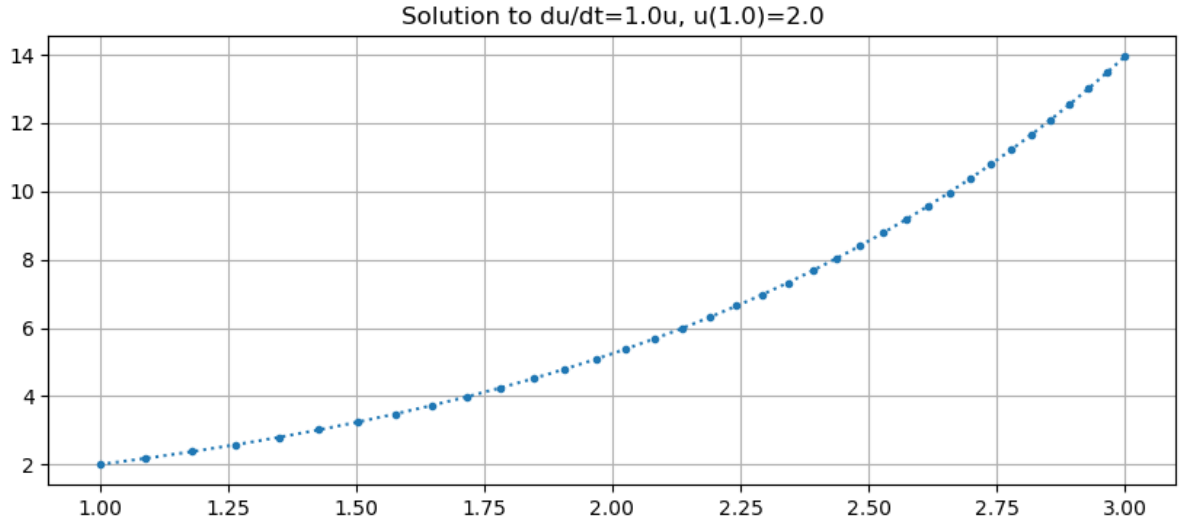
```
errortolerance = 1e-2
time_start = time()
(t, U) = eulermethod_errorcontrol(f, a, b, u_0; errortolerance=errortolerance,␣
 ↪demomode=true)
time_end = time()
time_elapsed = time_end - time_start

steps = length(U) - 1
h_ave = (b-a)/steps
U_exact = u.(t)
U_error = U_exact - U
U_max = norm(U_error, Inf)
println()
println("With error tolerance $errortolerance, this took $steps time steps, of␣
 ↪average length $(round(h_ave,4))")
```

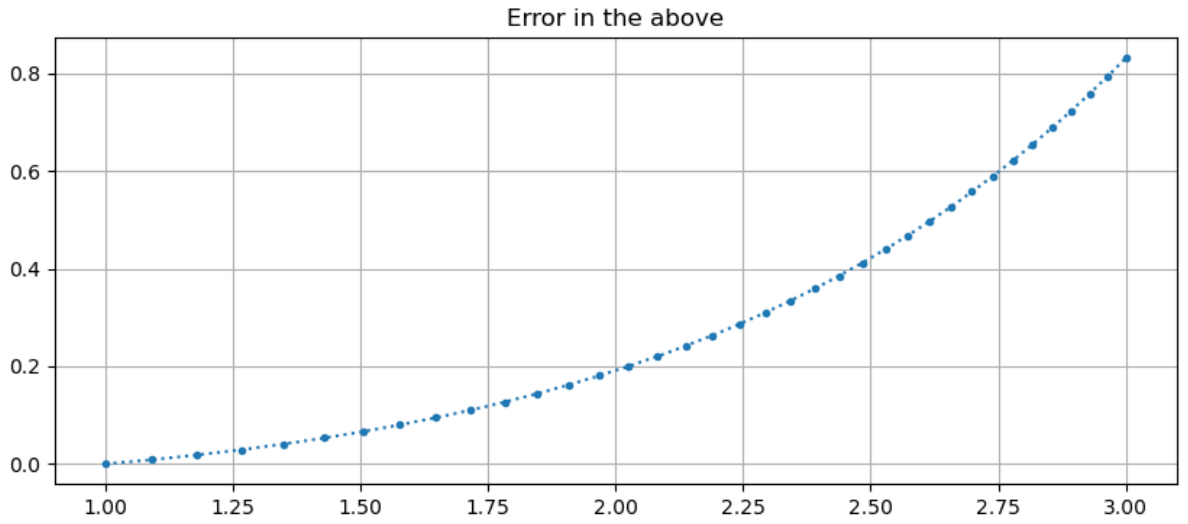
(continues on next page)

(continued from previous page)

```
println("The maximum absolute error is $(round(U_max,4))")
println("The maximum absolute error per time step is $(round(U_max/steps,4))")
println("The time taken to solve was $(round(time_elapsed,4)) seconds")

figure(figsize=[10,4])
title("Solution to du/dt=$(K)u, u($a)=$u_0")
plot(t, U, ".:")
grid(true)

figure(figsize=[10,4])
title("Error in the above")
plot(t, U_error, ".:")
grid(true);
```

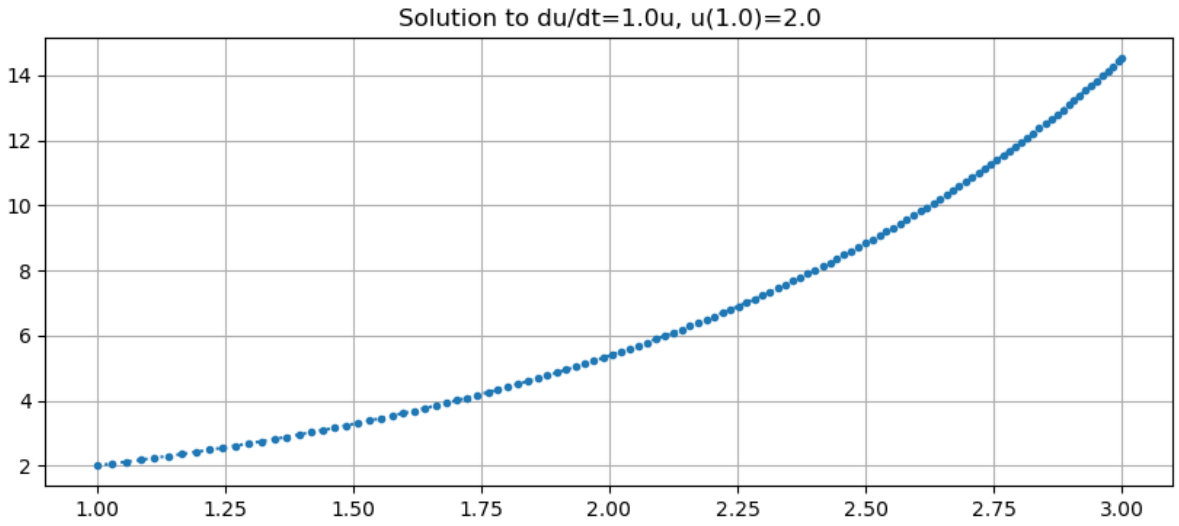
```
t_i=1.0: Decreasing step size to 0.09 and trying again.

With error tolerance 0.01, this took 36 time steps, of average length 0.05556
The maximum absolute error is 0.8311
The maximum absolute error per time step is 0.02309
The time taken to solve was 0.09479 seconds
```

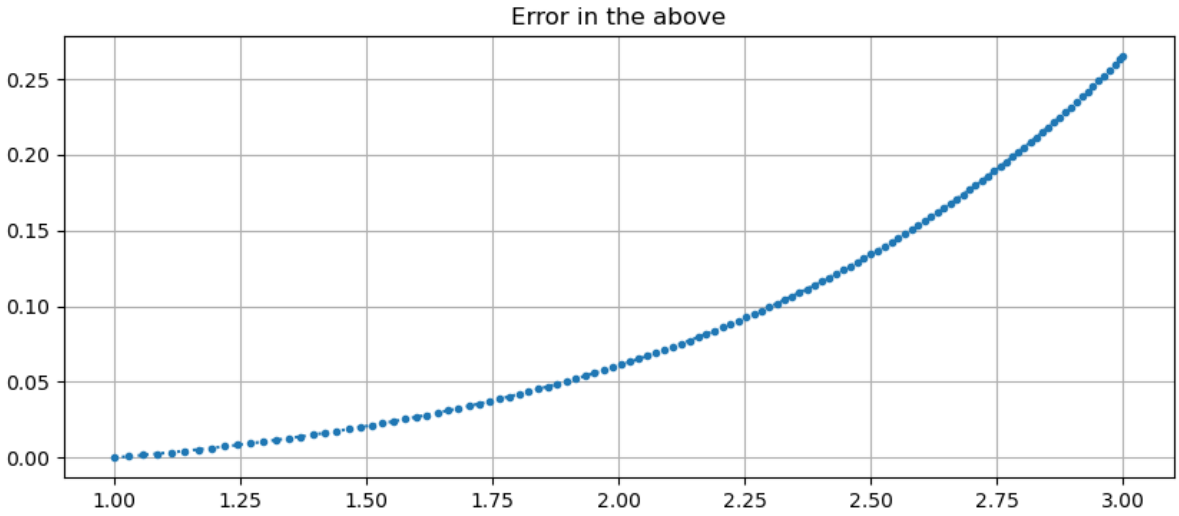





```
errortolerance = 1e-3
time_start = time()
(t, U) = eulermethod_errorcontrol(f, a, b, u_0; errortolerance=errortolerance,␣
 ↪demomode=true)
time_end = time()
time_elapsed = time_end - time_start

steps = length(U) - 1
h_ave = (b-a)/steps
U_exact = u.(t)
U_error = U_exact - U
U_max = norm(U_error, Inf)
println()
println("With error tolerance $errortolerance, this took $steps time steps, of␣
 ↪average length $(round(h_ave,4))")
println("The maximum absolute error is $(round(U_max,4))")
println("The maximum absolute error per time step is $(round(U_max/steps,4))")
println("The time taken to solve was $(round(time_elapsed,4)) seconds")

figure(figsize=[10,4])
title("Solution to du/dt=$(K)u, u($a)=$u_0")
plot(t, U, ".:")
grid(true)

figure(figsize=[10,4])
title("Error in the above")
plot(t, U_error, ".:")
grid(true);
```

```
t_i=1.0: Decreasing step size to 0.02846 and trying again.

With error tolerance 0.001, this took 119 time steps, of average length 0.01681
The maximum absolute error is 0.2648
The maximum absolute error per time step is 0.002225
The time taken to solve was 0.000586 seconds
```





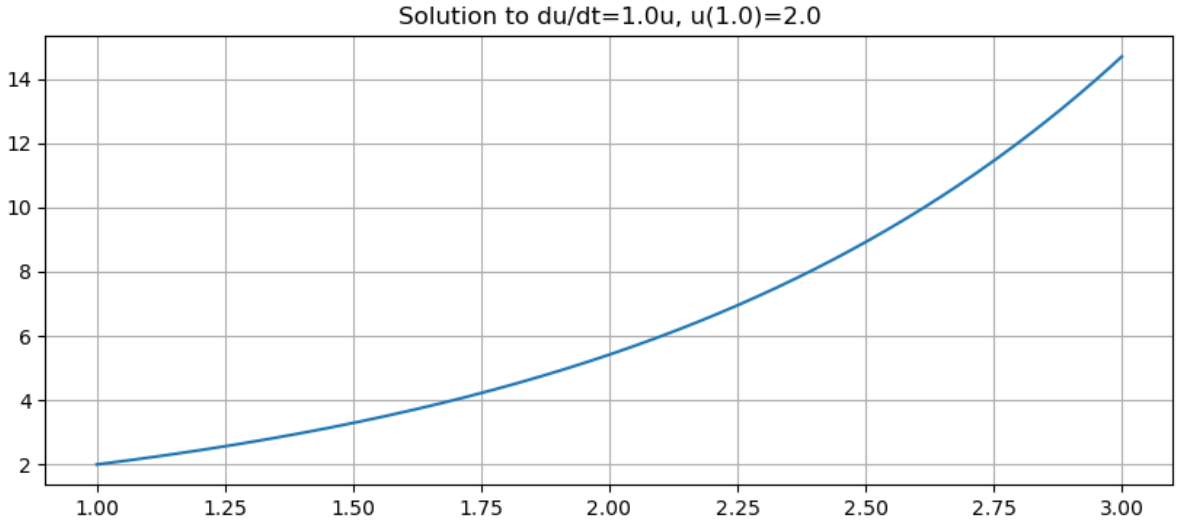

```
errortolerance = 1e-4
time_start = time()
(t, U) = eulermethod_errorcontrol(f, a, b, u_0; errortolerance=errortolerance,␣
 ↪demomode=true)
time_end = time()
time_elapsed = time_end - time_start

steps = length(U) - 1
h_ave = (b-a)/steps
U_exact = u.(t)
U_error = U_exact - U
U_max = norm(U_error, Inf)
println()
println("With error tolerance $errortolerance, this took $steps time steps, of␣
 ↪average length $(round(h_ave,4))")
println("The maximum absolute error is $(round(U_max,4))")
println("The maximum absolute error per time step is $(round(U_max/steps,4))")
println("The time taken to solve was $(round(time_elapsed,4)) seconds")
```

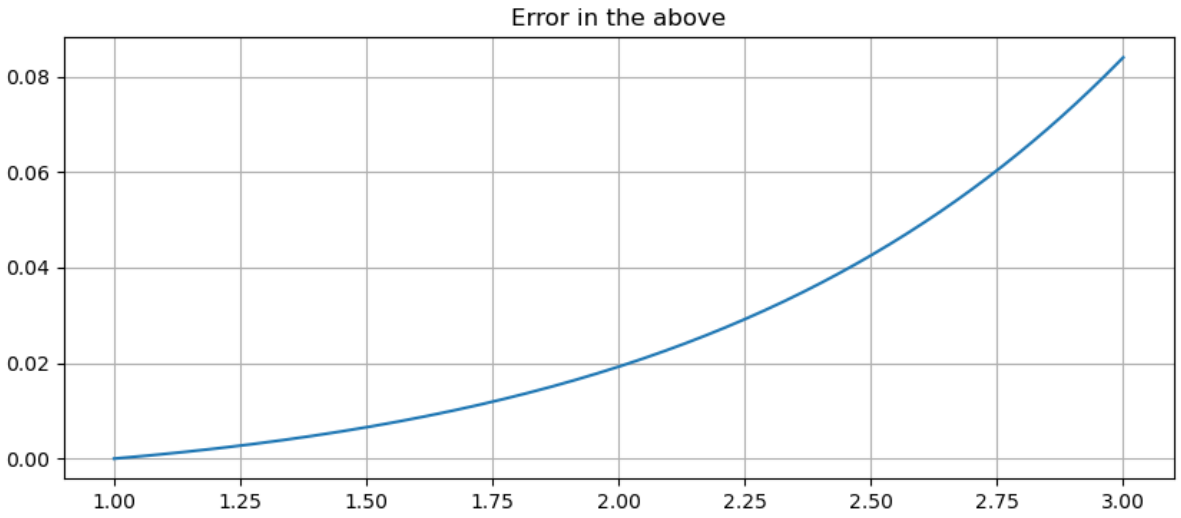
(continues on next page)

(continued from previous page)

```
figure(figsize=[10,4])
title("Solution to du/dt=$(K)u, u($a)=$u_0")
plot(t, U)
grid(true)

figure(figsize=[10,4])
title("Error in the above")
plot(t, U_error)
grid(true);
```

```
t_i=1.0: Decreasing step size to 0.009 and trying again.

With error tolerance 0.0001, this took 380 time steps, of average length 0.005263
The maximum absolute error is 0.08398
The maximum absolute error per time step is 0.000221
The time taken to solve was 0.0007212 seconds
```





7.5.5 The explicit trapezoid method with error control

In practice, one usually needs at least second order accuracy, and one approach to that is using computing a "candidates" for the next time step with a second order accurate Runge-Kutta method and also a third order accurate one, the latter used only to get an error estimate for the former.

Perhaps the simplest of these is based on adding error estimation to the Explicit Trapezoid Rule. Omitting the step size adjustment for now, the main ingredients are:

Algorithm 7.7

$K_1 = hf(t, U)$

$K_2 = hf(t + h, U + K_1)$

(So far, as for the explicit trapezoid method)

$K_3 = hf(t + h/2, U + (K_1 + K_2)/4)$

(a midpoint approximation, using the above)

$\delta_2 = (K_1 + K_2)/2$

(The order 2 increment as for the explicit trapezoid method)

$\delta_3 = (K_1 + 4K_3 + K_2)/6$

(An order 3 increment — note the resemblance to Simpson's Rule for integration. This is only used to get the final error estimate below)

$e_h = |\delta_2 - \delta_3|, = |K_1 - 2K_3 + K_2|/3$

Again, if this step is accepted, one uses the explicit trapezoid rule step: $U_{i+1} = U_i + \delta_2$.

Step size adjustment

The scale factor s for step size adjustment must be modified for a method order p (with $p = 2$ now):

- Changing step size by a factor s will change the error e_h in a single time step by a factor of about s^{p+1}.
- Thus, we want a new step with this rescaled error of about $s^{p+1}e_h$ roughly matching the tolerance T. Equating would give $s^{p+1}e_h = T$, so $s = (T/e_h))^{1/(p+1)}$, but as noted above, since we are using only an approximation $\widetilde{e}_h$ of e_h it is typical to include a "safety factor" of about 0.9, so something like

$$s = 0.9 \left(\frac{T}{|\widetilde{e}_h|} \right)^{1/(p+1)}$$

Thus for this second order accurate method, we then get

$$s = 0.9 \left(\frac{3T}{|K_1 - 2K_3 + K_2|} \right)^{1/3}$$

A variant: relative error control

One final refinement: it is more common in software to impose a *relative error* bound: aiming for $|e_h/u(t)| \leq T$, or $|e_h| \leq T|u(t)|$. Approximating $u(t)$ by U_i, this changes the step size rescaling guideline to

$$s = 0.9 \left| \frac{TU_i}{\widetilde{e}_h} \right|^{1/(p+1)}$$

Exercise C

Implement *The explicit trapezoid method with error control*, and test on the two familiar examples

$$du/dt = Ku$$
and
$$du/dt = K(\cos(t) - u) - \sin(t)$$

($K = 1$ is enough.)

7.5.6 Fourth order accurate methods with error control: Runge-Kutta-Felberg and some newer refinements

The details involve some messy coefficients; see the references above for those.

The basic idea is to devise a fifth order accurate Runge-Kutta method such that we can also get a fourth order accurate method from the same colection of *stages* K_i values. One catch is that any such fifth order method requires six stages (not five as you might have guessed).

The first such method, still widely used, is the Runge-Kutta-Felberg Method published by Erwin Fehlberg in 1970:

Algorithm 7.8 (RUnge-Kutta-Fehlberg)

$K_1 = hf(t, U)$

$K_2 = f(t + \frac{1}{4}h, U + K_1/4)$

$K_3 = f(t + \frac{3}{8}h, U + \frac{3}{32}K_1 + \frac{9}{32}K_2)$

$K_4 = f(t + \frac{12}{13}h, U + \frac{1932}{2197}K_1 - \frac{7200}{2197}K_2 + \frac{7296}{2197}K_3)$

$K_5 = f(t + h, U + \frac{439}{216}K_1 - 8K_2 + \frac{3680}{2565}K_3 - \frac{845}{4104}K_4)$

$K_6 = f(t + \frac{1}{2}h, U - \frac{8}{27}K_1 + 2K_2 - \frac{3544}{513}K_3 + \frac{1859}{4104}K_4 - \frac{11}{40}K_5)$

$\delta_4 = \frac{25}{216}K_1 + \frac{1408}{2565}K_3 + \frac{2197}{4104}K_4 - \frac{1}{5}K_5$

(The order 4 increment that will actually be used)

$\delta_5 = \frac{16}{135}K_1 + \frac{6656}{12825}K_3 + \frac{28561}{56430}K_4 - \frac{9}{50}K_5 + \frac{2}{55}K_6$

(The order 5 increment, used only to get the following error estimate)

$\tilde{e}_h = \frac{1}{360}K_1 - \frac{128}{4275}K_3 + \frac{2197}{75240}K_4 + \frac{1}{50}K_5 + \frac{2}{55}K_6$

This method is typically used with the relative error control mentioned above, and since the order is $p = 4$, the recommended step-size rescaling factor is

$$s = 0.9\left|\frac{TU_i}{\tilde{e}_h}\right|^{1/5}, = 0.9\left|\frac{TU_i}{\frac{1}{360}K_1 - \frac{128}{4275}K_3 + \frac{2197}{75240}K_4 + \frac{1}{50}K_5 + \frac{2}{55}K_6}\right|^{1/5},$$

7.5.7 ODE solvers in Julia package `DifferentialEquations`

Newer software often uses variants such as the method of Dormand–Prince method published in 1980 or that of Tsitouras published in 2011.

These (and many others) are available in the Julia package `DifferentialEquations` as `DP5` and `Tsit45` respectively.

Example using package `DifferentialEquations` — to be added

7.6 An Introduction to Multistep Methods

References:

- Section 6.7 *Multistep Methods* of [Sauer, 2019].
- Section 5.6 *Multistep Methods* of [Burden *et al.*, 2016].

7.6.1 Introduction

When approximating derivatives we saw that there is a distinct advantage in accuracy to using the centered difference approximation

$$\frac{df}{dt}(t) \approx \delta_h f(t) := \frac{f(t+h) - f(t-h)}{2h}$$

(with error $O(h^2)$) over the forward difference approximation

$$\frac{df}{dt}(t) \approx \Delta_h f(t) := \frac{f(t+h) - f(t)}{h}$$

(which has error $O(h)$).

However Euler's method used the latter, and all ODE methods seen so far avoid using values at "previous" times like $t - h$. There is a reason for this, as using data from previous times introduces some complications, but sometimes those are worth overcoming, so let us look into this.

In this section, we look at one simple multi-step method, based on the above centered-difference derivative approximation.

Future sections will look at higher order methods such as the Adams-Bashforth and Adams-Moulton methods.

7.6.2 The Leapfrog method

Inserting the above centered difference approximation of the derivative into the ODE $du/dt = f(t, u)$ gives

$$\frac{u(t+h) - u(t-h)}{h} \approx f(t, u(t))$$

which leads to the **leapfrog method**

$$\frac{U_{i+1} - U_{i-1}}{2h} = f(t_i, U_i)$$

or

$$U_{i+1} = U_{i-1} + 2hf(t_i, U_i)$$

This is the first example of a

Definition 7.1 (Multistep Method)

A **multistep method** for numerical solution of an ODE IVP $du/dt = f(t,u)$, $u(t_0) = u_0$ is one of the form

$$U_{i+1} = \phi(U_i, \ldots U_{i-s+1}, h), \quad s > 1$$

so that the new approximate value of $u(t)$ depends on approximate values at multiple previous times.

More specifically, this is called an s-step method.

This jargon is consistent with all methods seen in earlier sections being called *one-step methods*. For example, Euler's method can be written as

$$U_{i+1} = \phi_E(U_i, h) := U_i + hf(t_i, U_i)$$

and the explicit midpoint method can be written as the one-liner

$$U_{i+1} = \phi_{EMP}(U_i, h) := U_i + hf(t_i + h/2, U_i + hf(t_i, U_i)/2)h$$

The leapfrog method already illustrates two of the complications that arise with multistep methods:

- The initial data $u(a) = u_0$ gives U_0, but then the above formula gives U_1 in terms of U_0 and the non-existent value U_{-1}; a different method is needed to get U_1. More generally, with an s-step methods, one needs to compute the first $s-1$ steps, up to U_{s-1}, by some other method.
- leapfrog needs a constant step size h; the strategy of error estimation and error control using variable step size is still possible with some multistep methods, but that is distinctly more complicated than we have seen with one-step methods, and is not addressed in these notes.

Fortunately, many differential equations can be handled well by choosing a suitable fixed step size h. Thus, in these notes we work only with equal step sizes, so that our times are $t_i = a + ih$ and we aim for approximations $U_i \approx u(a + ih)$.

7.6.3 Second order accuracy of the leapfrog method

Using the fact that the centered difference approximation is second order accurate, one can verify that

$$\frac{u(t_{i+1}) - u(t_{i-1})}{2h} - f(t, u(t_i)) = O(h^2)$$

(Alternatively one can get this by inserting quadratic Taylor polynomials centered at t_i, and their error terms.)

The definition of local trunctation error needs to be extended slightly: it is the error $U_{i+1} - u(t_{i+1})$ when one starts with exact values for all previous steps; that is, assuming $U_j = u(t_j)$ for all $j \le i$.

The above results then shows that the local truncation error in each step is $U_{i+1} - u(t_{i+1}) = O(h^3)$, so that the "local truncation error per unit time" is

$\$\frac{U_{i+1} - u(t_{i+1})}{h} = O(h^2)\$$.

A theorem in the section on *A Global Error Bound for One Step Methods* says that when a one-step methods has local truncation error per unit time of $O(h^p)$ it also has global truncation error of the same order. The situation is a bit more complicated with multi-step methods, but loosely:

if the errors in a multistep method has local truncation $O(h^p)$ **and** it converges (i.e. the global error goes to zero at $h \to 0$) then it does so at the expected rate of $O(h^p)$.

But multi-step methods can fail to converge, even if the local truncation error is of high order! This is dealt with via the concept of **stability**; not considered here, but addressed in both references above, and a topic for future expansion of these notes.

In particular, when the leapfrog method converges it is second order accurate, just like the centered difference approximation of du/dt that it is built upon.

7.6.4 The speed advantage of multi-step methods like the leapfrog method

This second order accuracy illustrates a major potential advantage of multi-step methods: whereas any one-step Runge-Kutta method that is second order accurate (such as the explicit trapezoid or explicit midpoint methods) require at least two evaluations of $f(t, u)$ for each time step, the leapfrog methods requires only one.

More generally, for every s, there are s-step methods with errors $O(h^s)$ that require only one evaluation of $f(t, u)$ per time step — for example, the Adams-Bashforth methods, as seen at

- Section *Adams-Bashforth Multistep Methods*
- https://en.wikipedia.org/wiki/Linear_multistep_method#Adams-Bashforth_methods
- https://en.m.wikiversity.org/wiki/Adams-Bashforth_and_Adams-Moulton_methods
- [Sauer, 2019] Section 6.7.1 and 6.7.2
- [Burden *et al.*, 2016] Section 5.6

In comparison, any *explicit* one-step method order p require at least p evaluations of $f(t, u)$ per time step.

