Lecture Notes, Math 522, Spring 2019 Edition

Zachariah B. Etienne

February 11, 2020

The following is a set of lecture notes I compiled, wrote, and used over the course of the Spring 2016–2019 semesters of Math 522. All source material, including freely-available online lecture notes and encyclopedia articles, were very carefully chosen for strength in continuing the lecture narrative. Sources are cited, though the narrative is mine. I worked hard to distill and rewrite the material into small, easily-digested pieces, suitable for a 500-level Math class. It is my goal over the coming years to continue to improve and evolve the current material into a work that, in combination with original homework sets and exam problems already devised, will form the foundation of a textbook on Numerical Solutions of PDEs.

Table of Contents

0.1 Preface: Preliminaries

To do well in this class, it is expected that in addition to good programming skills, you are also competent in basic high school and undergraduate mathematics, including

- Scientific notation
- Significant digits/figures; can you perform arithmetic keeping a fixed number of significant digits? It is critically important to know this, as it is fundamental to understanding how computers do floating point arithmetic. Understanding the limitations of this arithmetic is a central skill in this class.
- Basic algebra: Solving nonlinear equations & inequalities involving Logarithms, Exponentials, and Quadratics.
- Calculus I: Differentiation using Chain Rule, Product Rule. Basic exposure to Taylor Series, series convergence, radius of convergence.
- Calculus II: Integration by variable substitution, Integration by parts
- Linear Algebra: Computing determinants, Properties of determinants, Computing Eigenvalues/Eigenvectors
- Calculus IV/Ordinary Differential Equations (ODEs): Basic solution strategies: method of separation, etc. Exposure to Even/Odd functions, Fourier series, etc.

In terms of coding abilities, on coding assignments you will be expected to design, write, and debug codes that are roughly 200 lines long. It is important that you start on coding assignments immediately after receiving the homework, or bugs may prevent you from completing the assignment on time.

If you consider yourself a bit "rusty" on any of these or any other topics covered in this course, I have prepared a "Suggested/Additional Online Reading List", which is linked to on the Course home page.

Now hand out the Skills Assessment. Give students 20 minutes to complete.

MATH 522, Spring 2019 Notes

Chapter 1: Introduction: Benefits and Pitfalls to Solving Partial Differential Equations (PDEs) on the Computer

Computers are remarkable machines, capable of performing *billions* (i.e., of order 10^9) mathematical operations each second. So when we are faced with a mathematical problem — like finding the solution to a nonlinear partial differential equation — a computer is often the only means to solve it.

The central focus of this class is to provide an introduction to the basic strategies used to solve partial differential equations (PDEs) on the computer. We call these strategies **algorithms**. It is assumed that you already know how to program computers, and this skill will be necessary to "code up" (i.e., program a computer to perform) these algorithms, which will be a major component to homework assignments. If you cannot understand how an algorithm works, you will quickly find yourself confused when the computer outputs the wrong result.

There is no "perfect algorithm" for solving mathematical problems requiring the solution of partial differential equations. Instead, there are a multitude of algorithms for solving any general problem. Even an algorithm designed for a particular problem will have shortcomings. It is therefore *essential* that we understand how these algorithms work, which requires **a strong foundation in undergraduate mathematics**, including algebra, trigonometry, scientific notation, matrix algebra, differentiation, solutions of ordinary differential equations, Taylor Series, Fourier Series, and integration. The strongest indicator of your final grade in this class will not be your ability to program computers (though this is a critical skill), but instead your proficiency in undergraduate mathematics.

Since computer algorithms we program can and will give the incorrect solution, someone who is skilled in solving mathematical problems on the computer must be *equally* skilled in understanding sources of error.

Continued on next page

1.1 The Four Horsemen of the Error-pocalypse

We classify errors into four categories when solving mathematical problems on the computer.

1. *Roundoff Error*: Computers have limited facilities for storing numbers, so to maximize efficiency, we usually represent numbers *approximately* in scientific notation with a *fixed number* of significant digits. As we will see in Chapter 8, error caused by storing only a fixed number of significant digits—i.e., **roundoff error**—can poison our numerical solution. We must therefore be able to determine how and where our algorithms lose precision.
2. *Undersampling Error / Insufficient Resources*: When solving mathematical problems on the computer, it is often essential that we first understand the *scale* of the solution—usually by computing a "back of the envelope" or "order of magnitude" estimate of the solution (Chapter 2). Without such an estimate, the algorithms we use may **undersample** the solution and yield the wrong answer, or we may find only after wasting a large amount of time writing a computer program that the required computing power to solve a mathematical problem far exceeds our available resources.
   Indeed, modern computers are wonders of technology, but for certain problems, they are simply not powerful enough. To determine the amount of computational resources needed to solve a problem on the computer, we can combine a back-of-the-envelope estimate of the solution with the computational *complexity* (**cost**) of the underlying algorithm itself (Chapter 15). If the needed resources exceeds our budget, we may need to choose or develop more efficient algorithms, optimize existing algorithms, or move to more powerful computational resources.
3. *Truncation / Approximation Errors*: Understanding the limitations and approximations made by our algorithms is critically important. Sometimes for example, our solution might depend on an infinite Taylor series. We cannot typically compute or store an infinite number of terms in this series, so we instead sum the series up to a finite term, ignoring all remaining terms. The error imposed by ignoring remaining terms in this way is called **truncation error**.
4. *Human Error*: Suppose we solve a mathematical problem on the computer based on some combination of algorithms taught in this class, and this problem is impossible to solve without a computer. How can we be confident that our solution is correct? To this end, we must always work to hone our skills in *code validation*—the process by which we eliminate **human errors** both in choice of algorithms and bug-free implementation of the algorithms—and gain confidence that the algorithms we have coded both work correctly and yield reliable solutions. To this end, every coding assignment will contain at least some component of code validation.

MATH 522, Spring 2019 Notes

Chapter 2: Determining the Scale of the Solution

We will find that many of the algorithms we encounter in this class *will not work properly* unless we first have some very rough idea (like, within a factor of a few) of the solution. Having some idea of the scale of the solution will also help us determine whether the computer's result is reasonable.

To get some sense of the scale of the solution without knowing the solution requires that we use all the information at our disposal to generate an estimate that is reasonably close. This is known as a "back-of-the-envelope" estimate or a "Fermi" problem, after Enrico Fermi—a very famous, Nobel-prize winning physicist—as he was a master at this sort of back-of-the-envelope calculations. Fermi worked on the Manhattan Project–the project that developed the first atomic bombs. During the first nuclear test, Fermi dropped pieces of paper from his viewing position to estimate the energy in the blast. His measurement came quite close to the official measurement, which made use of many sensors.

Being able to reason your way through back-of-the-envelope estimates make excellent job interview question for those of you who want to get really high-paying jobs in the financial or tech sectors. These sorts of question could be asked cold in an interview, meaning you have no access to books... just a whiteboard and your own numerical literacy. As you solve the problem in front of the interviewers, they will learn how you solve problems, as well as your basic numerical literacy.

2.1 Word Problems

The easiest Fermi problems boil down to basic word problems that require simple arithmetic to solve. Note that a typical Fermi problem will require basic numerical literacy (e.g., knowing the population of the United States), working knowledge of undergraduate mathematics (e.g., being able to integrate, differentiate, and compute Taylor expansions without help), and good analytical reasoning skills.

As a warm-up, let's suppose you are a graduate student who wishes to increase efficiency and productivity.

Example #1, Question #1

Graduate students are generally expected to work as a GTA or GRA 20 hours per week, spending another 20 hours on course work. This is of course the ideal scenario, assuming that there is not an urgent research project, midterm to grade, or exam to take. It also assumes that students are well prepared for their graduate level classes so that the amount of "brushing up" on previous material is minimized. If any of these assumptions are violated, the 40-hour week may become a 60-hour week or in some cases a 20 or 30-hour week.

Students on average take 3 courses per semester. Assume that each class is 150 minutes per week. If they are on average expected to spend 20 hours per week on classes, how many hours outside of class *per class* in an ideal situation are they expected to study?

Answer: First we compute the number of hours *in total* outside of class that should be spent studying. Notice we must first convert to a consistent unit of time (hours) so we can perform the necessary arithmetic:

```
20 hours/week - 150 minutes/course/week * (1 hour/(60 minutes)) * (3 courses)
   = (20 - 7.5) hours/week = 12.5 hours/week
```

Thus students are expected to spend 12.5 hours per week on their coursework outside of class, or about 12.5/3 hours per course, which comes to about 4 hours, 10 minutes studying outside of class for each class. Again, this is an ideal situation, as a student's level of preparation may be different for different classes, requiring additional time commitment for some classes.

Example #1, Question #2

Distracted Dan is not very focused on studying, keeping his cellphone out or the TV on. Each hour he studies is worth only 60% of the expected study time per week. Roughly how many hours would he need to study for each of his classes, if he is an otherwise average student? Give the answer as a simplified fraction and an approximate (i.e., within 10% of exact) decimal.

Answer: We determined in the previous problem that Dan must spend about 12.5 hours studying outside of class each week. Thus he must spend 12.5/3 hours per course per week, which amounts to a little over 4 hours per course. `12.5/3 = 25/6`, so the amount of time needed each week for studying outside of class per class for Distracted Dan will be `25/6 * 10/6 = 250/36 = 125/18 = (125*1.1)/(18*1.1)≈140/20 = 70/10 = 7` hours. The exact value is about 6h 57m.

Example #1, Question #3

Based on Question #2 above, about how many hours per week would Distracted Dan be expected to study on average outside of class to keep up with all of his classes?

Answer: Distracted Dan takes three courses each semester, and we just determined that the total study time outside of class required for Distracted Dan per course is nearly 7 hours, so the answer is that Distracted Dan must study about `3*7 = 21` hours outside of class to keep up with the average student.

Add this to the 7.5 hours spent inside of class, and Distracted Dan's 20 hour per week course load has ballooned to nearly 30 hours per week! The moral of the story is that students should not only work hard, but strive to work *efficiently* as well. Try to minimize distractions!

When we apply the concepts learned in this class to the real world, we will find that this sort of thinking is extremely valuable to determining the scale of physical systems as well.

2.2 Undersampling Error

Performing integrals, derivatives, interpolation, and extrapolation of functions on the computer is central to much of the work we do as when we solve mathematical problems on the computer. Unfortunately, performing these operations reliably on the computer usually requires that we first determine the basic scales over which the function we wish to integrate, differentiate, etc., varies.

For example, when evaluating a derivative approximately on the computer, we start from the definition of the derivative

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

to say that, *for Δx small enough*,

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}.$$

Approximations like this are made widely when solving mathematical problems on the computer, because

1. it is difficult to teach computers how to differentiate exactly;
2. sometimes we are only given the function values at a finite number of points; and
3. computers are designed to subtract, add, multiply, and divide extremely quickly.

The question we must answer is, *how small of a Δx will be necessary to obtain a reasonable approximation of the derivative?*. Or equivalently, *how finely must we sample the function to get a good approximation of its derivative at some arbitrary point x?*

The answer of course depends on the function. If x is in units of length for example, then we must first estimate the *lengthscale* over which $f(x)$ varies. If we chose to approximate the derivative of $f(x) = \sin(x)$ at any point x using the definition of derivative in this way, hopefully you would agree that if we chose $\Delta x = 10$, we should not expect the answer to be anywhere close to the correct result.[1] We refer to this type of error as **undersampling error**.

[1] After all, if x is related to units of length, then $\sin(x)$ varies over a lengthscale of $2\pi \approx 6.3$–its *wavelength*. If instead x is related to units of time, then we would say $\sin(x)$ varies over a timescale of $2\pi \approx 6.3$–its *period*.

Let's make this notion of undersampling just a little more rigorous by defining the minimal, or "Nyquist", sampling rate.

2.2.1 Minimal, or "Nyquist" Sampling

Suppose we have a function that is a wave with wavelength $\lambda = 1$ meter. We refer to λ as the *lengthscale* of the problem, and if we wish to interpolate the value of the function to any point, we will need to sample the function at a resolution of Δx that is some small fraction of λ.

In fact, the minimum sampling rate for a wave, called the *Nyquist sampling* frequency or rate, is $\Delta x = \lambda/2$. Figure 2.1 shows this graphically. This idea can be applied more generally, even to *non-periodic* functions, by simply replacing λ with *the scale over which the function varies*. The next section explores this more generic notion by analyzing the scale over which a solution will vary and applying it to estimate computational cost.

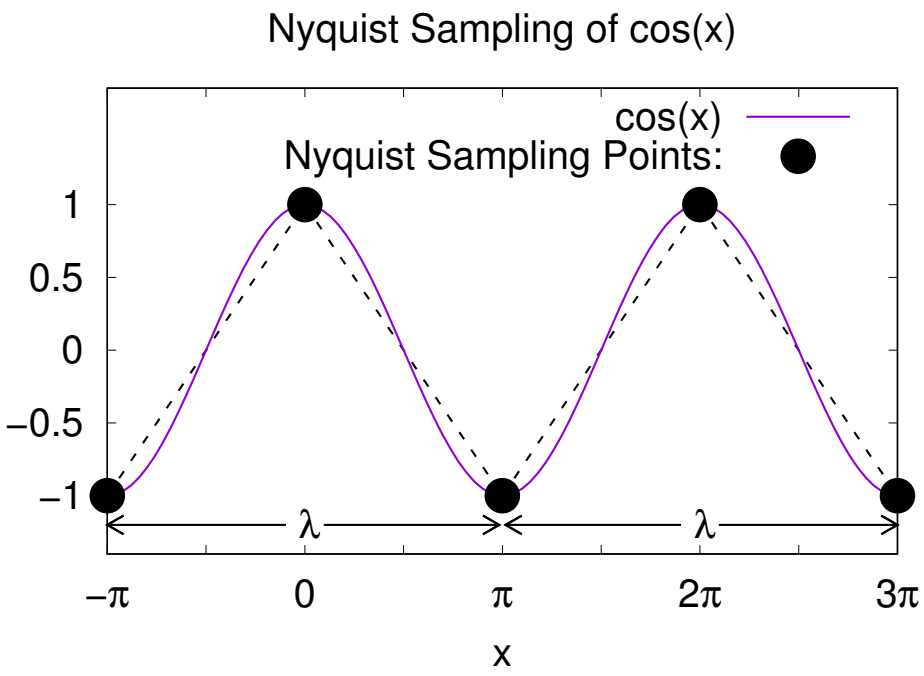

Figure 2.1: Minimal (Nyquist) sampling of a cosine function (solid curve). Nyquist-sampled points, at $\Delta x = \lambda/2$, are shown as black dots, which are connected by dashed lines.

2.3 Tying It All Together: Estimating the Cost of Weather Modeling

Example #2

Suppose we wish to produce a climate model that simulates cloud motion. Let's assume that the tiniest of clouds are negligible, so that our model is aiming to model large cloud movement. Provide an estimate for the minimum number of numerical points that we would need to Nyquist sample all the clouds in the Earth's atmosphere? Ignore the vertical direction.

One way of solving this problem is to estimate the number of distinct clouds we observe in the sky on average. Not every day we see clouds, so let's say 2 of 3 days are cloudy days, and when we consider that Morgantown is a bit cloudier than the average city (or desert for that matter), let's drop that to 1 of 2 days. Then on cloudy days, the average cloudiness is about 30%. So the average fraction of the local sky taken up by clouds is about 30% times $1/2 = 15\%$.

Now this is at our local position on Earth, so we need to know the fraction of the entire Earth's atmosphere we can view at a given time.

The Earth is roughly 6,000 km in radius (actual radius is around 6,300 km).

How do we figure out the fraction of the sky that we see? Well, this comes down to measuring our horizon. Our horizon is defined as the plane tangent to our location on the Earth. We obviously cannot see below the horizon (though there are some exceptions due to the way the atmosphere bends light passing through it). We can see clouds so long as they are above this tangent plane.

Suppose clouds are at 15,000 feet altitude on average. There are about 3 feet in a meter, so we have 5,000m or 5 km for the average altitude for clouds. So we get a right triangle: the line from us to the center of the Earth is about 6,000 km, and drawing another line from the center of the Earth to the cloud height on our horizon. Connecting these lines yields a right triangle, in which we stand at the right angle, with hypotenuse 6,005 km. Our goal is to find the subtended angle of our horizon from the center of the Earth.

What is this angle? Basic right-triangle geometry tells us that the angle θ satisfies

$$\cos(\theta) = \frac{6000}{6005}.$$

Of course this is very close to 1, which means that the angle must be very small. So let's Taylor expand $\cos(\theta)$ about $\theta = 0$:

$$\cos(\theta) = 1 - \frac{\theta^2}{2} + \frac{\theta^4}{4!} - ...$$

Since θ^2 is small, θ^4 is extremely small. so we can get a very good approximation of the angle by simply solving

$$1 - \frac{\theta^2}{2} = \frac{6000}{6005} \approx 1 - \frac{1}{1000}.$$

Thus the total horizon for us if we can see clouds at 15,000 feet is

$$\theta^2 \approx \frac{2}{1000} \implies \theta \approx \sqrt{\frac{2}{1000}} = \frac{1}{\sqrt{500}}.$$

What is the square root of 500? The square root of 100 is 10, so

$$\sqrt{100} = 10 \implies 2\sqrt{100} = 20 \implies \sqrt{400} = 20.$$

Also,

$$\sqrt{100} = 10 \implies 3\sqrt{100} = 30 \implies \sqrt{900} = 30.$$

So we're somewhere between 20 and 30, but closer to 20. Let's just keep 20 for our approximation.

Then the angle subtended by our horizon when looking in one direction is about 1/20 of a radian. Our goal then is to determine the area of the sky we can see as a total fraction of the overall surface area of the Earth.

Recall that the surface area of a sphere with radius R is derived from the integration formula

$$S_{\text{sphere}} = R^2 \int_0^{\pi} \int_0^{2\pi} \sin(\theta) d\theta = 4\pi R^2.$$

We are interested however in the surface area of some circular-like area subtended by an angle $\Delta\theta$ on the sphere, so

$$S_{\text{circ area}} = R^2 \int_0^{\Delta\theta} \int_0^{2\pi} \sin(\theta) d\theta = 2\pi[-\cos(\theta)]_0^{\Delta\theta} R^2.$$

We can use this formula, combined with the $\Delta\theta = 1/20$ to compute the surface area of the Earth's atmosphere over which clouds are visible to us:

$$\begin{aligned} S_{\text{area of our horizon}} &= 2\pi R^2[-\cos(\theta)]_0^{1/20} = 2\pi R^2 \left(-\cos\left(\frac{1}{20}\right) + 1\right) \\ &= 2\pi R^2 \left(1 - \cos\left(\frac{1}{20}\right)\right) = 2\pi R^2 \left(1 - \left(1 - \left(\frac{1}{20}\right)^2 \frac{1}{2!}\right)\right) = \pi R^2/400. \end{aligned}$$

From this we can compute what fraction of the Earth's atmosphere we can see at a given location on Earth:

$$\frac{S_{\text{area of our horizon}}}{E_{\text{sphere}}} = \frac{\pi/400}{4\pi} = 1/1600.$$

Notice the factors of R^2 canceled out.

Let's now use the assumed average fraction of the sky taken up by clouds from our perspective, of 15%. Let's assume the clouds are bunched up to take up 15% of our sky exactly. Then since our sky is only 1/1600 the total sky, this means that this one "fake" cloud takes up an angular fraction of the Earth's atmosphere of 15% of 1/1600, or $15/100 \times 1/1600 = 15/1600 \times 1/100 \approx 1/10,000$. So if we resolve one "cloud" by 3 points in each angular direction (Nyquist sampled), we need 9 points per cloud. Multiply that by 1,600 such "clouds", and we end up with about 15,000 numerical points required to Nyquist sample the cloud cover on Earth. Generally we'd like to resolve each feature in our solution by at least 10 points in each direction, so instead of 9 points, $10^2 = 100$ points per cloud would be far better. Also the clouds won't clump up as we assumed, so if there are a total of 20 clouds in the sky, we'll need 20 times more points. So a better model would use $20 \times 100 \times 15,000$, or about 30 million (3×10^7) points.