(See the *Implicit Methods: Adams-Moulton* for the distinction between explicit and implicit methods.)

```
using PyPlot
```

```
include("NumericalMethods.jl")
```

```
Main.NumericalMethods
```

```
import .NumericalMethods as NM
```

```
function leapfrog(f, a, b, U_0, U_1, n)
    n_unknowns = length(U_0)
    t = range(a, b, n+1)
    u = zeros(n+1, n_unknowns)
    u[1,:] = U_0
    u[2,:] = U_1
    h = (b-a)/n
    for i in 2:n
        u[i+1,:] = u[i-1,:] + 2*h*f(t[i], u[i,:])
    end
    return (t, u)
end;
```

Demo with the mass-spring system

As seen in the section *Systems of ODEs and Higher Order ODEs*

the damped mass-spring equation is

$$M\frac{d^2y}{dt^2} = -Ky - D\frac{dy}{dt}$$

with initial conditions

$$y(a) = y_0$$
$$\left.\frac{dy}{dt}\right|_{t=a} = v_0$$

with first-order system form

$$\frac{du_0}{dt} = u_1$$
$$\frac{du_1}{dt} = -\frac{K}{M}u_0 - \frac{D}{M}u_1$$

with initial conditions

$$u_0(a) = y_0$$
$$u_1(a) = v_0$$

The right-hand side f is given by

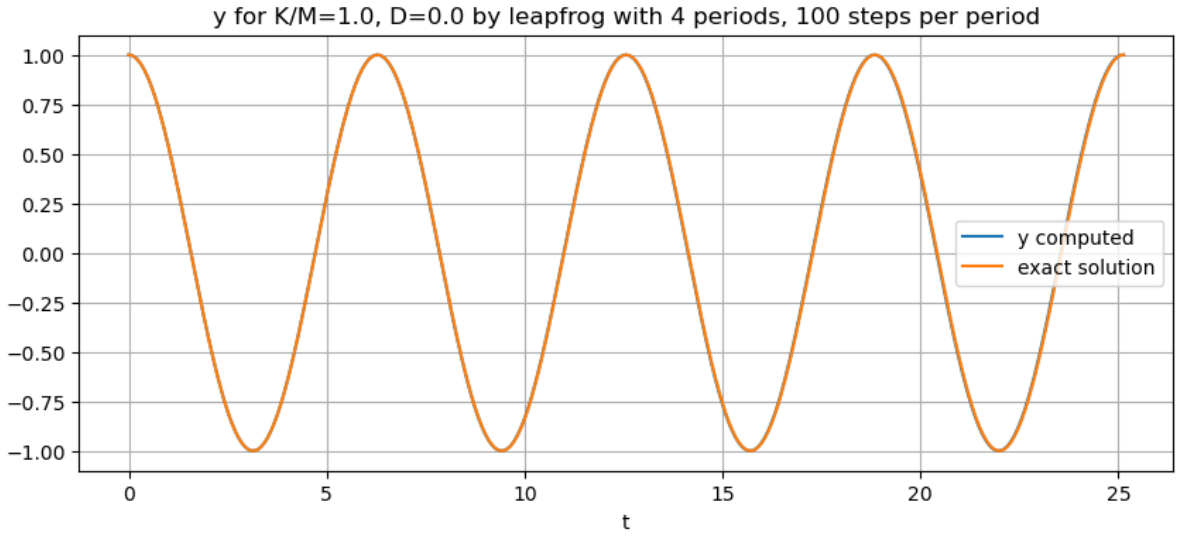
```
f_mass_spring(t, u) = [ u[2], -(K/M)*u[1] - (D/M)*u[2] ];
```

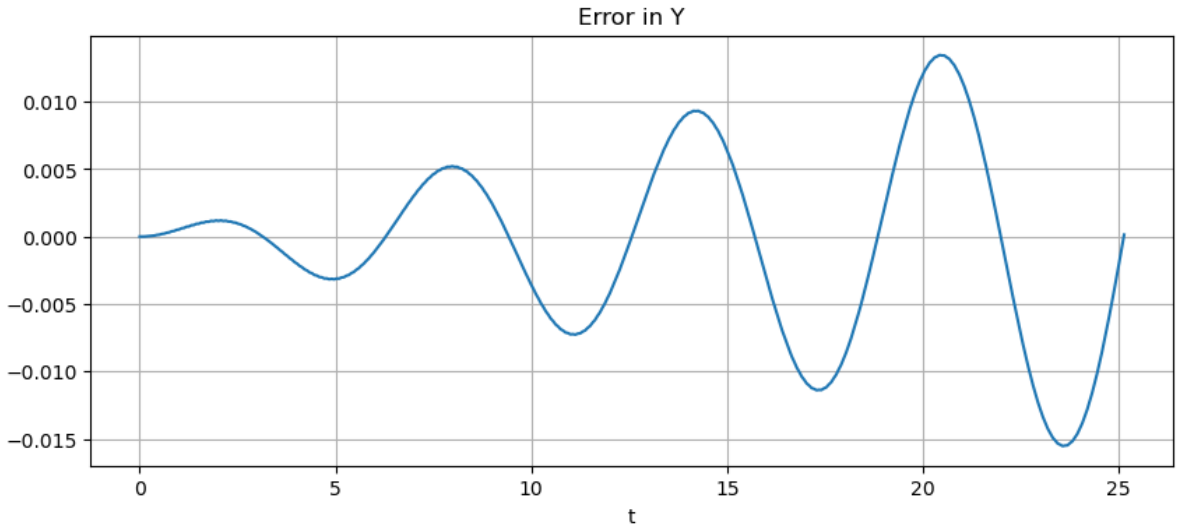
and the solutions seen in that section are given by function `y_mass_spring`

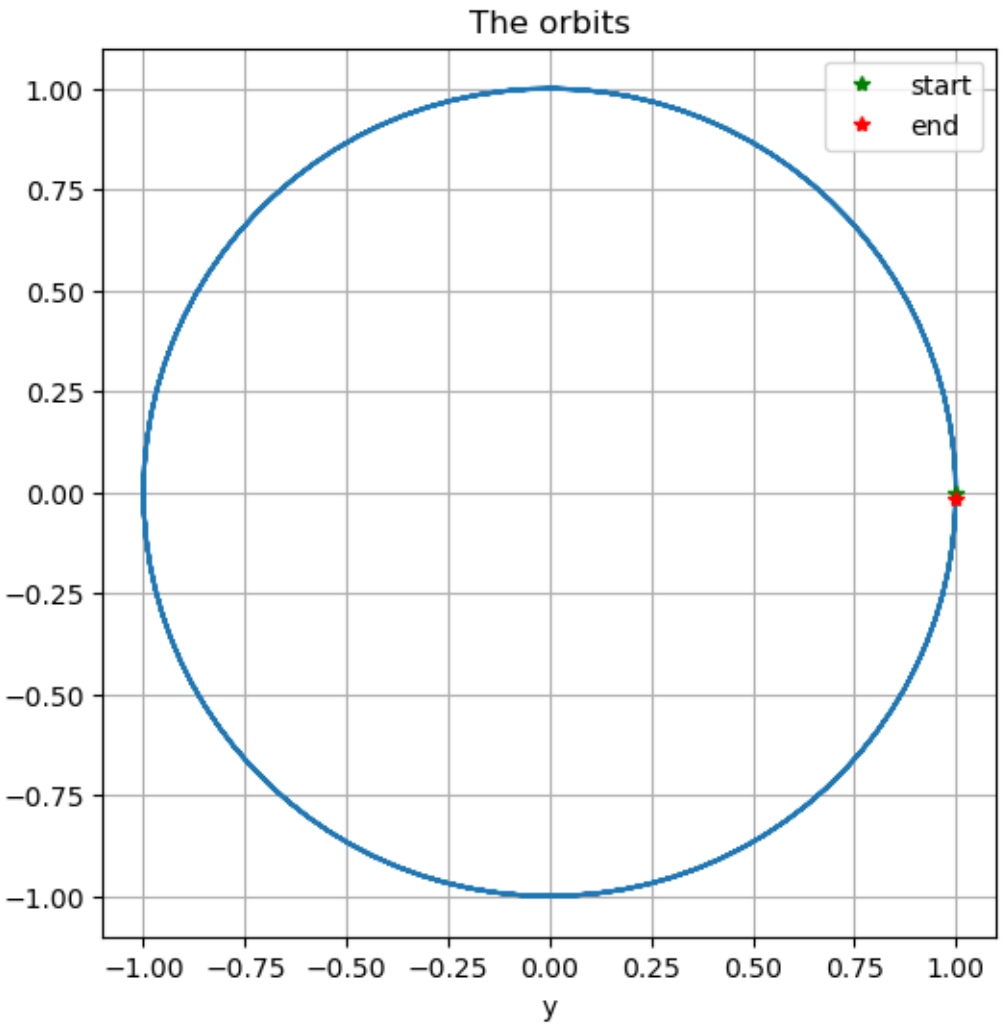
```
function y_mass_spring(t; t_0, u_0, K, M, D)
    (y_0, v_0) = u_0
    discriminant = D^2 - 4K*M
    if discriminant < 0  # underdamped
        omega = sqrt(4K*M - D^2)/(2M)
        A = y_0
        B = (v_0 + y_0*D/(2M))/omega
        return exp(-D/(2M)*(t-t_0)) * ( A*cos(omega*(t-t_0)) + B*sin(omega*(t-t_0)))
    elseif discriminant > 0  # overdamped
        Delta = sqrt(discriminant)
        lambda_plus = (-D + Delta)/(2M)
        lambda_minus = (-D - Delta)/(2M)
        A = M*(v_0 - lambda_minus * y_0)/Delta
        B = y_0 - A
        return A*exp(lambda_plus*(t-t_0)) + B*exp(lambda_minus*(t-t_0))
    else
        lambda = -D/(2M)
        A = y_0
        B = v_0 - A * lambda
        return (A + B*t)*exp(lambda*(t-t_0))
    end
end;
```

which alternatively could be imported from module `NumericalMethods`.

```
M = 1.0
K = 1.0
D = 0.0
U_0 = [1.0, 0.0]
a = 0.0
periods = 4
b = 2pi * periods

# Note: In the notes on systems, the second order methods were tested with 50 steps␣
 ↪per period
#stepsperperiod = 50  # As for the second order accurate explicit trapezoid and␣
 ↪midpoint methods
stepsperperiod = 100  # Equal cost per unit time as for the explicit trapezoid and␣
 ↪midpoint and Runge-Kutta methods

n = Int(stepsperperiod * periods)

# We need U_1, and get it with the Runge-Kutta method;
# this is overkill for accuracy, but since only one step is needed, the time cost is␣
 ↪negligible.
h = (b-a)/n
(t_1step, U_1step) = NM.rungekutta_system(f_mass_spring, a, a+h, U_0, 1)
U_1 = U_1step[end,:]
(t, U) = leapfrog(f_mass_spring, a, b, U_0, U_1, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("y for K/M=$(K/M), D=$D by leapfrog with $periods periods, $stepsperperiod␣
 ↪steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
 ↪"circular spirals"
title("The orbits")
plot(Y, DY)
xlabel("y")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```


y for K/M=1.0, D=0.0 by leapfrog with 4 periods, 100 steps per period
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
25
t
y computed
exact solution


Error in Y
0.010
0.005
0.000
−0.005
−0.010
−0.015
0
5
10
15
20
25
t

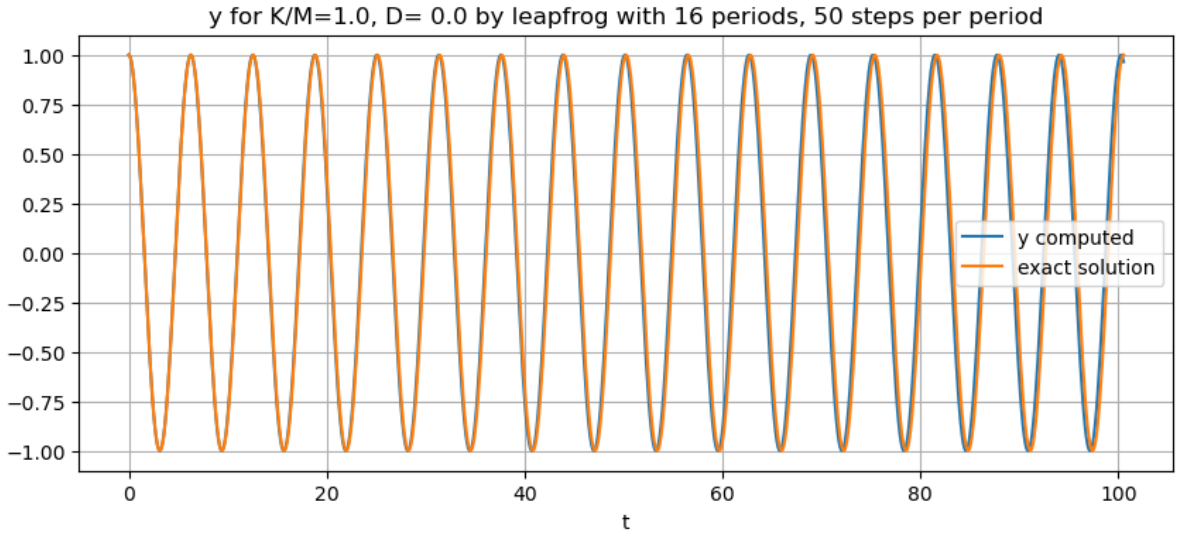


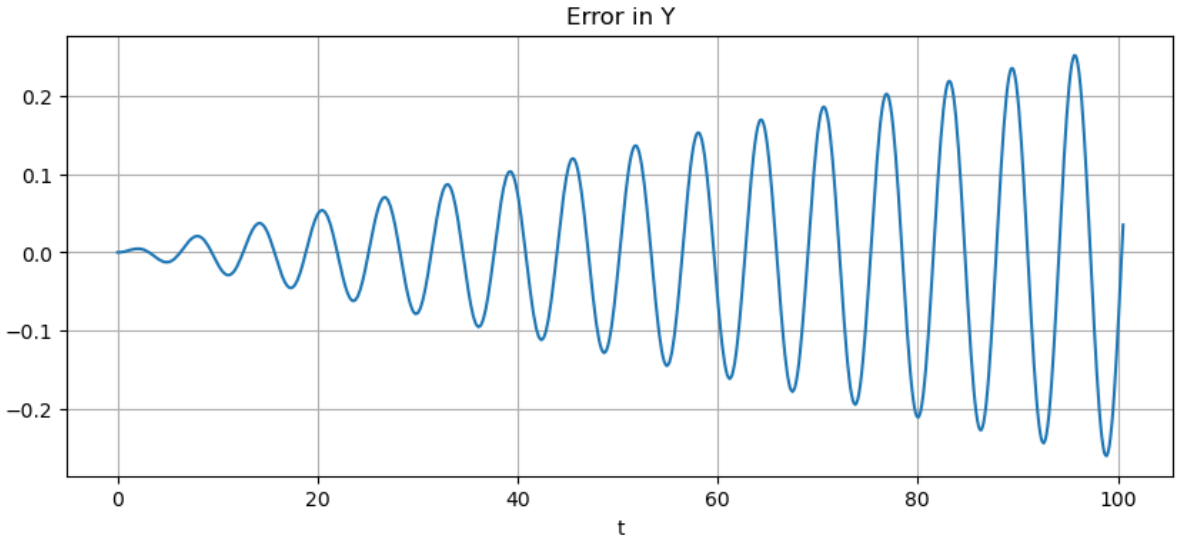

```
D = 0.0

periods = 16
b = 2pi * periods

# Note: In the notes on systems, the second order methods were tested with 50 steps␣
 ↪per period
stepsperperiod = 50  # As for the second order accurate explicit trapezoid and␣
 ↪midpoint methods
#stepsperperiod = 100  # Equal cost per unit time as for the explicit trapezoid and␣
 ↪midpoint and Runge-Kutta methods
n = Int(stepsperperiod * periods)

# We need U_1, and get it with the Runge-Kutta method;
# this is overkill for accuracy, but since only one step is needed, the time cost is␣
 ↪negligible.
h = (b-a)/n
(t_1step, U_1step) = NM.rungekutta_system(f_mass_spring, a, a+h, U_0, 1)
U_1 = U_1step[end,:]
(t, U) = leapfrog(f_mass_spring, a, b, U_0, U_1, n)
```

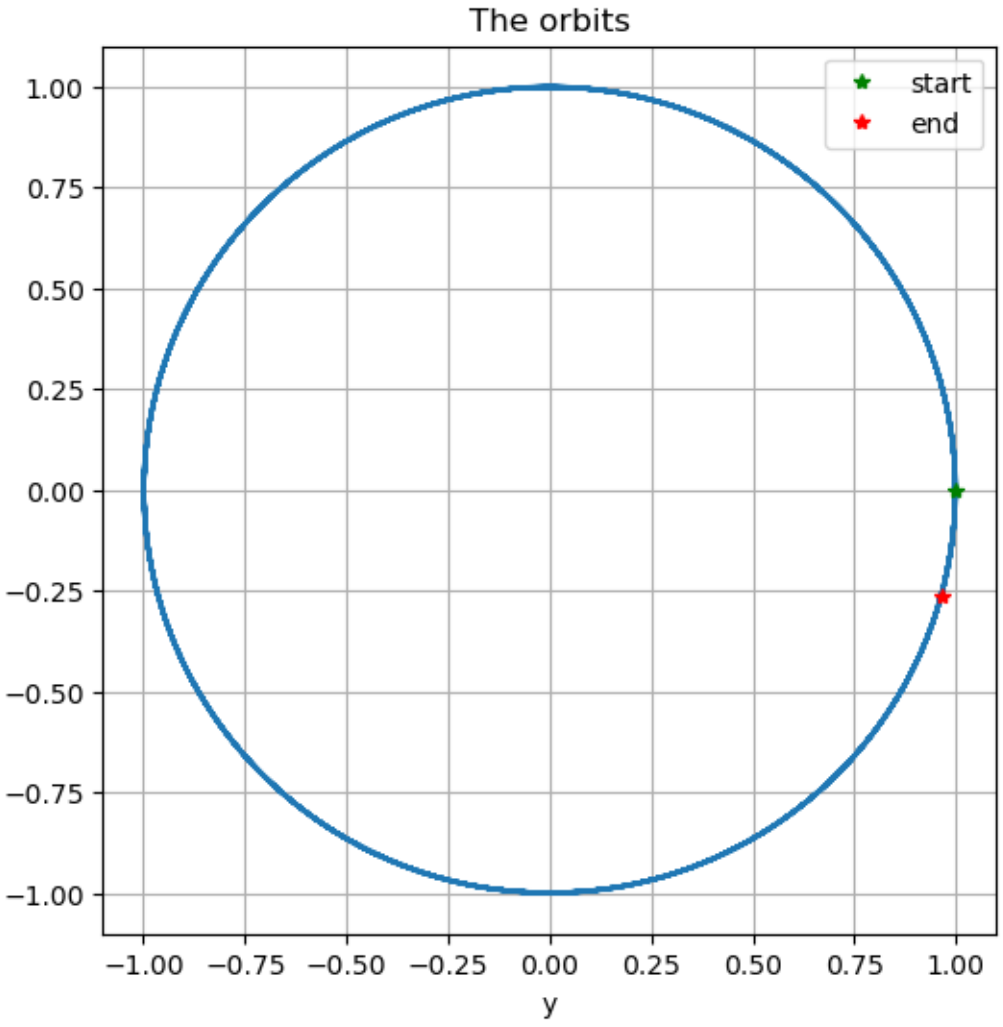
(continues on next page)

(continued from previous page)

```
Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("y for K/M=$(K/M), D= $D by leapfrog with $periods periods, $stepsperperiod
↪steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
↪"circular spirals"
title("The orbits")
plot(Y, DY)
xlabel("y")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```







The errors in leapfrog have an interesting feature: they are largely in timing, with its solutions rotating a little too fast, while the orbits stay on the correct circle: leapfrog respects the conserved "energy" $E(t) = \frac{1}{2}(y^2(t) + Dy^2(t))$. In the

section *Adams-Bashforth Multistep Methods*, this behavior wil be compared to a more "typical" methods.

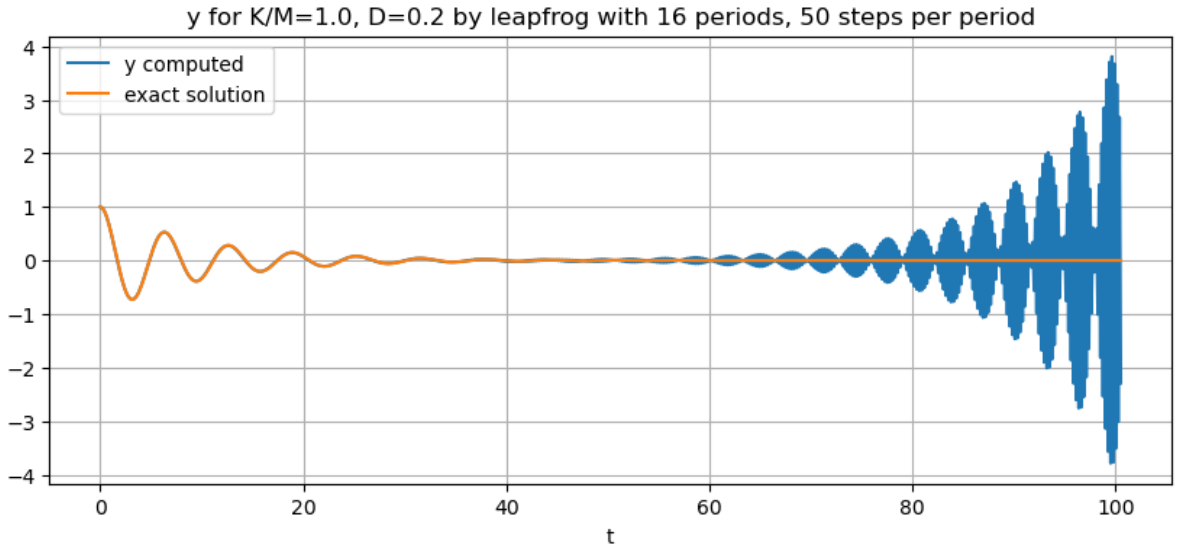
In some situations, this respecting of conserved quantities is very important, and the so-called **conservative methods** like leapfrog are then good choices.

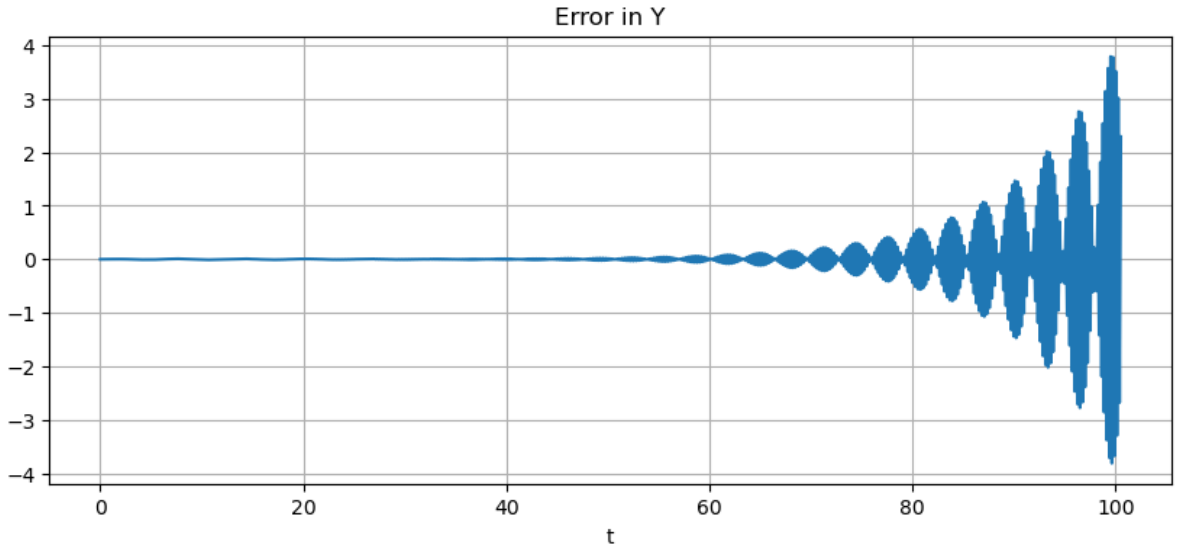
But with damping, things eventually go wrong!

This is an example in **instability**: reducing the step-size only postpones the problem, but does not avoid it.

In future sections it will be seen that the leapfrog method is stable (and a good choice) for "conservative" systems like the undamped mass-spring system, but unstable otherwise, such as for the damped case.

```
D = 0.2

periods = 16
b = 2pi * periods

# We need U_1, and get it with the Runge-Kutta method;
# this is overkill for accuracy, but since only one step is needed, the time cost is␣
 ↪negligible.
h = (b-a)/n
(t_1step, U_1step) = NM.rungekutta_system(f_mass_spring, a, a+h, U_0, 1)
U_1 = U_1step[end,:]
(t, U) = leapfrog(f_mass_spring, a, b, U_0, U_1, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("y for K/M=$(K/M), D=$D by leapfrog with $periods periods, $stepsperperiod␣
 ↪steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)
```





7.7 Adams-Bashforth Multistep Methods

References:

- Section 6.7 *Multistep Methods* in [Sauer, 2019].
- Section 5.6 *Multistep Methods* in [Burden *et al.*, 2016].

7.7.1 Introduction

Recall from *An Introduction to Multistep Methods*:

Definition 7.2 (Multistep Method)

A **multistep method** for numerical solution of an ODE IVP $du/dt = f(t,u)$, $u(t_0) = u_0$ is one of the form

$$U_i = \phi(U_{i-1}, \dots U_{i-s}, h), \quad s > 1$$

so that the new approximate value of $u(t)$ depends on approximate values at multiple previous times. (The shift of indexing to describe "present" in terms of "past" wil be convenient here.)

This is called an s-step method: the Runge-Kutta family of methods are all one-step.

We will be more specifically interted in what are called **linear multistep methods**, where the function at right is a linear combination of value of $u(t)$ and $f(t,u(t))$.

So for now we look at

$$U_i = a_0 U_{i-s} + \cdots + a_{s-1} U_{i-1} + h(b_0 f(t_{i-s}, U_{i-s}) + \cdots + b_{s-1} f(t_{i-1}, U_{i-1}))$$

The **Adams-Bashforth** methods are a case of this with the only a_i term being $a_{s-1} = 1$:

$$U_i = U_{i-1} + h(b_0 f(t_{i-s}, U_{i-s}) + \cdots + b_{s-1} f(t_{i-1}, U_{i-1}))$$

As wil be verified later, the s-step version of this is accurate to order s, so one can get arbitrarily high order of accuracy by using enough steps.

Aside. The case $s = 1$ is Euler's method, now written as

$$U_i = U_{i-1} + h f(t_{i-1}, U_{i-1})$$

The Adams-Bashforth methods are probably the most comomnly used **explicit**, **one-stage** multi-step methods; we will see more about the alternatives of **implicit** and **multi-stage** options in future sections. (Note that all Runge-Kutta methods (except Euler's) are multi-stage: the explicit trapezoid and midpoint methods are 2-stage; the classical Runge-Kutta method is 4-stage.)

The most basic **Adams-Bashforth** multi-step method is the 2-step method, which can be thought of this way:

1. Start with the two most recent values, $U_{i-1} \approx u(t_{i)-h}$ and $U_{i-2} \approx u(t_i - 2h)$
2. Use the derivative approximations $F_{i-1} := f(t_{i-1}, U_{i-1}) \approx u'(t_{i-1})$ and $F_{i-2} := f(t_{i-2}, U_{i-2}) \approx u'(t_{i-2})$ and linear extrapolation to "predict" the value of $u'(t_i - h/2)$; one gets: $u'(t_i - h/2) \approx \frac{3}{2}u'(t_i - h) - \frac{1}{2}u'(t_i - 2h) \approx \frac{3}{2}F_{i-1} - \frac{1}{2}F_{i-2}$
3. Use this in the centered difference approximation

$$\frac{u(t_i) - u(t_{i-1})}{h} \approx u'(t_i - h/2)$$

to get

$$\frac{U_i - U_{i-1}}{h} \approx \frac{3}{2}F_{i-1} - \frac{1}{2}F_{i-2}$$

That is,

$$U_i = U_{i-1} + \frac{h}{2}\left(3F_{i-1} - F_{i-2}\right), = U_{i-1} + \frac{h}{2}\left(3f(t_{i-1}, U_{i-1}) - f(t_{i-2}, U_{i-2})\right) \tag{7.11}$$

Equivalently, one can

1. Find the collocating polynomial $p(t)$ through (t_{i-1}, F_{i-1}) and (t_{i-2}, F_{i-2}) [so just a line in this case]
2. Use this on the interval (t_{i-1}, t_i) (*extrapolation*) as an approximation of $u'(t) = f(t, u(t))$ in that interval.
3. Use

$$u(t_i) = u(t_{i-1}) + \int_{t_{i-1}}^{t_i} u'(\tau) d\tau \approx u(t_{i-1}) + \int_{t_{i-1}}^{t_i} p(\tau) d\tau,$$

where the latter integral is easy to evaluate exactly.

One does not actually do this in each case; it is enough to verify that the integral gives $\left(\frac{3}{2}F_{i-1} - \frac{1}{2}F_{i-2}\right) h$.

See *Exercise 1*.

To code this algorithm, it is convenient to shift the indices, to get a formula for U_i. Also, note that what is $F_i = f(t_i, U_i)$ at one step is reused as $F_{i-1} = f(t_{i-1}, U_{i-1})$ at the next, so to avoid redundant evaluations of $f(t, u)$, these quantities should also be saved, at least till the next step:

$$U_i = U_{i-1} + \frac{h}{2}\left(3F_{i-1} - F_{i-2}\right)$$

```
using PyPlot
```

```
include("NumericalMethods.jl")
```

```
Main.NumericalMethods
```

```
import .NumericalMethods as NM
```

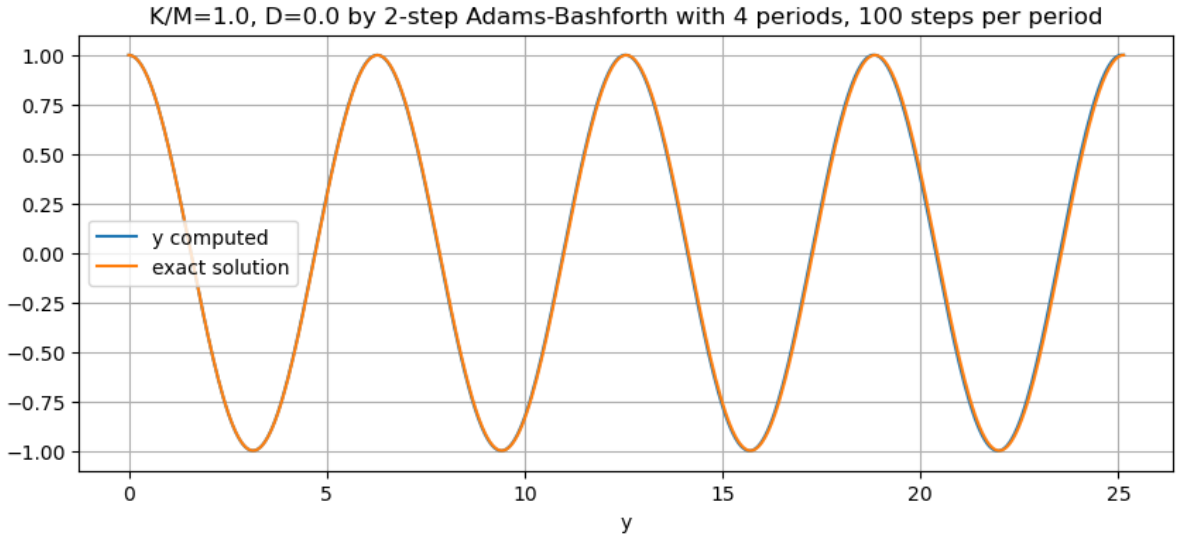
```
function adamsbashforth2(f, a, b, U_0, U_1, n)
    n_unknowns = length(U_0)
    t = range(a, b, n+1)
    u = zeros(n+1, n_unknowns)
    u[1,:] = U_0
    u[2,:] = U_1
    F_i_2 = f(a, U_0)  # F_0 to start when computing U_2
    h = (b-a)/n
    for i in 2:n  # i is the mathematical index, so "+1" for Julia array indices
        F_i_1 = f(t[i], u[i,:])
        u[i+1,:] = u[i,:] + (3*F_i_1 - F_i_2) * (h/2)
        F_i_2 = F_i_1
    end
    return (t, u)
end;
```

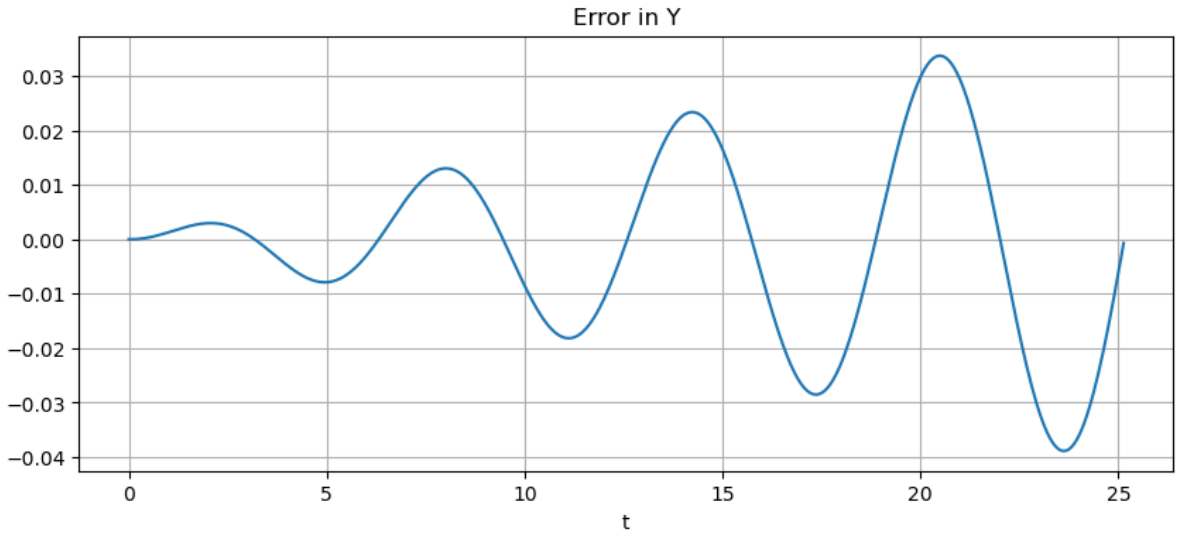
Demonstrations with the mass-spring system

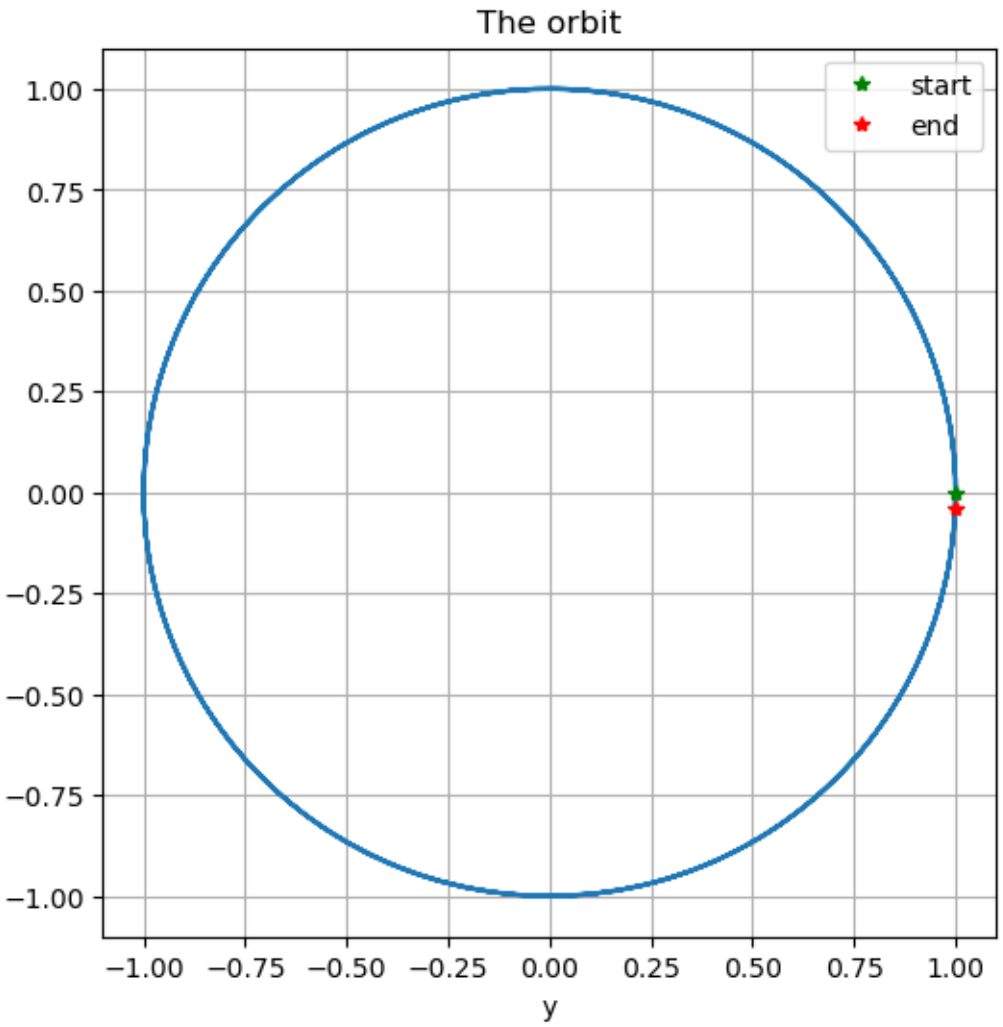
```
f_mass_spring(t, u) = [ u[2]; -(K/M)*u[1] - (D/M)*u[2] ];
```

```
using .NumericalMethods: y_mass_spring
```

```
M = 1.0
K = 1.0
D = 0.0
y_0 = 1.0
v_0 = 0.0
U_0 = [y_0, v_0]
a = 0.0
periods = 4
b = 2pi * periods

# Using the same time step size as with the leapfrog method in the previous section.
stepsperperiod = 100
n = Int(stepsperperiod * periods)

# We need U_1, and get it with the Runge-Kutta method;
# this is overkill for accuracy, but since only one step is needed, the time cost is␣
 ↪negligible.
h = (b-a)/n
(t_1step, U_1step) = NM.rungekutta_system(f_mass_spring, a, a+h, U_0, 1)
U_1 = U_1step[end,:]
(t, U) = adamsbashforth2(f_mass_spring, a, b, U_0, U_1, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 2-step Adams-Bashforth with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
xlabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
 ↪"circular spirals"
title("The orbit")
plot(Y, DY)
xlabel("y")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```


K/M=1.0, D=0.0 by 2-step Adams-Bashforth with 4 periods, 100 steps per period
y computed
exact solution
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
25
y


Error in Y
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.04
0
5
10
15
20
25
t

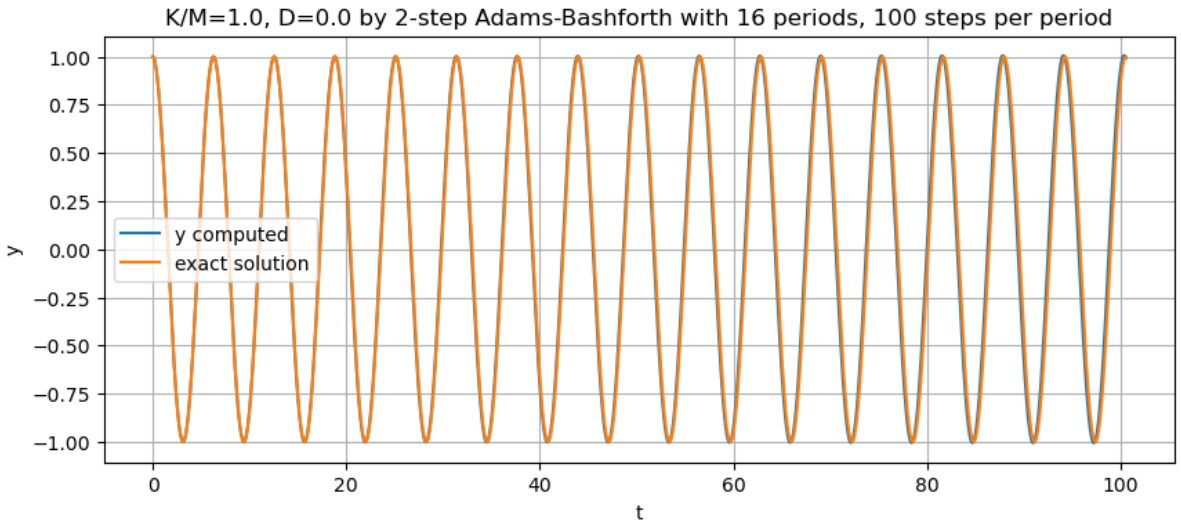


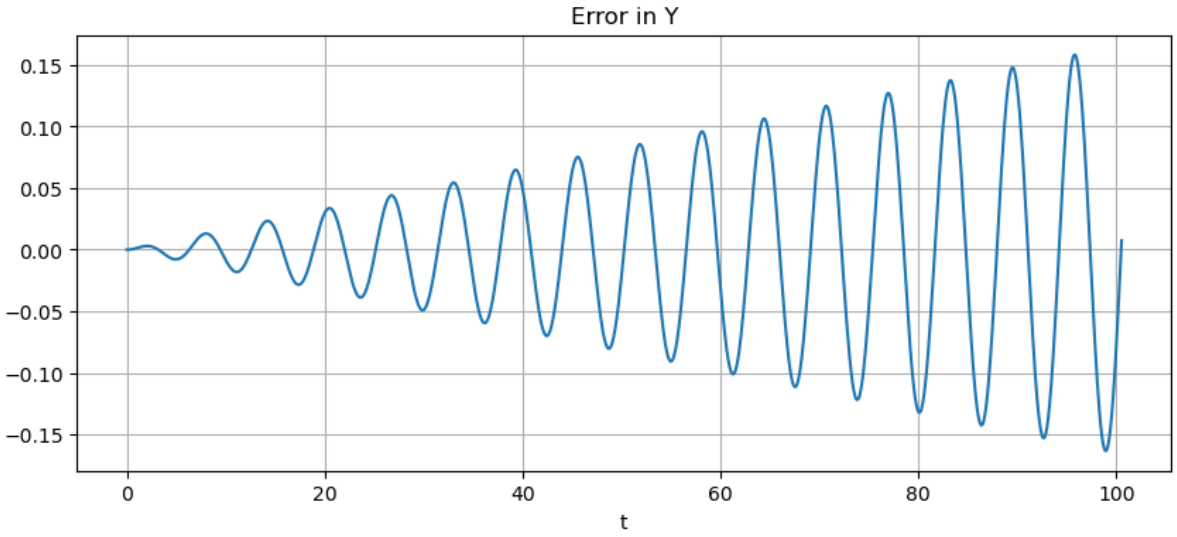

```
D = 0.0

periods = 16
b = 2pi * periods

# Using the same time step size as with the leapfrog method in the previous section.
stepsperperiod = 100
n = Int(stepsperperiod * periods)

# We need U_1, and get it with the Runge-Kutta method;
# this is overkill for accuracy, but since only one step is needed, the time cost is␣
 ↪negligible.
h = (b-a)/n
(t_1step, U_1step) = NM.rungekutta_system(f_mass_spring, a, a+h, U_0, 1)
U_1 = U_1step[end,:]
(t, U) = adamsbashforth2(f_mass_spring, a, b, U_0, U_1, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution
```

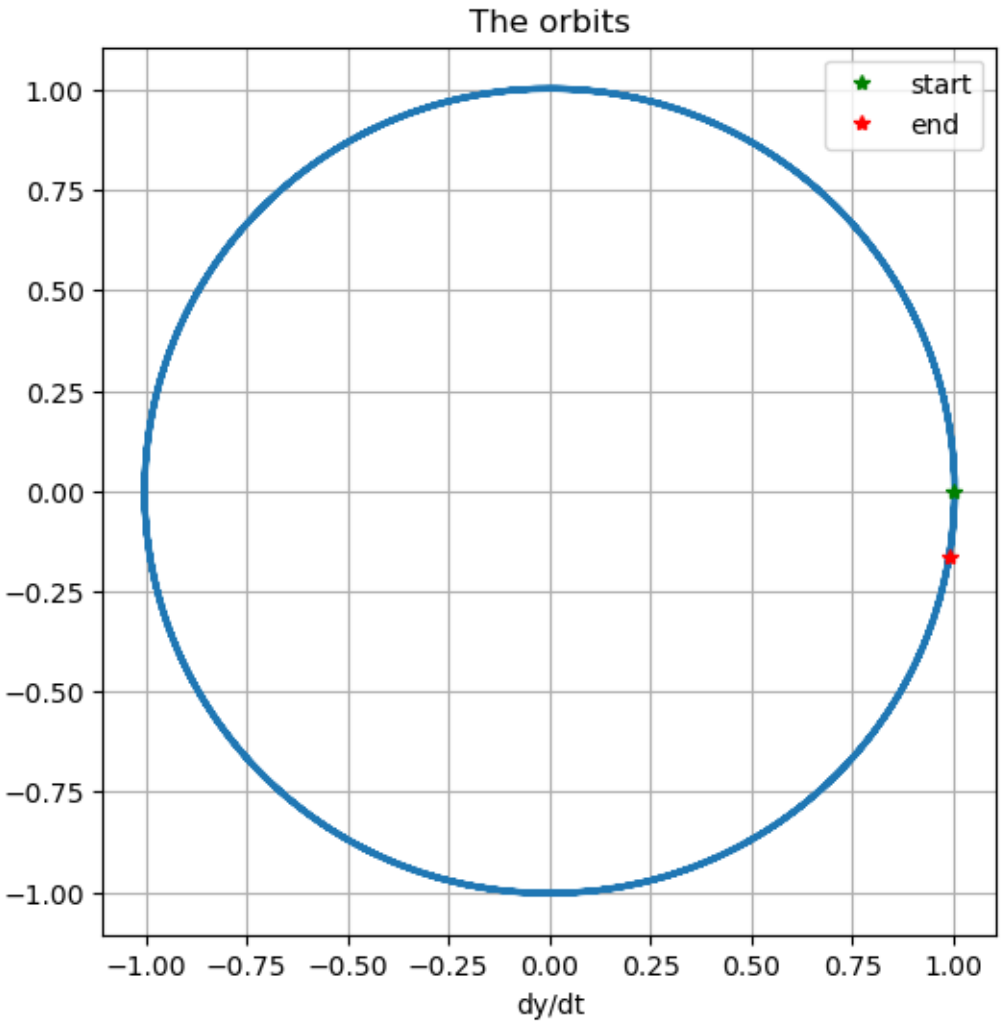
(continues on next page)

(continued from previous page)

```
figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 2-step Adams-Bashforth with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
ylabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
 ↪"circular spirals"
title("The orbits")
plot(Y, DY)
xlabel("y")
xlabel("dy/dt")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```







In comparison to the (also second order accurate) leap-frog method, this is distinctly worse; the errors are more than twice as large, and the solution fails to stay on the circle; unlike leapfrog, the energy $E(t) = \frac{1}{2}(y^2(t) + Dy^2(t))$ is not

conserved.

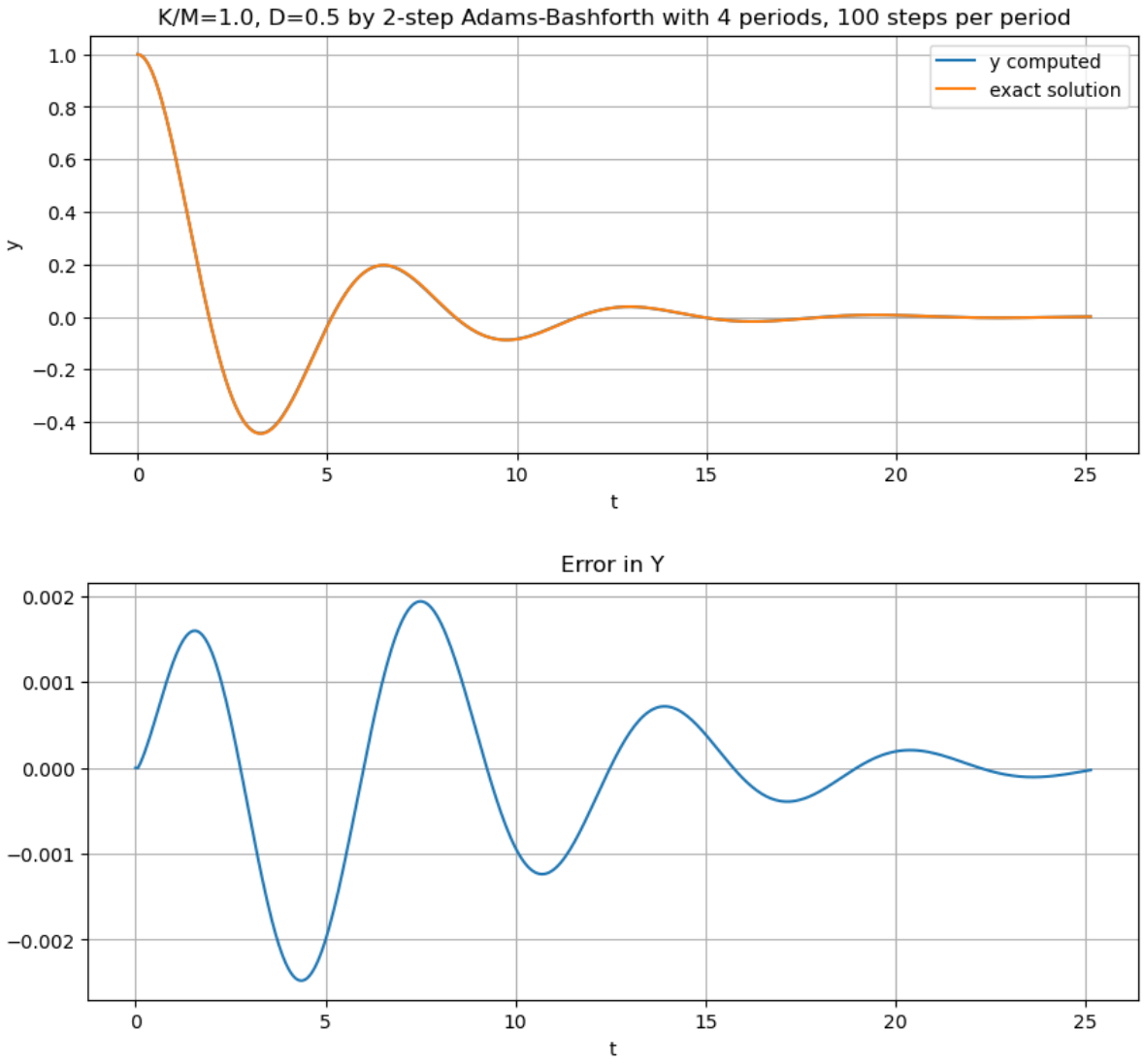
On the other hand …

This time with damping, nothings goes wrong!

This is an example in **stability**; in future sections it will be seen that the the Adams-Bashforth methods are all stable for these equations for small enough step size h, and so converge to the correct solution as $h \to 0$.

Looking back, this suggests (correctly) that while the leapfrog method is well-suited to *conservative* equations, Adams-Bashforth methods are much preferable for more general equations.

```
D = 0.5

periods = 4
b = 2pi * periods

# Using the same time step size as with the leapfrog method in the previous section.
stepsperperiod = 100
n = Int(stepsperperiod * periods)

# We need U_1, and get it with the Runge-Kutta method;
# this is overkill for accuracy, but since only one step is needed, the time cost is␣
 ↪negligible.
h = (b-a)/n
(t_1step, U_1step) = NM.rungekutta_system(f_mass_spring, a, a+h, U_0, 1)
U_1 = U_1step[end,:]
(t, U) = adamsbashforth2(f_mass_spring, a, b, U_0, U_1, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 2-step Adams-Bashforth with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
ylabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)
```



7.7.2 Higher order Adams-Bashforth methods

The strategy described above of polynomial approximation, extrapolation, and integration can be generalized to get the s step Adams-Bashforth method, of order s; to get the approximation U_i of $u(t_i)$ from data at the s most recent previous times t_{i-1} to t_{i-s}:

1. Find the collocating polynomial $p(t)$ of degree $s-1$ through $(t_{i-1}, F_{i-1}) \ldots (t_{i-s}, F_{i-s})$
2. Use this on the interval (t_{i-1}, t_i) (*extrapolation*) as an approximation of $u'(t) = f(t, u(t))$ in that interval.
3. Use $u(t_i) = u(t_{i-1}) + \int_{t_{i-1}}^{t_i} u'(\tau) d\tau \approx u(t_{i-1}) + \int_{t_{i-1}}^{t_i} p(\tau) d\tau$, where the latter integral can be evaluated exactly.

Again, one does not actually evaluate this integral; it is enough to verify that the resulting form will be

$$U_i = U_{i-1} + h(b_0 F_{i-s} + b_1 F_{i-s+1} + \cdots b_{s-1} F_{i-1})$$

with the coefficients being the same for any $f(t, u)$ and any h.

In fact, the polynomial fitting and integration can be skipped: thecoefficients can be derived by the method of undetermined coefficients as seen in *Approximating Derivatives by the Method of Undetermined Coefficients* and this also established that the local truncation error is $O(h^s)$:

- insert Taylor polynomial approximations of $u(t_{i-k}) = u(t_i) - kh)$ and $f(t_{i-k}, u(t_{i-k})) = u'(t_{i-k}) = u'(t_i - kh)$ into $$U_i = U_{i-1} + h(b_0 f(t_{i-s}, U_{i-s}) + \cdots + b_{s-1} f(t_{i-1}, U_{i-1}))$$
- solve for the s coefficients $b_0 \ldots b_{s-1}$ that give the highest power for the residual error: the terms in the first s powers of h (from $h^0 = 1$ to h^{s-1}) can be cancelled, leaving an error $O(h^s)$.

The first few Adams-Bashforth formulas are:

- $s = 1$: $b_0 = 1$, $$U_i = U_{i-1} + hF_{i-1} = U_{i-1} + hf(t_{i-1}, U_{i-1})$$ (Euler's method)$$
- $s = 2$: $b_0 = -1/2, b_1 = 3/2$, $$U_i = U_{i-1} + \frac{h}{2}(3F_{i-1} - F_{i-2})$$ (as above)$$
- $s = 3$: $b_0 = 5/12, b_1 = -16/12, b_2 = 23/12$, $$U_i = U_{i-1} + \frac{h}{12}(23F_{i-1} - 16F_{i-2} + 5F_{i-3})$$
- $s = 4$: $b_0 = -9/24, b_1 = 37/24, b_2 = -59/24, b_3 = 55/24$, $$U_i = U_{i-1} + \frac{h}{24}(55F_{i-1} - 59F_{i-2} + 37F_{i-3} - 9F_{i-4})$$

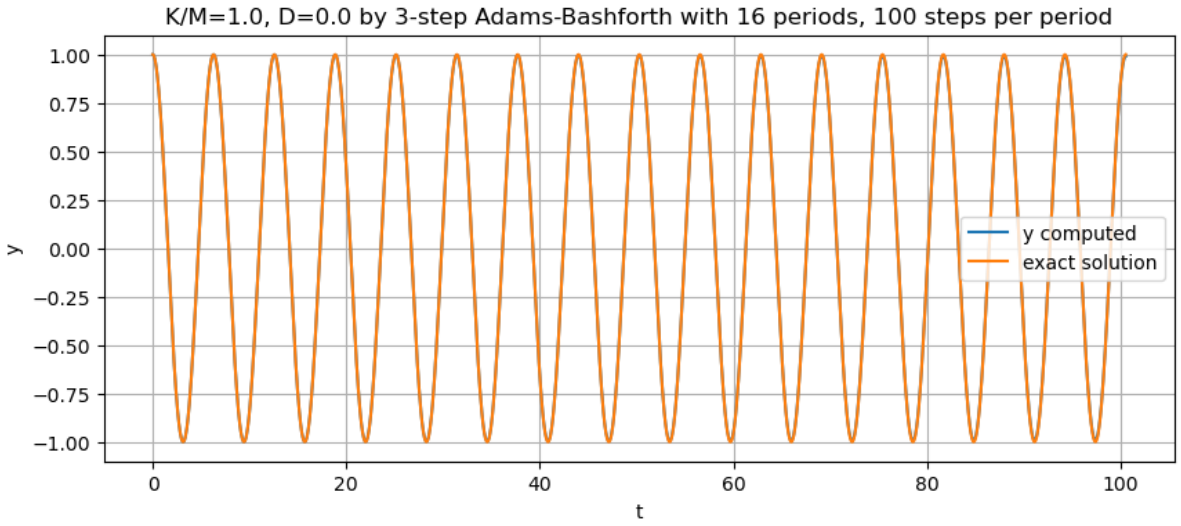
```
function adamsbashforth3(f, a, b, U_0, U_1, U_2, n)
    n_unknowns = length(U_0)
    h = (b-a)/n
    t = range(a, b, n+1)
    u = zeros(n+1, n_unknowns)
    u[1,:] = U_0
    u[2,:] = U_1
    u[3,:] = U_2
    F_i_3 = f(a, U_0)  # F_0 to start when computing U_3
    F_i_2 = f(a+h, U_1)  # F_1 to start when computing U_3
    for i in 3:n  # i is the mathematical index, so "+1" for Julia array indices
        F_i_1 = f(t[i], u[i,:])
        u[i+1,:] = u[i,:] + (23F_i_1 - 16F_i_2 + 5F_i_3) * (h/12)
        (F_i_2, F_i_3) = (F_i_1, F_i_2)
    end
    return (t, u)
end;
```

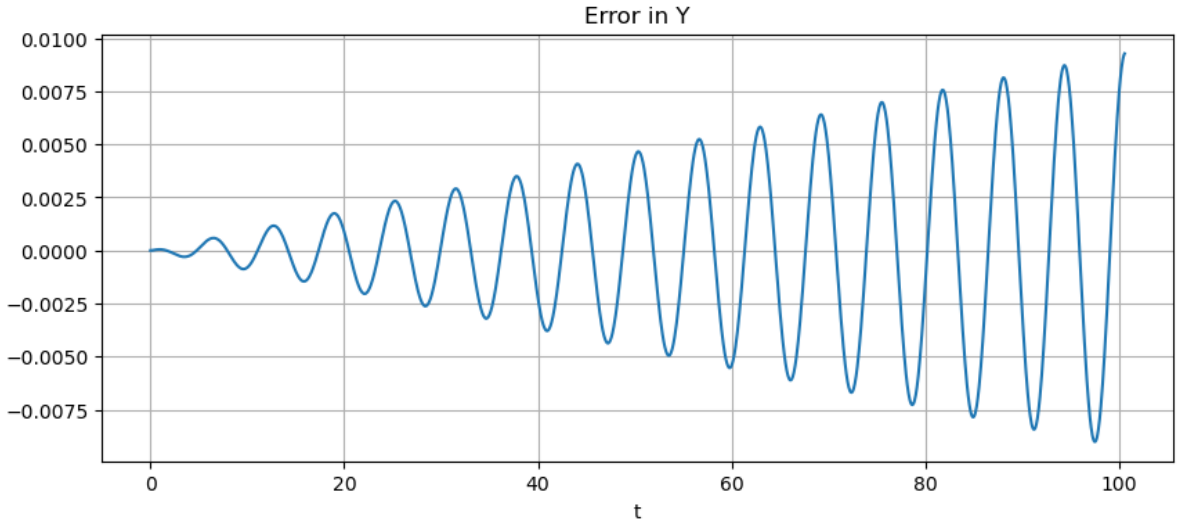
```
D = 0.0

periods = 16
b = 2pi * periods

# Using the same time step size as for leapfrog method in the previous section.
stepsperperiod = 100
n = Int(stepsperperiod * periods)

# We need U_1 and U_2, and get them with the Runge-Kutta method;
# this is overkill for accuracy, but since only two steps are needed, the time cost␣
 ↪is negligible.
h = (b-a)/n
(t_2step, U_2step) = NM.rungekutta_system(f_mass_spring, a, a+2h, U_0, 2)
U_1 = U_2step[2,:]
U_2 = U_2step[3,:]
(t, U) = adamsbashforth3(f_mass_spring, a, b, U_0, U_1,U_2, n)

Y = U[:,1]
DY = U[:,2]
```