2.4 Other Back-of-the-Envelope Estimation Examples

You should be able to estimate based on your existing knowledge alone, without the help of any external sources, answers within an order of magnitude to the following questions.

Exercises:

1. What is the total number of miles driven in passenger vehicles each day within the United States? Based on this, how much gasoline is consumed each day within the United States in gallons? (There are 1.6 km per mile, and 3.8 liters per gallon.)
2. You are working as a digital artist and want to create a realistic 3D model of a healthy forest in the summer time, but before starting the project you smartly decide to do some calculations to determine whether your workstation might be overwhelmed by the task. To this end, you must first compute the number of leaves on a typical large, healthy, leafy (deciduous) tree, to within an order of magnitude. Provide this estimate.
3. You would like to know how much faster your computer code will run in one year. Moore's Law says that CPU speed will double every 18 months. Use the Rule of 72 to compute by what factor CPU speed increases each year, according to Moore's Law. Your final answer must be correct up to and including the second significant digit. *Hint: Recall the Rule of 72 says that for a growth rate of x% per unit time, the amount of time necessary for doubling be given by 72/x.*
4. The Rule of 72 can also be used to estimate the time needed to remove half of some exponentially decaying quantity. Given this, use the Rule of 72, along with the inflation rate of 3% per year to estimate the value of a United States dollar 20 years and 100 from today. I.e., you are to assume that the dollar lowers in value 3% each year.
5. To estimate the distance to an object, we often use parallax. Parallax distances depend on the use of small-angle approximations, which in turn are based on Taylor series expansion of trigonometric functions. Use the Taylor series of the sine function $\sin(x)$ about $x_0 = 0$ to compute by hand $\sin(0.01)$ to four significant digits.
   You may find the following useful:
   Defining $f^{(n)}(x)$ to be the nth derivative of $f(x)$, the Taylor series expansion of a function $f(x)$ about a point $x = x_0$ is given by:
   $$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)(x - x_0)^n}{n!}$$

Solutions:

1. Assuming each driver in the US drives on average 30 miles per day, and one out of three people in the US drive every day (i.e., one-third of 3×10^8 people), we get about 3×10^9 miles per day, within an order of magnitude or so. At 20 miles per gallon average fuel economy, each driver on average consumes 1.5 gallons of gas. Thus we should expect roughly 1.5×10^8 gallons of gas per day. Official figures here: `https://tinyurl.com/oomagnitude1`
2. Roughly 10^6 leaves: `https://tinyurl.com/oomagnitude2`
3. Solution:

$$\begin{aligned} (72/x)\text{years} &= 1.5\text{years} \\ (72/x)\text{years} &= (3/2)\text{years} \\ \implies x &= 72 * 2/3 = 48 \end{aligned}$$

   Thus each year, CPU speed will increase by 48%.
4. The Rule of 72 says that the US dollar will decrease in value due to inflation by one-half after $72/3 = 24$ years. Thus after 20 years, each dollar will be able to purchase only about \$0.60 worth of goods, and after 100 years, each dollar will be able to purchase only about $\$1.00 \times \frac{1}{2^4} \approx \$0.06$ worth of goods.
5. The Taylor series expansion of $\sin(x)$ about $x_0 = 0$ is

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - ...$$

   Therefore $\sin(0.01)$ is

$$\begin{aligned} \sin(0.01) &= 0.01 - \frac{10^{-6}}{6} + \frac{10^{-10}}{120} - ... \\ &\approx 1 \times 10^{-2} - 1.7 \times 10^{-7} + 1 \times 10^{-12} - ... \\ &\approx 9.999 \times 10^{-3} \end{aligned}$$

MATH 522, Spring 2019 Notes

Chapter 3: Partial Differential Equations: An Introduction

Differential equations are equations relating functions and their derivatives. In the case of *Ordinary differential equations* or ODEs, we deal only with functions of one variable. *Partial differential equations* or PDEs relate partial derivatives of functions of more than one variable.

Notation: To denote the partial derivative of $u(x,t)$ with respect to t, we will use the following notation interchangeably:

$$\frac{\partial u(x,t)}{\partial t} = \partial_t u(x,t) = u_t(x,t)$$

The *order* of a PDE is given by its highest derivative.

A PDE is *linear* iff (i.e., if and only if) it can be written in a form where the coefficients are functions of the independent variable only.

We will mostly study first and second order linear PDEs in this class, as these are the PDEs most commonly seen in engineering, scientific, and applied mathematics contexts.

Such linear PDEs fall into 3 basic categories:

- Hyperbolic
- Parabolic
- Elliptic

3.1 The Prototypical Parabolic Equation: The Heat Equation

In one spatial dimension (1D), the heat equation is given by

$$\partial_t u(x,t) = \alpha^2 \partial_x^2 u(x,t),$$

Here, t represents time (e.g., units of seconds), and x represents space (e.g., units of meters).

The solution to the 1D heat equation describes the temperature $u(x,t)$ in a 1D rod, and α^2 denotes the *thermal diffusivity*, which varies depending on the material. α^2 tells us how fast heat will diffuse over a material, and has units of m^2/s. For gold, $\alpha^2 = 1.27 \times 10^{-4} m^2/s$. For 1% carbon steel, $\alpha^2 = 1.172 \times 10^{-5} m^2/s$, and for stainless steel, $\alpha^2 \approx 4 \times 10^{-6} m^2/s$.

For completeness, here is the heat equation in more than one spatial dimension.

Two spatial dimensions:

$$\partial_t u(x,y,t) = \alpha^2 (\partial_x^2 u(x,y,t) + \partial_y^2 u(x,y,t))$$

This might describe the temperature in a 2D spatial plane.

Three spatial dimensions:

$$\partial_t u(x,y,z,t) = \alpha^2 (\partial_x^2 u(x,y,z,t) + \partial_y^2 u(x,y,z,t) + \partial_z^2 u(x,y,z,t)) = \nabla^2 u(x,y,z,t)$$

This might describe the temperature in a 3D cube.

Note that the simple sum of second-order partial derivatives over all spatial dimensions is known as the Laplacian operator, given by ∇^2. So in 3D,

$$\partial_x^2 + \partial_y^2 + \partial_z^2 = \nabla^2.$$

Thus the heat equation in any dimension may be written

$$\partial_t u = \alpha^2 \nabla^2 u.$$

What is the order of this PDE? Answer: second, because the highest derivative is second-order. Is it linear or nonlinear? Answer: Linear because the coefficients in front of the solution u and its derivatives can all be written as simple functions of the independent variables and not u itself. In this case, the coefficients are constants: 1 for $\partial_t u$ and α^2 for $\nabla^2 u$.

3.2 Dimensional Analysis: Another Window into Determining the Scale of a Solution

Let's now try to understand the heat equation's behavior in terms of relevant timescales and lengthscales using the strategy of *dimensional analysis*.

Recall from the definition of partial derivative that

$$\partial_t u(x,t) = \lim_{\Delta t \to 0} \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t}.$$

So $\partial_t u(x,t)$ has units of temperature over time (Kelvin over seconds). To solve this PDE on the computer, we must first have some rough estimate of the timescales and lengthscales necessary to solving it, as we need to know *before solving the problem* how much computer resources will be necessary.

Here's how dimensional analysis works. $\partial_t u(x,t)$ is u's rate of change in time, which as a rough estimate can be written as

$$\partial_t u(x,t) \sim \frac{u(x,t)}{T},$$

where T is the *timescale* over which a rod will reach its equilibrum temperature. Similarly,

$$\partial_x^2 u(x,t) \sim \frac{u(x,t)}{L^2},$$

where L is the lengthscale for a temperature deviation in the rod. Thus we have

$$\begin{aligned} \frac{u(x,t)}{T} &\sim \alpha^2 \frac{u(x,t)}{L^2} \\ \implies \alpha^2 &\sim \frac{L^2}{T}. \end{aligned}$$

If we know α^2 for a material, this equation gives us very deep insights into the solution. Suppose we have an initial temperature perturbation in a, 1% carbon steel rod, that has a typical length scale of $1m$. Then the timescale over which we will need to solve the PDE (corresponding to the timescale over which the temperature will change) is

$$T \sim 1m^2/(10^{-5}m^2/s) \sim 10^5 s.$$

This analysis also yields deep insights into the nature of the PDE.

Suppose we start with a rod with thermal diffusion coefficient α^2 at uniform temperature. At $t = 0$ we touch to one side of this rod a very cold piece of metal. Then at $t = 0$, the solution $u(x,t)$ will have a discontinuity at one of the endpoints. What is the timescale over which this discontinuity will be smoothed out?

Based on dimensional analysis, we found that the timescale over which the solution varies will be related to the lengthscale over which the solution varies via:

$$T \sim \frac{L^2}{\alpha^2}.$$

The discontinuity at the endpoint of the rod has a lengthscale approaching zero, meaning that the timescale over which this discontinuity will be "smoothed out" will be

$$T \sim \lim_{L \to 0} \frac{L^2}{\alpha^2} = 0.$$

That is to say, the heat equation, and parabolic equations in general, smooth out sharp features extremely quickly. *This is a very useful feature*, as we will find later in this class.

3.3 The Prototypical Hyperbolic Equation: The Wave Equation

$$\frac{1}{c^2}\partial_t^2 u(x,t) = \partial_x^2 u(x,t)$$

is the 1D wave equation. The solution $u(x,t)$ measures the displacement in, e.g., meters of a wave, and c^2 has SI units of m^2/s^2, so c represents the propagation speed of the wave.

In 2 spatial dimensions, this equation becomes

$$\frac{1}{c^2}\partial_t^2 u(x,y,t) = (\partial_x^2 + \partial_y^2)u(x,y,t)$$

And in 3 spatial dimensions, this equation becomes

$$\frac{1}{c^2}\partial_t^2 u(x,y,z,t) = (\partial_x^2 + \partial_y^2 + \partial_z^2)u(x,y,z,t)$$

What is the order of this PDE?

3.4 The Prototypical Elliptic Equation: Poisson's Equation

$$\partial_x^2 u(x) = 4\pi G\rho(x)$$

is the 1D Poisson's equation, which is used to solve for the gravitational potential $u(x)$ in Newton's theory of gravity. $\rho(x)$ represents the local matter density, and G the gravitational constant.

Extensions to 2D and 3D are straightforward.

In the limit $\rho \to 0$, we obtain *Laplace's equation*:

$$\partial_x^2 u(x) = 0,$$

which is exactly the wave equation but with $c \to \infty$. Thus we can interpret Laplace's equation as transferring information *instantaneously*.

You'll notice that there is also no timescale in Poisson's equation, which is a reflection that information travels instantaneously in this case as well. As it turns out, the fastest speed at which information can propagate is the speed of light, so instantaneous transfer of information is not physically possible. However, Poisson's equation can be a good approximation for situations in which the timescale over which the solution varies is much slower than the travel time of a wave across the lengthscale of interest. .

MATH 522, Spring 2019 Notes

Chapter 4: Method of Separation of Variables

4.1 Solving the Heat Equation via Separation of Variables

Suppose we wish to solve the heat equation in one spatial dimension:

$$\partial_t u(x,t) = \alpha \partial_x^2 u(x,t)$$

The method of separation of variables assumes a solution of the form

$$u(x,t) = X(x)T(t),$$

allowing us to rewrite the PDE as

$$\alpha \partial_x^2 u(x,t) = \alpha X''(x)T(t) = \partial_t u(x,t) = T'(t)X(x).$$

Assuming that we are not interested in the trivial solution $u(x,t) = 0$, we may divide both sides by $\alpha X(x)T(t)$:

$$\frac{X''(x)}{X(x)} = \frac{T'(t)}{\alpha T(t)}.$$

Notice that the left-hand side of the equation varies with x and the right-hand side varies with t. Therefore, in order for the two sides to be equal, both sides must be constant. Let's call this arbitrary constant $-\lambda$.

Then we have:

$$\frac{X''(x)}{X(x)} = \frac{T'(t)}{\alpha T(t)} = -\lambda,$$

which implies

$$\begin{aligned} X''(x) + \lambda X(x) &= 0 \\ T'(t) + \lambda \alpha T(t) &= 0. \end{aligned}$$

Notice we have reduced the PDE to a set of two, uncoupled ODEs!

For $T(t)$, we are to attempt a trial solution of the form

$$\begin{aligned} T(t) = e^{rt} \quad &\implies \quad T'(t) + \lambda \alpha T(t) = 0 \\ \implies r + \lambda \alpha \quad &= \quad 0 \\ \implies T(t) = e^{-\lambda \alpha t}. & \end{aligned}$$

Now the motivation for choosing $-\lambda$ becomes clear; if $\lambda < 0$ then $T(t)$ will diverge as $t \to \infty$, which is unphysical. Thus $\lambda > 0$ is the only physically permissible constant.

Let's next solve for $X(x)$. Based on our experience solving ODEs, we are to attempt a trial solution

$$X(x) = e^{rx}$$

Substituting this into the ODE, we get

$$r^2 + \lambda = 0 \implies r = \pm i\sqrt{\lambda}.$$

Thus

$$X(x) = Ce^{i\sqrt{\lambda}x} + De^{-i\sqrt{\lambda}x}$$

But we know that Euler's equation implies

$$e^{ix} = \cos x + i \sin x,$$

so we get

$$\begin{aligned} X(x) &= C[\cos(\sqrt{\lambda}x) + i\sin(\sqrt{\lambda}x)] + D[\cos(\sqrt{\lambda}x) - i\sin(\sqrt{\lambda}x)] \\ &= (C+D)\cos(\sqrt{\lambda}x) + i(C-D)\sin(\sqrt{\lambda}x) \\ &= E\cos(\sqrt{\lambda}x) + F\sin(\sqrt{\lambda}x). \end{aligned}$$

Now let's suppose we wish to solve the heat equation for a rod of length L, with boundary conditions $u(0,t) = u(L,t) = 0$ (ends of rod are held at zero temperature for all times). Then we know that

$$X(0) = X(L) = 0$$

Since $\cos(\sqrt{\lambda}x)$ evaluated at $x = 0$ or $x = L$ cannot be zero, $E = 0$. Then we have

$$\begin{aligned} X(0) &= F \times 0 = 0 \\ X(L) &= F\sin(\sqrt{\lambda}L) = 0 \end{aligned}$$

The second equation is only true for λ such that $\sin(\sqrt{\lambda}L) = 0$. $\sin(x) = 0$ for $x = n\pi$, where n is an integer. Thus we have

$$\sqrt{\lambda}L = n\pi \implies \sqrt{\lambda} = \frac{n\pi}{L}$$

Since all n satisfy the heat equation, the general solution to this PDE is

$$u(x,t) = \sum_{n=1}^{\infty} c_n e^{-\alpha(n\pi/L)^2 t} \sin\left(\frac{n\pi x}{L}\right),$$

where c_n is a constant coefficent that depends on n.

But how do we set these c_n's? The answer: Initial conditions!

Suppose $u(x,0) = f(x)$. Then

$$u(x,0) = \sum_{n=1}^{\infty} c_n \sin\left(\frac{n\pi x}{L}\right) = f(x)$$

Multiply both sides of the equation by $\sin\left(\frac{m\pi x}{L}\right)$, where m is an integer, and integrate from 0 to L.

$$\implies \sum_{n=1}^{\infty} c_n \int_0^L \sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi x}{L}\right)dx = \int_0^L f(x)\sin\left(\frac{m\pi x}{L}\right)dx$$

Sines and cosines are orthogonal functions, meaning that for integers m and n

$$\int_0^L \sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi x}{L}\right)dx$$

is zero unless $m = n$. If $m = n$, then the integral evaluates to $L/2$. Note that if m or n is not an integer, then this analysis does not apply. Thus we have

$$c_m = \frac{2}{L}\int_0^L f(x)\sin\left(\frac{m\pi x}{L}\right)dx$$

and the solution becomes

$$u(x,t) = \sum_{n=1}^{\infty}\left\{\left[\frac{2}{L}\int_0^L f(x)\sin\left(\frac{n\pi x}{L}\right)dx\right]\sin\left(\frac{n\pi x}{L}\right)e^{-\alpha(n\pi/L)^2 t}\right\}.$$

Notice that this solution is simply a Fourier series in x multiplied by an exponential damping in time. Also, notice that the number of oscillations in the sine function in the interval is simply n, and the wavelength is given by $\lambda_n = \frac{2L}{n}$. Thus short-wavelength features are given by very large n. The solution indicates that the timescale for damping goes like $[L/(n\pi)]^2$, which for very large n (short wavelength) is damped very very fast, consistent with the intuitions we gained through dimensional analysis.

MATH 522, Spring 2019 Notes

Chapter 5: Polynomial Interpolation & Extrapolation

Suppose we are given data for a function $f(x)$ at a discrete number of points $x_0 < x_1 < ... < x_{N-1} < x_N$ only. Then, assuming $f(x)$ is smooth, how do we estimate values for other points $x \in [x_0, x_N]$? How about $x \notin [x_0, x_N]$?

Definition: The process of estimating values of $f(x)$ at $x \in [x_0, x_N]$ based on known values of the function $\{f(x_0), f(x_1), ..., f(x_N)\}$ is known as **interpolation**.

Definition: The process of estimating values of $f(x)$ at $x \notin [x_0, x_N]$ based on known values of the function $\{f(x_0), f(x_1), ..., f(x_N)\}$ is known as **extrapolation**.

In the following sections, we will exclusively refer to *interpolation*, but the discussion could equally well be applied to *extrapolation*.

5.1 Lagrange Polynomial Interpolation

Example #1

Suppose we have values of a function at only two points $x_0 < x_1$: $f_0 = f(x_0)$ and $f_1 = f(x_1)$. How do we interpolate values of the function between x_0 and x_1? How do we extrapolate the function outside of this range of x?

The equation of a line (i.e., a first-order polynomial $P_1(x)$) between f_0 and f_1 can be written

$$P_1(x) = c_1 x + c_0,$$

where c_1 is the slope and c_0 is the y-intercept. c_0 and c_1 are unknown coefficients, but we are given values of the function at two points x_0 and x_1. We wish for the line $P_1(x)$ to pass through the function at these two points. Thus we obtain two equations and two unknowns:

$$\begin{aligned} P_1(x_0) = f(x_0) = f_0 &= c_1 x_0 + c_0 \\ P_1(x_1) = f(x_1) = f_1 &= c_1 x_1 + c_0. \end{aligned}$$

Definition: Writing this set of equations in matrix form, we obtain the **Vandermonde matrix $\boldsymbol{A}$** for computing the unknown linear interpolation coefficients.

$$\boldsymbol{Ac} = \begin{bmatrix} 1 & x_0 \\ 1 & x_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \boldsymbol{f} = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}.$$

Remember that x_0, x_1, f_0, and f_1 are known, and c_0 and c_1 are unknowns. Solving this linear system of equations for c_0 and c_1, we obtain

$$\begin{aligned} c_0 &= f_0 - x_0 \frac{f_1 - f_0}{x_1 - x_0} \\ c_1 &= \frac{f_1 - f_0}{x_1 - x_0}. \end{aligned}$$

Notice the second equation yields the slope of the function, and in the limit where $x_1 \to x_0$, this is precisely the definition of derivative!

Thus the equation that yields the interpolated or extrapolated value of the function $f(x)$ at any x based on this linear approximation is given by

$$\begin{aligned} P_1(x) &= \frac{f_1 - f_0}{x_1 - x_0} x + f_0 - x_0 \frac{f_1 - f_0}{x_1 - x_0} \\ &= \frac{f_1 - f_0}{x_1 - x_0}(x - x_0) + f_0 \\ &= \left(1 - \frac{x - x_0}{x_1 - x_0}\right) f_0 + \frac{x - x_0}{x_1 - x_0} f_1 \\ &= \frac{x_1 - x_0 - x + x_0}{x_1 - x_0} f_0 + \frac{x - x_0}{x_1 - x_0} f_1 \\ &= \frac{x_1 - x}{x_1 - x_0} f_0 + \frac{x - x_0}{x_1 - x_0} f_1 \\ &= \frac{x - x_1}{x_0 - x_1} f_0 + \frac{x - x_0}{x_1 - x_0} f_1 \end{aligned} \tag{5.1}$$

Example #2

Repeat **Example #1**, but if the function $f(x)$ is given at three points $x_0 < x_1 < x_2$.

The lowest degree polynomial with 3 coefficients is a quadratic:

$$P_2(x) = c_2 x^2 + c_1 x + c_0. \tag{5.2}$$

Knowing the value of $f(x)$ at x_0, x_1, and x_2 provides us with three equations and three unknowns:

$$\begin{aligned} P_2(x_0) = f(x_0) = f_0 &= c_2 x_0^2 + c_1 x_0 + c_0 \\ P_2(x_1) = f(x_1) = f_1 &= c_2 x_1^2 + c_1 x_1 + c_0 \\ P_2(x_2) = f(x_2) = f_2 &= c_2 x_2^2 + c_1 x_2 + c_0, \end{aligned}$$

where again we require that the polynomial fitting the function overlaps the function at these three points. Once we have these unknowns c_2, c_1, and c_0, we can immediately perform quadratic interpolation and extrapolation of the function.

Writing this set of equations in matrix form, we obtain the **Vandermonde matrix** for computing the unknown quadratic interpolation coefficients:

$$\boldsymbol{A}\boldsymbol{c} = \begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} = \boldsymbol{f}.$$

The inverse of the Vandermonde matrix, $\boldsymbol{A}^{-1}$, yields the coefficients:

$$\begin{aligned} \boldsymbol{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \boldsymbol{A}^{-1}\boldsymbol{f} &= \begin{bmatrix} \frac{x_1 x_2}{(x_0 - x_1)(x_0 - x_2)} & \frac{x_0 x_2}{(x_1 - x_0)(x_1 - x_2)} & \frac{x_0 x_1}{(x_2 - x_0)(x_2 - x_1)} \\ -\frac{x_1 + x_2}{(x_0 - x_1)(x_0 - x_2)} & -\frac{x_0 + x_2}{(x_1 - x_0)(x_1 - x_2)} & -\frac{x_0 + x_1}{(x_2 - x_0)(x_2 - x_1)} \\ \frac{1}{(x_0 - x_1)(x_0 - x_2)} & \frac{1}{(x_1 - x_0)(x_1 - x_2)} & \frac{1}{(x_2 - x_0)(x_2 - x_1)} \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} \\ &= \begin{bmatrix} x_1 x_2 & x_0 x_2 & x_0 x_1 \\ -(x_1 + x_2) & -(x_0 + x_2) & -(x_0 + x_1) \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \frac{f_0}{(x_0 - x_1)(x_0 - x_2)} \\ \frac{f_1}{(x_1 - x_0)(x_1 - x_2)} \\ \frac{f_2}{(x_2 - x_0)(x_2 - x_1)} \end{bmatrix} \\ &= \begin{bmatrix} x_1 x_2 & x_0 x_2 & x_0 x_1 \\ -(x_1 + x_2) & -(x_0 + x_2) & -(x_0 + x_1) \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \end{bmatrix} \end{aligned}$$

Next, let's plug these coefficients into the original quadratic expression (Eq. 5.2) to obtain the expression for any interpolated value $f(x)$ for a quadratic polynomial:

$$\begin{aligned}
P_2(x) &= c_2x^2 + c_1x + c_0 \\
&= \left[\tilde{f}_0 + \tilde{f}_1 + \tilde{f}_2\right]x^2 - \left[(x_1+x_2)\tilde{f}_0 + (x_0+x_2)\tilde{f}_1 + (x_0+x_1)\tilde{f}_2\right]x + \left[x_1x_2\tilde{f}_0 + x_0x_2\tilde{f}_1 + x_0x_1\tilde{f}_2\right] \\
&= \tilde{f}_0\left[x^2 - (x_1+x_2)x + x_1x_2\right] + \tilde{f}_1\left[x^2 - (x_0+x_2)x + x_0x_2\right] + \tilde{f}_2\left[x^2 - (x_0+x_1)x + x_0x_1\right] \\
&= \tilde{f}_0\left[x(x-x_1) + x_2(x_1-x)\right] + \tilde{f}_1\left[x(x-x_0) + x_2(x_0-x)\right] + \tilde{f}_2\left[x(x-x_1) + x_0(x_1-x)\right] \\
&= \tilde{f}_0\left[x(x-x_1) - x_2(x-x_1)\right] + \tilde{f}_1\left[x(x-x_0) - x_2(x-x_0)\right] + \tilde{f}_2\left[x(x-x_1) - x_0(x-x_1)\right] \\
&= (x-x_2)(x-x_1)\tilde{f}_0 + (x-x_2)(x-x_0)\tilde{f}_1 + (x-x_0)(x-x_1)\tilde{f}_2 \\
&= \frac{\prod_{i\neq 0}(x-x_i)}{\prod_{i\neq 0}(x_0-x_i)}f_0 + \frac{\prod_{i\neq 1}(x-x_i)}{\prod_{i\neq 1}(x_1-x_i)}f_1 + \frac{\prod_{i\neq 2}(x-x_i)}{\prod_{i\neq 2}(x_2-x_i)}f_2 \\
&= \sum_{i=0}^{2}\ell_i(x)f(x_i), \quad \text{where} \quad \ell_i(x) = \prod_{\substack{0\leq j\leq 2,\\ i\neq j}}\frac{(x-x_j)}{(x_i-x_j)}
\end{aligned} \tag{5.3}$$

Notice that the above expression holds true for first-order (linear) polynomials as well (cf. Eq. 5.1), if each two in the bottom expression is replaced by a one.

Looking at the inverse of the Vandermonde matrix for third-order polynomial interpolation,

$$\boldsymbol{A}^{-1}\boldsymbol{f} = \begin{bmatrix} -x_1x_2x_3 & -x_0x_2x_3 & -x_0x_1x_3 & -x_0x_1x_2 \\ x_1x_2 + x_1x_3 + x_2x_3 & x_0x_2 + x_0x_3 + x_2x_3 & x_0x_1 + x_0x_3 + x_1x_3 & x_0x_1 + x_0x_2 + x_1x_2 \\ -(x_1+x_2+x_3) & -(x_0+x_2+x_3) & -(x_0+x_1+x_3) & -(x_0+x_1+x_2) \\ 1 & 1 & 1 & 1 \end{bmatrix}\begin{bmatrix}\tilde{f}_0\\ \tilde{f}_1\\ \tilde{f}_2\\ \tilde{f}_3\end{bmatrix},$$

where, extending the pattern for $N = 2$ order polynomials to $N = 3$ order, we define

$$\tilde{f}_i = f_i \Big/ \prod_{\substack{0\leq j\leq 3,\\ j\neq i}}(x_i - x_j)\,,$$

we will find by following the same steps as with second order that at third order

$$\begin{aligned}
P_3(x) =& c_3x^3 + c_2x^2 + c_1x + c_0 \\
=& \sum_{i=0}^{3}\ell_i(x)f(x_i) \quad \text{where} \quad \ell_i(x) = \prod_{\substack{0\leq j\leq 3,\\ i\neq j}}\frac{(x-x_j)}{(x_i-x_j)}
\end{aligned} \tag{5.4}$$

as well. As we have shown, the above formula holds for $P_N(x)$, where the upper limits on the sum and product are $n = 1$, 2, or 3. It can be proven that the pattern we found in Eqs. 5.1, 5.3, and 5.4 continues to hold for *all* polynomial degrees N, so that

$$P_N(x) = \sum_{i=0}^{N}c_ix^i = \sum_{i=0}^{N}\ell_i(x)f(x_i), \quad \text{where} \quad \ell_i(x) = \prod_{\substack{0\leq j\leq N,\\ i\neq j}}\frac{(x-x_j)}{(x_i-x_j)}. \tag{5.5}$$

Of course, this assumes that we have evaluated $f(x)$ at $N+1$ points (because a sum running from 0 to N contains $N+1$ terms). Edward Warring may be the discoverer of this formula, but Joseph Louis Lagrange published it about 15 years after Warring's discovery, in 1795. Thus it is known as the **Lagrange interpolation formula**.

5.2 Numerical Implementation

Suppose we wish to implement the Lagrange interpolation formula of Eq. 5.5 numerically. As input, we will need some function $f(x)$ evaluated at N points x_j, where $x_0 < x_1 < ... < x_j < x_{j+1} < ... < x_N$, and as output, we will be able to approximately evaluate $f(x)$ at any $x \in [x_0, x_N]$. This does not exclude use of this formula for *extrapolation*, however; it can be evaluated at any x outside the interval as well, and such an extrapolation can be quite accurate for some range of $x \notin [x_0, x_N]$ depending on the underlying function.

We might find this strategy quite useful if $f(x)$ is extremely expensive to compute, and we wish to evaluate an approximate expression for $f(x)$ at a very large number of points x in the neighborhood of $[x_0, x_N]$. It could also be useful if we cannot evaluate the function at points between x_i and x_{i+1} (e.g., because our measuring instrument does not provide the data), yet we have a reasonable expectation that the function $f(x)$ is smooth.

In pseudocode, the Lagrange interpolation formula can be written

Lagrange Polynomial Interpolation Pseudocode

```
output = 0.0
do i=0,N
   l_i = 1.0
   do j=0,N
      if(j not equal to i) then
         l_i = l_i * (x - x[j])/(x[i] - x[j])
      end if
   end do
   output = output + l_i * f[i]
end do
```

MATH 522, Spring 2019 Notes

Chapter 6: Numerical Representations of Derivatives

Here are three of the most common ways to represent partial derivatives of some function u:

1. Finite Difference: u is sampled on a numerical grid, then a low-degree polynomial is fitted to the function *near* the point at which the derivative is to be evaluated. The derivative of the fitting polynomial is taken as an approximation for the derivative of the function.
2. Spectral: u is approximated by a single high-order polynomial or Fourier series over the entire domain. Derivatives of the polynomial or Fourier series approximates the function's derivatives.
3. Finite element/Spectral element: The numerical grid is broken into multiple subdomains. The function is approximated with non-overlapping polynomials/Fourier series, one per subdomain. Derivatives of polynomials/Fourier series approximates function's derivatives in that subdomain.

In Sec. 6.1 finite difference methods are reviewed, and in Sec. 6.2 spectral methods are introduced. Finite/Spectral element combines a bit of both, so are not reviewed. **Finite difference representations of derivatives are the most straightforward to implement, so we will focus on this approach almost exclusively in this class. Therefore, you will find that Sec. 6.1 is the most detailed.**

6.1 Finite Difference Techniques

Finite difference techniques are based on the notion that *smooth* (i.e., multiple-times-differentiable) functions can at points x near $\bar{x}$ (i.e., in the *neighborhood* of $\bar{x}$) be locally represented by a power series:

$$f(x) = \sum_{n=0}^{\infty} a_n (x-\bar{x})^n.$$

For infinitely differentiable functions (also known as *analytic functions*), the coefficients a_n are given by

$$a_n = \frac{f^{(n)}(\bar{x})}{n!}.$$

That is to say, the power series representation of an analytic function is equal to the Taylor series of that function! For all analytic functions, the Taylor series must *converge* to the value of the function at all points in the neighborhood as the number of terms in the series (starting at $n = 0$) approaches infinity. Once we have chosen a point $\bar{x}$, we call the sum "the Taylor series of the function $f(x)$ about $\bar{x}$".

Finite difference derivatives rely on the assumption that the underlying function $f(x)$ is smooth, so that the function may be approximated near a given point $\bar{x}$ by a Taylor series with a small number of terms. Since a Taylor series is simply a polynomial, this means that the function $f(x)$ near the point $\bar{x}$ can be well approximated by a low-degree polynomial.

For example, consider the function $f(x) = \sin(x)$ at the point $\bar{x} = 0$. How rapidly does the Taylor series converge to the exact value of the function at neighboring point $x = 10^{-6}$? Using the above formula for a_n, we will find that The Taylor series of this function about $\bar{x} = 0$ is given by

$$\begin{aligned} f(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, \quad \text{so} \\ f(10^{-6}) &= 10^{-6} - \frac{(10^{-6})^3}{3!} + \frac{(10^{-6})^5}{5!} - \frac{(10^{-6})^7}{7!} + \ldots = 10^{-6} - \frac{1}{6}10^{-18} + \frac{1}{120}10^{-30} - \ldots \end{aligned}$$

Thus the value of the function at $x = 10^{-6}$ is extremely well approximated by *just the first term in the sum*, and even better approximated (to about 24 significant digits) by the first two terms in the sum.

Now if we take the first derivative of the Taylor series, we find

$$\begin{aligned} f'(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, \quad \text{so} \\ f'(10^{-6}) &= 1 - \frac{(10^{-6})^2}{2!} + \frac{(10^{-6})^4}{4!} - \frac{(10^{-6})^6}{6!} + \ldots = 1 - \frac{1}{2}10^{-12} - \frac{1}{24}10^{-24} + \frac{1}{720}10^{-36} - \ldots \end{aligned}$$

Therefore if we approximate the function with only the *first-degree polynomial* $P_1(x) = x$, we find the derivative of $f(x) = \sin(x)$ at $x = 10^{-6}$ is correct to about 12 significant digits! (I.e., the derivative matches the exact value to about 1 part in 10^{12}.) Further, if we instead approximate the function with a third-degree polynomial $P_3(x) = x - \frac{x^3}{3!}$, then the derivative will be correct to about *24* significant digits!

This behavior holds for all sufficiently smooth functions, provided we choose a value x close enough to $\bar{x}$.

Finite difference derivatives take advantage of this fact that smooth functions like $f(x) = \sin(x)$ are very well approximated near some point $\bar{x}$ by a low-degree polynomial.

Let's next solidify the notion of finite difference derivative.

Definition of Finite-Difference Derivative

Given some function $f(x)$, approximate $f'(x)$ at $x = \bar{x}$ by sampling the function at $N+1$ unique points in the neighborhood of $\bar{x}$. Number these points $x_0, x_1, x_2, ..., x_N$, and the value of the function evaluated at these points as $f(x_0) = f_0, f(x_1) = f_1, ..., f(x_N) = f_N$, respectively. There is exactly one Nth-degree polynomial

$$P_N(x) = \sum_{n=0}^{N} a_n (x - \bar{x})^n$$

that is guaranteed to match the function's value at all of these points. In other words, $P_N(x) = f(x)$ at all $x \in \{x_0, x_1, x_2, ..., x_N\}$. (Prove this to yourself by noticing that a degree-1 polynomial [a line] is uniquely determined by 2 points; a degree-2 polynomial [a quadratic] is uniquely determined by 3 points, etc.) We say that $P'_N(\bar{x})$ represents a **finite difference approximation of** $f'(\bar{x})$.

As a corollary to the fact that $P_N(x) = f(x)$ at all $x \in \{x_0, x_1, x_2, ..., x_N\}$, we can solve for the a_n coefficients by constructing a system of $N+1$ equations with the $N+1$ unknowns $\{a_0, a_1, ..., a_N\}$:

$$\begin{aligned} P_N(x_0) = f_0 &= a_0 + a_1(x_0 - \bar{x}) + a_2(x_0 - \bar{x})^2 + \ldots + a_N(x_0 - \bar{x})^N \\ P_N(x_1) = f_1 &= a_0 + a_1(x_1 - \bar{x}) + a_2(x_1 - \bar{x})^2 + \ldots + a_N(x_1 - \bar{x})^N \\ \vdots &= \vdots \\ P_N(x_N) = f_N &= a_0 + a_1(x_N - \bar{x}) + a_2(x_N - \bar{x})^2 + \ldots + a_N(x_N - \bar{x})^N. \end{aligned}$$

In matrix form, this linear system of equations can be written

$$\boldsymbol{f} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_N \end{pmatrix} = \boldsymbol{V}\boldsymbol{a} = \begin{pmatrix} 1 & (x_0 - \bar{x}) & (x_0 - \bar{x})^2 & \ldots & (x_0 - \bar{x})^N \\ 1 & (x_1 - \bar{x}) & (x_1 - \bar{x})^2 & \ldots & (x_1 - \bar{x})^N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (x_N - \bar{x}) & (x_N - \bar{x})^2 & \ldots & (x_N - \bar{x})^N \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{pmatrix}.$$

The matrix $\boldsymbol{V}$ is known as the **Vandermonde matrix**, and the solution to this matrix equation is $\boldsymbol{a} = \boldsymbol{V}^{-1}\boldsymbol{f}$. Adopting index form for $\boldsymbol{V}^{-1}$, where $\boldsymbol{V}^{-1}_{ni}$ refers to the nth row and the ith column of matrix $\boldsymbol{V}^{-1}$, we can write the matrix equation completely in index form:

$$a_n = \sum_{i=0}^{N} \boldsymbol{V}^{-1}_{ni} f_i = \sum_{i=0}^{N} f_i \boldsymbol{V}^{-1}_{ni},$$

where $\boldsymbol{V}^{-1}$ depends only on $\{\bar{x}, x_0, x_1, x_2, ..., x_N\}$. Thus,

$$P_N(x) = \sum_{n=0}^{N} a_n (x - \bar{x})^n = \sum_{n=0}^{N} \sum_{i=0}^{N} f_i \boldsymbol{V}^{-1}_{ni} (x - \bar{x})^n = \sum_{i=0}^{N} \sum_{n=0}^{N} f_i \boldsymbol{V}^{-1}_{ni} (x - \bar{x})^n = \sum_{i=0}^{N} f_i \sum_{n=0}^{N} \boldsymbol{V}^{-1}_{ni} (x - \bar{x})^n = \sum_{i=0}^{N} b_i(x) f_i.$$

Thus **the mth finite difference derivative in canonical form** will be given by

$$P^{(m)}_N(x) = \sum_{i=0}^{N} b^{(m)}_i(x) f_i,$$

where

$$b_i(x) = \sum_{n=0}^{N} \boldsymbol{V}^{-1}_{ni} (x - \bar{x})^n,$$

and $b^{(m)}_i(x)$ are known as the **finite difference coefficients**, which **do not depend on the function** $f(x)$.

Indeed the expression we just derived for the finite difference coefficients appears rather complicated, but we will find later there are shortcuts that simplify their computation. First, let's follow the prescription based on this definition:

Approximating the first derivative with a first-degree polynomial

Suppose we have a smooth function $f(x)$ and we wish to evaluate $f'(x)$ at $\bar{x} = x_0$. According to the prescription above, for a finite difference derivative based on an Nth-degree polynomial, we must sample the function at $N+1$ points in the neighborhood of $\bar{x}$. So for a linear polynomial, let's sample the function at x_0 and $x_1 = x_0 + \Delta x$, which are assumed to both be in the neighborhood of $\bar{x}$.
We wish to evaluate $f'(x)$ at $\bar{x} = x_0$ using a first-degree-polynomial-based finite difference derivative approximation at x_0. I.e., we first first fit the function $f(x)$ at these two points by the unique linear polynomial centered at x_0:

$$P_1(x) = a_0 + a_1(x - x_0).$$

Then $P_1'(x_0)$ is the first-degree-polynomial-based finite-difference derivative approximation of $f'(x_0)$. To evaluate $P_1(x)$, we first find the coefficients a_0 and a_1 by constructing the Vandermonde matrix:

$$\boldsymbol{f} = \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} = \boldsymbol{V}\boldsymbol{a} = \begin{pmatrix} 1 & (x_0 - \bar{x}) \\ 1 & (x_1 - \bar{x}) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & (x_1 - x_0) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & \Delta x \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

and then inverting it:
Notice that f_0, f_1, x_0, x_1, and Δx are known, so we have here two equations to solve for the two unknowns a_0 and a_1. Inverting the Vandermonde matrix, we get

$$\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \boldsymbol{V}^{-1}\boldsymbol{f} = \begin{pmatrix} 1 & 0 \\ -\frac{1}{\Delta x} & \frac{1}{\Delta x} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \end{pmatrix}.$$

Multiplying, we find

$$\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} f_0 \\ \frac{f_1 - f_0}{\Delta x} \end{pmatrix}$$

By definition, $P_1(x) = a_0 + a_1(x - x_0)$, which implies that

$$P_1'(x_0) = a_1 = \frac{f_1 - f_0}{\Delta x}.$$

Notice in the limit of $\Delta x \to 0$, this *is precisely the definition of* $f'(x_0)$!