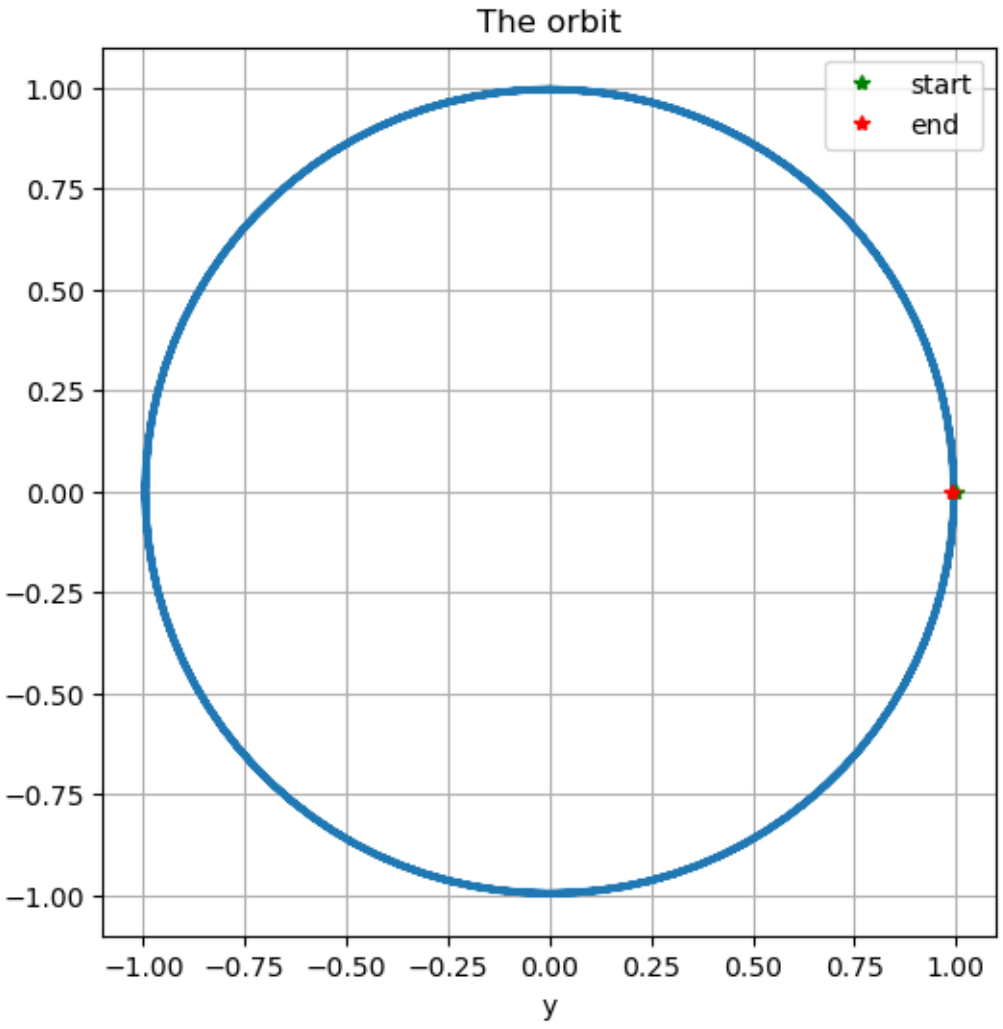
(continues on next page)

(continued from previous page)

```
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 3-step Adams-Bashforth with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
ylabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
 ↪"circular spirals"
title("The orbit")
plot(Y, DY)
xlabel("y")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```

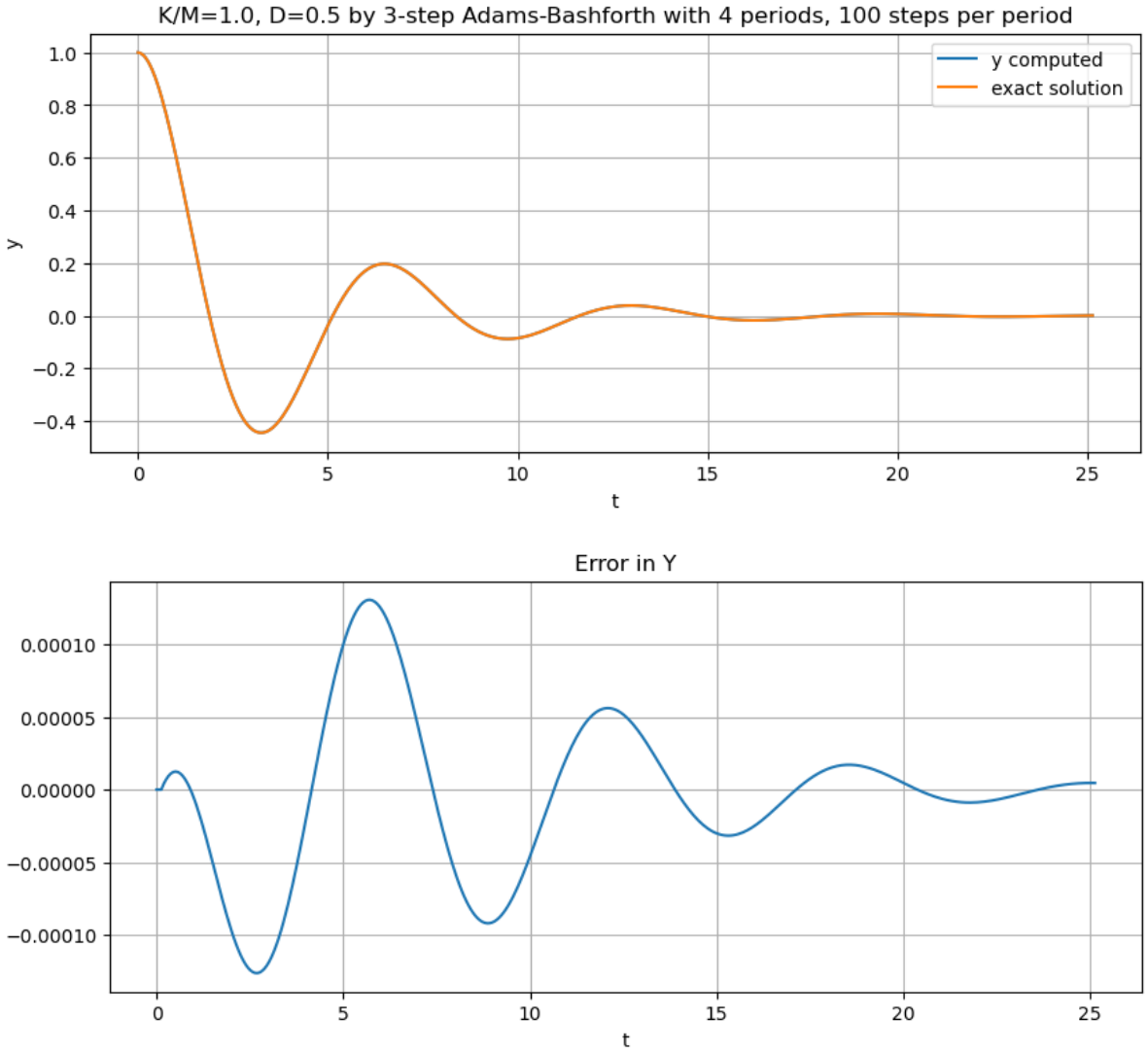






Comparing to the leap-frog method, this higher order method at last has smaller errors (and they can be got even smaller by increasing the number of steps) but the leapfrog method is still better at keeping the solutions on the circle.

```
D = 0.5

periods = 4
b = 2pi * periods

# Note: In the notes on systems, the second order Runge-Kutta methods were tested␣
 ↪with 50 steps per period
#stepsperperiod = 50  # As for the second order accurate explicit trapezoid and␣
 ↪midpoint methods
stepsperperiod = 100  # Equal cost per unit time as for the explicit trapezoid and␣
 ↪midpoint and Runge-Kutta methods
n = Int(stepsperperiod * periods)

# We need U_1 and U_2, and get them with the Runge-Kutta method;
# this is overkill for accuracy, but since only two steps are needed, the time cost␣
 ↪is negligible.
h = (b-a)/n
(t_2step, U_2step) = NM.rungekutta_system(f_mass_spring, a, a+2h, U_0, 2)
U_1 = U_2step[2,:]
U_2 = U_2step[3,:]
(t, U) = adamsbashforth3(f_mass_spring, a, b, U_0, U_1,U_2, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 3-step Adams-Bashforth with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
ylabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)
```



The fourth-order, four step method does at last appear to surpass leap-frog on the conservative case:

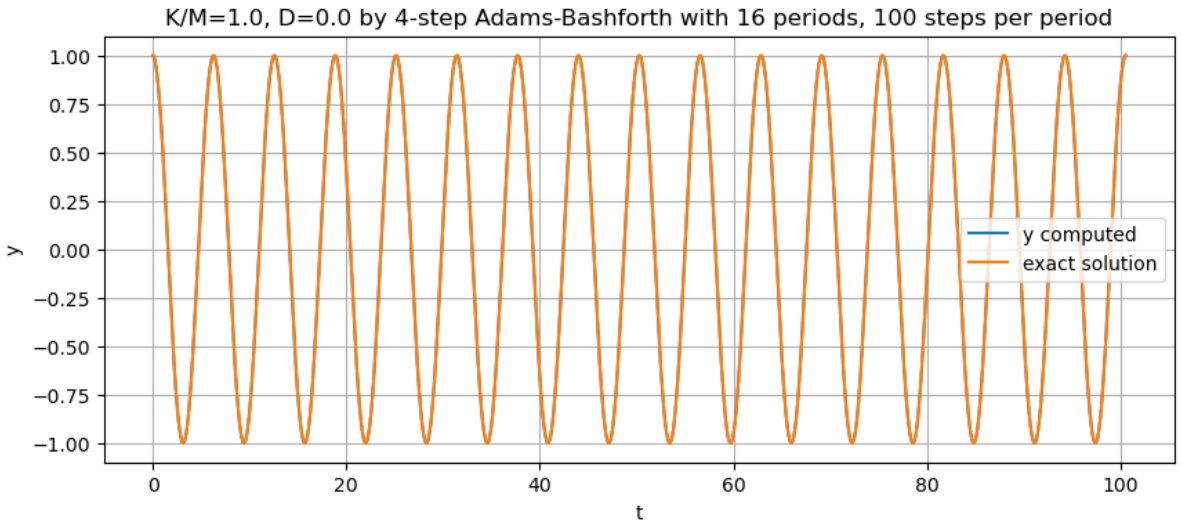
```
function adamsbashforth4(f, a, b, U_0, U_1, U_2, U_3, n)
    n_unknowns = length(U_0)
    h = (b-a)/n
    t = range(a, b, n+1)
    u = zeros(n+1, n_unknowns)
    u[1,:] = U_0
    u[2,:] = U_1
    u[3,:] = U_2
    u[4,:] = U_3
    F_i_4 = f(a, U_0)  # F_0 to start when computing U_4
    F_i_3 = f(a+h, U_1)  # F_1 to start when computing U_4
    F_i_2 = f(a+2h, U_2)  # F_1 to start when computing U_4
    h = (b-a)/n
    for i in 4:n  # i is the mathematical index, so "+1" for Julia array indices
        F_i_1 = f(t[i], u[i,:])
        u[i+1,:] = u[i,:] + (55F_i_1 - 59F_i_2 + 37F_i_3 - 9F_i_4) * (h/24)
        (F_i_2, F_i_3, F_i_4) = (F_i_1, F_i_2, F_i_3)
```

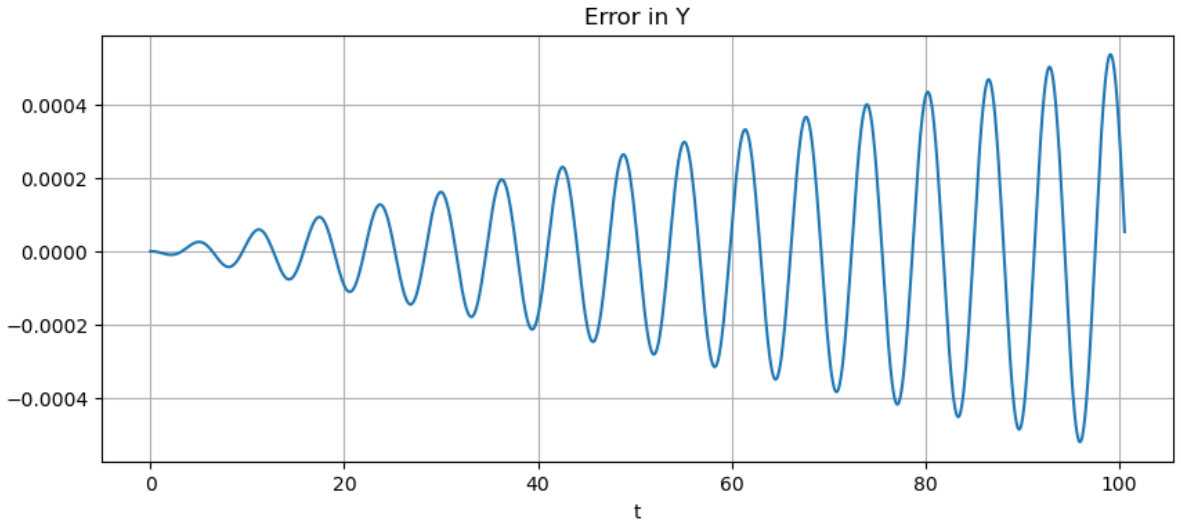
(continues on next page)

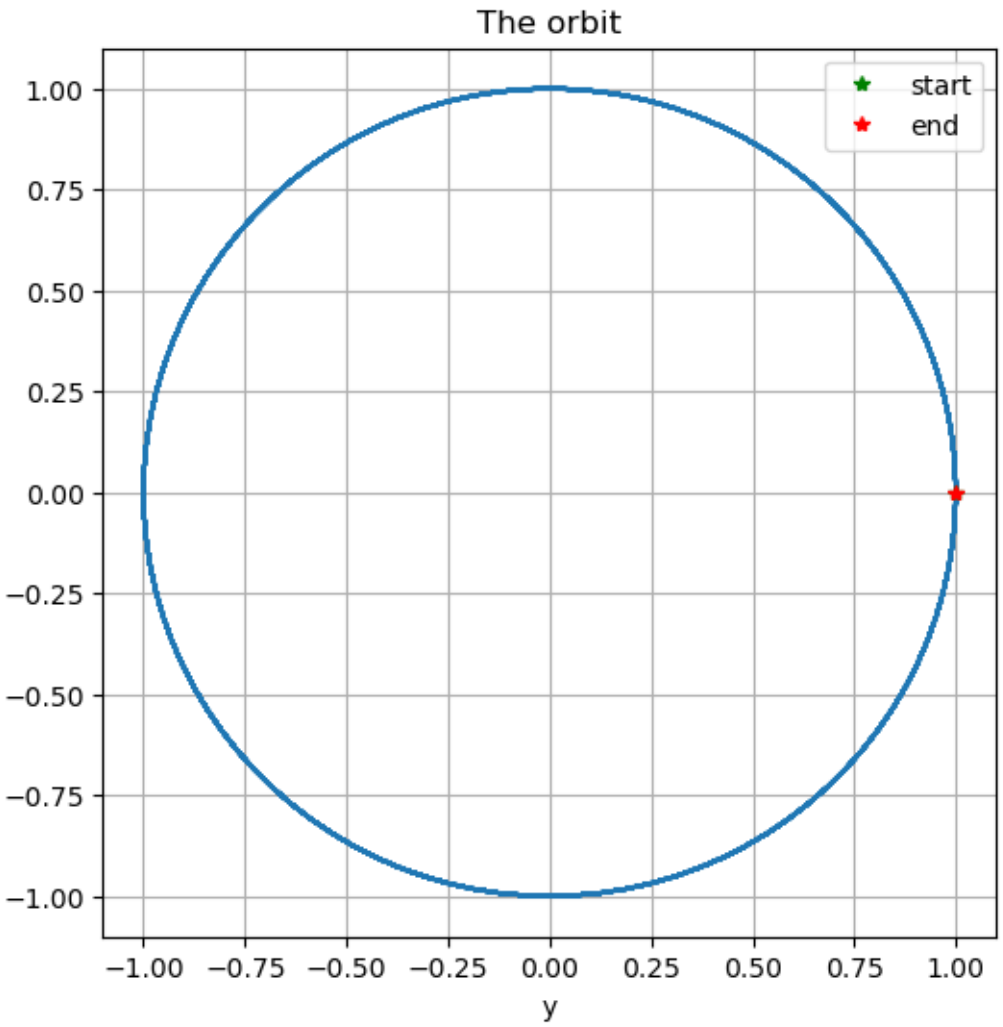
(continued from previous page)

```
    end
    return (t, u)
end;
```

```
D = 0.0

periods = 16
b = 2pi * periods

# Using the same time step size as for leapfrog method in the previous section.
stepsperperiod = 100
n = Int(stepsperperiod * periods)

# We need U_1, U_2 and U_3, and get them with the Runge-Kutta method;
# this is overkill for accuracy, but since only three steps are needed, the time cost␣
 ↪is negligible.
h = (b-a)/n
(t_3step, U_3step) = NM.rungekutta_system(f_mass_spring, a, a+3h, U_0, 3)
U_1 = U_3step[2,:]
U_2 = U_3step[3,:]
U_3 = U_3step[4,:]
(t, U) = adamsbashforth4(f_mass_spring, a, b, U_0, U_1, U_2, U_3, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 4-step Adams-Bashforth with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
ylabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
 ↪"circular spirals"
title("The orbit")
plot(Y, DY)
xlabel("y")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```


K/M=1.0, D=0.0 by 4-step Adams-Bashforth with 16 periods, 100 steps per period
y computed
exact solution
y
t
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
20
40
60
80
100


Error in Y
t
0.0004
0.0002
0.0000
−0.0002
−0.0004
0
20
40
60
80
100

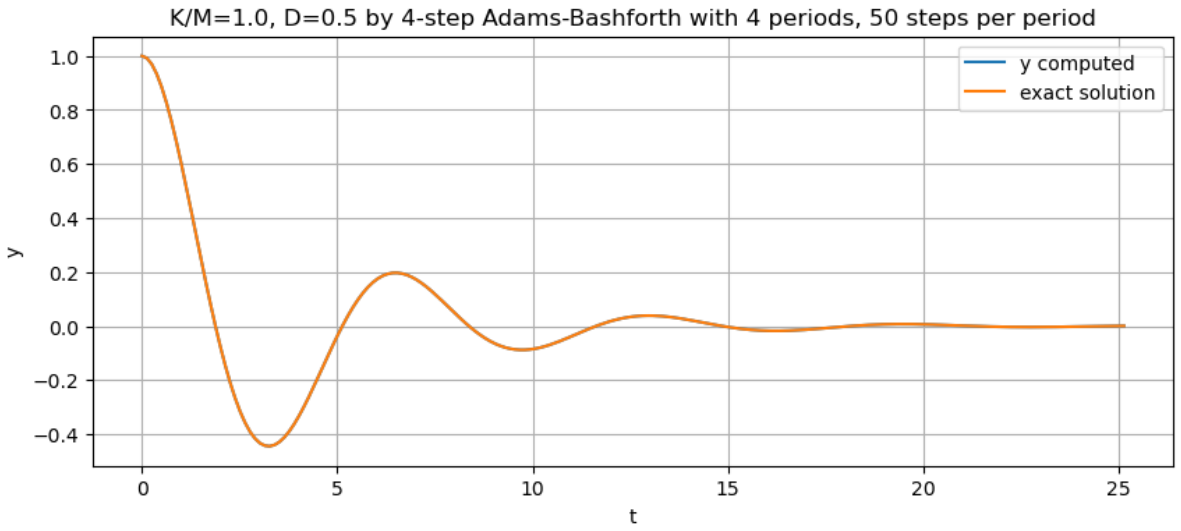


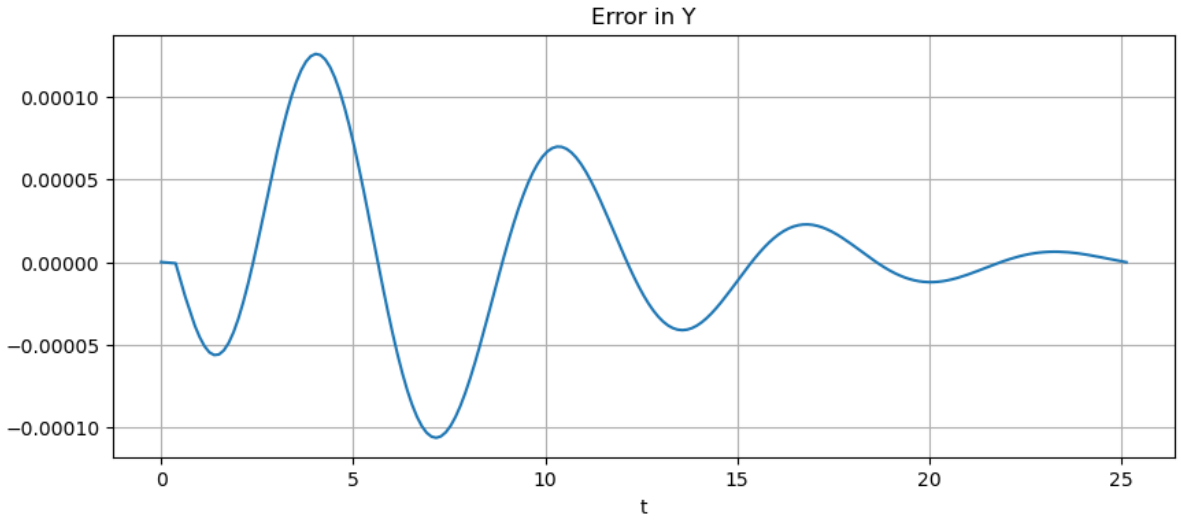

```
D = 0.5

periods = 4
b = 2pi * periods

# Using the same time step size as for leapfrog method in the previous section.
stepsperperiod = 50
n = Int(stepsperperiod * periods)

# We need U_1, U_2 and U_3, and get them with the Runge-Kutta method.
h = (b-a)/n
(t_3step, U_3step) = NM.rungekutta_system(f_mass_spring, a, a+3h, U_0, 3)
U_1 = U_3step[2,:]
U_2 = U_3step[3,:]
U_3 = U_3step[4,:]
(t, U) = adamsbashforth4(f_mass_spring, a, b, U_0, U_1, U_2, U_3, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution
```

(continues on next page)

(continued from previous page)

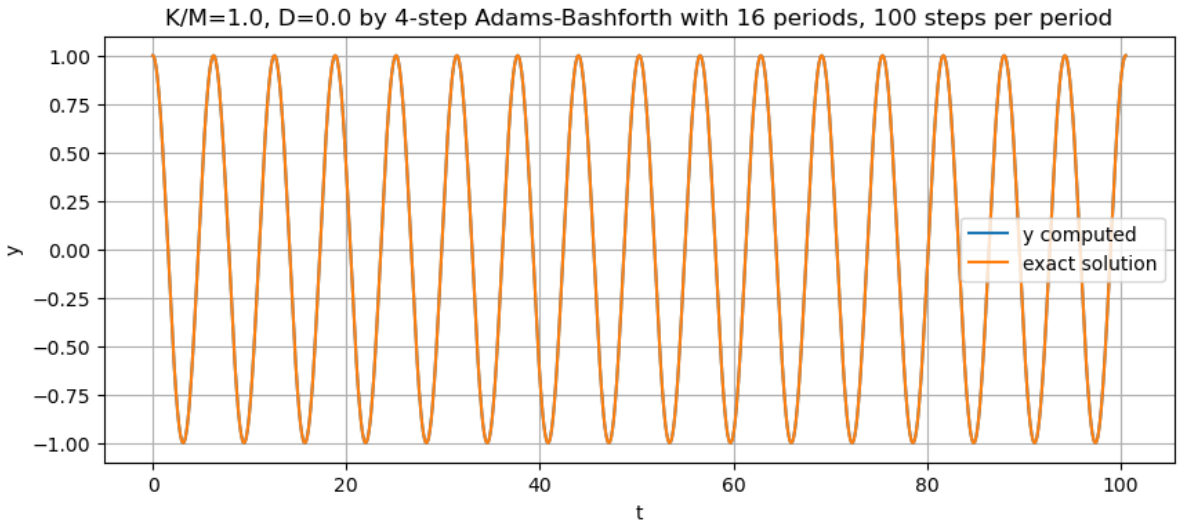
```
figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 4-step Adams-Bashforth with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
ylabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)
```