What if we increase the degree of the fitting polynomial used when evaluating a finite-difference derivative? This amounts to adding up more terms in the Taylor series approximation of the function near $x = \bar{x}$, and we will find that (for smooth functions) the derivative approximation becomes far better in this case.

Approximating the first derivative with a quadratic polynomial

Suppose we are given a function at three points $f(x_0) = f_0$, $f(x_1) = f_1$, and $f(x_2) = f_2$. We wish to evaluate $f'(x)$ at $\bar{x} = x_1$ using a second-degree-polynomial-based finite difference derivative approximation at x_1. I.e., we first first fit the function $f(x)$ at these three points by the unique quadratic polynomial centered at x_1:

$$P_2(x) = a_0 + a_1(x - x_1) + a_2(x - x_1)^2$$

Then $P_2'(x_1)$ is the second-degree-polynomial-based finite-difference derivative approximation of $f'(x_1)$. To evaluate $P_2(x)$, we first find the coefficients a_0, a_1, and a_2 by constructing the Vandermonde matrix:

$$\boldsymbol{f} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix} = \boldsymbol{V}\boldsymbol{a} = \begin{pmatrix} 1 & (x_0 - x_1) & (x_0 - x_1)^2 \\ 1 & 0 & 0 \\ 1 & (x_2 - x_1) & (x_2 - x_1)^2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}$$

and then inverting it:

$$\boldsymbol{V}^{-1}\boldsymbol{f} = \begin{pmatrix} 0 & 1 & 0 \\ \frac{1}{x_2 - x_0} + \frac{1}{x_0 - x_1} & \frac{1}{x_1 - x_2} + \frac{1}{x_1 - x_0} & \frac{1}{x_2 - x_1} + \frac{1}{x_0 - x_2} \\ \frac{1}{(x_0 - x_1)(x_0 - x_2)} & \frac{1}{(x_1 - x_0)(x_1 - x_2)} & \frac{1}{(x_0 - x_2)(x_1 - x_2)} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix} = \boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix},$$

the coefficients $\boldsymbol{a}$ can be computed immediately. After some simplification, we find:

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ \frac{f_0 - f_1}{x_0 - x_1} + \frac{f_2 - f_0}{x_0 - x_2} + \frac{f_1 - f_2}{x_1 - x_2} \\ \frac{\frac{f_0 - f_2}{x_0 - x_2} + \frac{f_2 - f_1}{x_1 - x_2}}{x_0 - x_1}. \end{pmatrix}.$$

Again, $P_2(x) = a_0 + a_1(x - x_1) + a_2(x - x_1)^2$, so $P_2'(x) = a_1 + 2a_2(x - x_1)$, so $P_2'(x_1) = a_1$ and we get

$$P_2'(x_1) = \frac{f_0 - f_1}{x_0 - x_1} + \frac{f_2 - f_0}{x_0 - x_2} + \frac{f_1 - f_2}{x_1 - x_2}.$$

In the case of uniform sampling, where x_1 is the center point evaluated, we have $x_2 = x_1 - \Delta x$ and $x_0 = x_1 + \Delta x$, the above equation simplifies to

$$\begin{aligned} P_2'(x_1) &= \frac{f_0 - f_1}{-\Delta x} + \frac{f_2 - f_0}{-2\Delta x} + \frac{f_1 - f_2}{-\Delta x} \\ &= \frac{1}{\Delta x}\left(f_1 - f_0 + \frac{1}{2}(f_0 - f_2) + f_2 - f_1\right) \\ &= \frac{1}{2\Delta x}(f_2 - f_0) = \frac{f(x_1 + \Delta x) - f(x_1 - \Delta x)}{2\Delta x} \end{aligned}$$

where the finite difference coefficients may be written as $b_0'(x_1) = 1/(2\Delta x)$, $b_1'(x_1) = 0$, $b_2'(x_1) = -1/(2\Delta x)$. The above provides an alternative definition of the derivative; just take the limit as $\Delta x \to 0$, and we have the *centered definition of derivative*. Going to higher degree fitting polynomials or choosing different sampling points can produce an infinite number of alternative definitions of derivative.

6.1.1 Second-Degree-Polynomial-Based Finite Difference Coefficients from Quadratic Functions $f(x)$

Key Idea

Since the quadratic polynomial $P_2(x)$ must pass through $f(x)$ at all points $x \in \{x_{-1}, x_0, x_1\}$, if $f(x)$ happens to be a quadratic polynomial itself, $P_2(x) \equiv f(x)$ at *all* points x, and the derivative $P_2'(x)$ is identically equal to $f'(x)$ at all points x as well.

This fact implies that for $f(x) = A(x-\bar{x})^2 + B(x-\bar{x}) + C$, $P_2^{(m)}(x) \equiv f^{(m)}(x)$, *regardless of what* m, A, B, C, x, *or* $\bar{x}$ *we choose.*

The same idea holds true for finite-difference approximations based on polynomials of *any* degree; if $f(x) = P_N(x)$, then the Nth-degree-polynomial finite-difference derivative will be *exact.*

Recall in our definition of finite difference derivative, that the finite difference coefficents $b_i^{(m)}(x)$ *do not depend on the sampled function* $f(x)$. Thus $b_i^{(m)}(x)$ will be the same, regardless of the function $f(x)$ that we choose to sample, **even if we choose** $f(x) = P_N(x)$. In this section, we will find that when we choose $f(x) = P_N(x)$, then the coefficients $b_i^{(m)}(x)$ can be computed quite easily.

To illustrate the technique, let's find the second-degree-polynomial-based finite-difference coefficients choosing an evenly-sampled grid; i.e., with constant grid spacing Δx. For a quadratic (second-degree) polynomial, this means that we will sample the function at three points x_i where $x_{i+1} = x_i + \Delta x$. Following the **Key Idea** above, we will choose $f(x) = P_2(x)$, and since this must hold for all x, let's evaluate the derivative at $\bar{x} = 0$, choosing the central sampling point to be at $x = 0$ also.

Therefore we will sample the function

$$f(x) = Ax^2 + Bx + C$$

at three points $x \in \{x_{-1}, x_0, x_1\} = \{\Delta x, 0, \Delta x\}$.

Our goal now is to find the finite-difference coefficients $b_i'(\bar{x})$ *without inverting a Vandermonde matrix*, and instead applying the **Key Idea** above, which states that regardless of the chosen A, B, or C, $f(x)$ will be equivalent to $P_2(x)$.

- Let's first choose $A = B = 0$. Then $f(x) = C$, $P_2(x) = f(x)$ and $P_2'(x) = 0 \equiv f'(x)$ *at all points* x. Returning to the canonical finite difference form, this implies

$$\begin{aligned} P_2'(x) = 0 &= \frac{c_{-1}f_{-1} + c_0 f_0 + c_1 f_1}{\Delta x} \\ &= \frac{C(c_{-1} + c_0 + c_1)}{\Delta x}. \end{aligned}$$

  Thus we have found that c_{-1}, c_0, and c_1 must satisfy $c_{-1} + c_0 + c_1 = 0$.

- Next let's choose $A = C = 0$. In this case $f(x) = Bx$, $P_2(x) = f(x)$ and $P_2'(x) = B \equiv f'(x)$ *at all points* x. This implies

$$\begin{aligned} P_2'(x) = B &= \frac{c_{-1}f_{-1} + c_0 f_0 + c_1 f_1}{\Delta x} \\ &= \frac{c_{-1}B(-\Delta x) + c_1 B(\Delta x)}{\Delta x} \\ &= \frac{B\Delta x(c_1 - c_{-1})}{\Delta x} \end{aligned}$$

  Thus $c_1 - c_{-1} = 1$.

- If $B = C = 0$, then $f(x) = Ax^2$, $P_2(x) = f(x)$ and $P_2'(x) = 2Ax \equiv f'(x)$ *at all points* x. This implies

$$\begin{aligned} P_2'(\bar{x}) = P_2'(0) = 0 &= \frac{c_{-1}f_{-1} + c_0 f_0 + c_1 f_1}{\Delta x} \\ &= \frac{c_{-1}A(-\Delta x)^2 + c_1 A(\Delta x)^2}{\Delta x} \\ &= A\Delta x(c_{-1} + c_1) \end{aligned}$$

This must be true for all A and Δx, so $c_{-1} + c_1 = 0$.

To summarize, we have found 3 equations that unknowns c_{-1}, c_0, and c_1 must simultaneously satisfy:

$$\begin{aligned} c_{-1} + c_0 + c_1 &= 0 \\ c_1 - c_{-1} &= 1 \\ c_{-1} + c_1 &= 0 \end{aligned}$$

Since $c_{-1} + c_1 = 0$, we know that $c_{-1} + c_0 + c_1 = 0 \implies c_0 = 0$. Also $c_{-1} + c_1 = 0 \implies c_{-1} = -c_1$, and plugging this into the second equation yields $c_1 - (-c_1) = 1 \implies c_1 = \frac{1}{2}$. And finally, $c_{-1} + c_1 = 0 = c_{-1} + \frac{1}{2} \implies c_{-1} = -\frac{1}{2}$.

So our finite-difference approximation to $f'(\bar{x})$ is given by

$$f'(\bar{x}) = \frac{f_1 - f_{-1}}{2\Delta x} = \frac{f(\bar{x} + \Delta x) - f(\bar{x} - \Delta x)}{2\Delta x},$$

which is exactly the same as the approximation we found above by inverting the 3×3 Vandermonde matrix.

As you might imagine, we could improve the accuracy of our derivative by fitting the function to a higher-degree polynomial... so long as the underlying function is smooth, and well-enough sampled in neighboring points. Equivalently, the error of the derivative written in this way should be related to the order of the polynomial we've fitted to it, as well as Δx, the spacing over which the function is sampled.

Finally, note that this finite-difference approximation also immediately applies to the *partial* derivative $\partial_x f(x, y, z, ...)$, since when taking partial derivatives with respect to an independent variable, we treat other independent variables *as constants*.

6.1.2 Finite Difference Coefficients from Taylor Series; Determining Truncation Error

The Taylor series expansion of a function provides not only a much more efficient way of deriving finite difference coefficients, but also immediate estimates for the error of the finite difference derivative.

Suppose we want to evaluate the finite difference derivative of $f(x_i)$ using data at points $f(x_{i,i\pm1})$ only. The Taylor expansion of the function at each point is given by:

$$\begin{aligned} f_{i-1} &= f_i + f_i'(-\Delta x) + f_i''\frac{(-\Delta x)^2}{2!} + f_i'''\frac{(-\Delta x)^3}{3!} + ... \\ f_i &\equiv f_i \\ f_{i+1} &= f_i + f_i'(\Delta x) + f_i''\frac{(\Delta x)^2}{2!} + f_i'''\frac{(\Delta x)^3}{3!} + ... \end{aligned}$$

Combining these equations, we find that

$$\begin{aligned} f_{i+1} - f_{i-1} &= 2f_i'(\Delta x) + 2f_i'''\frac{(\Delta x)^3}{3!} + ... \\ \implies f_i' &= \frac{f_{i+1} - f_{i-1}}{2\Delta x} + f_i'''\frac{(\Delta x)^2}{3!} + ... \end{aligned}$$

So we have found that our approximation involving a second-order polynomial fit has a dominant error term that is proportional to $f_i'''(\Delta x)^2$. f_i''' is simply the exact third derivative of f at x_i, so as we decrease Δx (increasing our

numerical resolution), we can expect in the worst case that the error will drop off as $(\Delta x)^2$, provided the underlying function is smooth and is not too sparsely sampled.

But why can we neglect higher-order terms? First, the denominator of higher-order terms drops rapidly, at a rate proportional to $\frac{1}{n!}$. Second, for a function that is sufficiently well-sampled (beyond Nyquist), it will vary on length scales $L \gg \Delta x$, so terms like

$$f_i'''(\Delta x)^2 \sim f_i \frac{\Delta x^2}{L^3} \ll \frac{f_i}{L} \sim f_i'$$

will fall off very rapidly as the order of the derivative increases. Again, we assume that the underlying function is smooth and does not vary significantly over the scale of Δx. If this is not the case, high-order derivatives of the function may be very large and the Taylor series expansion may not converge.

Provided these assumptions are met, the Taylor series will converge so rapidly that we will usually say

$$\text{the error } E(\Delta x) \underset{\sim}{\propto} (\Delta x)^2, \text{ so to } \textit{very good approximation } E \approx k(\Delta x)^2,$$

for some proportionality constant k.

Because the leading-order error term is very nearly proportional to Δx^2, we say that this finite difference representation of the derivative

$$f_i' \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x} + k(\Delta x)^2$$

is **second-order** accurate in Δx. Since the error in this approximation comes from the fact that we have **truncated** the Taylor series at some finite order, we refer to this error as **truncation error**.

Next, let's derive the finite difference expression for f_i''. Again,

$$\begin{aligned} f_{i-1} &= f_i + f_i'(-\Delta x) + f_i'' \frac{(-\Delta x)^2}{2!} + f_i''' \frac{(-\Delta x)^3}{3!} + ... \\ f_i &\equiv f_i \\ f_{i+1} &= f_i + f_i'(\Delta x) + f_i'' \frac{(\Delta x)^2}{2!} + f_i''' \frac{(\Delta x)^3}{3!} + ... \end{aligned}$$

Our goal is to have f_i'' on the left-hand side, and only terms including f_i and error terms not depending on f_i' on the right-hand side.

Notice that $f_{i+1} + f_{i-1}$ neatly cancels the f' term, so we end up with

$$\begin{aligned} f_{i+1} + f_{i-1} &= 2f_i + 2f_i'' \frac{(-\Delta x)^2}{2!} + 2f_i^{(4)} \frac{(-\Delta x)^4}{4!} + ... \\ \implies f_i'' &= \frac{f_{i+1} - 2f_i + f_{i-1}}{(\Delta x)^2} + 2f_i^{(4)} \frac{(\Delta x)^2}{4!} + ... \end{aligned}$$

Notice again that our second derivative is second-order accurate in truncation error. So we say that the finite difference approximation

$$f_i'' \approx \frac{f_{i+1} - 2f_i + f_{i-1}}{(\Delta x)^2} + c(\Delta x)^2$$

is second-order accurate in grid spacing Δx, since the error term is proportional to Δx^2.

6.2 Spectral Techniques

Spectral techniques employ *basis functions* $\phi_n(x)$ to approximate the solution $f(x)$ of a PDE:

$$f(x) \approx \sum_{n=0}^{N} a_n \phi_n(x)$$

Note that $\phi_n(x)$ could be sines/cosines, Chebyshev polynomials, Bessel functions, just to name a few.

For Fourier series, to get a_n, use the fact that sines and cosines are orthogonal functions:

$$\int f(x)\phi_m(x)dx \approx \sum_{n=0}^{N} a_n \int \phi_n(x)\phi_m(x)dx,$$

where $\int \phi_n(x)\phi_m(x)dx = \delta_{nm}$ for properly normalized basis functions $\phi_m(x)$. Thus we get

$$a_n \approx \int f(x)\phi_m(x)dx$$

If we have a numerical grid of M points, performing a single integral numerically requires the sum over M points, and we have a total of N coefficients, meaning that to compute all coefficients requires the summation of $M \times N$ terms. For sines and cosines, $M = N$, so the computational cost of spectral methods using direct summation goes like the number of grid points N, squared.

For sines and cosines, it turns out that this operation can be made much faster via the Fast Fourier Transform (FFT), which reduces the cost to $N \ln N$.

Thus spectral methods turn out to be often quite efficient; as the number of gridpoints (also called collocation points) increases linearly, the error drops *exponentially*, provided the solution is smooth and sufficiently sampled with the chosen number of gridpoints.

MATH 522, Spring 2019 Notes

Chapter 7: Scientific Notation, Significant Digits, and Relative Error

7.1 Scientific Notation

Proper Scientific Notation

Proper scientific notation rewrites real numbers in the form

$$a.bcdefghij \times 10^X,$$

where a–j denote individual digits of the **mantissa** or **significand**, such that $a \neq 0$ unless *all digits are zero*, and b–j are all decimal digits (e.g., $b = 2$, $c = 8$). X denotes the **exponent**, which can be any integer. In this class, the real number would generally correspond to some measure, like distance, time, pressure, energy, power, etc.
The total number of digits given in the significand in scientific notation must correspond to the number of **significant digits** in the number we are given.

The number `131,511,234` is written as 1.31511234×10^8 in scientific notation. While 0.131511234×10^9 would represent the same number, it is *not* in proper scientific notation (because $a \neq 0$ unless *all digits are zero*).

As another example, the number `0.000034100` would be 3.4100×10^{-5} in scientific notation. Notice that we kept the two zeroes after the 1 because these are assumed to be significant; the number `0.000034100` is assumed to have 5 significant digits.

Additionally, the number `1020000` would be 1.02×10^6 in scientific notation. If we wished to indicate that the zeroes after the 2 *were* significant, we would add a decimal point after the last zero; `1020000.` would be 1.020000×10^6 in scientific notation.

Finally, the number `1.5121` would be 1.5121×10^0 in scientific notation; notice the exponent is zero in this case.

In this class, we will make use of scientific notation both in the context of measurements, as well as in the context of performing fixed-precision arithmetic on the computer. We review the former in Sec. 7.1.1, and the latter in Sec. 7.1.2.

7.1.1 The Use of Scientific Notation for Measurements

Scientific notation is typically used to record measurements in science and engineering contexts, so that **the number of significant digits is related to the precision of our measuring device**. Suppose we have a standard ruler. With this ruler, we might be able to measure the thickness of a dictionary to a tolerance of 1 millimeter (mm), or equivalently, 0.1 cm. For example, our measurement with the ruler may indicate that the thickness is 11.0 cm.

Now suppose we wish to construct a bookshelf that can hold exactly 1,000 dictionaries in the standard position. How long must the bookshelf be? Because our measuring device is limited to the 1 millimeter tolerance, it would be *meaningless* and *misleading* to claim that, based on our measurement, the bookshelf must be exactly 11,000 cm in length. So it is very important that we keep track of our measurement tolerances, and **scientific notation** provides an easy means to accomplish this.

Being comfortable with scientific notation will be essential for succeeding as a professional in this field. **If you are rusty, you are strongly encouraged to next work through all material and quizzes on Scientific Notation & Significant Figures in** `https://tinyurl.com/sigdigitsreview`.

7.1.2 Scientific Notation on the Computer

As in our example with the books, the number of significant digits generally corresponds to the precision of our measuring device. We use computers in this class not to measure physical quantities directly, but to solve mathematical problems. Computers represent our numerical solutions and perform all arithmetic in scientific notation.

Computer scientific notation is unique in that a *fixed number* of significant digits is kept in all arithmetic operations, regardless of the number's relation to actual measurements. *Therefore we must be careful when interpreting the results our computer gives us, as the computer does not obey the standard rules of arithmetic with scientific notation.* For example, if our computer could only store three significant digits and we were to evaluate $7.13 \times 10^4 - 7.11 \times 10^4$, the result would be $2.XY \times 10^4$, where XY could be any two digit number. In other words, a computer that stores three significant digits will always give the result to three significant digits, despite the fact that in this case, the result is actually only valid to one significant digit.

This is quite different from the normal use of scientific notation for measurements, in which our measurement apparatus might be able to provide a precision of $\pm 0.01 \times 10^4$, meaning that the number 1.11×10^3 would contain a digit of significance *beyond* the capabilities of the measuring device.