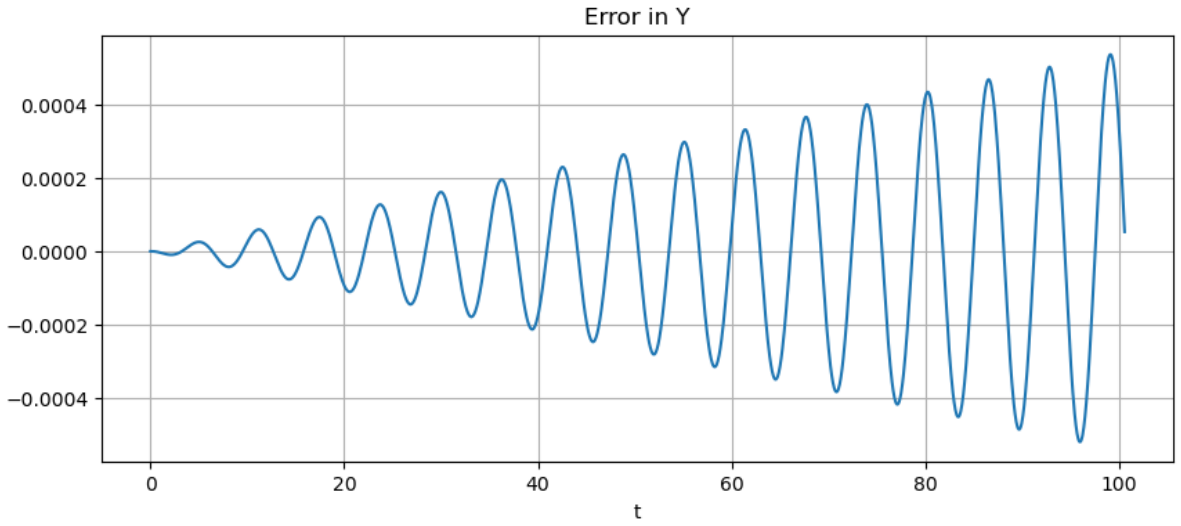


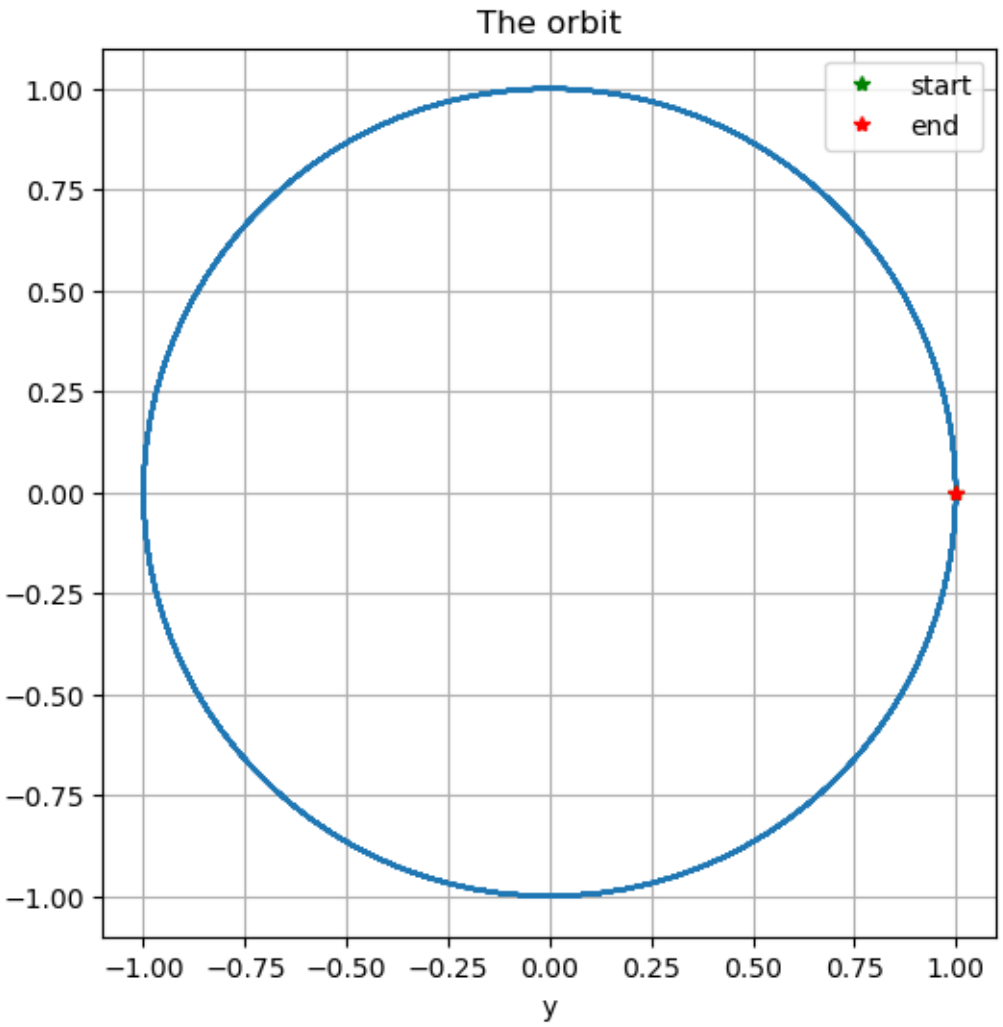


Finally, an "equal cost" comparison to the forht order Runge-Kutta method results in section *Systems of ODEs and Higher Order ODEs* with four times as many steps per unit time: the fourth order Adams-Bashforth method come out ahead in these two test cases.

```
D = 0.0

periods = 16
b = 2pi * periods

stepsperperiod = 100
n = Int(stepsperperiod * periods)

# We need U_1, U_2 and U_3, and get them with the Runge-Kutta method;
# this is overkill for accuracy, but since only three steps are needed, the time cost␣
↪is negligible.
h = (b-a)/n
(t_3step, U_3step) = NM.rungekutta_system(f_mass_spring, a, a+3h, U_0, 3)
U_1 = U_3step[2,:]
U_2 = U_3step[3,:]
U_3 = U_3step[4,:]
(t, U) = adamsbashforth4(f_mass_spring, a, b, U_0, U_1, U_2, U_3, n)

Y = U[:,1]
DY = U[:,2]
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 4-step Adams-Bashforth with $periods periods,
↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
ylabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)

figure(figsize=[6,6])  # Make axes equal length; orbits should be circular or
↪"circular spirals"
title("The orbit")
plot(Y, DY)
xlabel("y")
plot(Y[1], DY[1], "g*", label="start")
plot(Y[end], DY[end], "r*", label="end")
legend()
grid(true)
```


K/M=1.0, D=0.0 by 4-step Adams-Bashforth with 16 periods, 100 steps per period
y computed
exact solution
y
t
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
20
40
60
80
100


Error in Y
t
0.0004
0.0002
0.0000
−0.0002
−0.0004
0
20
40
60
80
100

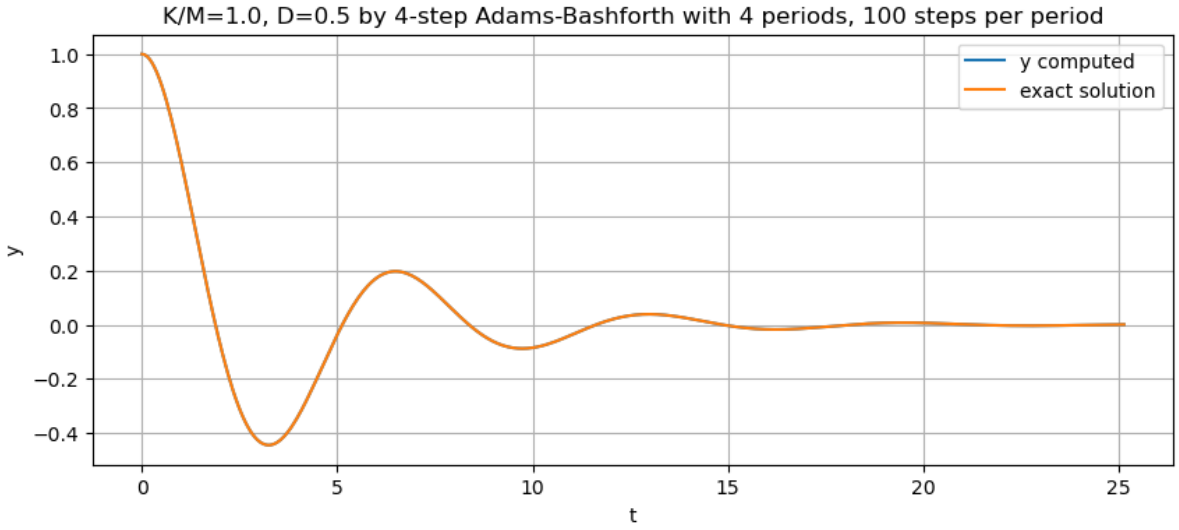


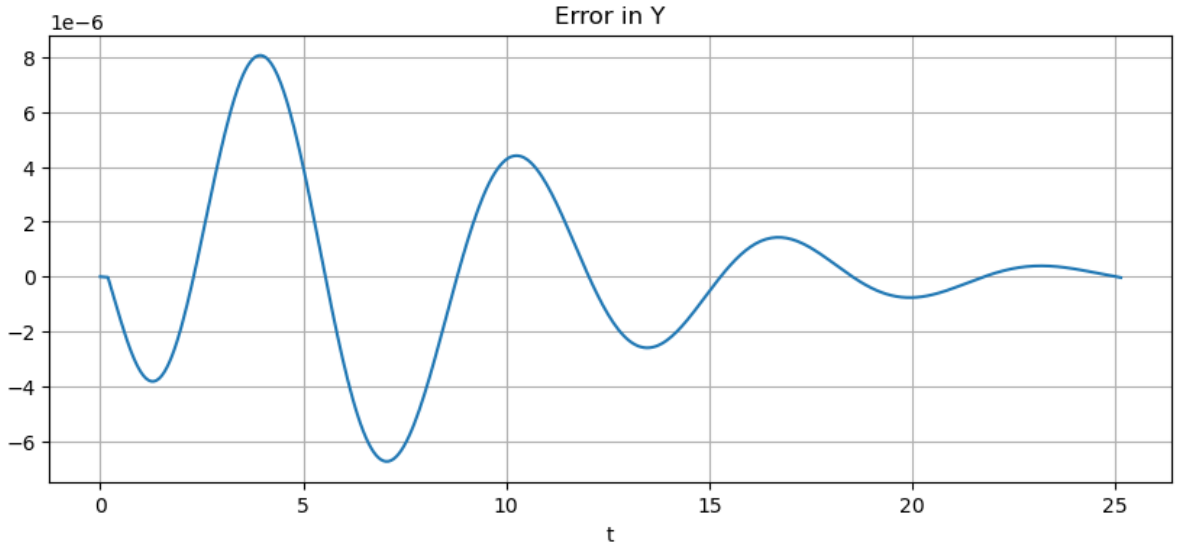

```
D = 0.5

periods = 4
b = 2pi * periods

# Using the same time step size as for leapfrog method in the previous section.
stepsperperiod = 100
n = Int(stepsperperiod * periods)

# We need U_1, U_2 and U_3, and get them with the Runge-Kutta method;
# this is overkill for accuracy, but since only three steps are needed, the time cost␣
 ↪is negligible.
h = (b-a)/n
(t_3step, U_3step) = NM.rungekutta_system(f_mass_spring, a, a+3h, U_0, 3)
U_1 = U_3step[2,:]
U_2 = U_3step[3,:]
U_3 = U_3step[4,:]
(t, U) = adamsbashforth4(f_mass_spring, a, b, U_0, U_1, U_2, U_3, n)

Y = U[:,1]
DY = U[:,2]
```

(continues on next page)

(continued from previous page)

```
y = y_mass_spring.(t; t_0=a, u_0=U_0, K=K, M=M, D=D)  # Exact solution

figure(figsize=[10,4])
title("K/M=$(K/M), D=$D by 4-step Adams-Bashforth with $periods periods,
 ↪$stepsperperiod steps per period")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
xlabel("t")
ylabel("y")
grid(true)

figure(figsize=[10,4])
title("Error in Y")
plot(t, y-Y)
xlabel("t")
grid(true)
```





7.7.3 Exercises

Exercise 1

Verify the derivation of Equation (7.11) for the second order Adams-Bashforth method, via polynomial collocation and integration.

Exercise 2

Verify the above result for $s = 3$ by the method of undetermined coefficients.

7.8 Implicit Methods: Adams-Moulton

References:

- Section 6.7 *Multistep Methods* in [Sauer, 2019].
- Section 5.6 *Multistep Methods* in [Burden *et al.*, 2016].

7.8.1 Introduction

So far, most methods we have seen give the new approximation value with an explicit formula for it in terms of previous and so already known) values; the general **explicit s-step method** seen in *Adams-Bashforth Multistep Methods* was

$$U_i = \phi(U_{i-1}, \ldots U_{i-s+}, h), \quad s > 1$$

However, we briefly saw two **implict methods** back in *Runge-Kutta Methods*, in the process of deriving the explicit trapezoid and explicit midpoint methods: the **Implicit Trapezoid Method** (or just the **Trapezoid Method**, as this is the real thing, before the further approximations were used to get an explicit formula)

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

and the **Implicit Midpoint Method**

$$U_{i+1} = U_i + hf\left(t + h/2, \frac{U_i + U_{i+1}}{2}\right)$$

These are clearly not as simple to work with as explicit methods, but the equation solving can often be done. In particular for linear differential equations, these give linear equations for the unknown U_{i+1}, so even for systems, they can be solved by the method seen earlier in these notes.

Another strategy is noting that these are fixed point equations, so that fixed point iteration can be used. The factor h at right helps; it can be shown that for small enough h (how small depends on the function f), these are contraction mappings and so fixed point iteration works.

This idea can be combined with linear multistep methods, and one important case is modifying the Adams-Bashforth method by allowing $F_i = f(t_i, U_i)$ to appear at right: this gives the Adams-Moulton form

$$U_i = U_{i-1} + h(b_0 f(t_{i-s}, U_{i-s}) + \cdots + b_s f(t_i, U_i))$$

where the only change from Adams-Bashforth methods is that $f(t_i, U_i)$ term.

The coefficients can be derived much as for Adams-Bashforth methods, by the method of undetermined coefficients; one valuable difference is that there at now $s+1$ undetermined coefficients, so all error terms up to $O(h^s)$ can be cancelled and the error made $O(h^{s+1})$: one degree higher.

The $s = 1$ case is familiar:

$$U_i = U_{i-1} + h(b_0 f(t_{i-1}, U_{i-1}) + b_1 f(t_i, U_i))$$

and as symmetry suggests, the solution is $b_0 = b_1 = 1/2$, giving

$$U_i = U_{i-1} + h\frac{f(t_{i-1}, U_{i-1}) + f(t_i, U_i)}{2}$$

which is the (implicit) trapzoid rule in the new shifted indexing.

This is much used for numerical solution of partial differential equations of evoluton type (after first approximating by a large system of ordinary differnetial equations). In that context it is often known as the **Crank-Nicholson method**.

We can actually start at $s = 0$; the first few Adams-Moulton methods are:

$s = 0 : b_0 = 1$

$$U_i - hf(t_i, U_i)) = U_{i-1}$$ The backward Euler method

$s = 1 : b_0 = b_1 = 1/2$

$$U_i - \frac{h}{2} f(t_i, U_i) = U_{i-1} + \frac{h}{2}(F_{i-1})$$ The (implicit) trapezoid method

$s = 2 : b_0 = -1/12, b_1 = 8/12, b_2 = 5/12$

$$U_i - \frac{5h}{12} f(t_i, U_i) = U_{i-1} + \frac{h}{12}(-F_{i-2} + 8F_{i-1})$$

$s = 3 : b_0 = 1/24, b_1 = -5/24, b_2 = 19/24, b_3 = 9/24$

$$U_i - \frac{9h}{24} f(t_i, U_i) = U_{i-1} + \frac{h}{24}(F_{i-3} - 5F_{i-2} + 19F_{i-1})$$

The use of F_{i-k} notation emphasizes that these earlier values of $F_{i-k} = f(t_{i-k}, U_{i-k})$ are known from a previous step, so can be stored for reuse.

The backward Euler method has not been mentioned before; it comes from using the backward counterpart of the forward difference approximation of the derivative:

$$u'(t) \approx \frac{u(t) - u(t-h)}{h}$$

Like Euler's method it is only first order accurate, but it has excellent *stability* properties, which makes it useful in some situations.

Example 7.7 (An equation with fast and slow time scales)

The equation

$$y'' + 101y' + 100y = 0, \quad y(0) = y_0, y'(0) = v_0$$

has the general solution

$$y(t) = Ae^{-t} + Be^{-100t}$$

With explicit methods, the time-step size has to be smal enought to resove the fast time scale, so h of order $1/100$, but the second term decays very rapdly, and with some equations like this it is enough to resolve the dominant behavior of the e^{-t} term, which some implict method can resolve with far larger step sizes.

Let us compare how this is handled by the implicit trapezoid and explicit trapezoid methods.

The system form is

$$\frac{d}{dt}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -100 & -101 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

The trapezoid method is thus

$$U_i - \frac{h}{2}\begin{bmatrix} 0 & 1 \\ -100 & -101 \end{bmatrix}U_i = U_{i-1} + \frac{h}{2}\begin{bmatrix} 0 & 1 \\ -100 & -101 \end{bmatrix}$$

or

$$\begin{bmatrix} 1 & -h/2 \\ 50h & 1+101h/2 \end{bmatrix}U_i = \begin{bmatrix} 1 & h/2 \\ -50h & 1-101h/2 \end{bmatrix}U_{i-1}$$

```
using PyPlot
```

```
include("NumericalMethods.jl")
using .NumericalMethods: explicittrapezoid_system, approx3
```

```
function f_fast_slow(t, u)
    return [u[2], -K*u[1] - (K+1)*u[2]]
end;
```

```
function y_fast_slow(t; t_0, y_0, v_0, K)
    B = -(y_0 + v_0)/(K-1)
    A = y_0 - B
    return A*exp(-(t-t_0)) + B*exp(-K*(t-t_0))
end;
```

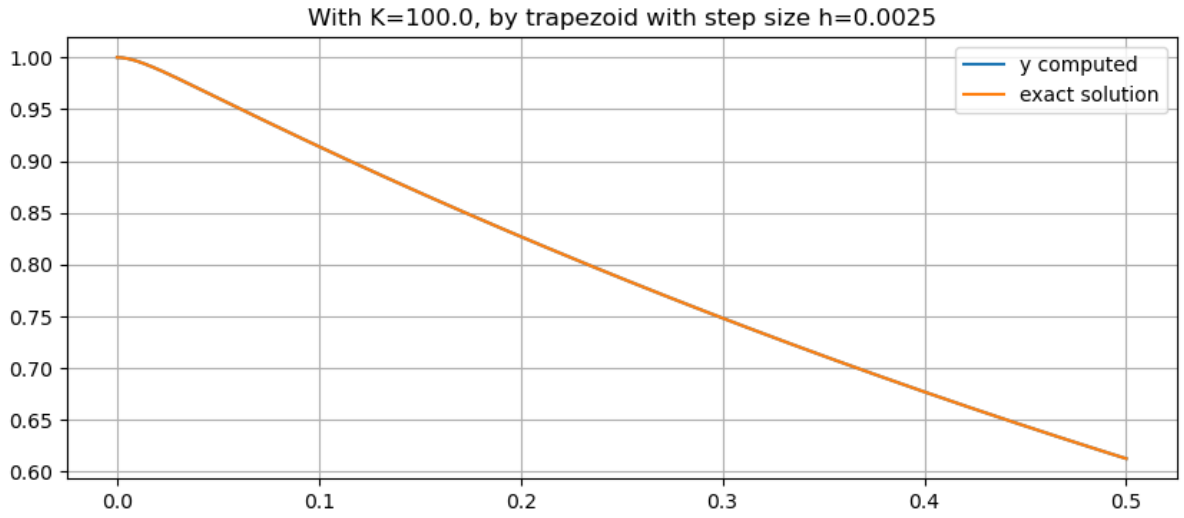
```
K = 100.0
y_0 = 1.0
v_0 = 0.0
u_0 = [y_0; v_0]
a = 0.0
b = 0.5;
```

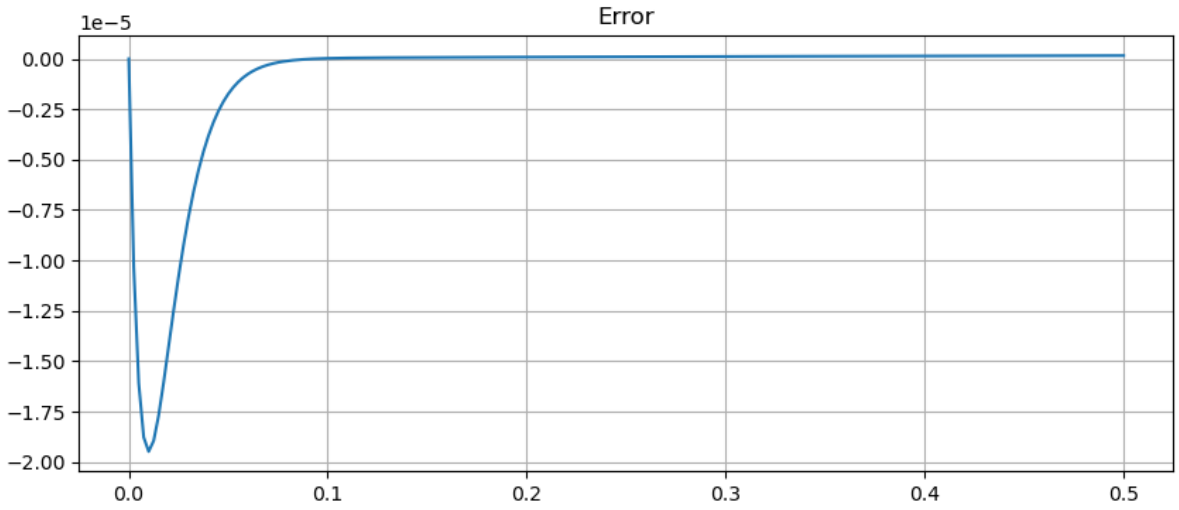
```
n = 200  # explicit trapezoid works
#n = 50  # explicit trapezoid works, barely

h = (b-a)/n
t = range(a, b, n+1)
M = [(1)   (-h/2);
```

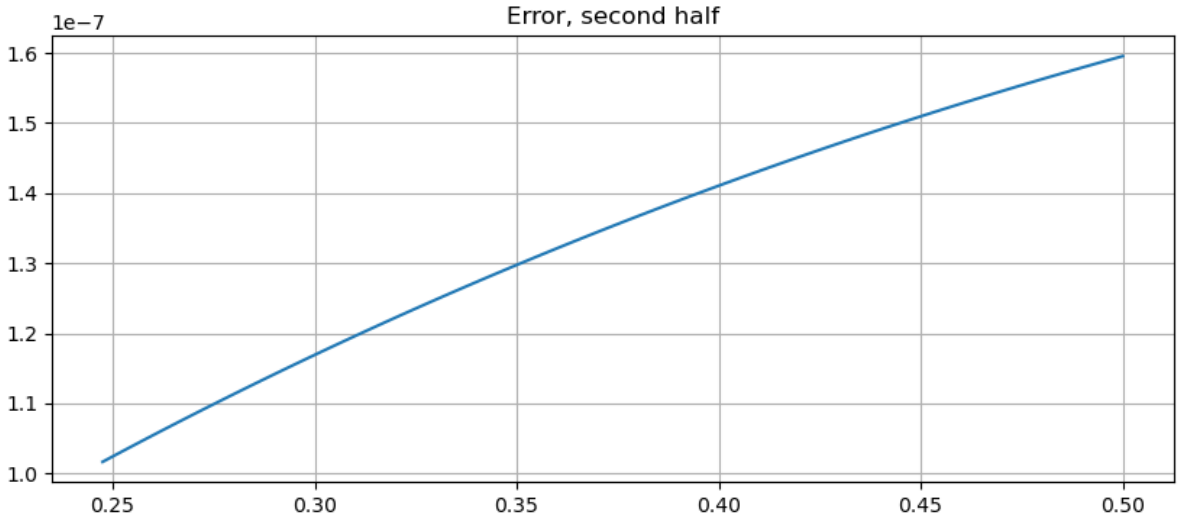
(continues on next page)

(continued from previous page)

```
     (50h) (1 + 101h/2)]
N = [(1)     (h/2);
     (-50h) (1 - 101h/2)]
U = zeros(n+1, 2)
U[1,:] = u_0
for i in 1:n
    U[i+1,:] = M\(N*U[i,:])
end

Y = U[:,1]
y = y_fast_slow.(t, t_0 = 0.0, y_0=y_0, v_0=v_0, K=K)
figure(figsize=[10,4])
title("With K=$K, by trapezoid with step size h=$(approx3(h))")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
grid(true)

Yerror = y-Y
figure(figsize=[10,4])
title("Error")
plot(t, Yerror)
grid(true)

n_middle = n ÷ 2
figure(figsize=[10,4])
title("Error, second half")
plot(t[n_middle:end], Yerror[n_middle:end])
grid(true)
```







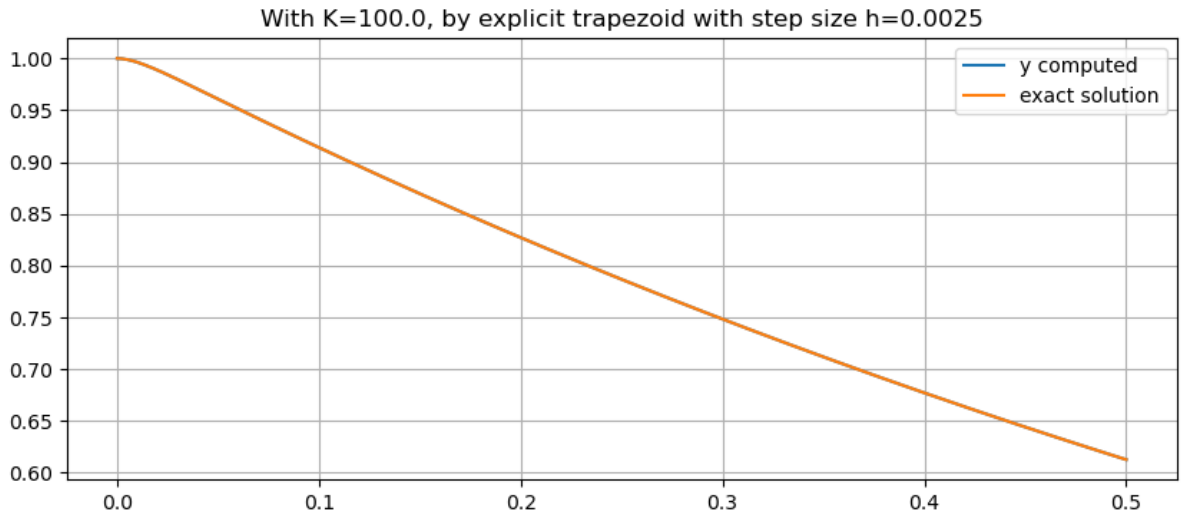
Compare to the explicit trapezoid method

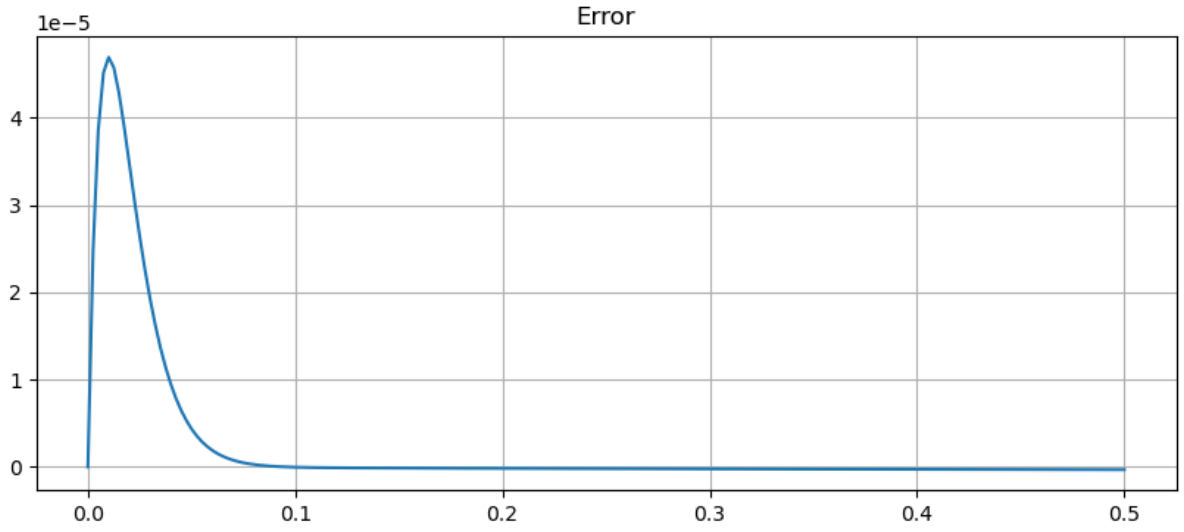
```
(t_ET, U) = explicittrapezoid_system(f_fast_slow, a, b, u_0, n)
Y = U[:,1]
y = y_fast_slow.(t, t_0 = 0.0, y_0=y_0, v_0=v_0, K=K)

figure(figsize=[10,4])
title("With K=$K, by explicit trapezoid with step size h=$(approx3(h))")

plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
grid(true)

yerror = y-Y
figure(figsize=[10,4])
title("Error")
plot(t, yerror)
```

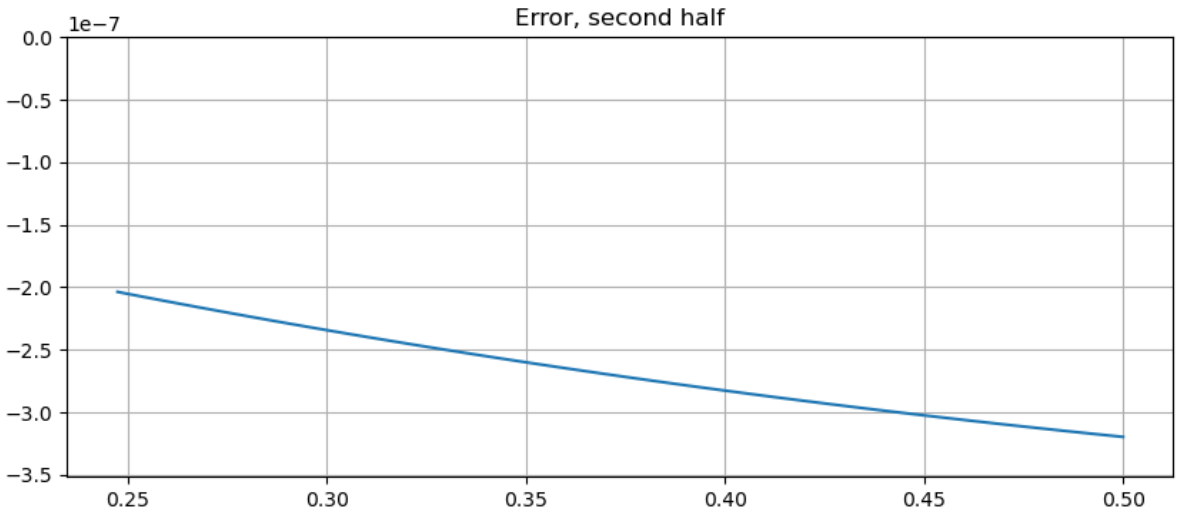
(continues on next page)

(continued from previous page)

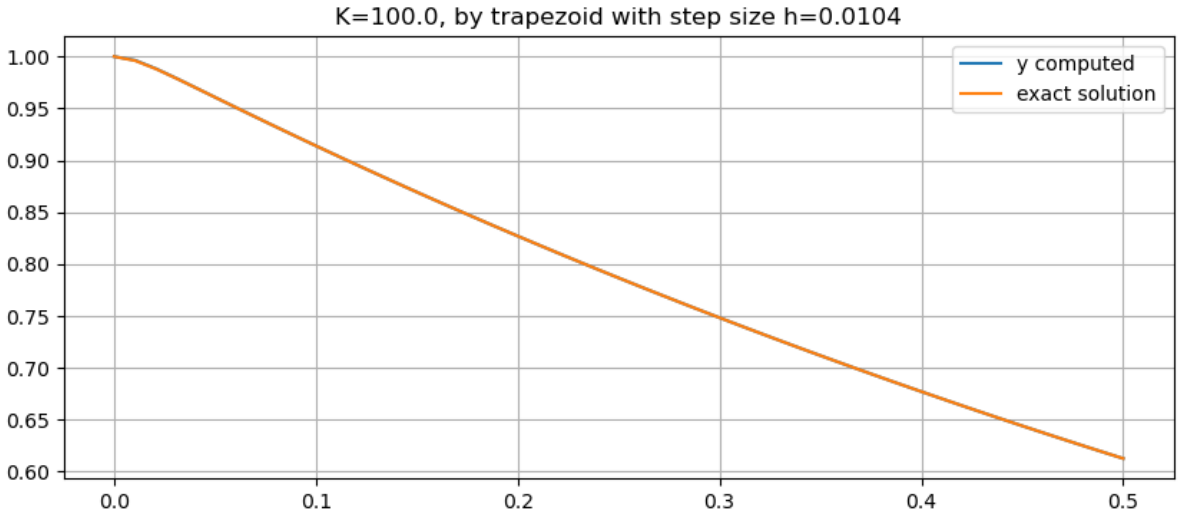
```
grid(true)

n_middle = n ÷ 2
t_secondhalf = t[n_middle:end]
yerror_secondhalf = yerror[n_middle:end]
yerrormin = minimum([minimum(yerror_secondhalf), 0.0])*1.1
yerrormax = maximum([maximum(yerror_secondhalf), 0.0])*1.1

figure(figsize=[10,4])
title("Error, second half")
ylim(yerrormin, yerrormax)
plot(t_secondhalf, yerror_secondhalf)
grid(true)
```

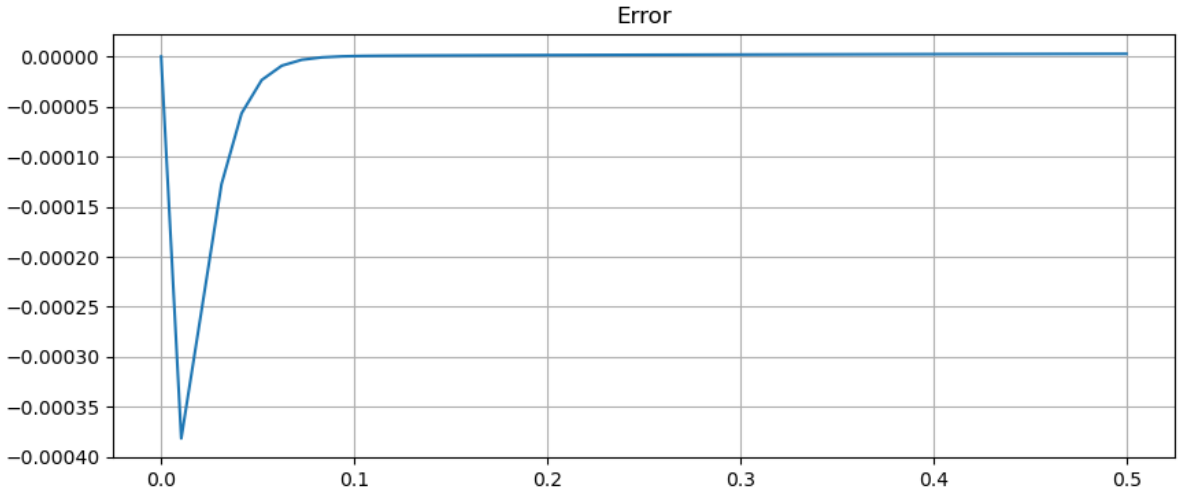


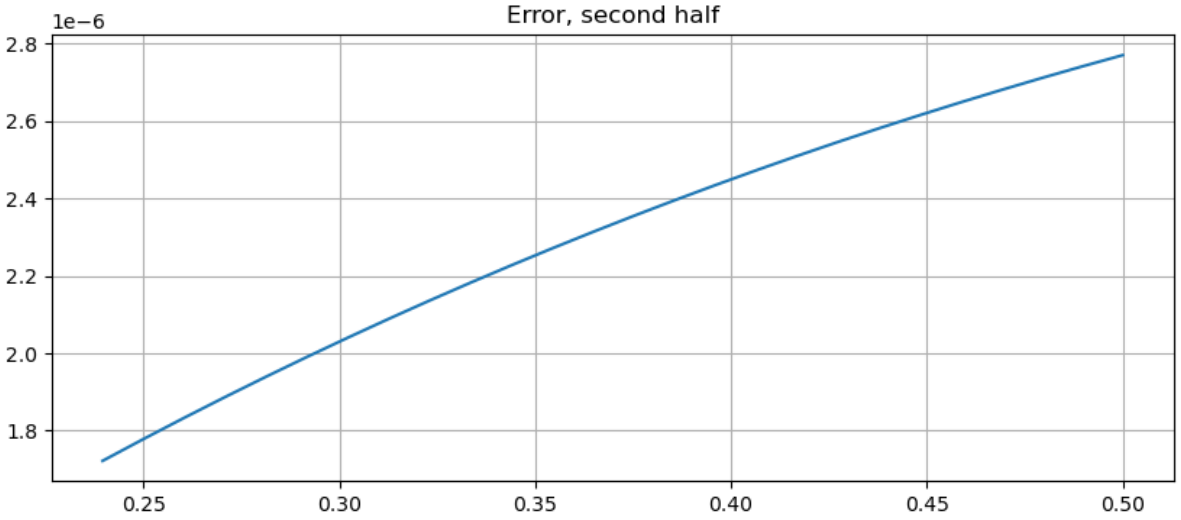





```
n = 48  # explicit trapezoid fails

h = (b-a)/n
t = range(a, b, n+1)
M = [(1)    (-h/2);
     (50h) (1 + 101h/2)]
N = [(1)     (h/2);
     (-50h) (1 - 101h/2)]
U = zeros(n+1, 2)
U[1,:] = u_0
for i in 1:n
    U[i+1,:] = M\(N*U[i,:])
end

Y = U[:,1]
y = y_fast_slow.(t, t_0 = 0.0, y_0=y_0, v_0=v_0, K=K)

figure(figsize=[10,4])
title("K=$K, by trapezoid with step size h=$(approx3(h))")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
grid(true)

Yerror = y-Y
figure(figsize=[10,4])
title("Error")
plot(t, Yerror)
grid(true)

n_middle = n ÷ 2
figure(figsize=[10,4])
title("Error, second half")
plot(t[n_middle:end], Yerror[n_middle:end])
grid(true)
```





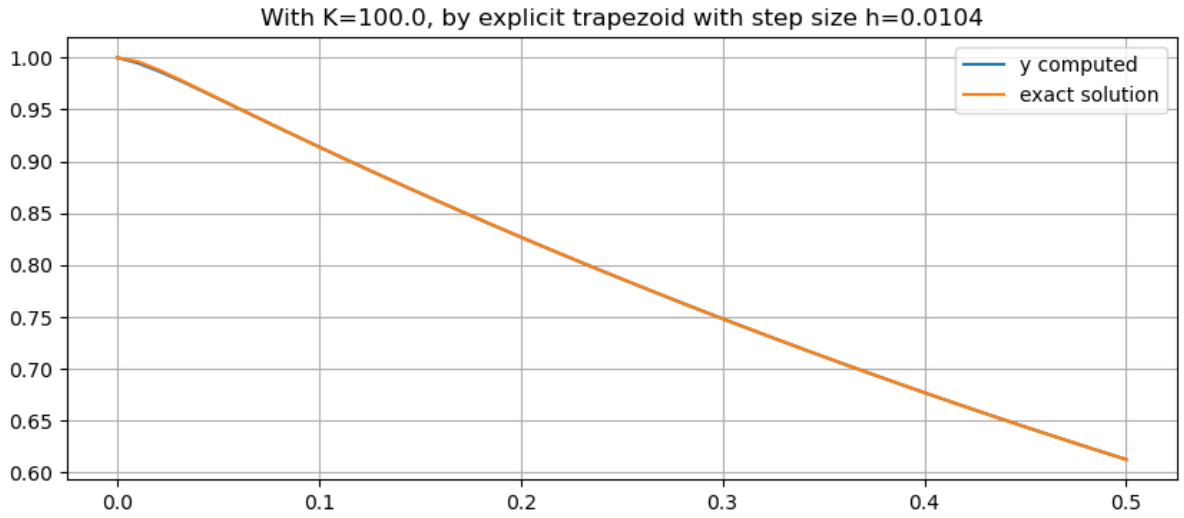


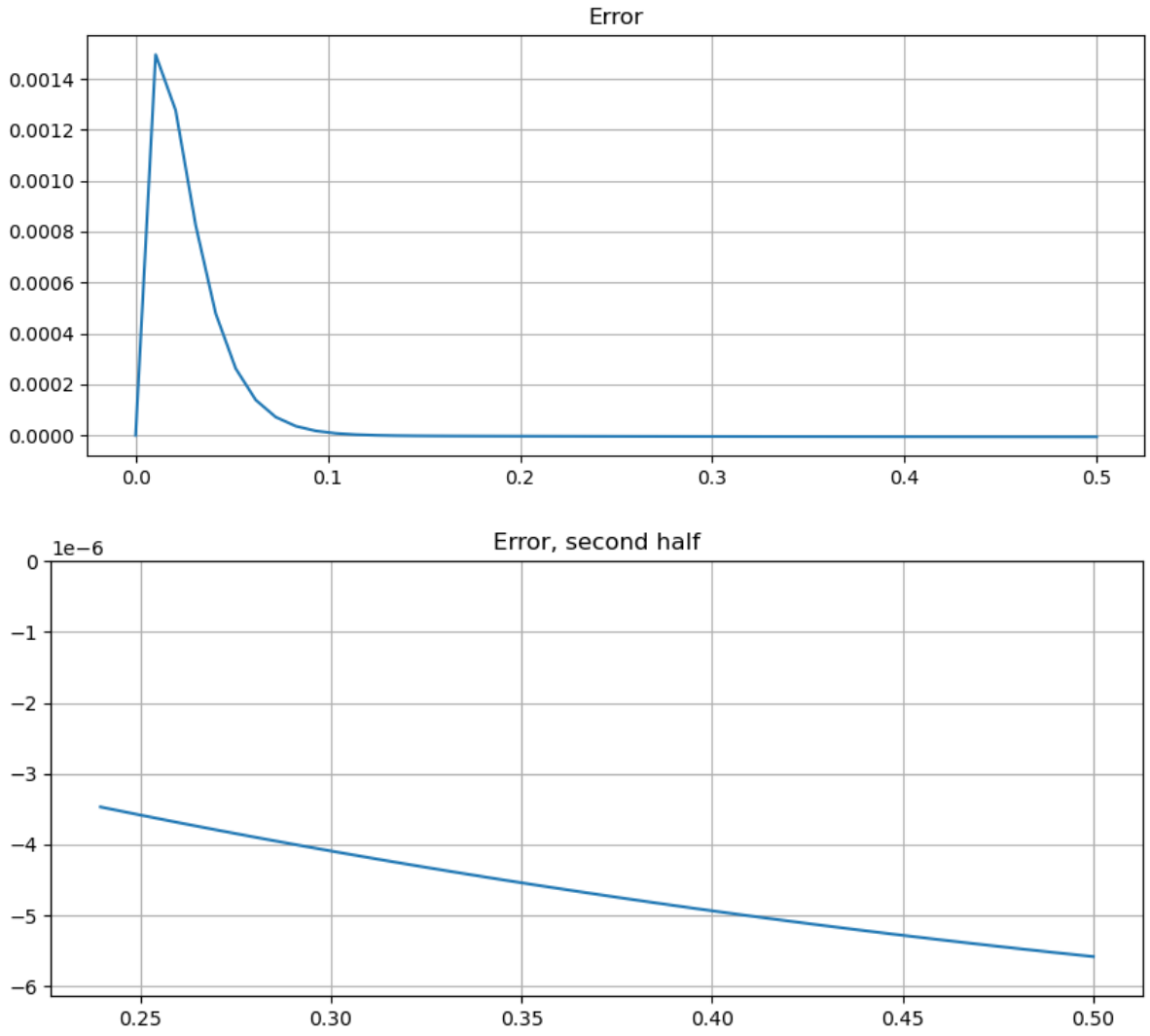

```
(t_ET, U) = explicittrapezoid_system(f_fast_slow, a, b, u_0, n)
```

(continues on next page)

(continued from previous page)

```
Y = U[:,1]
y = y_fast_slow.(t, t_0 = 0.0, y_0=y_0, v_0=v_0, K=K)

figure(figsize=[10,4])
title("With K=$K, by explicit trapezoid with step size h=$(approx3(h))")
plot(t, Y, label="y computed")
plot(t, y, label="exact solution")
legend()
grid(true)

yerror = y-Y
figure(figsize=[10,4])
title("Error")
plot(t, yerror)
grid(true)

n_middle = n ÷ 2
t_secondhalf = t[n_middle:end]
yerror_secondhalf = yerror[n_middle:end]
yerrormin = minimum([minimum(yerror_secondhalf), 0.0])*1.1
yerrormax = maximum([maximum(yerror_secondhalf), 0.0])*1.1

figure(figsize=[10,4])
title("Error, second half")
ylim(yerrormin, yerrormax)
plot(t_secondhalf, yerror_secondhalf)
grid(true)
```





Example 7.8 (Comparing 4th order methods: Adams-Moulton vs Runge-Kutta (coming soon))

Implementing the $s = 3$ case above, which is fourth order accurate, and compare it to the classical Runge-Kutta method *Algorithm 7.3*.

Rather than implementing any of these in general, the next section introduces a strategy for converting these to explicit methods, much as in section *Runge-Kutta Methods* Euler's method (Adams-Bashforth $s = 1$) was combined with the trapezoid method (Adams-Moulton $s = 1$) to get the explicit trapozoid method: an explicit method with the same order of accuracy as the latter of this pair.

7.8.2 Exercises

Coming soon.

CHAPTER

EIGHT

BIBLIOGRAPHY

CHAPTER

NINE

APPENDICES

TO DO: add an overview of the appendices.

9.1 Installing Julia and some useful add-ons

Remark 9.1 (TO DO)

Reformat using package prf.

To use Julia with the code and notebook in this book, there are several steps:

1. *Install the Julia language itself*
2. Make Julia usable in notebooks
    1. *Install Anaconda*, which contains the JupyerLab notebook tool.
    2. *Enable use of Julia from within JupyerLab*.
3. *Install a few additional packages* (Step 3 can be done before step 2.)

9.1.1 Installing Julia

Get it from the Julia download site.

9.1.2 Installing Anaconda (for Jupyterlab)

First, download the installer from www.anaconda.com. This page should detect your OS and hardware type (i.e. x86 or Apple silicon) and point you to the appropriate version. If not, click the logo for your OS below and select from the list.

Installation (at least on macOS) might best be done by deviating from the default, which is to install for the current user only, inside your file space (in cse I supose you do not have admin pvivileges to put it elsewhere.) If instead you want it available to all users in the standard place, select “Install on a specific disk” and then navigate to the appropriate folder.

9.1.3 Enabling Julia in JupyterLab

This is done by *adding package "IJulia"*, so read on.

9.1.4 Adding some useful Julia packages

The following `Pkg.add` commands should only need to be done "once per computer"; each one requires first making `Pkg` available with

```
using Pkg
```

All these commands can be done with the basic Julia comand line tool installed as *above*; alternativel once `IJulia` is installed, they can be run from a notebook; for example by running this one.

Package "IJulia", which enables Julia in JupyterLab

This adds the option of creating and using Jupyter notebooks that use Julia to the basic option of Python; it also adds the option of opening a Julia console as a tab inside JupyterLab, for running Julia commands interactively.

```
Pkg.add("IJulia")
```

Use an interface to the Python package Matplotlib.pyplot

```
Pkg.add("PyPlot")
```

Deployed in a particular notebook or Julia session with

```
using PyPlot
```

Allow LaTeX notation in creating strings, using prefix "L".

For example L"y = \cos(x^2)" gives $y = \cos(x^2)$

```
Pkg.add("LaTeXStrings")
```

Deployed with

```
using LaTeXStrings
```

Add some random number stuff

```
Pkg.add("Random")
```

Deploy with

```
using Random
```

9.2 Notes on the Julia Language

Revised September 23, 2022

These notes are not intended to teach the Julia language from scratch, or comprehensively; instead they are addressed to readers who already know somewhat similar tools like Matlab or "Python Scientific" (Python + Numpy + Matplotlib), and they focus on the parts of Julia actually used in this book.

A very short version of the story is that what works in Matlab often also work in Julia; one design principle is offering familiarity for Matlab users by having the Matlab syntax work (at least as one option) except where that would get in the way of moderization. Though some syntax comes from Python and some from C. And one bit is from Perl.

An intermediate version is this summary of Julia's noteworthy differences from other languages, with comparisons to Matlab and Python (and also to R and C/C++).

For the full story, see resources like the official Julia documentation. (Don't read it all at once: the PDF version is over 1500 pages.)

9.2.1 Characters and strings

The character set is Unicode via UTF-8, so Greek letters and much more are usable.

I generally advise against going beyond plain old ASCII characters, but see the functions plu and forwardsubstitution in the collection *Module NumericalMethods* where Unicode is used to illustrate the possibility of following mathematical notation more closely.

The usual style is that the names of variable and functions use only (Roman alphabet) letter, digits, and possibly underscores to separate words in names that are descriptive phrases.

```
println("π/2 = ", π/2)
```

```
π/2 = 1.5707963267948966
```

(That function `println` will be explained below in *Displaying values*.)

So not only is π is the character set; it is the name of predefined constant. (which can also be referred to as `pi`)

Typing such characters is done using LaTeX notation and tabbing: for π, type the three character sequence `\pi` and then press TAB.

The above characters are in single right quotes (to be pedantic, apostrophes), whereas strings of characters must be surrounded, more properly, by double quote characters:

```
"This is a string"
'This is a syntax error'
```

This one comes from C, I think.

Aside: here is a quirky use of Unicode built in to Julia:

```
√4
```

```
2.0
```

String concatenation and duplication

To concatenate two strings, "multiply" them with `*`. (Versus "adding" them as in Python.)

Also, consistent with that, making multiple copies of a string is done by "exponentiation", with `^`:

```
greeting = "Hello"
audience = "world"
sentence = greeting * ' ' * audience * '.'
println(sentence)
println((greeting*' ')^3)
```

```
Hello world.
Hello Hello Hello
```

Note that single characters can also be concatenated into strings, and exponentiated:

```
gamma = 'g' * 'a' * 'm'^2 * 'a'
```

```
"gamma"
```

9.2.2 Displaying values: `println`, `print` and just saying the name

We have seen above two basic ways of getting output displayed on the screen:

- the function `println`
- putting an expression on the last line of a cell (or typing it into the interactive Julia command line), which causes that expression to be evaluated and the result displayed. This is as in both Matlab and Python, and with the same method for supressing output: ending that line with a semi-colon.

The basic usage of function `println` is as for the synonymous function in Matlab and `print` in Python: input is a sequence of strings whose values are ouptut on a line, after which output moves to a newline.

There is also a variant `print` which does not go to a new line at the end; useful for assembling a line of output piece-by-piece:

```
println("The"," end.")
println()
print("That's")
print(" all")
print(" folks")
println("!")
```

```
The end.

That's all folks!
```

Note that in each case, the spaces must be explicitly specified. Also, the sequence of strings can be empty: get just a blank line with `println()`.

9.2.3 Displaying the values of variables and expressions

As already seen above, there are several ways of display the value of a variable, or more generally of an expression.

One is to put such expressions in the sequence of arguments to `println` (so I lied slightly about the arguments being strings):

```
a = 2
println("a = ", a, " and its cube is ", a^3)
```

```
a = 2 and its cube is 8
```

Another is to use '$' interpolation, adopted from the Perl language:

```
println("a = $a and its cube is $(a^3)")
```

```
a = 2 and its cube is 8
```

When the expression whose value to interpolate is just a variable name, it is enough to prefix that name by '$. However more involved expressions must be parenthesised, as with `(a^3)`.

Fun fact: the string in the last example is an expression whose value is the string displayed, so its value can also be displayed by simply having it as the last (and semicolon-free) line of a cell:

```
"a = $a and its cube is $(a^3)"
```

```
"a = 2 and its cube is 8"
```

This is useful to know since it reveals a nice "orthogonality of features": interpolation is not a special feature of function `println` but something that can be used in other situations; probably the most common use in this book will be putting annotations on graphs.

Sometimes it is illuminating to use the "last line automatic display" feature rather than `print` because it reveals some information about the type of a quantity rather than just its value. This book uses that often when discussing Julia language features rather than when actually doing numerical computing; see for example the notes on *arrays* below.

On the other hand, many expressions give output values that you might not expect, like the definitions of *functions* and function `plot` in the notes on *modules* below; thus you might want to end many cells with a semi-colon to supress unneeded output. An end-of-line semi-colon never hurts, even where redundant, so you may type like a C programmer if you wish.

Semicolons can also be used to combine statements on a single line. This is often considered as poor style, but I sometimes find that it improves readability with several short and closely related statements:

```
a = 1; b = 3
print("a=",a,", b=",b)
```

```
a=1, b=3
```

9.2.4 Boolean values (true-false)

These are `true` and `false` and nothing else:

- not capitalized as they are in Python
- not "1" and "0" or "nonzero" vs "zero" or any other laziness.

The logical operators are much as in C: "&&", "||", "&", "|", and "!" for negation.

Be careful with that last one; logical negation is not "~" as in Matlab.

The doubled forms "&&" and "||" use lazy or "short-circuiting" evaluation: if the value the left-hand term determines the truth value of the whole, then the second term is not evaluated. For example, the following avoids division by zero:

```
if q != 0 && -1 < p/q < 1
    println("$p/$q is a proper fraction")
else
    println("$p/$q is not a proper fraction")
end
```

9.2.5 Comparisons

The usual comparisons of numbers and tests for equality exist,

```
< <= == >= !=
```

with a couple of extra twists.

the first is the comparisons can be chained, as seen above:

```
-1 < p/q < 1
```

is equivalant to

```
-1 < p/q && p/q < 1
```

but both more readable and more efficient, because the middle term is only evaluated once. Like `&&`, this is short-circuiting.

(This one comes from Python, the only other language I know of with this feature.)

This can even be done in cases where usual mathematical style forbids, with reversing of the direction of the inequalities:

```
2 < 4 > 3
```

```
true
```

The second extra is the function `isequal(a, b)`. This is mostly the same as == but with some special handling for the special values `-0.0` and `NaN` of IEEE floating point numbers:

```
NaN == NaN
```

```
false
```

```
isequal(NaN, NaN)
```

```
true
```

```
-0.0 == 0.0
```

```
true
```

```
isequal(-0.0, 0.0)
```

```
false
```

9.2.6 Numbers

As just mentioned, the default floating point numbers and arithmetic are IEEE Float64, and the default integers are Int64; others are available (like Int32 and unsigned integers) but they will not be mentioned again in this book.

Thus the main novelty here is a notation: the underscore can be used within numbers to improve readabilty, as commas are (and periods in some countries.) They can be put wherever you like:

```
two_thousand_million = 2_000_000000  # British English
println("two_thousand_million is ",two_thousand_million)
crore = 1_00_00_000  # This is where Indian style puts the commas
println("In the Indian number naming system, one crore is $crore")
almostpi = 3.1415_9265
println("pi is approximately $almostpi")
```

```
two_thousand_million is 2000000000
In the Indian number naming system, one crore is 10000000
pi is approximately 3.14159265
```

A point on recommended style: although floating point numbers can be typed with the decimal point at the beginning or the end, as with `2.` and `.5`, it is recommended style to always have digits around the decimal point, as with `2.0` and `0.5`.

One reason is that the period '.' is used with many other meanings, so this style helps to avoid ambiguities; see below about *vectorization*.

Complex numbers

Julia uses `im` for the square root of -1 rather than `i` or `j`, and `im` cannot be used as the name of a variable.

In general, the complex number $a + bi$ is expressed as `a + bim` where 'b' is a literal number, not the name of a variable. Then `im` is immediately juxtaposed with that number, no intervening space allowed).

```
z = 3 + 4im
println("z = ", z)
println("Its absolute value is ", abs(z))
```

```
z = 3 + 4im
Its absolute value is 5.0
```

As you might expect, imaginary numbers can be written without the real part a, as `bim` and when $b = 1$, it can be ommited

However, the imaginary part `b` is always needed, even when $b = 0$, to announce that the nuber is to br treated a complex.

```
println(2im)
```

```
0 + 2im
```

```
println(im)
```

```
im
```

```
println(im^2)
```

```
-1 + 0im
```

```
println(-1im)
```

```
0 - 1im
```

```
println(-im)
```

```
0 - 1im
```

However, the imaginary part `b` is always needed, even when $b = 0$, to announce that the number is to be treated a complex.

```
sqrt(-1 + 0im)
```

```
0.0 + 1.0im
```

```
sqrt(-1)
```

```
DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. Try␣
↪sqrt(Complex(x)).

Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math ./math.jl:33
 [2] sqrt
   @ ./math.jl:591 [inlined]
 [3] sqrt(x::Int64)
   @ Base.Math ./math.jl:1372
 [4] top-level scope
   @ In[158]:1
 [5] eval
   @ ./boot.jl:368 [inlined]
 [6] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,␣
↪filename::String)
   @ Base ./loading.jl:1428
```

The name `im` is reserved for this role; it cannot be used as a variable name:

```
im = 100
```

```
cannot assign a value to variable Base.im from module Main

Stacktrace:
 [1] top-level scope
   @ In[159]:1
 [2] eval
   @ ./boot.jl:368 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,␣
↪filename::String)
   @ Base ./loading.jl:1428
```

9.2.7 Arithemetic operators

With arithmetic on numbers, the main items to note are

1. exponentiation
2. division with integers, and
3. multiplication by juxtoposition

- Exponentiation a^b is given by `a^b` (as in Matlab and different from Python's `a**b`).
- `p/q` with `p` and `q` both integers promotes to floating point arithmetic.

- To do integer division with integer result, use `p ÷ q` (or `div(p,q)` to avoid that Unicode: all operators also have function forms).
- The remainder is given by `p%q`, or `rem(p,q)`. Thus `(p ÷ q)* q + p%q = p`
- There is also "backward division": `q\p` is the same as `p/q`; see below.
- The product of a literal number by a variable or function value can be indicated by juxtoposition, no '*' needed:

```
println(2π)
println(4tan(pi/4))
```

```
6.283185307179586
3.9999999999999996
```

9.2.8 Arrays

Julia has numerical arrays bult in, and the basic construction notation is much as in Matlab, with semicolons separating rows. However, not everything is a matrix of real numbers:

- Integer values are supported.
- There is a distinction between matrices and vectors, with the latter being equivalent to single column matrices.
- Single row matrices are still considered as matrices.

This all follows common mathematical conventions, and so does multiplication.