In addition, instead of writing the exponent symbols $\times 10^X$ for all numbers, computers adopt the `eX` symbol. That is to say, instead of outputting 1.31041×10^{-21}, the number `1.31041e-21` will be output.

Examples of Computer Scientific Notation

Suppose we have a computer that can only store **three** significant digits in the mantissa in scientific notation. Assume for the purpose of this problem that there is no limit on the range of the exponent. Evaluate the following expressions, keeping the number of significant digits fixed at three. Also assume that the computer will completely ignore digits beyond the third significant digit; this is equivalent to always rounding down in any arithmetic operation. Your answer must be in proper computer scientific notation and correclty.

- `1.49e-22 + 3.11e-21`: The answer is `3.25e-21`; we were told to ignore the all significant digits beyond the third.
- `1.49e22 + 3.11e2`: The answer is `1.49e22`; we were told to ignore all significant digits beyond the third. Notice that `3.11e2` is completely insignificant and is therefore *ignored*.

7.2 Quantifying Numerical Error

When we solve mathematical problems on the computer, we typically do so in a way that *approximates* the exact solution to some number of significant digits, which might be impossible or impractical to compute by hand. Determining whether we can trust the computer's solution *is a central focus of this class*, and depends entirely on how much error we are willing to accept in our (typically) approximate solution. So to succeed in this class and as a professional in the field requires that we understand the error in our approximate numerical solution.

One quite useful strategy for measuring the error in our numerical solution is to force our numerical code to solve a problem that we can solve **exactly** by hand. Let's suppose our numerical solution gives the result 1.0229×10^3 seconds, and we find by hand the exact solution to be 1.0220×10^3 seconds.

So what is the error in the numerical solution?

Definition: The **Absolute Error**, corresponding to the absolute difference between numbers n_1 and n_2 is given by

$$E_{\text{Abs}} \equiv |n_1 - n_2|,$$

so for our example, the absolute error between 1.0220×10^3 seconds and 1.0229×10^3 seconds is

$$E_{\text{Abs}} = |1.0220 \times 10^3 - 1.0229 \times 10^3| \text{ seconds} = 0.0009 \times 10^3 \text{ seconds} = 9 \times 10^{-1} \text{ seconds}$$

Notice that the numbers going into this expression are accurate to *five* significant digits, but after the subtraction, we have a number that contains only *one* significant digit of precision. **Definition:** A reduction in the number of significant digits due to an addition or subtraction is known as a **loss of significance**, a **catastrophic cancellation**, or a **catastrophic subtraction**.

We must be very careful when a catastrophic cancellation occurs when doing arithmetic on the computer, because the computer will by default store a *fixed* number of significant digits, regardless of whether a catastrophic cancellation has occurred. For the example above, a computer might very well give us the answer 9.0129×10^{-1}. It is *not* the computer's responsibility to tell us that this number is only valid to a single significant digit; we must be able to figure this out on our own!

While absolute error can be a useful measure, we are often more interested in the *percent error* or *relative error*, which as we will find can give us the number of significant digits to which we can trust our answer. **Definition:** The **relative error** is the absolute difference, rescaled by the *magnitude* of the numbers $|n_1|$ and/or $|n_2|$, and the **percent error** is the relative error expressed in percentage.

There are at least two strategies for computing relative error, and both give similar results for this case, though only the second provides a means for directly and unambiguously computing relative error used when an exact solution is unknown. **Definition:** When the exact solution is unknown, the relative error might instead be referred to as the **relative difference**.

The first strategy for computing relative error compares two numbers n_1 and n_2 and scales the difference by the magnitude (absolute value) of n_2:

$$E_{\text{Rel},1} = \left| \frac{n_2 - n_1}{n_2} \right|.$$

Notice that this measure emphasizes n_2 as setting the magnitude, so if the exact value is known, we set n_2 to that value, and n_1 as the numerical solution. When n_2 and n_1 agree to more than a couple of significant digits, the choice of n_1 or n_2 in the denominator *will not significantly change the relative error*. You should be able to prove this to yourself.

The second strategy for computing relative error could be referred to instead as the *relative difference*, as this strategy is used primarily when an exact solution is unknown, and we are interested only in how well two approximate solutions agree. This second definition of relative error, i.e., relative difference, treats the magnitudes of n_1 and n_2 equally, by replacing the $|n_2|$ in the denominator by the average of $|n_1|$ and $|n_2|$:

$$E_{\text{Rel},2} = \frac{|n_1 - n_2|}{(|n_1| + |n_2|)/2}.$$

Applying both definitions to our two numbers (assuming 1.0220×10^3 is the "exact" value) yields results that agree to all significant digits:

$$\begin{aligned} E_{\text{Rel},1} &= \frac{0.0009 \times 10^3}{1.0220 \times 10^3} = 9 \times 10^{-4} = 9 \times 10^{-2}\% \\ E_{\text{Rel},2} &= \frac{0.0009 \times 10^3}{1.02245 \times 10^3} = 9 \times 10^{-4} = 9 \times 10^{-2}\%. \end{aligned}$$

Notice that we only kept one significant digit in our result, because 0.0009×10^3 is only known to one significant digit. Often Relative Error is used interchangeably with Percent Error; in this case, the Percent Error is $9 \times 10^{-2}\% = 0.09\%$.

Now consider a very basic question: To how many significant digits do two numbers agree? For example, suppose the numbers are 1.0220×10^3 and 1.0229×10^3? Clearly we would say they agree up to and including the fourth significant digit, so our answer is 4. But what about 1.0220×10^3 and 1.0221×10^3? Clearly they agree better, so it seems reasonable to say that the number of digits of agreement *does not need to be an integer*! In the former case, we'd say that the two numbers agree to *between 4 and 5 digits, but closer to 4*, and in the latter case, we'd say that the two numbers agree to *between 4 and 5 digits, but closer to 5.*

It is easy to count the number of digits of agreement between two numbers by eye, but how could we teach a computer to do this automatically? We could attempt a complicated algorithm that manually scans the digits of the number, seeing that, e.g., 100. minutes $= 1.00 \times 10^2$ minutes and 101 minutes$= 1.01 \times 10^2$ minutes agree to almost 3 significant digits. But notice that if we chose a different unit of time, say "ticks", where 100. minutes=99.0 ticks, then the two measurements would be 99.0 ticks and 100.0 ticks. Notice that the two numbers differ by only 1 part in 100, just like 100 and 101. Clearly we do not wish to define the number of digits of agreement based on our choice of unit, so this notion of counting digits of agreement is surely wrong, as it would depend on our number base and our choice of unit.

How do we solve this problem?

Consider the fact that the relative error between 1.0220×10^3 and 1.0229×10^3 is 9×10^{-4}, and these numbers agree to about 4 significant digits.

Similarly, the relative error between 1.02200×10^3 and 1.02209×10^3 is 9×10^{-5}, and these numbers agree to about 5 significant digits.

So it seems there is a pattern: look at the *exponent* of the relative error to get approximately the correct number of digits of agreement. What mathematical expression would convert 9×10^{-5} to a number that is nearly 5? If we can answer this question, then we can program a computer to give us the correct number of significant digits of agreement, SDA.

Based on this pattern, we can see that SDA must satisfy

$$SDA = -\log_{10} E_{\text{Rel}} + 1$$

Using this expression, we find that 1.01000 and 1.00000 yield an SDA of approximately 3, as expected. In addition, notice that 0.9900000 and 1.0000000 will give about 2.998 significant digits—i.e., almost 3 significant digits of agreement as well.

In this course, when you are asked to compute the logarithmic relative error between two numbers $\log_{10} E_{\text{Rel}}$, it is very important that you remember the very close relationship between this difference and the number of significant digits of agreement SDA.

Exercises:

1. How would you write the number `1020000` in proper scientific notation if the first *four* digits (starting with the 1 on the left) were significant?
2. Use scientific notation to write the thickness of the dictionary (given in the first example of the chapter) based on the 1 millimeter tolerance measurement. Based on the fact that digits after the least (rightmost) significant digit *are completely unknown*, write the range of widths that the bookshelf might require. Assume that the ruler always rounds *down* to the nearest millimeter; i.e., if the book's thickness falls between 11.00 cm and 11.09999... cm, we round to 11.0 cm.
3. Suppose we measure the thickness of a dictionary in our laboratory to be 11.0 cm, and our colleagues in Austria measure their lab notebook to be 4.5 cm thick. How many significant digits do we obtain in our measurement? How many significant digits do our colleagues measure? Assuming that we use the same ruler as described in the previous exercise, if we were to stack the dictionary and lab notebook atop one another, to how many significant digits do we know the total height, based only on our current measurements?
4. To how many significant digits do 1.023151×10^{-30} and $1.\bar{0} \times 10^{-30}$ (i.e., the exact number 10^{-30}) agree? Your answer must be in the form "Between N and $N+1$ significant digits", where you must fill in the integers N and $N+1$.
5. To how many significant digits do $n_1 = 1.023151 \times 10^{-30}$ and $n_2 = 0.000000 \times 10^{-30}$ agree?

Solutions:

1. 1.020×10^6
2. Thickness of dictionary: 1.10×10^1 cm. Since digits after the least significant digit *are completely unknown*, the width of the bookshelf will range from 1.10×10^4 cm to $1.10\bar{9} \times 10^4$ cm$= 1.11 \times 10^4$ cm. So to be safe, we should construct our bookshelf to be 1.11×10^4 cm in width.
3. *How many significant digits do we obtain in our measurement?* Answer: 3. *How many significant digits do our colleagues measure?* Answer: 2. *Assuming that we use the same ruler as described in the previous exercise, if we were to stack the dictionary and lab notebook atop one another, to how many significant digits do we know the total height, based only on our current measurements?* Answer: Between 2 and 3, but closer to 3. Can use our measure of significant digits to get a better idea.
4. Between 2 and 3 significant digits of agreement.
5. 0 significant digits of agreement. Must use $E_{\text{rel},2}$ here. Notice that we could simply swap n_1 and n_2 and use $E_{\text{rel},1}$ instead; there is nothing special about assigning zero to n_1 instead of n_2.

MATH 522, Spring 2019 Notes

Chapter 8: Number Storage and Arithmetic on the Computer

Great resource: `http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`

Computers store all information, including numbers, in a binary number base system consisting of ones and zeroes. So when doing arithmetic on the computer, we must be very cautious because computers allocate a limited number of binary digits to any given number.

In practice, it is rare that you would need to convert from decimal to binary format, but unless you understand *how* numbers are stored in binary format on the computer, you will sometimes be very confused by your computer's results. Thus the focus in this section will be on better understanding the limitations of finite precision arithmetic and how these limitations can sometimes be addressed.

In general, numbers on the computer are stored in one of two basic formats: *Integers* and *Floating Point*. Section 8.1 reviews the former and Sec. 8.2 the latter.

8.1 Integers on the Computer

In most modern programming languages, integers are a *type* of number that get exactly 4 *bytes* or 32 *bits* (i.e., a set of 32 ones and zeroes) of storage space. This means that 2^{32} integers n can be represented using this data type, typically all numbers n satisfying $-2^{31} \leq n \leq 2^{31}-1$ (using **two's complement**: `http://tinyurl.com/twoscomp`), meaning that integers from about -2.147×10^9 to $+2.147 \times 10^9$ can be represented exactly.

Integer arithmetic on computers will be *exact* as long as no part of the calculation exceeds the bounds of storage allocated to an integer in your given calculation. When this happens the results will usually be complete nonsense.

So we must either have foresight that no number will exist beyond these bounds before jumping in to a calculation using this data type, or regularly check that we are getting close. Otherwise our results will in general be untrustworthy.

Words of caution

Before doing integer arithmetic, first check how many bytes are allocated to the "integer type" in your chosen programming environment. Different programming environments adopt different standards for integer types. For example, in some implementations of C, an integer is defined as having 2 *bytes* or 16 bits of storage space (there are 8 bits in a byte). For 16 bits of storage, an integer n spans only the much more limited range $-2^{15} \leq n \leq 2^{15}-1$, or $-32{,}768$ to $32{,}767$.

16 or 32-bit integer calculations can be useful for very basic arithmetic. Unfortunately this is a very limited data type. For example, in the case of number theory we can contend with absolutely gigantic integers. What are we to do in this case? The answer is to use so-called "arbitrary precision arithmetic", which is built-in to Matlab and some symbolic calculators like Mathematica. For those of us using compiled languages like C/C++/FORTRAN, arbitrary precision arithmetic libraries exist and can be used for this purpose.

8.1.1 Exercises

Exercises:

1. Extending the pattern given in the notes when using two's complement, what range of *unsigned* integers n will be stored in 6, 8, and 10 bits?
2. Repeat the above problem, but for *signed* integers.
3. Assuming the following bits are ordered such that the rightmost bit is the 2^0 column, the second-to-rightmost bit is the 2^1 column, etc., convert the following 6-bit binary numbers to unsigned integers in base-10.
   (a) 010011
   (b) 100000
   (c) 011111
   (d) 111110
   (e) 011101
4. Assume that we have an integer format in which the leftmost bit sets the sign of the integer, where the number is negative if the bit is 1 and positive otherwise. Convert the following 6-bit binary numbers in this format to signed integers in base-10. For all bits except the leftmost bit, assume the same bit ordering as in the first problem.
   (a) 010011
   (b) 100000
   (c) 111111

Solutions to Exercises:

1. $0 \leq n \leq 63$, $0 \leq n \leq 255$, and $0 \leq n \leq 1023$, respectively
2. $-32 \leq n \leq 31$, $-128 \leq n \leq 127$, and $-512 \leq n \leq 511$, respectively.
3. (a) through (e): 19, 32, 31, 62, and 29, respectively.
4. (a) through (c): 19, -0 (signed zero), and -31, respectively.

8.2 Floating Point Numbers on the Computer

But what are we to do if our calculations involve rational, irrational, real, or complex numbers that may span many orders of magnitude and do not need to be exact? This is quite often the case in science and engineering applications for which we use computers to model systems for which we do not have exact measurements, and for which exactitude is therefore not a strict requirement. In this case, your best bet be to use numbers stored in floating point format, where the "point" denotes the decimal point, which "floats" according to the exponent of the number. Proper scientific notation, described in Chapter 7, is one example of a floating point number format.

How floating point numbers are stored is very easy to understand if you are familiar with scientific notation. For example, suppose we measure some event to be 154,132,000 nanoseconds in duration using a clock that can only measure to the nearest 1,000 nanosecond increment. We conclude that our clock measures the duration of this event to six significant digits. This measurement can be represented in scientific notation as 1.54132×10^8 nanoseconds.

Definitions: Notice that 1.54132×10^8 consists of two numbers; the number before the $\times$ and the number in the exponent above the 10. The former is called the **mantissa** or **significand**, which in this case is stored to 6 significant digits and the latter the **exponent**. So on a base-10 computer, storing such a number would require a total of 6+1 "deci-bits" (i.e., each "deci-bit" can store numbers zero through nine; notice that if the first "deci-bit" can be zero, in order to denote the number zero).

However, computers are base-2 machines, storing numbers as sets of ones and zeroes. We have reviewed how computers store integers, but how can they store floating point numbers?

Definition: In this class, we will make extensive use of **computer scientific notation**, which takes the form *mantissa*`e`*exponent*. So for example, the number 9.183×10^{-5} is written `9.183e-5` in computer scientific notation.

8.2.1 Floating-Point Arithmetic with Fixed Storage: Base-10 Example

Since we are more familiar with numbers written in *decimal* (base-10) and not *binary* (base-2), let us first imagine a computer that stores floating point numbers using a limited number of base-10 "deci-bits":

Example: Base-10 computer that stores 2 significant digits

Imagine you have a *base-10* computer number format that can only store 2 significant **decimal** digits of precision in the mantissa (with an added sign bit), and can only store integer exponents in the range $[-5, +4]$ (because there are 10 digits between -5 and 4, inclusive). In addition,

- **Definition**: **Overflow**: Numbers greater in magnitude than the largest representable number in a finite-precision format will be evaluated to `Infinity` if the number is positive or `-Infinity` if the number is negative. For this number format, expressions that — when evaluated — result in numbers greater than to `9.9e4` or less than `-9.9e4` are evaluated to `+Infinity` and `-Infinity`, respectively.
- **Definition**: **Underflow**: Positive numbers smaller than the smallest nonzero number will be evaluated as zero. Negative numbers larger than the smallest nonzero number will be evaluated as negative zero. For this number format, numbers smaller than `1.0e-5` but larger than zero will be evaluated to `0.0e0`. Numbers larger than `-1.0e-5` but smaller than zero will be evaluated to `-0.0e0`.

1. What is the largest number ϵ storable in a number format such that $1 + \epsilon$ will evaluate to 1? (In this class, we will define this ϵ as **machine epsilon**. Warning: There are multiple definitions of "machine epsilon" in the literature; this is the one we will consistently use.)
2. Next evaluate the following expressions using this computer number format; you must abide by the restrictions of the computer number format in your calculations, and your answers must be in the same number format:
   (a) `1.0e0 + 3.1e-1`
   (b) `4.5e-3 + 5.5e-1`
   (c) `1.0e-3*5.5e4`
   (d) `1.0e3/5.5e-5`

Solutions to Exercises:

1. `9.9e-2`
2.
   (a) `1.3e0`
   (b) `5.5e-1`. Let's rewrite the expression as `4.5e-3 + 5.5e-1 = 5.5e-1*(1 + 4.5e-3/5.5e-1)`. Since `4.5e-3/5.5e-1`$< \epsilon_m$, this will evaluate to `5.5e-1`.
   (c) `5.5e1`. Note that `5.5e5` is representable in this number format.
   (d) `Infinity`. Note that `5.5e-5` is representable in this number format, and `1/5.5e-5`$\approx$`1.8e4` is also representable. But when multiplying `1.8e4` by `1.0e3`, we get a number that exceeds the bounds of our number format.

8.2.2 Double Precision Floating Point Arithmetic

Computers store floating point numbers in base-2, most commonly using either 32 or 64 bits, with the former being called *single precision* and the latter *double precision*. In this class, and in most scientific, mathematical, and engineering contexts, we will exclusively use double precision.

Source (vetted):
`https://en.wikipedia.org/w/index.php?title=Double-precision_floating-point_format&oldid=688044028`

Double Precision Floating Point Format

According to the IEEE 754 technical standard, double precision floating point numbers are stored in 64 bits on the computer as follows:

- Bit 63: Sign bit S
- Bits 62–52: exponent (11 bits) X.
- Bits 51–0: mantissa or significand (52 bits) M

With few exceptions (see box below), double precision numbers N are represented as follows, given mantissa bits b_{51}–b_0:

$$N = (-1)^S (1.b_{51}b_{50}...b_0) \times 2^{X-1023}, \quad \text{where } 1.b_{51}b_{50}...b_0 \text{ is a } \textit{binary} \text{ floating point number, } \textit{not} \text{ a decimal}$$

or equivalently,

$$N = (-1)^S \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right) \times 2^{X-1023},$$

where

$$X = \sum_{i=0}^{10} b_{52+i} 2^{+i}.$$

Exceptions to the above expressions

The exponent X, having 11 bits, can represent $2^{11} = 2048$ unique integers, of which only the 2046 numbers between -1022 ($X = 1$) to 1023 ($X = 2046$) are consistent with the above definition for N. The exceptional cases $X = 0$ and $X = 2047$ are as follows:

- The exponent $X = 0$ and a mantissa of
  - all zeroes corresponds to $N = 0$. If the sign bit is set to 1, then $N = -0$, the signed zero.
  - anything other than zero typically corresponds to $N = \pm 0$ (depending on the sign bit), but this is implementation-dependent. Sometimes (rarely) these bits are used to store nonzero numbers smaller in absolute magnitude than roughly $2^{-1022} \approx 10^{-308}$.
- The exponent $X = 2047$ and a mantissa of
  - all zeroes corresponds to ±Infinity, where the sign of the infinity is given by the sign bit.
  - any value except zero corresponds to `NaN` (Not a Number, which is output if an undefined arithmetic operation, like 1/0, is attempted).