Vectors (1-index arrays) are created with bracketed, comma separated lists; note the column vector presentation.

```
v = [1, 2, 3]
```

```
3-element Vector{Int64}:
 1
 2
 3
```

Matrices are created with rows separated by semicolons and elements with a row separated only by spaces:

```
A = [ 1.0 2.0 3.0 ; 4.0 5.0 6.0 ]
```

```
2×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
```

Note the difference from vector `v` above:

```
u = [ 1 2 3 ]
```

```
1×3 Matrix{Int64}:
 1  2  3
```

On the other hand, a vector can be created as a one-column matrix:

```
v_as_matrix = [ 1 ; 2 ; 3 ]
```

```
3-element Vector{Int64}:
 1
 2
 3
```

```
v == v_as_matrix
```

```
true
```

```
v == u
```

```
false
```

```
u * v  # like u-transpose times v; the inner product
```

```
1-element Vector{Int64}:
 14
```

```
outer = v * u  # The outer product
```

```
3×3 Matrix{Int64}:
 1  2  3
 2  4  6
 3  6  9
```

```
v * v  # vector times vector, a mismatch
```

```
MethodError: no method matching *(::Vector{Int64}, ::Vector{Int64})
Closest candidates are:
  *(::Any, ::Any, ::Any, ::Any...) at operators.jl:591
  *(::StridedMatrix{T}, ::StridedVector{S}) where {T<:Union{Float32, Float64,␣
↪ComplexF32, ComplexF64}, S<:Real} at /Applications/Julia-1.8.app/Contents/
↪Resources/julia/share/julia/stdlib/v1.8/LinearAlgebra/src/matmul.jl:49
  *(::StridedVecOrMat, ::LinearAlgebra.Adjoint{<:Any, <:LinearAlgebra.LQPackedQ})␣
↪at /Applications/Julia-1.8.app/Contents/Resources/julia/share/julia/stdlib/v1.8/
↪LinearAlgebra/src/lq.jl:269
  ...

Stacktrace:
 [1] top-level scope
   @ In[169]:1
 [2] eval
   @ ./boot.jl:368 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,␣
↪filename::String)
   @ Base ./loading.jl:1428
```

A row matrix can be converted to a vector with function `collect`, which we will see again below.

```
collect(v)
```

```
3-element Vector{Int64}:
 1
 2
 3
```

Array indexing and slicing

Indices run from 1 (like Matlab, unlike Python), **but** indices are indicated with brackets (like Python, unlike Matlab — which uses parentheses for too many things!)

```
v[1]
```

```
1
```

```
A[2,3]
```

```
6.0
```

Be careful with single indexing a higher dimensional array.

This treats the elements of the array as if laid out in a singe row ("flattened"), rather than selecting a row as in Python.

```
A[1], A[2], A[3], A[4], A[5], A[6]
```

```
(1.0, 4.0, 2.0, 5.0, 3.0, 6.0)
```

Also note the order: the columns are joined end-to-end, not the rows: Julia follows Matlab (and Fortran) in storing arrays in **column-major order**, as opposed to the **row-major order** of Python (and C ,and Java, and in general in languages that count fom zero.)

One can also index expressions directly

```
(v*u)[2,2]
```

```
4
```

Indexing part of an array can be done with **slicing** which works mostly as in Matlab.

The basic form is `first:last`, selecting indices from `first` to `last` *inclusive* (like Matlab, unlike Python)

```
v[2:3]
```

```
2-element Vector{Int64}:
 2
 3
```

```
A[1:2,2:3]
```

```
2×2 Matrix{Float64}:
 2.0  3.0
 5.0  6.0
```

The final index value can be indicated by `end`, and index arithmetic can be done on this:

```
println("The last entry in v is ", v[end], ", the penultimate is ", v[end-1])
println("The last two columns of A are")
A[1:end, end-1:end]
```

```
The last entry in v is 3, the penultimate is 2
The last two columns of A are
```

```
2×2 Matrix{Float64}:
 2.0  3.0
 5.0  6.0
```

When the slice on an index is all values, one can use `:` instead of `1:end`. So the previous example could also be done as

```
A[:, end-1:end]
```

```
2×2 Matrix{Float64}:
 2.0  3.0
 5.0  6.0
```

One can also specify an arbitrary collection of indices by giving a bracketed list (a one index array) of indices.

For example, The "corners" of the outer product `u * v`, flipped vertically, are

```
outer[[3,1], [1,3]]
```

```
2×2 Matrix{Int64}:
 3  9
 1  3
```

and we can permute rows like this

```
outer[[3,1,2],:]
```

```
3×3 Matrix{Int64}:
 3  6  9
 1  2  3
 2  4  6
```

A tricky point: slicing to a single row matrix vs slicing to a vector.

For some matrix-vector caclulations is is necessary to ensure that an object is a "1 by n" row matrix rather than an n-vector (which is an "n by 1" column matrix) but unfortunately, **slicing out a single row of matrix returns a vector:**

```
A[1,:]  # gives a vector, a.k.a. a 1 column matrix
```

```
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

To keep the slice as a row, indicating the row as `[i]` rather than just `i`:

```
A[[1],:]  # gives a 1 row matrix
```

```
1×3 Matrix{Float64}:
 1.0  2.0  3.0
```

So this works as an outer product

```
[3, 4, 5] * A[[1],:]
```

```
3×3 Matrix{Float64}:
 3.0   6.0   9.0
 4.0   8.0  12.0
 5.0  10.0  15.0
```

but this fails as a "vector times vector""

```
[3, 4, 5] * A[1,:]
```

```
MethodError: no method matching *(::Vector{Int64}, ::Vector{Float64})
Closest candidates are:
  *(::Any, ::Any, ::Any, ::Any...) at operators.jl:591
  *(::StridedMatrix{T}, ::StridedVector{S}) where {T<:Union{Float32, Float64,␣
↪ComplexF32, ComplexF64}, S<:Real} at /Applications/Julia-1.8.app/Contents/
↪Resources/julia/share/julia/stdlib/v1.8/LinearAlgebra/src/matmul.jl:49
  *(::StridedVecOrMat, ::LinearAlgebra.Adjoint{<:Any, <:LinearAlgebra.LQPackedQ})␣
↪at /Applications/Julia-1.8.app/Contents/Resources/julia/share/julia/stdlib/v1.8/
↪LinearAlgebra/src/lq.jl:269
  ...

Stacktrace:
 [1] top-level scope
   @ In[184]:1
 [2] eval
   @ ./boot.jl:368 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,␣
↪filename::String)
   @ Base ./loading.jl:1428
```

Slicing can also select equally spaced values with syntax `first:step:last`

Warning to Python users: the order is different (Python uses `first:last:step`)

```
A[:,1:2:3]
```

```
2×2 Matrix{Float64}:
 1.0  3.0
 4.0  6.0
```

In particular, this allows counting backwards, with step of -1:

```
A[:,end:-1:1]
```

```
2×3 Matrix{Float64}:
 3.0  2.0  1.0
 6.0  5.0  4.0
```

One big difference from Matlab or Python is some "orthogonality": rather than just being a notation used in this context of array indexing, this colon notation is a function (named `:`) whose value is a data type called a **range** that can be used in a variety of contexts, and assigned to variables.

One other usage will be seen soon in the notes on *iteration*. Meanwhile:

```
oddcolumnsorrows = 1:2:3
```

```
1:2:3
```

```
print("The odd numbered rows of A are")
A[:,oddcolumnsorrows]
```

```
The odd numbered rows of A are
```

```
2×2 Matrix{Float64}:
 1.0  3.0
 4.0  6.0
```

```
print("The odd numbered rows of the outer product 'v u' are")
(v*u)[oddcolumnsorrows,:]
```

```
The odd numbered rows of the outer product 'v u' are
```

```
2×3 Matrix{Int64}:
 1  2  3
 3  6  9
```

Function `range`

Closely related to this slice syntax is the function `range`, which produces a range value as above, and has several flavors:

- `range(start, stop)` returns `start:stop`.
- `range(start, stop, length)` returns a range running from `start` to `stop` with `length` values — like `linspace` in both Matlab and Python.
- `range(start, stop, step=stepsize)` returns `start:stepsize:stop`. But note that this flexibility requires specifying what the third parameter means by using its name: it is a **keyword parameter**.
- More generally, there are four parameters `start`, `stop`, `length` and `step` and you can specify any three (which determines the fourth). However, all other combinations require specifying some or all of them by name, as keyword parameters.

Keyword parameters will be discussed in the section *Functions, Part 2* once that is written.

Meanwhile, some examples.

```
range(1,11)
```

```
1:11
```

```
range(1,2,11)
```

```
1.0:0.1:2.0
```

```
range(1, 2, step=0.1)
```

```
1.0:0.1:2.0
```

```
range(step=0.1, stop=2, length=11)  # A weird but legal way to do it
```

```
1.0:0.1:2.0
```

9.2.9 Tuples

Julia also has tuples, much as in Python.

These resemble one index arrays, excpr that

- The elements can be anythingm and do not need to be of the same type.
- They are **immutable**: leemts can be accessed by indexing, *but cannnot be changed*.

Tuples are created as a comma-separated lists, optionally parenthesized:

```
tuple123 = ('1', "two", 3)
```

```
('1', "two", 3)
```

```
or equivalently
```

```
syntax: extra token "equivalently" after end of expression

Stacktrace:
 [1] top-level scope
   @ In[195]:1
 [2] eval
   @ ./boot.jl:368 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,␣
↪filename::String)
   @ Base ./loading.jl:1428
```

```
tuple123 = '1', "two", 3
```

```
('1', "two", 3)
```

```
tuple123[2]
```

```
"two"
```

but this is not allowed:

```
tuple123[3] = 4
```

```
MethodError: no method matching setindex!(::Tuple{Char, String, Int64}, ::Int64,␣
↪::Int64)

Stacktrace:
 [1] top-level scope
   @ In[198]:1
 [2] eval
   @ ./boot.jl:368 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,␣
↪filename::String)
   @ Base ./loading.jl:1428
```

One very common use of tuples is in functions that return more than one quantity, such as `rowreduction` in *Module NumericalMethods* which returns its results with

```
return (U, c)
```

Strictly a single quantity is returned, but it is a tuple.

9.2.10 Arithmetic operations on arrays: vectorization and broadcasting

- With arrays, addition and subtraction and multiplicato or division by numbers work as one would expect with vector matrices and such.
- Multiplication of arrays as with `A*b` does matrix-matrix or matrix-vector product (again as with Matlab).
- `A/b` and `A\b` are also matrix and vector friendly; in particular `x = A\b` means $x = A^{-1}b$ and so solves $Ax = b$.
- Also copied from Matlab is the "dot" or "pointwise" versions of operators, but this goes beyond what Matlab does. In the following I use lower case lettters for numbers, upper case for arrays, and illustrate only for 1D arrays though it works in higher dimensions too.
- The first case is **vectorization**, creating implicit loops over one or more indices:
  - `C = A .* B` computes the product point-wise product, so `C[i] = A[i]*B[i]`.
  - `C = a.^B` gives `C[i] = a^B[i]`
  - `C = A.^b` gives `C[i] = A[i]^b`
  - `C = A.^B` gives `C[i] = A[i]^B[i]`
- The second concept is **broadcasting** where a number is promoted to an appropriate array with that number in each element.
- For example `a + B` is an error, but `a .+ B` gives array `C` with `C[i] = a + B[i]`.

9.2.11 Condititional statements

One example probably says it all:

```
x = 11.0
```

```
11.0
```

```
if x > 0
    println("x is positive")
elseif x < 0
    println("x is negative")
else
    println("x is zero")
end
```

```
x is positive
```

In other words, "as in Matlab", again.

Note also the use of the Matlab style of using `end` to end blocks of code, which will of course also be seen with loops, function definition and so on. Indentation is optional but "four spaces per level (no tabs)" is the usual style.

Also note the `println`; more on output to the screen (and to files) later.

9.2.12 Iteration with `for`, `while`, `break` and `continue`

The title tells much of the story: this is all very much as in Matlab — and as in Python except with `end` statements.

The only novelty is the details of the `for` statement, which are roughly the union of the Matlab and Python options.

Julia denotes ranges of values with Matlab-style notation:

- `a:b` is the values from `a` to `b` by steps of one,
- `a:step:b` is the values from `a` to `b` by steps of `step`.

This gives the first, Matlab-style `for` loop syntax:

```
for i = 1:5
    print(i) # no end of lines or spaces
end
```

```
12345
```

```
for i = 1:2:5
    print(i," ")
end
```

```
1 3 5
```

```
for i = 5:-1:1
    print(i," ")
end
```

```
5 4 3 2 1
```

However, there is a far more flexible syntax for looping over all kinds of lists and even more general collections. The general form is

```
for item in list
    ...
end
```

but until we see all the sorts of things that the "list" can be, just a few examples:

```
for i in 1:4
    println("$i^2 is $(i^2)")
end
```

```
1^2 is 1
2^2 is 4
3^2 is 9
4^2 is 16
```

```
for x in [1, 4/3, sqrt(11)]
    println("x is $x")
end
```

```
x is 1.0
x is 1.3333333333333333
x is 3.3166247903554
```

A final example using some exotic stuff that will be explained later:

```
theta = pi/6
for f in [sin, cos, tan]
    fname = String(Symbol(f))
    println("$fname($theta) = $(f(theta))")
end
```

```
sin(0.5235987755982988) = 0.49999999999999994
cos(0.5235987755982988) = 0.8660254037844387
tan(0.5235987755982988) = 0.5773502691896257
```

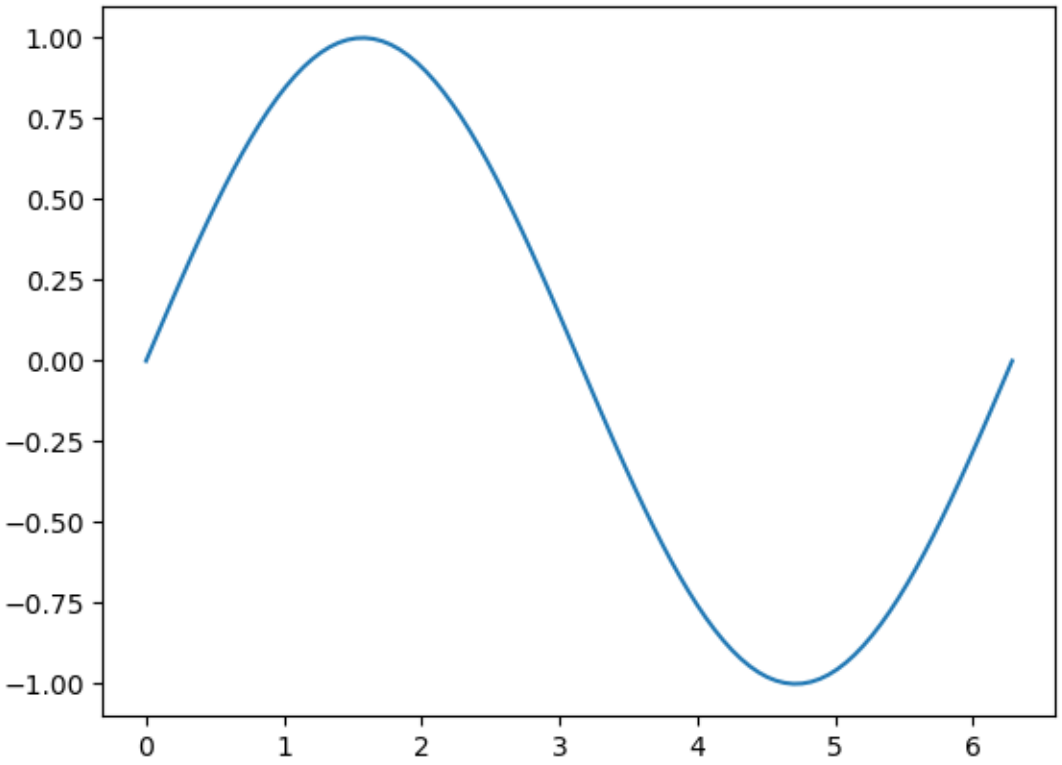
9.2.13 Using modules and packages, and some graph plotting

Much like Python, collections of functions (and other stuff like constants) can be created and their contents used.

There are several ways to do this; here I describe some but not all, and only for a package that already exists (creating your own comes later): `PyPlot`, which is for plotting, and is very similar to both Matlab's plotting commands and the Python package `matplotlib.pyplot`. (There is a bit more more about *PyPlot* below.)

Method 1 `import` a module: make the module (or package) available, and access its contents by "full name"; much like the same statement in Python:

```
import PyPlot
x = range(0.0, 2pi, 100)
y = sin.(x)
PyPlot.plot(x, y);
```

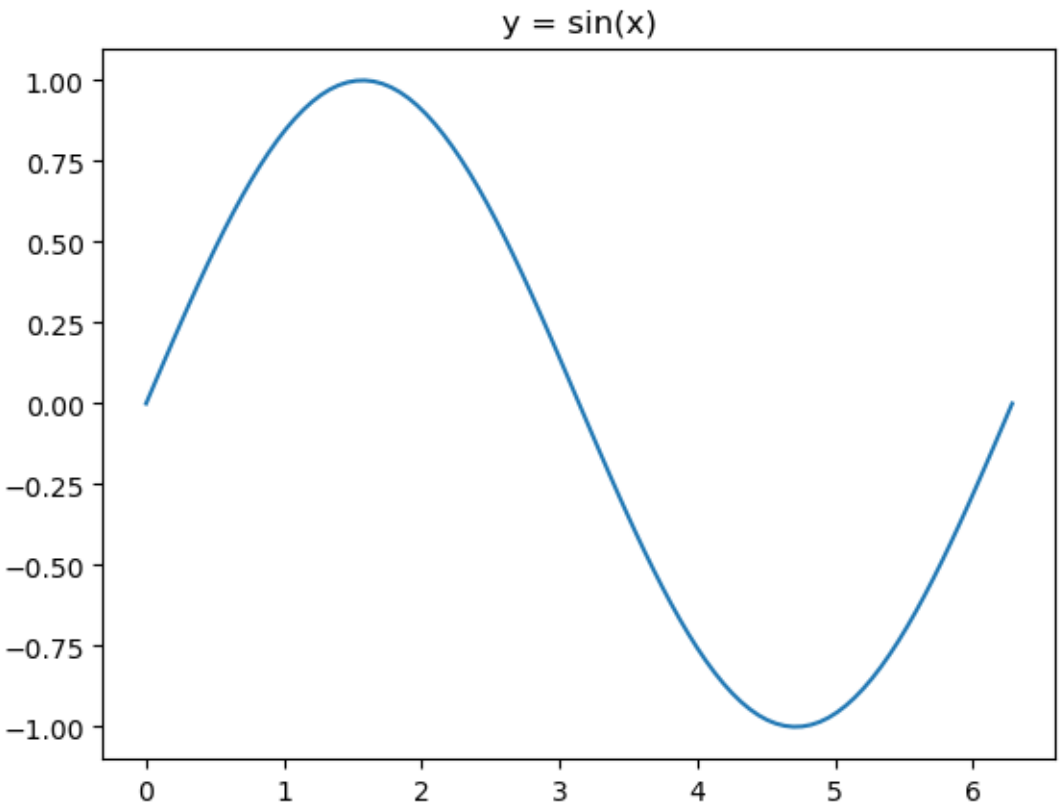


Notes:

- Function `range` is similar to `linspace` in both Matlab and Python; the usage here is `range(first, last, number_of_points)`, giving that many equally spaced values.
- This is our first example of vectorizing a function; it must be `sin.` not just `sin`

Method 2 `import` specific items from a module, making them available by "first name" only; much like `from PyPlot import plot, title` In Python:

```
import PyPlot: plot, title
x = range(0, 2pi, 100)
y = sin.(x)
plot(x, y)
title("y = sin(x)");
```

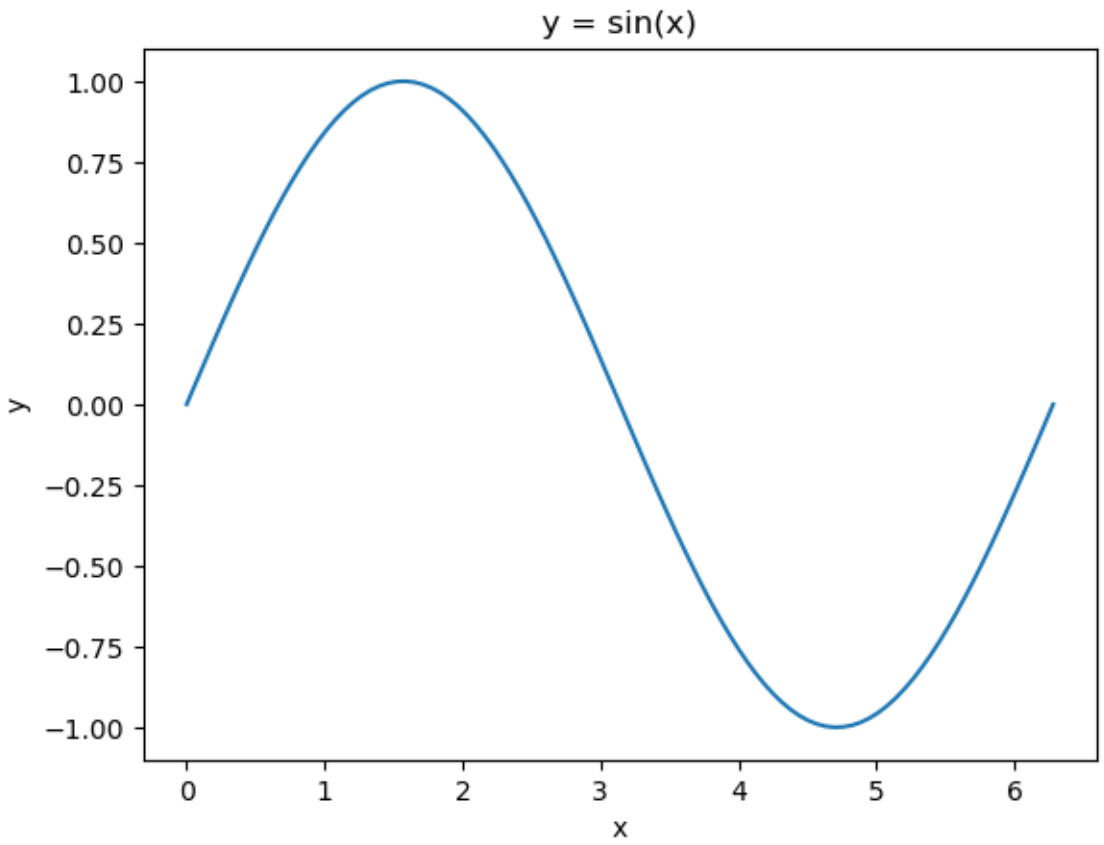


Method 3. `using` a module, which makes all items available on a first-name basis full; much like a wild import `from PyPlot import *` in Python.

But whereas Python style recommends against wild imports, this is common usage in Julia. In this book I will avoid this with user defined modules, to avoid uncertaintly about where an item comes from.

(Confession: the above is not quite true: a module can choose to "export" only some of its items, but not others; then `using` provides only those exported items on a first-name basis; other items must still be accessed by their full name.)

```
using PyPlot
x = range(0, 2pi, 100)
y = sin.(x)
plot(x, y)
title("y = sin(x)")
xlabel("x")
ylabel("y");
```



Note: those one character labels still need to be in double quotes; the argument is a string, not a character.

9.2.14 Functions, part I

Julia's function system is quite innovative, powerful, flexible — and therefore rather complicated to describe. So I start with just the simplest cases.

One liners

A very simple syntax for function that just evaluates a single formula is:

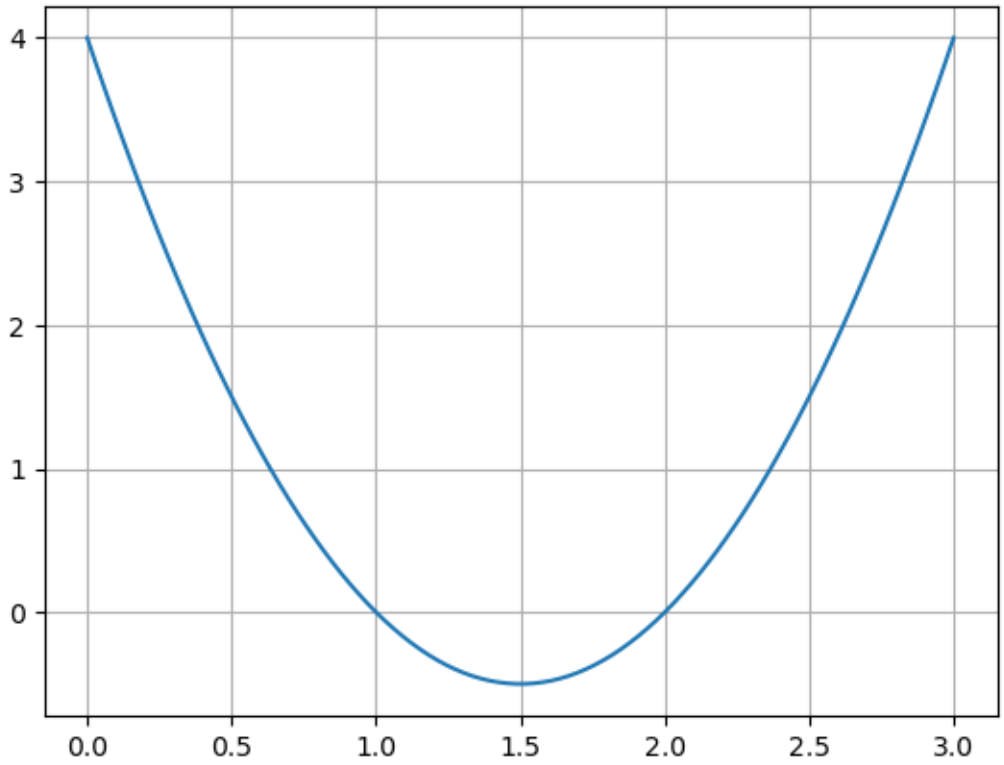
```
f(x) = 2x^2 - 6x + 4;
```

(Multiplication by juxtoposition is nice here!)

Note: semi-colon suppression is used here because the expression defining a function returns a value: some information about the function.

This can be used for graphing, but must be vectorized:

```
x = range(0, 3, 100)
plot(x, f.(x))
grid(true)
```



Functions defined by a block of code

The more general syntax is `function ... end`, which for the above example is

```
function f(x)
    2x^2 - 6x + 4
end;
```

```
a = 2
b = -6
vertex = -b/(2a)
println("The vertex is at $vertex, giving minimum value $(f(vertex))")
```

```
The vertex is at 1.5, giving minimum value -0.5
```

The keyword `return`

This is fine if the output value is always computed on the last line of code, but for more flexibility (and to my mind, better readability) there is the keyword `return`:

```
function f(x)
    return 2x^2 - 6x + 4
end;
```

This time the style is more "Pythonic", since the output variables are specifed in the `return` line rather than on the `function` line as Matlab does. This allows different calls of the function to return different type of value:

```
function squareroots(x)
    if x > 0
        return sqrt(x), -sqrt(x)
    elseif x == 0
        return 0
    else
        println("I can only handle real roots; sorry.")
    end
end;
```

```
squareroots(3.0)
```

```
(1.7320508075688772, -1.7320508075688772)
```

```
squareroots(0.0)
```

```
0
```

```
squareroots(-1.0)
```

```
I can only handle real roots; sorry.
```

9.2.15 Vectorization of Functions

One innovation in Julia is that the dot notation introduced above for *arithmetic on arrays* can also be used to vectorize a function; both ones provided by Julia and ones defined in your code.

A function `f(x)` whose code expects a number `x` or one `g(x, y)` that expects several numbers can be used on compatable arrays with

```
Y = f.(X)
Z = g.(X,Y)
```

The former is thus roughly a short-hand for

```
for i in length(X)
    Y[i] = f(X[i])
end
```

This is used quite a lot in the next section.

9.2.16 Plotting graphs: a bit more about PyPlot

There are many graphics packages for Julia; this book uses `PyPlot`. The name is because this is a front end to the Python package `matplotlib.pyplot` and in turn that name reflects the fact that Matplotlib mimics Matlab's plotting tools, so this choice makes it easiest for readers familiar with one of those languages.

To install PyPlot, see the notes on getting PyPlot in *Installing Julia and some useful add-ons*

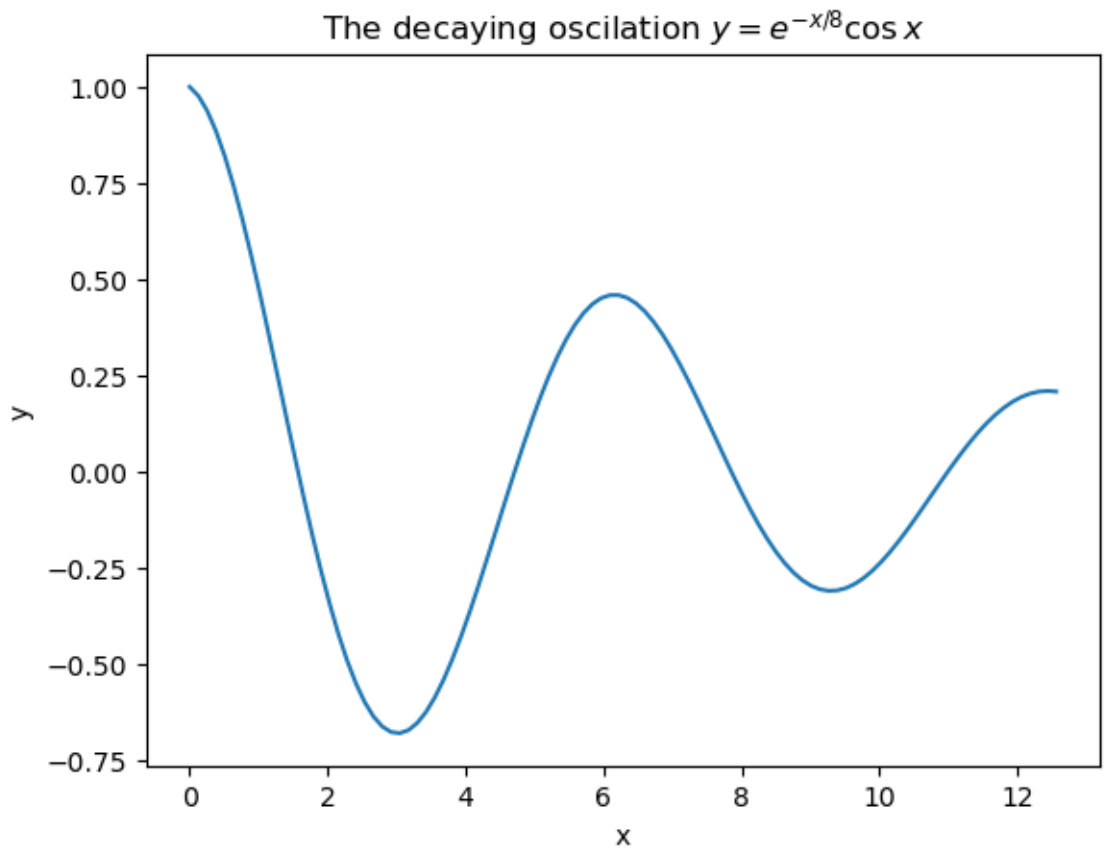
L-Strings for inserting LaTeX mathematical markup

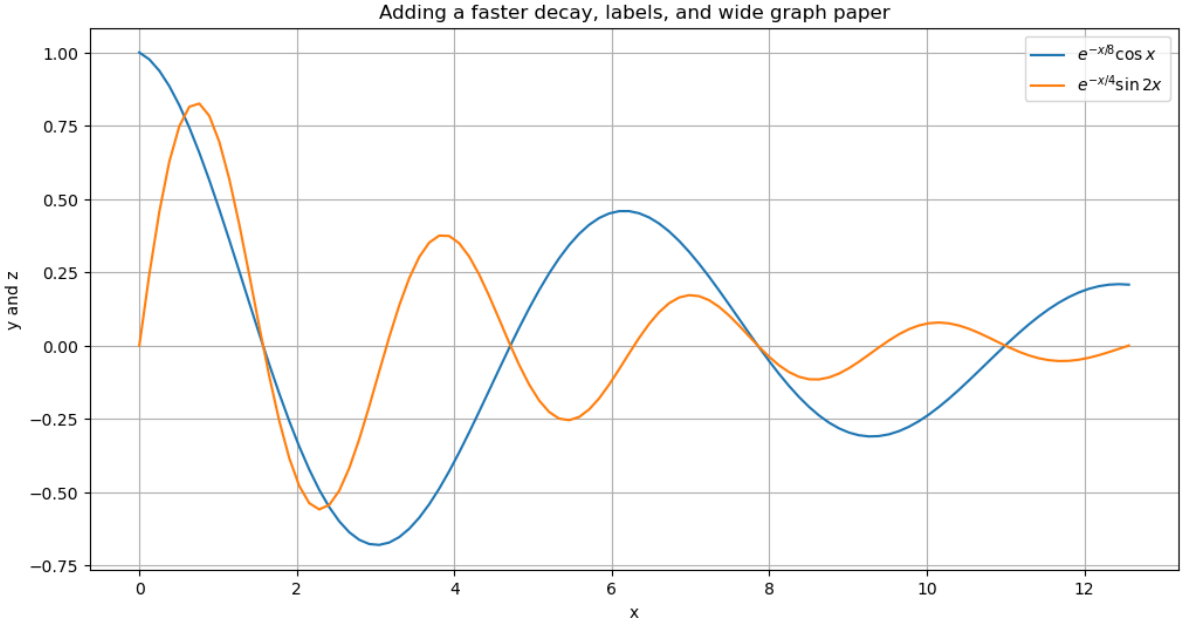
Mathematical formulas can be used in the annotations on graphs by the use of **L-strings**: string prefixed with 'L' and containing LaTeX mathematical markup between '$' signs.

For now I just illustrate all this with a few examples.

```
using PyPlot
```

```
x = range(0, 4pi, 100)
y = exp.(-x/8) .* cos.(x)

figure()
title(L"The decaying oscilation $y = e^{-x/8} \cos x$")
plot(x, y)
xlabel("x")
ylabel("y")

figure(figsize=[12,6])
z = exp.(-x/4) .* sin.(2x)
title("Adding a faster decay, labels, and wide graph paper")
plot(x, y, label=L"$e^{-x/8} \cos x$")
plot(x, z, label=L"$e^{-x/4} \sin 2x$")
xlabel("x")
ylabel("y and z")
legend()
grid(true)
```





To Be Continued …

9.3 Module `NumericalMethods`

Version of 2022-11-13

The following code cells are the content of file `NumericalMethods.jl`, used to define the module `NumericalMethods`

This file exists for two reasons:

1. It can be convenient to gather the cells defining various functions for the module in a notebook like this one, which allows documentation, and then convert to the the ".jl" file with the JupterLab command `File > Save and Export Notebook As ... > Executable Script` This gathers the contents of the code cells, ignorig any markdown cells.
2. This description of the module's definitions can be used as a section in a Jupyter Book.

Usage is:

```
include("NumericalMethods.jl")
```

then

```
using .NumericalMethods
```

or for a particular function, like

```
import .NumericalMethods: newtonmethod
```

```
# Module `NumericalMethods`

# Version of 2022-11-13

# The following code cells are the content of file `NumericalMethods.jl`, used to␣
↪define the module `NumericalMethods`

# The notebook file version exists for two reasons:
#
# 1. It can be convenient to gather the cells defining various functions for the␣
↪module in a notebook like this one,
#    which allows documentation, and then convert to the the ".jl" file with the␣
↪JupterLab command
#    `File > Save and Export Notebook As ... > Executable Script`
#
# This gathers the contents of the code cells, ignorig any markdown cells.

# 2. This description of the module's definitions can be used as a section in a␣
↪Jupyter Book.
```

```
module NumericalMethods
```

9.3.1 Root-finding

Newton's method

```
function newtonmethod(f, Df, x0, errortolerance; maxiterations=20, demomode=false)
    # Basic usage is:
    # (rootApproximation, errorEstimate, iterations) = newton(f, Df, x0,␣
↪errorTolerance)
    # There is an optional input parameter "demomode" which controls whether to
    # - println intermediate results (for "study" purposes), or to
    # - work silently (for "production" use).
    # The default is silence.

    if demomode
        println("Solving by Newton's Method.")
        println("maxiterations = $maxiterations")
        println("errortolerance = $errortolerance")
    end
    x = x0
    global errorestimate  # make it global to this function; without this it would be␣
↪local to the "for" loop.
    for iteration in 1:maxiterations
        fx = f(x)
        Dfx = Df(x)
        # Note: a careful, robust code would check for the possibility of division by␣
↪zero here,
        # but for now I just want a simple presentation of the basic mathematical␣
↪idea.
        dx = fx/Dfx
        x -= dx  # Aside: this is shorthand for "x = x - dx"
        errorestimate = abs(dx);
        if demomode
            println("At iteration $iteration, x = $x with estimated error
↪$errorestimate and backward error $(abs(f(x)))")
        end
        if errorestimate <= errortolerance
            if demomode
                println("Done!")
            end
            return (x, errorestimate, iteration)
        end
    end
    # Note: if we get to here (no "return" above), it completed maxIterations␣
↪iterations without satisfying the accuracy target,
    # but we still return the information that we have.
    return (x, errorestimate, maxiterations)
end;
```

The secant method

```
function secantmethod(f, a, b, errortolerance; maxiterations=20, demomode=false)
    # Solve f(x)=0 in the interval [a, b] by the Secant Method.
    if demomode
        print("Solving by the Secant Method.")
    end;
    # Some more descriptive names
    x_older = a
    x_more_recent = b
    f_x_older = f(x_older)
    f_x_more_recent = f(x_more_recent)
    for iteration in 1:maxiterations
        global x_new, errorestimate
        if demomode
            println("\nIteration $(iteration):")
        end;
        x_new = (x_older * f_x_more_recent - f_x_older * x_more_recent)/(f_x_more_
↪recent - f_x_older)
        f_x_new = f(x_new)
        (x_older, x_more_recent) = (x_more_recent, x_new)
        (f_x_older, f_x_more_recent) = (f_x_more_recent, f_x_new)
        errorestimate = abs(x_older - x_more_recent)
        if demomode
            println("The latest pair of approximations are $x_older and $x_more_
↪recent,")
            println("where the function's values are $f_x_older and $f_x_more_recent␣
↪respectively.")
            println("The new approximation is $x_new with estimated error
↪$errorestimate and backward error $(abs(f_x_new))")
        end;
        if errorestimate < errortolerance
            break
        end;
    end;
    # Whether we got here due to accuracy of running out of iterations,
    # return the information we have, including an error estimate:
    return (x_new,  errorestimate)
end;
```

9.3.2 Linear Algebra and Simultaneous Equations

Row Reduction

(with no pivoting)

```
function rowreduce(A, b)
    # To avoid modifying the matrix and vector specified as input,
    # they are copied to new arrays, with the function copy().
    # Warning: it does not work to say "U = A" and "c = b";
    # this makes these names synonyms, referring to the same stored data.

    U = copy(A)  # not "U=A", which makes U and A synonyms
    c = copy(b)
```

(continues on next page)

(continued from previous page)

```
    n = length(b)
    L = zeros(n, n)
    for k in 1:n-1
        for i in k+1:n
            # compute all the L values for column k:
            L[i,k] = U[i,k] / U[k,k]  # Beware the case where U[k,k] is 0
            for j in k+1:n
                U[i,j] -= L[i,k] * U[k,j]
            end
            # Put in the zeros below the main diagonal in column k of U;
            # this is not important for calculations, since those elements of U are␣
↪not used in backward substitution,
            # but it helps for displaying results and for checking the results via␣
↪residuals.
            U[i,k] = 0.

            c[i] -= L[i,k] * c[k]
        end
    end
    for i in 2:n
        for j in 1:i-1
            U[i,j] = 0.
        end
    end
    return (U, c)
end;
```

Backward substitution

```
function backwardsubstitution(U, c; demomode=false)
    n = length(c)
    x = zeros(n)
    x[end] = c[end]/U[end,end]
    if demomode
        println("x_$n = $(x[n])")
    end
    for i in n-1:-1:1
        if demomode
            println("i=$i")
        end
        x[i] = ( c[i] - sum(U[i,i+1:end] .* x[i+1:end]) ) / U[i,i]
        if demomode
            print("x_$i = $(x[i])")
        end
    end
    return x
end;
```

Solve a linear system (no pivoting)

```
solvelinearsystem(A, b) = backwardsubstitution(rowreduce(A, b)...);
```

LU factorization

```
function lu_factorize(A; demomode=false)
    # Compute the Doolittle LU factorization of A.
    # Sums like $\sum_{s=1}^{k-1} l_{k,s} u_{s,j}$ are done as matrix products;
    # in the above case, row matrix L[k, 1:k-1] by column matrix U[1:k-1,j] gives the sum for a give j,
    # and row matrix L[k, 1:k-1] by matrix U[1:k-1,k:n] gives the relevant row vector.
    n = size(A)[1]  # First component of the array's size; size(A) returns "(rows, columns)"
    # Initialize U as a zero matrix;
    # correct below the main diagonal, with the other entries to be computed and filled below.
    U = zeros(n,n)
    # Initialize L as a zero matrix;
    # correct above the main diagonal, with the other entries to be computed and filled in below.
    L = zeros(n,n)
    # Column and row 1 are special:
    U[1,:] = A[1,:]
    L[1,1] = 1.
    L[2:end,1] = A[2:end,1]/U[1,1]
    if demomode
        println("After step k=1")
        println("U="); printmatrix(U)
        println("L="); printmatrix(L)
    end;
    for k in 2:n-1
        # Julia note: it is necessary to use indices "[k]" and so on to get a one-row matrix instead of a vector.
        U[[k],k:end] = A[[k],k:end] - L[[k],1:k] * U[1:k,k:end]
        L[k,k] = 1.
        L[k+1:end,k] = ( A[k+1:end,k] - L[k+1:end,1:k] * U[1:k,k] )/U[k,k]
        if demomode
            println("After step k=$k")
            println("U="); printmatrix(U)
            println("L="); printmatrix(L)
        end;
    end;
    # The last row is also special: nothing to do for L
    L[end,end] = 1.
    U[end,end] = A[end,end] - sum(L[[n],1:end-1] * U[1:end-1,end])
    if demomode
        println("After step k=$n")
        println("U="); printmatrix(U)
    end;
    return [L, U]
end;
```

Forward substitution

(without pivoting)

```
function forwardsubstitution(L, b)
    # Solve L c = b for c.
    n = length(b)
    c = zeros(n)
    c[1] = b[1]
    for i in 2:n
        c[i] = b[i] - sum(L[i:i,1:i] * c[1:i])
    end;
    return c
end;
```

PLU factorization

```
function plu(A; demomode=false)
    # Compute the Doolittle PA=LU factorization of A —
    # but with the permutation recorded as permutation vector, not as the permutation␣
↪matrix P.
    # Sums like $\sum_{s=1}^{k-1} l_{k,s} u_{s,j}$ are done as matrix products;
    # in the above case, row matrix L[k, 1:k-1] by column matrix U[1:k-1,j] gives the␣
↪sum for a give j,
    # and row matrix L[k, 1:k-1] by matrix U[1:k-1,k:n] gives the relevant row vector.

    n = size(A)[1]  # gives the number of rows in the 2D array.
    π = zeros(Int, n)
    # Julia can use Greek letters (and in fact, UNICODE):
    # to insert character π, type \pi, hit tab, and select "π" from the menu.
    # Or just call it "perm" or such.
    π = collect(1:n)
    # Julia language note: function "collect" converts the abstract entity "1:n" into␣
↪an array of numbers.

    # Initialize U as the zero matrix;
    # correct below the main diagonal, with the other entries to be computed below.
    U = zeros(n,n)

    # Initialize L as zeros;
    # correct above the main diagonal, with the other entries to be computed below,
    # including the ones on the diagonal.
    L = zeros(n,n)

    for k in 1:n-1
        if demomode; println("k=$k"); end
        # Find the pivot element in column k:
        pivotrow = k
        abs_u_ik_max = abs(A[π[k],k])
        for row in k+1:n
            abs_u_ik = abs(A[π[row],k])
            if abs_u_ik > abs_u_ik_max
                pivotrow = row
                abs_u_ik_max = abs_u_ik
            end
```

(continues on next page)

(continued from previous page)

```
        end
        if pivotrow > k # swap rows, virtually
            if demomode; println("Swap row $k with row $pivotrow"); end
            (π[k], π[pivotrow]) = (π[pivotrow], π[k])
        else
            if demomode; println("No row swap needed."); end
        end
        U[k,k:end] = A[[π[k]],k:end] - L[[π[k]],1:k] * U[1:k,k:end]
        L[π[k],k] = 1.
        for row in k+1:n
            L[π[row],k] = ( A[π[row],k] - L[π[row],1:k] ⋅ U[1:k,k] ) / U[k,k]
            # Julia note: To enter the centered dot notation for the dot product,␣
↪type "\cdot" and then hit the tab key.
        end
        if demomode
            println("permuted A is:")
            for row in 1:n
                println(A[π[row],:])
            end
            println("Intermediate L is"); printmatrix(L)
            println("Intermediate U is"); printmatrix(U)
        end
    end
    # The last row (index "end") is special: nothing to do for L except put in the 1␣
↪on the "permuted main diagonal"
    L[π[end],end] = 1.
    U[end,end] = A[π[end],end] - L[π[end],1:end-1] ⋅ U[1:end-1,end]
    if demomode
        println("After the final step, k=$(n-1)")
        println("L is"); printmatrix(L)
        println("U is"); printmatrix(U)
    end
    return (L, U, π)
end;
```

Forward substitution with pivoting

```
function forwardsubstitution(L, b, π)
    # Version 2: with permutation of rows
    # Solve L c = b for c, with permutation of the rows of L and of b.
    n = length(b)
    c = zeros(n)
    c[1] = b[π[1]]
    for i in 2:n
        c[i] = b[π[i]] - L[π[i], 1:i] ⋅ c[1:i]
    end
    return c
end;
```

9.3.3 Collocation and Data Fitting

Polynomial collocation

```
function polyfit(x, y)
    # Version 1: exact collocation.
    # Compute the coeffients c_i of the polynomial of lowest degree that collocates␣
 ↪the points (x[i], y[i]).
    # These are returned in an array c of the same length as x and y, even if the␣
 ↪degree is less than the normal length(x)-1,
    # in which case the array has some trailing zeroes.
    # The polynomial is thus p(x) = c[1] + c[2]x + ... c[d+1] x^d where d =length(x)-
 ↪1, the nominal degree.
    n_nodes = length(x)
    degree = n_nodes - 1
    V = zeros(n_nodes,n_nodes)
    for i in 0:degree
        for j in 0:degree
             V[i+1,j+1] = x[i+1]^j  # Shift the array indices up by one, since Julia␣
 ↪counts from 1, not 0.
        end
    end
    c = solvelinearsystem(V, y)
    return c
end;
```

Least squares polynomial approximation

```
function polyfit(x, y, n)
    # Version 2: least squares fitting.
    # Compute the coeffients c_i of the polynomial of degree n that give the best␣
 ↪least squares fit to data (x[i], y[i]).
    N = length(x)
    m = zeros(2n+1)
    for k in 0:2n
        m[k+1] = sum(x.^k)  # Here and below, shift the indices up by one, since␣
 ↪Julia counts from 1, not 0.
    end
    M = zeros(n+1,n+1)
    for i in 0:n
        for j in 0:n
             M[i+1, j+1] = m[i+j+1]
        end
    end
    p = zeros(n+1)
    for k in 0:n
        p[k+1] = sum(x.^k .* y)
    end
    c = solvelinearsystem(M, p)
    return c
end;
```

Evaluate a polynomial

```
function polyval(x; coeffs)  # coeffs has to be a keyword argument in order that only x gets vectorized
    # Evaluate the polynomial with coefficients in c (as given by polyfit, for example).
    # If x is an array, the usage becomes y = polyval.(c, x)
    # for each element of array x.
    y = coeffs[1]
    for i in 2:length(coeffs)
        y += coeffs[i]*x^(i-1)
    end
    return y
end;
```

9.3.4 Derivatives and Definite Integrals

9.3.5 Minimization

9.3.6 Differential Equations

Euler's method

```
function eulermethod(f, a, b, u_0; n=100)
    # Solve du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    h = (b-a)/n
    t = range(a, b, n+1)  # Note: "n" counts steps, so there are n+1 values for t.
    u = zeros(n+1)
    u[1] = u_0
    for i in 1:n
        u[i+1] = u[i] + f(t[i], u[i])*h
    end
    return (t, u)
end;
```

```
function eulermethod_errorcontrol(f, a, b, u_0; errortolerance=1e-3, h_min=1e-6, h_max=0.1, steps_max=1000, demomode=false)
    steps = 0
    t_i = a
    U_i = u_0
    t = [t_i]
    U = [U_i]
    h = h_max  # Start optimistically!
    while t_i < b && steps < steps_max
        K_1 = h*f(t_i, U_i)
        K_2 = h*f(t_i + h/2, U_i + K_1/2)
        errorestimate = abs(K_1 - K_2)
        s = 0.9 * sqrt(errortolerance/errorestimate)
        if errorestimate <= errortolerance  # Success!
            t_i += h
            U_i += K_1
            append!(t, t_i)
```

(continues on next page)

(continued from previous page)

```
            append!(U, U_i)
            # Adjust step size up, but not too big
            h = min(s*h, h_max)
        else  # Innacurate; reduce step size and try again
            h = max(s*h, h_min)
            if demomode
                println("t_i=$t_i: Decreasing step size to $(about(h)) and trying␣
↪again.")
            end
        end
        # A refinement not mentioned above; the next step should not overshoot t=b:
        if t_i + h > b
            h = b - t_i
        end
        steps += 1
    end
    return (t, U)
    # Note: if the step count ran out, this does not reach t=b, but at least it is␣
↪correct as far as it goes
end;
```

```
function eulermethod_system(f, a, b, u_0, n)
    # TO DO: one could use multiple dispatch to keep the name "eulermethod".
    # The conversion for the system version is mainly "U[i] -> U[i,:]"

    h = (b-a)/n
    t = range(a, b, n+1)

    # The following three lines and the one in the for loop below change for the␣
↪system version
    n_unknowns = length(u_0)
    U = zeros(n+1, n_unknowns)
    U[1,:] = u_0  # Only for system version

    for i in 1:n
        U[i+1,:] = U[i,:] + f(t[i], U[i,:])*h  # For the system version
    end
    return (t, U)
end;
```

The explicit trapezoid method

```
function explicittrapezoid(f, a, b, u_0; n=100, demomode=false)
    # Use the Explict Trapezoid Method (a.k.a Improved Euler) to solve
    #     du/dt = f(t, u)
    # for t in [a, b], with initial value u(a) = u_0

    h = (b-a)/n
    t = range(a, b, n+1)  # Note: "n" counts steps, so there are n+1 values for t.
    u = zeros(n+1)
    u[1] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i])*h
```

(continues on next page)

(continued from previous page)

```
        K_2 = f(t[i]+h, u[i]+K_1)*h
        u[i+1] = u[i] + (K_1 + K_2)/2.0
    end;
    return (t, u)
end;
```

```
function explicittrapezoid_system(f, a, b, u_0, n)
    # Use the Explict Trapezoid Method (a.k.a Improved Euler) to solve the system
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    # The conversion for the system version is mainly "u[i] -> u[i,:]"

    h = (b-a)/n
    t = range(a, b, n+1)
    n_unknowns = length(u_0)
    u = zeros(n+1, n_unknowns)
    u[1,:] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i,:])*h
        K_2 = f(t[i]+h, u[i,:]+K_1)*h
        u[i+1,:] = u[i,:] + (K_1 + K_2)/2.0
    end
    return (t, u)
end;
```

The explicit midpoint method

```
function explicitmidpoint(f, a, b, u_0; n=100, demomode=false)
    # Use the Explicit Midpoint Method (a.k.a Modified Euler) to solve
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0

    h = (b-a)/n
    t = range(a, b, n+1)  # Note: "n" counts steps, so there are n+1 values for t.
    u = zeros(length(t))
    u[1] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i])*h
        K_2 = f(t[i]+h/2, u[i]+K_1/2)*h
        u[i+1] = u[i] + K_2
    end;
    return (t, u)
    end;
```

```
function explicitmidpoint_system(f, a, b, u_0, n)
    # Use the Explict Midpoint Method (a.k.a Modified Euler) to solve the system
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    # The conversion for the system version is mainly "u[i] -> u[i,:]"

    h = (b-a)/n
    t = range(a, b, n+1)
    n_unknowns = length(u_0)
    u = zeros(n+1, n_unknowns)
    u[1,:] = u_0
```

(continues on next page)

(continued from previous page)

```
    for i in 1:n
        K_1 = f(t[i], u[i,:])*h
        K_2 = f(t[i]+h/2, u[i,:]+K_1/2)*h
        u[i+1,:] = u[i,:] + K_2
    end
    return (t, u)
end;
```

The Runge-Kutta method

```
function rungekutta(f, a, b, u_0; n=100, demomode=false)
    # Use the (classical) Runge-Kutta Method to solve
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    h = (b-a)/n
    t = range(a, b, n+1)  # Note: "n" counts steps, so there are n+1 values for t.
    u = zeros(length(t))
    u[1] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i])*h
        K_2 = f(t[i]+h/2, u[i]+K_1/2)*h
        K_3 = f(t[i]+h/2, u[i]+K_2/2)*h
        K_4 = f(t[i]+h, u[i]+K_3)*h
        u[i+1] = u[i] + (K_1 + 2*K_2 + 2*K_3 + K_4)/6
    end;
    return (t, u)
end;
```

```
function rungekutta_system(f, a, b, u_0, n)
    # Use the (classical) Runge-Kutta Method to solve
    #    du/dt = f(t, u) for t in [a, b], with initial value u(a) = u_0
    # The conversion for the system version is mainly "u[i] -> u[i,:]"

    h = (b-a)/n
    t = range(a, b, n+1)
    n_unknowns = length(u_0)
    u = zeros(n+1, n_unknowns)
    u[1,:] = u_0
    for i in 1:n
        K_1 = f(t[i], u[i,:])*h
        K_2 = f(t[i]+h/2, u[i,:]+K_1/2)*h
        K_3 = f(t[i]+h/2, u[i,:]+K_2/2)*h
        K_4 = f(t[i]+h, u[i,:]+K_3)*h
        u[i+1,:] = u[i,:] + (K_1 + 2*K_2 + 2*K_3 + K_4)/6
    end
    return (t, u)
end;
```

9.3.7 Some auxilliary functions

For examples, presentation of results, etc.

Helper function `printmatrix`

```
function printmatrix(A)
    # A helper function to "pretty print" matrices
    (rows, cols) = size(A)
    print("[ ")
    for row in 1:rows
        if row > 1
            print("  ")
        end
        for col in 1:cols
            print(A[row,col], " ")
        end
        if row < rows;
            println()
        else
            println("]")
        end
    end
end;
```

```
# A helper function for rounding some output to three significant digits
approx3(x) = round(x, sigdigits=3);
```

```
# A helper function for rounding some output to four significant digits
approx4(x) = round(x, sigdigits=4);
```

9.3.8 For some examples in Chapter Initial Value Problems for Ordinary Differential Equations

```
f_mass_spring(t, u) = [ u[2], -(K/M)*u[1] - (D/M)*u[2] ];

function y_mass_spring(t; t_0, u_0, K, M, D)
    (y_0, v_0) = u_0
    discriminant = D^2 - 4K*M
    if discriminant < 0  # underdamped
        omega = sqrt(4K*M - D^2)/(2M)
        A = y_0
        B = (v_0 + y_0*D/(2M))/omega
        return exp(-D/(2M)*(t-t_0)) * ( A*cos(omega*(t-t_0)) + B*sin(omega*(t-t_0)))
    elseif discriminant > 0  # overdamped
        Delta = sqrt(discriminant)
        lambda_plus = (-D + Delta)/(2M)
        lambda_minus = (-D - Delta)/(2M)
        A = M*(v_0 - lambda_minus * y_0)/Delta
        B = y_0 - A
```

(continues on next page)

(continued from previous page)

```
        return A*exp(lambda_plus*(t-t_0)) + B*exp(lambda_minus*(t-t_0))
    else
        lambda = -D/(2M)
        A = y_0
        B = v_0 - A * lambda
        return (A + B*t)*exp(lambda*(t-t_0))
    end
end;

function damping(K, M, D)
    if D == 0
        println("Undamped")
    else
        discriminant = D^2 - 4K*M
        if discriminant < 0
            println("Underdamped")
        elseif discriminant > 0
            println("Overdamped")
        else
            println("Critically damped")
        end
    end
end;
```

```
end;
```

BIBLIOGRAPHY

[BFB16] Richard L. Burden, J. Douglas Faires, and Annette M. Burden. *Numerical Analysis*. Cengage, 10th edition, 2016.

[CK12] Ward Chenney and David Kincaid. *Numerical Mathematics and Computing*. Cengage, 7 edition, 2012.

[KC90] David Kincaid and Ward Chenney. *Numerical Analysis*. Brooks/Cole, 1990.

[Sau19] Timothy Sauer. *Numerical Analysis*. Pearson, 3rd edition, 2019.

PROOF INDEX

remark-LU-with-P

remark-dolittle

remark-importing-backwardsubstitution

remark-julia-arrays

remark-julia-style

remark-multiple-dispatch-polyfit

remark-other-matrix-norms

remark-positive-definite-also-works

remark-positive-definite-matrices-also-work

remark-vector-derivative notation

richardson-forward-differences

richardson0n-to-kn

rkf

robust

romberg-integration

runge-kutta

secant-method

separatrices

stiffness

super-linear

swapping-values-in-julia

taylors-theorem-a

taylors-theorem-h

theorem-1

theorem-Crout-SDD

theorem-LU-SDD

theorem-collocation

theorem-gaus-seidel-convergence

theorem-jacobi-convergence

theorem-loss-of-precision

theorem-matrix-iteration-convergence

theorem-no-pivoting-columnwise-sdd

theorem-row-reduction-preserves-sdd

to-do-Installing

trapezoid-rule-error

trapezoid-step-size

triangular-matrix

uniformly-contracting

vector-valued-contraction-mapping-theorem

vectorization-broadcasting-in-julia

well-posed

zip-in-julia