8.2.2.1 Exactly Representable Numbers in Double Precision

In general we should expect a number to be represented to only 15–16 significant digits in double precision. For example, the decimal number $0.1 \equiv \frac{1}{10}$ is known exactly in base-10 floating point as

$$\begin{aligned} 0.1 &\equiv \frac{0}{10^0} + \frac{1}{10^1} + \frac{0}{10^2} + \sum_{n=3}^{\infty} \frac{0}{10^n} \\ &= 1.\bar{0} \times 10^{-1}. \end{aligned}$$

But when we convert this number to base-2 floating point, notice that we'll have to represent it as

$$0.1 = \frac{b_0}{2^0} + \frac{b_1}{2^1} + \frac{b_2}{2^2} + ...,$$

which happens to be a *repeating binary decimal!* And since double precision only stores 52 significant bits, the repeating binary decimal is artificially stopped at the 52^{nd} bit, yielding an error that is about 1 part in 10^{16}.

But what numbers can we store exactly in double precision? Clearly powers of two like $\frac{1}{2} = 1 \times 2^{-1}$, or $\frac{1}{2^{1000}} = 1 \times 2^{-1000}$ are exactly representable. Let's dig further with an example:

Example #1

Consider the double precision floating point number `0:1[10 zeroes]:1[51 zeroes]`. To what does this correspond in decimal? Is the number exact?

- The sign bit is 0, so this is a positive number.
- Regarding the exponent, Bit 62 is 1, all others zero, so $X = 2^{10} = 1024$, and our exponent term is $2^{1024-1023} = 2^1 = 2$.
- Regarding the mantissa, Bit 51 is 1, all others zero, so $\sum_{i=1}^{52} b_{52-i} 2^{-i} = 2^{-1} = 1/2$.

So our number is $+(1 + 1/2) * 2 = 3$, which is an exact number.

Next notice that if we set *all* significand bits to zero, then the number becomes $+(1 + 0) * 2 = 2$. If we then "flip" all the bits in the exponent, we'll have `0[10 ones]`, which is the same number when counting in binary on our fingers as if we held up all of our fingers: 1023. Thus the result would be 1. We already determined that the number zero is represented in double precision (recall that +0 is represented by all bits being zero). So we have demonstrated that integers 0, 1, 2, and 3 are all *exactly* representable in double precision. But we also know that if we flip the sign bit, we can immediately get -3, -2, and -1.

Therefore, we have proven that the *contiguous* set of integers $N \in [-3 : 3]$ is exactly representable in double precision.

Example #2

What is the largest range of contiguous integers that can be stored in double precision?

First let's consider the **largest** integer we could possibly store exactly in double precision. Clearly we'd just need to maximize the exponent at 1023, and set all the bits in the mantissa to 1's. This will result in the integer N_{large}, where

$$\begin{aligned} N_{\text{large}} &= (1 + 2^{-1} + 2^{-2} + ... + 2^{-52}) \times 2^{1023} \\ &= 2^{1023} + 2^{1022} + 2^{1021} + ... + 2^{1023-52}. \end{aligned}$$

The result is nothing more than the sum of some very large integers, and the result is an integer.

But what is the next integer smaller than N_{large} that can be represented exactly in double precision?

The next smaller integer should be the same double-precision number, but with the least significant bit flipped to zero:

$$N_{\text{next largest}} = (1 + 2^{-1} + 2^{-2} + ... + 2^{-51} + 0 \times 2^{-52}) \times 2^{1023}.$$

Notice that the difference between the largest and second-largest integer exactly representable in double precision is $N_{\text{largest}} - N_{\text{next largest}} = 2^{1023-52} \gg 1$, so these two integers *are not contiguous*. So what would be the largest integer such that the next largest integer is just one less?

Since the significand is a number that is 52 bits, and we multiply by 2^L, where L ranges from -1022 to 1023, how about we multiply the largest possible significand by 2^{52}? Then we'd have

$$\begin{aligned} N_{52} &= (1 + 2^{-1} + 2^{-2} + ... + 2^{-52}) \times 2^{52} \\ &= (2^{52} + 2^{51} + 2^{50} + ... + 1) \\ &= 2^{53} - 1 \approx 9 \times 10^{15}. \end{aligned}$$

Ah, but we can also represent 1×2^{53} exactly as well! Notice also that the next integer smaller than N_{52} would be

$$\begin{aligned} N_{52,\text{ next largest}} &= (1 + 2^{-1} + 2^{-2} + ... + 2^{-51} + 0 \times 2^{-52}) \times 2^{52} \\ &= (2^{52} + 2^{51} + 2^{50} + ... + 2^1 + 0) \\ &= N_{52} - 1. \end{aligned}$$

Next, you should be able to show that $2^{53} + 1$ is not an integer representable exactly in double precision. Thus we have found that $2^{53} - 1$ and 2^{53} are the largest two consecutive integers that can be stored exactly in double precision. You may also find it instructive to prove that $N_{52} - 2$ and $N_{52} - 3$ can also be stored exactly, and that the pattern continues all the way to $N = 0$. But we also know that if we flip the sign bit, we can get the negative versions of each of these integers.

Therefore double precision arithmetic can store the complete set of contiguous integers between -2^{53} ($\approx -9 \times 10^{16}$) and 2^{53} ($\approx 9 \times 10^{16}$), inclusive.

8.2.2.2 Summary: Limitations of Double Precision

Limitations of Double Precision

- **Definition**: **Machine epsilon** is the largest number ϵ representable in a given number format such that $1 + \epsilon = 1$. In double precision, this number is $2^{-53} \approx$`1.11e-16`. **Warning #1:** There is no IEEE 754 standard definition of machine epsilon; you fill find other texts define machine epsilon differently.[a] **Warning #2**: Different implementations of the IEEE 754 standard might behave differently near machine epsilon, yielding machine epsilons of `2.22e-16` or `4.44e-16` instead. **For simplicity, we will normally assume 16 decimal digits of significance for double precision.**
- **Magnitude of smallest nonzero number**: $2^{-1022} \approx$`2.22e-308`. We will typically round to 10^{-308} in this class.
- **Magnitude of largest non-infinite number**:

  $$2^{1023}(1 + 2^{-1} + 2^{-2} + ... + 2^{-51} + 2^{-52}) = 2^{1023} + 2^{1022} + ... + 2^{1023-52} \approx \texttt{1.8e308}$$

  We will typically round to 10^{+308} in this class.
- **Largest set of contiguous integers**: From $-2^{53} \approx$ `-9e16` and $2^{53} \approx$`9e16`.

[a] One common alternative definition is the smallest number in a given number format that, when added to 1 yields a number not equal to one. This is not the definition we will use in this class.

8.2.2.3 Loss of Significance

(References/Sources: `https://en.wikipedia.org/wiki/Loss_of_significance`
`http://math.stackexchange.com/questions/136634/rewriting-to-avoid-catastrophic-cancellation`)

Definition: Loss of significance or **catastrophic cancellation** occurs when significant digits are lost in an arithmetic operation. Suppose our number format stores 8 significant digits. Then

```
1.1204002e-1 - 1.1204000e-1
```

will yield `2.e-8`. However, because our number format stores 8 significant digits, *we can expect that for this case, the numbers after the first significant digit will be nonzero!*

Loss of significance can be a major problem when performing double precision arithmetic, because we typically assume that we can trust our double precision results to about 16 significant digits. Your computer will not warn you when a catastrophic cancellation has occurred, and doing so would greatly slow your calculation. We must therefore be able to identify when a loss of significance has occurred *by analyzing the expressions ourselves.* Typically this will involve adding `print()` statements to our code when we obtain results that are unusual. Numerical analysis can be hard work, but when we are successful, our efforts can be greatly rewarded!

Exercises:

You have added `print()` statements into your code to try and diagnose why your numerical solution (using double precision) appears to be strange. Below is a list of arithmetic operations that your code has performed, and you are to write the number of significant digits that remain after the following expressions are evaluated in double precision. You are to assume that double precision numbers input into all expressions are known to 16 significant digits.

1. `1.1204002e-1 - 1.1204000e-1` (same as example above, but in double precision)
2. `5.2101e-5 - 5.2101e-4`
3. `5.2111e-5 - 5.2101e-5`
4. `5.2111e-5 - 5.2101e-6`
5. `5.211104e-5 - 5.211101e-5`
6. `5.2101e-5 - 5.2101e-5`

Solutions to Exercises:

1. 7 significant digits are lost, so 9 remain.
2. No digits are lost, so 16 significant digits remain.
3. 3 significant digits are lost, so 13 remain.
4. No digits are lost, so 16 significant digits remain.
5. 6 significant digits are lost, so 10 significant digits remain.
6. There is exact cancellation in this case (each number is represented by exactly the same bits), so double precision will yield zero exactly, resulting in a *gain* of an *infinite* number of significant digits. In practice, this will happen rarely.

8.2.2.4 Fixing Loss of Significance in an Algorithm

Let us now consider an algorithm that suffers from a severe loss of significance in certain cases: the quadratic formula.

The quadratic formula solves for the roots x_1 and x_2 that solve the following quadratic polynomial

$$ax^2 + bx + c = 0.$$

The quadratic formula is given by

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

In pseudocode, we could represent this algorithm as follows:

Algorithm for Solving Quadratic Polynomials:

```
function quadratic(a, b, c, x)
   x[1] = ( -b + sqrt(b^2 - 4*a*c) ) / (2*a)
   x[2] = ( -b - sqrt(b^2 - 4*a*c) ) / (2*a)
end function
```

Notice that when $b^2 \gtrsim 10^{16} \times 4ac$, $\sqrt{b^2-4ac} \equiv b$ in double precision. Thus for the first root we would get $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} = \frac{-b+b}{2a} = 0$, which will in general be incorrect to *all* significant digits.

For example, consider the case in which $a = 1$, $b =$`1e10`, and $c =$`1e-10`. The exact roots are given by $x_1, x_2 = -10^{-20}, -10^{10}$, but the roots computed in double precision will be given by

$$\begin{aligned} x_1, x_2 &= \frac{-10^{10} \pm \sqrt{(10^{10})^2 - 4(1)(10^{-10})}}{2(1)} \\ &= \frac{-10^{10} \pm \sqrt{10^{20} - 4 \times 10^{-10}}}{2} \\ &= \frac{-10^{10} \pm \sqrt{10^{20}(1 - 4 \times 10^{-30})}}{2} \\ &= \frac{-10^{10} \pm \sqrt{10^{20}(1)}}{2} \\ &= 0,\ -10^{10}, \end{aligned}$$

where in the second-to-last step, we applied the fact that in double precision $(1-4\times10^{-30}) \equiv 1$, since $4\times10^{-30} < \epsilon_m$ (note that machine epsilon equally applies whether adding to *or subtracting from* 1). So we have found that x_1 is incorrect at *all* significant digits, while x_2 will be correct to about 16 significant digits. We conclude that was a *complete loss of significance* in computing x_1.

Sometimes, *but not always* there is an alternative strategy for getting around limitations of double precision arithmetic. In this case, it turns out that we can rewrite the quadratic formula for x_1 as follows:

$$\begin{aligned} x_1 &= \frac{-b+\sqrt{b^2-4ac}}{2a}\left(\frac{-b-\sqrt{b^2-4ac}}{-b-\sqrt{b^2-4ac}}\right) \\ &= \frac{b^2-(b^2-4ac)}{2a(-b-\sqrt{b^2-4ac})} \\ &= \frac{2c}{-b-\sqrt{b^2-4ac}} \end{aligned}$$

Now let's apply this example, but with this alternative strategy for computing the root of x_1:

$$\begin{aligned} x_1 &= \frac{2c}{-b-\sqrt{b^2-4ac}} \\ &= \frac{2(10^{-10})}{-(10^{10})-\sqrt{(10^{10})^2-4(1)(10^{-10})}} \\ &= \frac{2\times10^{-10}}{-10^{10}-\sqrt{10^{20}-4\times10^{-10}}} \\ &= \frac{2\times10^{-10}}{-10^{10}-\sqrt{10^{20}(1-4\times10^{-30})}} \\ &= \frac{2\times10^{-10}}{-10^{10}-\sqrt{10^{20}(1)}} \\ &= \frac{2\times10^{-10}}{-2\times10^{10}} \\ &= -10^{-20}, \end{aligned}$$

which will be consistent with the exact result to 16 significant digits. Notice that while we did not eliminate the loss of significance, we did prevent it from influencing our result.

Exercise:

Rewrite the above algorithm for solving quadratic polynomials to make it more robust for cases in which $b^2 \gtrsim 10^{16} \times 4ac$.

Solution:

```
function quadratic(a, b, c, x)
    if(b^2 > 1e16*4*a*c)
        x[1] = (2 c) / ( -b - sqrt(b^2 - 4*a*c) )
    else
        x[1] = ( -b + sqrt(b^2 - 4*a*c) ) / (2*a)
    end if
    x[2] = ( -b - sqrt(b^2 - 4*a*c) ) / (2*a)
end function
```

8.2.2.5 Tips for minimizing finite precision arithmetic errors

Here are a few tips for minimizing finite precision arithmetic errors on the computer.

1. Avoid obvious addition or subtraction of numbers that are many orders of magnitude different. This may require rescaling your problem so that extremely large or small exponents do not appear. (Top priority)
2. **Definition**: Choose numerical algorithms that have small **unavoidable errors**; i.e., algorithms that naturally avoid loss of significance (like the adjusted quadratic equation formula given above). Using an algorithm that has small unavoidable errors *is no guarantee* that catastrophic cancellation will not occur for certain inputs! (High priority)
3. **Definition: Roundoff error** is the build-up of numerical errors due to loss of significance. Once the above problems are addressed, try to minimize the number of floating point operations (i.e., generally more efficient algorithms have lower roundoff errors). Roundoff errors will generally grow at a rate between $\sqrt{N}\epsilon$ (due to the expected distance traveled from the origin in a random walk after N steps) and $N\epsilon$ (a non-random walk, adding a single ϵ error at each step), *unless the numerical algorithm exhibits chaotic behavior*. If chaotic behavior is observed, tiny perturbations due to roundoff errors can grow at a rate proportional to e^N. (Medium priority)

8.2.2.6 Tips for determining how many significant digits of agreement *SDA* (see Chapter 7) you can expect between double-precision and exact arithmetic

Note that this neglects guard digits.

1. If the final result is not an exactly representable number in double precision, then the maximum number of significant digits of agreement between the double-precision and exact results will be 16.
2. Evaluating the expression according to proper order-of-operation, check for arithmetic steps that go out of bounds (i.e., check for **underflow** or **overflow** errors) in double precision arithmetic. To this end, recall that the smallest nonzero number is roughly plus or minus `1e-308` and the largest non-infinite number has magnitude `1e+308`. If this happens, evaluate to zero or $\pm\infty$ as appropriate.
3. Check for catastrophic cancellation (a.k.a., loss of significance).
4. Check for numbers that are exactly representable by double precision (e.g., all integers between -2^{53} and 2^{53} (inclusive), as well as $\pm\frac{1}{2}$ to some power). If these exist, they are known to an infinite number of significant digits. If not, they are generally known to only 15–16 significant digits.
5. Dividing or subtracting two numbers that are guaranteed identical to all significant bits will yield *exactly* one or zero, respectively.
6. Transcendental functions like `sin()`, `cos()`, `log()`, and `log10()` are typically evaluated in double precision using a Taylor series approximation. So even if the input and result are both numbers exactly representable in double precision, you should generally expect only 16 significant digits of precision in the result.

8.2.3 Examples of Finite Precision Arithmetic Using Double Precision

Exercises:

When the following expressions are evaluated by the computer, **to how many significant decimal digits will the numerical result agree with the exact result?** Your answer will consist of a single integer (∞ is an acceptable answer). If the integer is finite and nonzero, your answer will be accepted if it is within 1 decimal digit of the exact answer.
We use computer scientific notation, such that, e.g., `5.63e22` $\equiv 5.63 \times 10^{22}$. For the purpose of this problem, apply IEEE 754 standard-compliant **double precision** arithmetic, assuming all given *integers* represented in integer or floating point format between -2^{53} and 2^{53} (i.e., integers between $\approx -9 \times 10^{15}$ and $\approx 9 \times 10^{15}$ inclusive) are **exact** (for example, `2.01e2` is exact), as well as powers of $\pm\frac{1}{2}$. Otherwise, assume that in double precision the number is only known to 16 significant digits, and that machine epsilon is `4e-16`.

1. `4 - 1`
2. `1e23/2.312e101 + 1e-20`
3. `1e20 - 1e200/1e-200`
4. `(1e-250/1e-250)`
5. `1.3512e-45 - 1.3512e-45`
6. `(1e55 - 1e35) + 1e55`
7. `1 - 1e-12`
8. `(1e120 * 1e120 * 1e120)`
9. `1.0/256.0`
10. `(2.51-2.50)/(2.51-2.50)`
11. `log10(1e-145) - 2e-200`
12. `5e-200*5e-2 + 2`
13. `7 + 2.4e-230/5e280`
14. `7 + 2.4e230/5e280`
15. `1/100`
16. `log10(2e-230) - 2e-200`
17. `(3.1415-3.14)/(3.1415-3.14)`
18. `1/1024`
19. `3.2e-200/5e-202 + 2`
20. `(5.0e-200 + 1)/(1.2e1+4e0)`

Additional problem set. Same instructions as previous problem. Solutions provided in class.

Additional Exercises:

1. `1e-1/1.0e-1`
2. `2e10 - 1e-100`
3. `4.34/2`
4. `1.0 + (5.0e33 - 1e-14)`
5. `8.0e200 + 1.0`
6. `(2.104e4 - 2.103e4)/1.0e1`

Solutions to Exercises:

1. `4 - 1`: 4 and 1 are exactly represented in double precision, and we have already proven that 3 is also represented exactly in double precision. The double-precision result agrees with the exact result to an infinite number of significant digits.
2. `1e23/2.312e101 + 1e-20`: Evaluating, we get about $0.5 \times 10^{23-101}$. $23 - 101 = -78$, so we get $0.5 \times 10^{-78} + 10^{-20}$. The first number in the sum is about 59 orders of magnitude less than the second. Additions in double precision are only sensitive to about 16 significant digits, so the first number is irrelevant and the result in double precision is 10^{-20}. *Ignoring the addition*, the number 10^{-20} cannot be exactly represented in double precision. Therefore this result is consistent with the exact solution to 16 significant digits.
3. `1e20 - 1e200/1e-200`: `1e200/1e-200=1e400`, which exceeds the bounds of double precision, thus we get Infinity, so the answer is zero digits of precision.
4. `(1e-250/1e-250): 1e-250/1e-250=1`. Note that even if `1e-250` is not exactly representable in double precision, its representation in bits is the same for both numerator and denominator. So imagine you have a string of bits divided by the same string of bits. You'll get 1 exactly, and 1 is exactly representable by double precision. Thus the answer is: the double-precision result is consistent with the exact result to ∞-many digits of precision.
5. `1.3512e-45 - 1.3512e-45`: Imagine you have a string of bits subtracting the same string of bits. You'll get 0 exactly, and 0 is exactly representable by double precision. Thus the answer is: the

double-precision result is consistent with the exact result to ∞-many digits of precision.

6. `(1e55 - 1e35) + 1e55`: Notice the catastrophic cancellation that occurs within the parentheses. So `(1e55 - 1e35) + 1e55 = 1e55 + 1e55 = 2e55`, which is consistent with the exact result to 16 significant digits.
7. `1 - 1e-12`: Notice that 1 is exact, but `1e-12` is not. There is no catastrophic cancellation here; we get an answer that is consistent with the exact result to 16 significant digits.
8. `(1e120 * 1e120) * 1e120`: The result, `1e360`, exceeds the bounds of double precision. Thus we get an overflow: Our answer will be correct to zero significant digits.
9. `1.0/256.0`: Both 1 and 256 are exact. Also, $1/256 = 1/2^8 = 2^{-8}$ is exactly representable in double precision as well. Thus the answer is: the double-precision result is consistent with the exact result to ∞-many digits of precision.
10. `(2.51-2.50)/(2.51-2.50)`: Let's look at the numerator first: `(2.51-2.50)`. Neither of these terms is an exact power of $\frac{1}{2}$ or an integer, and the result, 0.01 is also not an exact power of $\frac{1}{2}$. We will get 0.01 to 16 significant digits. But the denominator will result in exactly the same number: 0.01 to 16 significant digits. Therefore, our result will be identical strings of bits dividing each other: 1 exactly. So the answer is ∞.
11. `log10(1e-145) - 2e-200`: Note that `log10(1e-145)` will be computed via a Taylor series on the computer, so despite the fact that the exact result is an integer, -145, which can be exactly represented in double precision, the Taylor series approximation can only be trusted to 16 significant digits! So we get `-145 - 2e-200`. Notice the catastrophic cancellation on the second term. So the answer is `-1.45e2` to 16 significant digits.
12. `5e-200*5e-2 + 2`: Notice the first term evaluates to `25e-202=2.5e-201`, so we get catastrophic cancellation. Since 2 is exactly represented in double precision, the double precision result is consistent with the exact result to about 201 significant digits.
13. `7 + 2.4e-230/5e280`: 7 is exactly representable, and the expression `2.4e-230/5e280` will yield an underflow in double precision, giving a double precision result of 7 *exactly*. The exact result is 7 plus a perturbation at the 511th digit. Thus the double precision and exact results agree to about 511 significant digits.
14. `7 + 2.4e230/5e280`: 7 is exactly representable, and the expression `2.4e230/5e280` will yield a number that is of order `1e-51`. Recall that machine epsilon is defined as the largest ϵ such that $1 + \epsilon = 1$. In double precision machine epsilon is approximately `4e-16`, so `7+1e-51 ~ 7 (1 + 1e-52)` is 7 exactly. Thus the double precision and exact results agree to about 51 significant digits.
15. `1/100`: The division is not a power of 2, so the answer is 16 significant digits of agreement between double precision and the exact answer.
16. `log10(2e-230) - 2e-200`: `log10(2e-230)` is computed via Taylor series, thus will yield a double precision result that matches the exact result to 16 significant digits. The `-2e-200` will not affect the result. The answer is 16.
17. `(3.1415-3.14)/(3.1415-3.14)`: The numerator and denominator will evaluate to exactly the same bits, despite being consistent with the exact result to 16 significant digits. Thus the answer in double precision is 1 exactly, matching the exact result to ∞ significant digits.
18. `1/1024`: $1024 = 2^{10}$, so the expression is 2^{-10}, an exactly representable number in double precision that matches the exact result to ∞ significant digits.
19. `3.2e-200/5e-202 + 2`: `5e-202` and `3.2e-200` must be assumed to be known only to 16 significant digits, and so their ratio is a number of order `1e2`, known to only 16 significant digits. Adding `2` has no effect on the significance, so the double-precision result will be consistent with the exact solution to 16 significant digits.
20. `(5.0e-200 + 1)/(1.2e1+4e0)`: `(5.0e-200 + 1)` in double precision yields `1` exactly, which differs from the exact result at the 200th significant digit. The denominator will evaluate to an exact integer since `1.2e1` and `4e0` are both less than 2^{52} and there is no loss of significance. The double precision result then is `1/16`, which is an exact power of 2, yielding the number `1/16` exactly. This is consistent

with the exact result to about 199 significant digits.

DO NOT PRINT

Solutions to Additional Exercises:

1. `1e-1/1.0e-1`: The numerator and denominator are represented by exactly the same bits, so the double precision result is the number `1`, which is exactly representable and consistent with the exact result to Infinitely many significant digits.
2. `2e10 - 1e-100`: `2e10` $< 2^{52}$, so is exactly representable in double precision. `1e-100` is not exactly representable. There is a catastrophic cancellation in double precision, leading to the double precision result of `2e10`, which is exactly represented as an integer. The exact result is not an integer, and differs from the double precision result at the 110th significant digit. Thus the answer is 110.
3. `4.34/2`: `4.34` is not exactly representable in double precision, and it is known only to 16 significant digits. Thus dividing it by two will yield a number correct to only 16 significant digits.
4. `1 + (5.0e33 - 1e-14)`: `1` is exactly representable in double precision, but `5.0e20` and `1e-14` are represented only to 16 significant digits. `(5.0e20 - 1e-14)` will yield `5.0e20` to 16 significant digits due to catastrophic cancellation. Then `1 + (5.0e20)` will yield `5.0e20` to 16 significant digits, again due to loss of precision.
5. `8.0e200 + 1`: `8.0e200` is represented in double precision to 16 significant digits, and `1` is exact. The sum yields `8.0e200` to 16 significant digits due to loss of precision.
6. `(2.104e4 - 2.103e4)/1.0e1`: All numbers in the numerator and denominator are represented exactly in double precision. Evaluating numerator and denominator separately, we obtain `1.0e1/1.0e1`, an exactly representable number in double precision. Thus the double precision result agrees with the exact result to an infinite number of significant digits.

MATH 522, Spring 2019 Notes

Chapter 9: Lax Equivalence Theorem

Source: `http://ta.twi.tudelft.nl/users/vuik/wi3097/lecture6_consistency.pdf`

The Lax Equivalence Theorem

The Lax Equivalence Theorem states that when solving a linear PDE on the computer,

$$\textbf{Stability} + \textbf{Consistency} \equiv \textbf{Convergence}.$$

9.1 Basic Definitions

Let's next define each of these terms.

9.1.1 Stability

Stability refers to much small perturbations affect solution; an unstable numerical scheme is one in which small perturbations in solution grow rapidly toward infinity.

There are two criteria for stability:

1. Stability of the initial value problem (IVP):

$$\epsilon(t) \equiv \tilde{y}(t) - y(t),$$

where $\tilde{y}(t)$ is the perturbed solution and $y(t)$ is the underlying solution.

- Unstable if and only if $\lim_{t\to\infty} |\epsilon(t)| = \infty$.
- Stable if and only if $\lim_{t\to\infty} |\epsilon(t)| \neq \infty$.
- Unconditionally stable if and only if $\lim_{t\to\infty} |\epsilon(t)| = 0$.

Example: Stability of the IVP

Given the equation

$$\begin{aligned} y'(t) &= \alpha y(t), \quad \alpha \text{ is a constant} \\ y(0) &= y_0 \end{aligned}$$

The solution is $y(t) = y_0 e^{\alpha t}$. The perturbed initial data will be $y_0 + \bar{y}$, so the perturbed solution will be

$$\tilde{y}(t) = (y_0 + \bar{y})e^{\alpha t}$$

so the difference between perturbed and unperturbed is

$$\epsilon(t) = |\bar{y}|e^{\alpha t}$$

which goes to zero as $t \to \infty$ only if $\alpha < 0$.

2. Stability of the numerical method: We will solve initial value problem PDEs using an iterative technique, so that the solution at iteration $n+1$, y_{n+1} depends on the solution at iteration n, y_n, via some operator $\boldsymbol{G}$. For the case of our example ODE $\boldsymbol{G}$ will reduce to a simple function that depends on Δt and α; i.e., $\boldsymbol{G} = G(\Delta t, \alpha)$.

Example: Stability of the Numerical Method

Given the stable IVP (as discussed in the previous example above)

$$\begin{aligned} y'(t) &= \alpha y(t), \quad \alpha < 0 \text{ is a constant} \\ y(0) &= y_0, \end{aligned}$$

define the solution after one step in time Δt to be $y_1 = y(t = \Delta t)$. Then the operator $\boldsymbol{G}$ will satisfy

$$y_1 = G(\Delta t, \alpha) y_0,$$

Our iterative method is such that at the next iteration, the solution will be given by

$$y_2 = G(\Delta t, \alpha) y_1 = G(\Delta t, \alpha)^2 y_0.$$

Thus the solution at iteration $n+1$ will be given by

$$y_{n+1} = G(\Delta t, \alpha)^{n+1} y_0.$$

We say the numerical method is:

- **Unstable** if and only if $\lim_{\Delta t \to 0} |G(\Delta t, \alpha)| > 1$.
- **Stable** if and only if $\lim_{\Delta t \to 0} |G(\Delta t, \alpha)| \leq 1$.

This is why we generally refer to $G(\Delta t, \alpha)$ as the **growth factor** of our numerical algorithm for solving the PDE (or, in the example above, ODE).

9.1.2 Consistency

If all numerical differential operators (e.g., finite difference derivatives) are identical to the exact differential operators in the limit of zero grid spacing, we say that the numerical scheme is **consistent**.

9.1.3 Convergence

A numerical system is globally **convergent** if and only if the truncation error goes to zero with grid spacing $\Delta x, \Delta y, \Delta t$, etc. going to zero for all independent variables x, y, t, etc.

9.2 The Meaning of the Lax Equivalence Theorem

If you are able to demonstrate that the linear PDE you are trying to solve with some numerical scheme is stable, and that all numerical differential operators are consistent in the limit of zero grid spacing, then by the Lax Equivalence theorem your numerical solution of the PDE is guaranteed to converge to the exact solution as the numerical grid spacing goes to zero.

Conversely, if your numerical result converges at the expected rate as grid spacing goes to zero (due to truncation error only), then you may conclude that your numerical scheme is both consistent and stable. *It is critically important that you verify that your numerical result converges at the expected rate.* Chapter 12 reviews an efficient strategy for measuring numerical convergence, as part of a set of broader code validation tests.

Caveat: Before checking for numerical convergence, remember that if your grid spacing does not properly sample important features in the solution, or if roundoff errors are large, convergence *may not* be dominated by truncation error. Such considerations are not part of the Lax Equivalence Theorem.

MATH 522, Spring 2019 Notes

Chapter 10: Solving Initial Value Problems on the Computer, The von Neumann Stability Analysis

As you will recall from Chapter 9, the Lax Equivalence Theorem states that when solving PDEs on the computer, "Stability + Consistency ≡ Convergence". Here, "Stability" refers to both stability in terms of the initial value problem (i.e., that the solution to the initial value problem does not tend toward infinity as time approaches infinity), as well as stability in terms of the numerical scheme, where stable numerical schemes possess a growth factor G with at most unit magnitude.

How do we determine this growth factor? The von Neumann stability analysis takes as input the numerical representation of the PDE and outputs conditions that the numerical scheme must satisfy in order for the growth factor G to satisfy $|G| \leq 1$.

In short, the analysis injects a wave with some fixed, arbitrary wavelength λ into our numerical solution, and determines the conditions under which the resulting growth factor for this λ satisfies $|G| \leq 1$. Generally, we will use the *wavenumber* $k = 2\pi/\lambda$ instead of wavelength in our analyses.

With a uniform grid with grid spacing $\Delta x > 0$, it is impossible to model waves with arbitrarily small wavelengths. So before performing a stability analysis, we must first understand what wavelength waves can be resolved on our numerical grids. After all, we need not worry about the growth waves with wavelengths that cannot possibly be resolved on our numerical grids!

What is the range of resolvable wavelengths on uniform numerical grids?

The Fourier Convergence Theorem states that any *smooth* function $u(x,t)$ (for example, the solution of a PDE on our numerical grid) at a given time t_0 can be represented as the linear combination of such waves:

$$u(x,t_0) = \sum_{\ell=-\infty}^{\infty} A_\ell e^{ik_\ell x} = \sum_{\ell=-\infty}^{\infty} A_\ell \left(\cos(k_\ell x) + i\sin(k_\ell x)\right),$$

where the amplitudes A_ℓ are complex, and k_ℓ denotes the wavenumber of the ℓth wave with wavelength $\lambda_\ell = 2\pi/(k_\ell)$.

If we are solving the PDE over a domain with length L, then $\lambda_\ell \propto L/\ell$ and $k_\ell \propto \ell/L$. Therefore, as ℓ goes to infinity, the wavelength λ_ℓ approaches zero. With a uniform numerical grid, with constant grid spacing $\Delta x > 0$ (i.e., uniform sampling of the solution in space at spatial points $x_i = x_0 + i\Delta x$, where i is an integer), we cannot possibly model waves at wavelengths corresponding to sampling higher than the Nyquist rate. That is to say, the shortest wavelength resolvable on uniform numerical grids with grid spacing Δx will be the Nyquist wavelength $\lambda_{\rm Nyq} = 2\Delta x$. Thus the shortest wavelength resolvable on a uniform numerical grid is

$$\lambda_{\rm Nyq} = 2\Delta x = 2\pi/k_{\rm Nyq} \implies k_{\rm Nyq} = \pi/\Delta x.$$

Looking at the opposite limit, i.e. the limit $\ell \to 0$, the wavelength λ_ℓ stretches to infinity and the solution tends to a constant, which is trivially resolved on a uniform numerical grid.

We conclude that the range of resolvable wavenumbers k on uniform numerical grids is

$$\boxed{0 \leq k \leq \frac{\pi}{\Delta x}}.$$

This means that by sampling the function uniformly, we are modeling the solution with a Fourier series from $|\ell| = 0$ up to the $|\ell|$ corresponding to $\lambda_\ell = \lambda_{\rm Nyq} = 2\Delta x$.

Armed with the range of wavenumbers k that are possibly resolvable on uniform numerical grids, let us now

inject a wave with wavenumber k in this range into a numerical algorithm for solving an initial value problem. The growth of the amplitude of this wave will yield the growth factor G for this wave, and the range of G that satisfy $|G| \leq 1$ will corresponds to a stable numerical scheme. Typically we will find that $|G| \leq 1$ only if Δx and Δt satisfy an inequality. This inequality is known as the Courant-Friedrichs-Lewy, or CFL, condition, and the approach taken to derive the CFL is known as the von Neumann stability analysis.

10.1 Solving a PDE on a Uniform Numerical Grid in 1-Dimension of Space and Time

The von Neumann stability analysis is a very useful strategy for determining whether our algorithm for solving a PDE on the computer is stable. But how might we construct an algorithm for solving a PDE on the computer in the first place?

Let's consider the one-directional wave equation (also known as the *advection equation*) in one spatial dimension:

$$\partial_t u(x,t) = -v\partial_x u(x,t), \quad u(x,0) = f(x).$$

This initial value problem (IVP) models a wave propagating in positive x-direction at velocity $v > 0$. The general solution of this IVP is $u(x,t) = f(x - vt)$, which you can confirm by plugging in to the PDE and initial condition.

Notation

Assuming a uniform numerical grid implies that spatial gridpoints are uniquely determined by their index j, where spatial point x_j is given by $x_j = x_0 + j\Delta x$. Similarly, a uniform numerical grid in time implies that the solution at iteration n corresponds to the solution at time $t = t_n$, where $t_n = n\Delta t$. For short we will specify the solution to the PDE at spatial point j and iteration n, $u(x_j, t_n)$, as $\boxed{u_j^n}$.

Our goal is to represent the advection equation as an iterative numerical method that represents the unknown solution at iteration $n+1$, u_j^{n+1}, in terms of the solution at an iteration we know or are given, u_j^n. That is to say, we want the value of the solution u at time $t = (n+1)\Delta t$ in terms of solution at time $t = n\Delta t$, which we will compute iteratively from the initial conditions.

In this way, we can start with the initial conditions at $n = 0$, and then march the solution forward in time to u_j^1, then using u_j^1 we can compute u_j^2, etc.

Our discussion to this point assumes that the numerical representation of the advection equation is written in terms of the solution at iterations n and $n+1$ only. So when writing $\partial_t u$ at iteration n and spatial index j, we must be careful to write the corresponding finite difference derivative that only includes the solution at iterations n and $n+1$ only; i.e., we must choose the first-order-accurate finite difference representation of the time derivative:

$$(\partial_t u)_j^n = \frac{u_j^{n+1} - u_j^n}{\Delta t} + \mathcal{O}(\Delta t).$$

(You can confirm the dominant error term is proportional to Δt by evaluating $(u_j^{n+1} - u_j^n)/\Delta t$ as a Taylor series, adopting the strategy of Chapter 6.)

Next, let's rewrite the differential operator for $\partial_x u$. Note that we have data for u at all points x at time $t = 0$, so let's use a higher-order finite difference operator to represent the spatial derivative:

$$(\partial_x u)_j^n = \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} + \mathcal{O}\left((\Delta x)^2\right)$$

In this case, if we were to derive this operator using Taylor series, we'd find that the leading error term goes like $(\Delta x)^2$.

Then we can write our numerical scheme as:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x},$$

which will converge to the exact PDE as Δx and Δt go to zero.

This is called the **Forward Time, Centered Space, or FTCS** representation of this PDE.

Let's rearrange terms:

$$u_j^{n+1} = u_j^n - \frac{v\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n).$$

Notice this form indicates that so long as we know the solution everywhere in space at iteration n (right-hand-side terms), we can immediately get the solution at iteration $n+1$ (left-hand side).

- Is this numerical scheme **consistent**? *Answer: Yes, as Δt and Δx go to zero, we obtain the original PDE.*
- Is the IVP **stable**? *Answer: Yes, the solution simply translates to the right as time progresses forward.*
- Is the numerical scheme **stable**? *Answer: It turns out that the answer is* **no***, and the von Neumann Stability Analysis will demonstrate why this is the case.*

10.1.1 von Neumann Stability Analysis of the 1-directional Wave Equation

The von-Neumann stability analysis assumes that our solution u_j^n at any given time $n\Delta t$ can be represented as a sum of sines and cosines (i.e., a Fourier series):

$$u_j^n = \sum_\ell B_\ell e^{ik_\ell(j\Delta x)}$$

where k_ℓ is the wavenumber, with units of one over the distance; so that the wavelength λ_ℓ is given by $2\pi/k_\ell$. Notice that we've replaced our earlier expression $e^{ik_\ell x}$ in the sum with $e^{ik_\ell(j\Delta x)}$, since we are on a uniform numerical grid, and we have absorbed the $e^{ik_\ell x_0}$ constant term into B_ℓ, where $B_\ell = e^{ik_\ell x_0}A_\ell$.

Since the solution to the PDE can be locally (in space and time) be represented by a Fourier series, each term in the series must individually satisfy the PDE. Let us choose the ℓth term in the series, which corresponds to a wave with wavelength $2\pi/k_\ell$: $e^{ik_\ell(j\Delta x)}$, and plug it in to the FTCS prescription we have devised for the 1-directional wave equation:

$$\begin{aligned}
u_j^1 &= u_j^0 - \frac{v\Delta t}{2\Delta x}(u_{j+1}^0 - u_{j-1}^0) \\
&= \sum_{k_\ell}\left[e^{ik_\ell(j\Delta x)} - \frac{v\Delta t}{2\Delta x}(e^{ik_\ell([j+1]\Delta x)} - e^{ik_\ell([j-1]\Delta x)})\right] \\
&= \sum_{k_\ell} e^{ik_\ell(j\Delta x)}\left[1 - \frac{v\Delta t}{2\Delta x}(e^{ik_\ell\Delta x)} - e^{-ik_\ell\Delta x)})\right].
\end{aligned}$$

The von Neumann stability analysis measures the *growth* of this wave from one iteration to the next. So between iterations $n=0$ and $n=1$, the **growth factor** $G(k\Delta x)$ for this wave $\tilde{u}_j^n$ with arbitrary (but resolvable) wavenumber $k = k_\ell$ can be derived as follows:

$$\begin{aligned}
\tilde{u}_j^1 &= \tilde{u}_j^0 - \frac{v\Delta t}{2\Delta x}(\tilde{u}_{j+1}^0 - \tilde{u}_{j-1}^0) \\
&= e^{ik(j\Delta x)} - \frac{v\Delta t}{2\Delta x}(e^{ik([j+1]\Delta x)} - e^{ik([j-1]\Delta x)}) \\
&= \left(1 - \frac{v\Delta t}{2\Delta x}(e^{ik\Delta x} - e^{-ik\Delta x})\right)e^{ik(j\Delta x)} \\
&= \left(1 - \frac{2iv\Delta t}{2\Delta x}\sin(k\Delta x)\right)e^{ik(j\Delta x)} \\
&= G(k\Delta x)e^{ik(j\Delta x)} \\
&= G(k\Delta x)\tilde{u}_j^0.
\end{aligned}$$

We consider $G(k\Delta x)$ a function of $k\Delta x$ instead of k because $k\Delta x$ is a dimensionless number on our uniform numerical grids that will range from 0 (infinite wavelength) to π (Nyquist wavelength).

From this, we can then immediately write the growth factor at the next iteration:

$$\tilde{u}_j^2 = G(k\Delta x)\tilde{u}_j^1 = G(k\Delta x)\left[G(k\Delta x)\right]\tilde{u}_j^0 = \left[G(k\Delta x)\right]^2\tilde{u}_j^0.$$

Continuing this line of reasoning, at the n'th iteration, our solution will be given by

$$\tilde{u}_j^{n+1} = G(k\Delta x)^{n+1}\tilde{u}_j^0$$

Clearly the only way to prevent $\tilde{u}_j^{n+1}$ from diverging to infinity as $n \to \infty$ is if $|G(k\Delta x)| \leq 1$. Such a divergence is equivalent to the statement that the numerical scheme is *unstable*.

So for this numerical scheme to be stable

$$\begin{aligned} |G(k\Delta x)| &= \left|1 - \frac{iv\Delta t}{\Delta x}\sin(k\Delta x)\right| \\ &= \sqrt{1 + \left(\frac{v\Delta t}{\Delta x}\sin(k\Delta x)\right)^2} \leq 1. \end{aligned}$$

Our numerical scheme will always have $\Delta x > 0$ and $\Delta t > 0$, and the $k = 0$, infinite-wavelength (constant), solution clearly has a growth factor of 1 (implying stability). In fact, you'll notice that in the resolvable range of k: $0 \leq k \leq \pi/\Delta x$, only the endpoints result in a stable numerical scheme; only Nyquist-resolved waves and constant functions do not exponentially grow in this numerical scheme. (But be careful; in some situations, roundoff errors might perturb Nyquist-sampled waves or constant functions to vary in different ways, effectively adding waves with other wavelengths and thus causing the numerical scheme to become unstable.)

We conclude that the FTCS numerical scheme for solving the 1-directional wave equation in one spatial dimension is *unstable*.

10.2 von Neumann Stability analysis of FTCS Heat Equation

Consider the heat equation written in the form:

$$u_t = \alpha^2 u_{xx}.$$

Now represent the operators in a way consistent with the previous example:

$$(u_t)_j^n = \frac{u_j^{n+1} - u_j^n}{\Delta t} + \mathcal{O}(\Delta t)$$

Similarly,

$$(u_{xx})_j^n = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} + \mathcal{O}\left((\Delta x)^2\right)$$

which is the second derivative operator with dominant truncation error term $\mathcal{O}\left(\Delta x^2\right)$.

Then our PDE becomes:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \alpha^2 \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}.$$

we know that these derivative operators are **consistent**, as we've derived them from Taylor series expansions. This means that as $\Delta t \to 0$ and $\Delta x \to 0$, the LHS and RHS become exact derivatives (i.e., truncation error will go to zero on both sides of the equation as the gridspacing in space and time drops to zero).

Now let's rewrite the above equation as an iterative scheme:

$$u_j^{n+1} = u_j^n + \alpha^2 \frac{\Delta t}{(\Delta x)^2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n)$$

The von Neumann stability analysis determines the growth rate $G(k\Delta x)$ of an arbitrary term in the Fourier series at $t = 0$, $\tilde{u}_j^0 = e^{ikj\Delta x}$, where $\tilde{u}_j^1 = G(k\Delta x)\tilde{u}_j^0$:

$$\begin{aligned}
\tilde{u}_j^1 &= e^{ikj\Delta x} + \alpha^2 \frac{\Delta t}{(\Delta x)^2} e^{ikj\Delta x}(e^{ik\Delta x} - 2 + e^{ik\Delta x}) \\
&= \left[1 + \alpha^2 \frac{\Delta t}{(\Delta x)^2}(2\cos(k\Delta x) - 2)\right] \tilde{u}_j^0 \\
&= [1 - 2r(1 - \cos(k\Delta x))]\tilde{u}_j^0, \quad r = \alpha^2 \frac{\Delta t}{(\Delta x)^2}
\end{aligned}$$

Note that $\sin^2(x) = \frac{1-\cos(2x)}{2} \implies 1 - \cos(2x) = 2\sin^2 x$, so we can rewrite the expression for the growth factor $G(k\Delta x)$ as:

$$G(k\Delta x) = 1 - 2r(2\sin^2(k\Delta x/2)) = 1 - 4r\sin^2(k\Delta x/2).$$

So stability for this scheme requires

$$\begin{aligned}
|G(k\Delta x)| &\leq 1 \implies -1 \leq G(k\Delta x) \leq 1 \\
\implies -1 &\leq 1 - 4r\sin^2(k\Delta x/2) \leq 1 \\
\implies -2 &\leq -4r\sin^2(k\Delta x/2) \leq 0 \\
\implies 0 &\leq 4r\sin^2(k\Delta x/2) \leq 2.
\end{aligned}$$

Our goal is to determine for what range of r this numerical scheme will satisfy this inequality, knowing that the resolvable range of $k\Delta x$ is given by $0 \leq k\Delta x \leq \pi$. So let's focus first on the term involving $k\Delta x$: $\sin^2(k\Delta x/2)$. Since $0 \leq k\Delta x \leq \pi$, $0 \leq k\Delta x/2 \leq \pi/2$. What does this sine function do in this interval? Well, it starts at 0 at $k\Delta x/2 = 0$, and increases monotonically to 1 at $k\Delta x/2 = \pi/2$. Next notice that if the sine function is zero, then the inequality is immediately satisfied and our numerical solution is stable. Great! Now what about the other values?

We will find it useful to define

$$\Gamma = \sin^2(k\Delta x/2),$$

which satisfies $0 < \Gamma \leq 1$ in the interval $0 < k\Delta x/2 \leq \pi/2$. Since Γ cannot be zero, we are free to divide both sides of the inequality by Γ:

$$0 \leq 4r \leq \frac{2}{\Gamma}.$$

So we have derived an inequality that will constrain our value of r. Notice that as Γ approaches zero but remains positive, the inequality becomes *less restrictive*, meaning that as we approach the infinite wavelength limit $k\Delta x \to 0$, r (which is a positive number) can become larger and larger, yet we retain numerical stability. But we will resolve wavelengths ranging from infinite wavelength all the way down to Nyquist. So in considering this inequality, we must determine the $k\Delta x$ and therefore the Γ that provides the *most restrictive* range for r.

Clearly the most restrictive range of r will occur when $\Gamma = 1$, corresponding to $k\Delta x = \pi$, the Nyquist sampling rate. So the most restrictive inequality becomes

$$0 \leq 4r \leq 2 \implies 0 \leq r \leq \frac{1}{2}.$$

Thus for stability in all resolvable wavelengths on our numerical grid, we must have

$$r = \alpha^2 \frac{\Delta t}{(\Delta x)^2} \leq 1/2,$$

or equivalently, Δx and Δt must be set so that

$$\frac{\alpha^2 \Delta t}{(\Delta x)^2} \leq \frac{1}{2}.$$

An inequality restricting our choices of Δx and Δt is known in some literature as the Courant-Friedrichs-Lewy condition or simply the "CFL condition" or "CFL criterion" and in other literature as the "Courant condition".

Why is it that the FTCS scheme for the heat equation is stable, but for the one-directional wave equation is not? The Lax Method provides some insights into this.

10.3 The Lax Method

Recall that the FTCS algorithm for the 1D, one-directional wave equation

$$\partial_t u(x,t) = -v\partial_x u(x,t)$$

is given by

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x},$$

or equivalently in iterative form,

$$u_j^{n+1} = u_j^n - \frac{v\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n).$$

Recall also that this scheme is guaranteed to be unstable for practically all waves resolvable on our numerical grids.

The Lax Method cures this by replacing u_j^n with its average from neighboring points:

$$u_j^n \to \frac{1}{2}(u_{j+1}^n + u_{j-1}^n)$$

Note that this is a completely valid operation, as when $\Delta x \to 0$, $u_{j\pm1}^n \to u_j^n$. Then we get:

$$u_j^{n+1} = \frac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{v\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n).$$

Let's perform a von Neumann stability analysis to this revised algorithm for solving the one-directional wave equation. Recall that we wish to consider the growth of an arbitrary term in the Fourier series $\tilde{u}_j^n = e^{ikj\Delta x}$:

$$\begin{aligned}
\tilde{u}_j^{n+1} &= \frac{1}{2}(e^{ik(j+1)\Delta x} + e^{ik(j-1)\Delta x}) - \frac{v\Delta t}{2\Delta x}(e^{ik(j+1)\Delta x} - e^{ik(j-1)\Delta x}) \\
&= \frac{1}{2}e^{ikj\Delta x}(e^{ik\Delta x} + e^{-ik\Delta x}) - \frac{v\Delta t}{2\Delta x}e^{ikj\Delta x}(e^{ik\Delta x} - e^{-ik\Delta x}) \\
&= [\cos(k\Delta x) - i\frac{v\Delta t}{\Delta x}\sin(k\Delta x)]e^{ikj\Delta x}
\end{aligned}$$

Thus for stability we must have:

$$\begin{aligned}
|G(k\Delta x)| &= |\cos(k\Delta x) - i\frac{v\Delta t}{\Delta x}\sin(k\Delta x)| \le 1 \\
&= \sqrt{\cos^2(k\Delta x) + \left(\frac{v\Delta t}{\Delta x}\right)^2 \sin^2(k\Delta x)} \le 1.
\end{aligned}$$

The expression under the square root is a bit ugly, so let's take advantage of the fact that

$$|G(k\Delta x)| \le 1 \implies 0 \le |G(k\Delta x)|^2 \le 1,$$

so we rewrite the inequality as

$$0 \le |G(k\Delta x)|^2 = \cos^2(k\Delta x) + \left(\frac{v\Delta t}{\Delta x}\right)^2 \sin^2(k\Delta x) \le 1.$$

So under what conditions will this inequality be satisfied, thus indicating that our numerical scheme is stable? We should always turn our attention first to terms that are functions of only $k\Delta x$, so simplifying these terms *as much as possible* first will usually be very beneficial. If possible, we should try and write $|G(k\Delta x)|^2$ in the form

$$|G(k\Delta x)|^2 = f(k\Delta x) \times g(\Delta t, \Delta x, ...).$$

To this end, let's write everything in terms of sines, using the trigonometric identity $\cos^2 x = 1 - \sin^2 x$:

$$\begin{aligned} 0 &\leq 1 - \sin^2(k\Delta x) + \left(\frac{v\Delta t}{\Delta x}\right)^2 \sin^2(k\Delta x) \leq 1 \\ \implies -1 &\leq -\sin^2(k\Delta x) + \left(\frac{v\Delta t}{\Delta x}\right)^2 \sin^2(k\Delta x) \leq 0 \\ \implies -1 &\leq \sin^2(k\Delta x)\left[\left(\frac{v\Delta t}{\Delta x}\right)^2 - 1\right] \leq 0. \end{aligned}$$

Now that we have properly simplified the expression, we turn our attention first to the function of $k\Delta x$. In particular, we must address the question: What is the range over which $\sin^2(k\Delta x)$ varies in the resolvable interval $0 \leq k\Delta x \leq \pi$? Clearly at $(k\Delta x) = 0$, $\sin^2(k\Delta x) = 0$, which satisfies the inequality regardless of what we choose for $\frac{v\Delta t}{\Delta x}$. So this point is a stable point.

But what about the infinite number of other resolvable values of $k\Delta x$ on our grid, namely $0 < k\Delta x \leq \pi$? Obviously we cannot analyze all of them one at a time, so let's try to mathematically reason our way to the solution, using the same basic strategy we applied when deriving the CFL condition for the FTCS heat equation.

First notice that $0 < \sin^2(k\Delta x) \leq 1$ for $k\Delta x$ in the range $0 < k\Delta x \leq \pi$, so since $\sin^2(k\Delta x) \neq 0$, let's define $\sin^2(k\Delta x) = \Gamma > 0$ and divide both sides of the inequality by Γ (we cannot do this if it is possible for $\Gamma = 0$, as this would result in a division by zero):

$$\begin{aligned} -1 &\leq \sin^2(k\Delta x)\left[\left(\frac{v\Delta t}{\Delta x}\right)^2 - 1\right] \leq 0 \\ \implies -1 &\leq \Gamma\left[\left(\frac{v\Delta t}{\Delta x}\right)^2 - 1\right] \leq 0 \\ \implies -\frac{1}{\Gamma} &\leq \left[\left(\frac{v\Delta t}{\Delta x}\right)^2 - 1\right] \leq 0. \end{aligned}$$

Notice that if Γ is a very small nonzero number, the inequality is *not very restrictive*, allowing for a wider range of $\frac{v\Delta t}{\Delta x}$ than if Γ is at its maximum value of 1. Since the most restrictive inequality occurs when $\Gamma = 1$ (corresponding to $k\Delta x = \pi$, the Nyquist sampling limit), we need only choose $\Gamma = 1$ to be sure that the range of possible $\frac{v\Delta t}{\Delta x}$ that satisfy the inequality will be true regardless of our choice of $k\Delta x$:

$$\begin{aligned} -1 &\leq \left[\left(\frac{v\Delta t}{\Delta x}\right)^2 - 1\right] \leq 0 \\ \implies 0 &\leq \left(\frac{v\Delta t}{\Delta x}\right)^2 \leq 1 \\ \implies 0 &< \frac{v\Delta t}{\Delta x} \leq 1, \text{ since we assumed } \frac{v\Delta t}{\Delta x} > 0 \end{aligned}$$

In other words, $\frac{v\Delta t}{\Delta x} \leq 1$ is the CFL stability criterion/condition for the Lax Method. Amazingly, our numerical scheme has been stabilized, via a very straightforward replacement!

Now let's see if we can make sense of this CFL criterion. It basically says that if the propagation speed of waves v is too slow to move Δx distance in Δt, then our numerical scheme is stable. In other words, the CFL condition states that our timestep must be small enough so that a wave does not propagate farther than one full gridpoint Δx in time Δt. If we chose a larger Δt than the CFL criterion would allow, then our numerical scheme would be unable to update the next gridpoint fast enough.

So how did we know to replace u_j^n with $(u_{j+1}^n + u_{j-1}^n)/2$? Why does this stabilize the numerical algorithm for solving the 1D, one-directional wave equation? At this point, the numerical algorithm looks like a real kludge; we tried a dirty trick to make an unstable algorithm stable again.

To gain some insights, let's rewrite the equation in a clever way:

$$\begin{aligned} u_j^{n+1} &= \frac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{v\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n) \\ \implies u_j^{n+1} - u_j^n &= \frac{1}{2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n) - \frac{v\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n) \\ \implies \frac{u_j^{n+1} - u_j^n}{\Delta t} &= \frac{1}{2\Delta t}(u_{j+1}^n - 2u_j^n + u_{j-1}^n) - \frac{v}{2\Delta x}(u_{j+1}^n - u_{j-1}^n) \\ (u_t + \mathcal{O}(\Delta t)) &= \frac{(\Delta x)^2}{2\Delta t}\left(u_{xx} + \mathcal{O}\left(\Delta x^2\right)\right) - v\partial_x u + \mathcal{O}\left(\Delta x^2\right) \end{aligned}$$

The CFL criterion allows for $0 < v\Delta t/\Delta x \leq 1$, so let's set $\beta = v\Delta t/\Delta x$, where $0 < \beta \leq 1$. Then $\Delta x = v\Delta t/\beta$. Substituting into the above equation, we get

$$\begin{aligned} (u_t + \mathcal{O}(\Delta t)) &= \frac{(\Delta x)^2}{2\Delta t}\left(u_{xx} + \mathcal{O}\left(\Delta x^2\right)\right) - v\partial_x u + \mathcal{O}\left(\Delta x^2\right) \\ &= \frac{(v\Delta t)^2}{2\beta^2\Delta t}\left(u_{xx} + \mathcal{O}\left(\Delta x^2\right)\right) - v\partial_x u + \mathcal{O}\left(\Delta x^2\right) \\ &= \frac{v^2\Delta t}{2\beta^2}\left(u_{xx} + \mathcal{O}\left(\Delta x^2\right)\right) - v\partial_x u + \mathcal{O}\left(\Delta x^2\right), \end{aligned}$$

so the u_{xx} term vanishes in the limit of $\Delta t \to 0$, meaning that the Lax method is still consistent with the original PDE!

But why is it that parabolic terms stabilize our numerical solution?

To get a deeper understanding, let's solve the heat equation

$$u_t(x,t) = \alpha^2 u_{xx}(x,t)$$

in the case where the initial condition is given by $u(x,0) = \cos(kx)$. Further, let's suppose that the spatial derivative $u_{xx}(x,t)$ is computed *exactly*. Then we may write our iterative scheme for the first iteration as follows:

$$\frac{u^1 - u^0}{\Delta t} = \alpha^2\partial_{xx}(\cos(kx)) = -\alpha^2 k^2\cos(kx) = -\alpha^2 k^2 u^0$$

Thus

$$\begin{aligned} u^1 &= u^0 - \alpha^2\Delta t k^2 u^0 \\ &= (1 - \alpha^2\Delta t k^2)u^0 \\ \implies u^2 &= (1 - \alpha^2\Delta t k^2)u^1 \\ &= (1 - \alpha^2\Delta t k^2)^2 u^0 \\ \implies u^{n+1} &= (1 - \alpha^2\Delta t k^2)^{n+1} u^0 \end{aligned}$$

Clearly $\alpha^2\Delta t k^2 > 0$, so as long as we choose Δt small enough, we are guaranteed to have $\alpha^2\Delta t k^2 < 1$, and therefore a growth factor $|G(\Delta t)| < 1$. Notice also that the smallest growth factors will occur for the largest values of k, corresponding to the shortest wavelengths. So we say that parabolic terms add *dissipation* to our numerical system of equations, and are most effective at dissipating the shortest wavelengths.

Recall that the FTCS scheme for the 1-directional wave equation yields a growth factor of:

$$\begin{aligned} |G(k\Delta x)|^2 &= \left|1 - \frac{iv\Delta t}{\Delta x}\sin(k\Delta x)\right|^2 \\ &= 1 + \left(\frac{v\Delta t}{\Delta x}\sin(k\Delta x)\right)^2 \end{aligned}$$

Notice that this growth factor is actually small at the lower and upper limits of $k\Delta x$, but is maximized at $k\Delta x = \pi/2$, or $\lambda = 2\pi/k = 4\Delta x$... half the Nyquist sampling frequency! So while we may be stable to Nyquist-sampled waves for this scheme, we are maximally unstable for half-Nyquist-sampled waves—still very short wavelengths, and the parabolic term dissipates these quite effectively, stabilizing our numerical scheme.

10.4 Numerically solving the full, second-order wave equation in one spatial dimension, via FTCS

For ease of numerical implementation, it is common to write this second-order in time PDE as two, coupled first-order-in-time PDEs, defining $v(x,t) = \partial_t u(x,t)$. Then

$$\begin{aligned} \partial_t u(x,t) &= v(x,t) \\ \partial_t v(x,t) &= c^2 \partial_x^2 u(x,t). \end{aligned}$$

The FTCS representation is given by:

$$\begin{aligned} u_j^{n+1} &= u_j^n + \Delta t v_j^n \\ v_j^{n+1} &= v_j^n + \frac{\Delta t c^2}{(\Delta x)^2}(u_{j-1}^n - 2u_j^n + u_{j+1}^n) \end{aligned}$$

To analyze the growth of a wave of arbitrary amplitude ($\tilde{u}^0$ and $\tilde{v}^0$) and arbitrary wavenumber in the initial conditions of u and v:

$$\begin{pmatrix} \tilde{u}_j^0 \\ \tilde{v}_j^0 \end{pmatrix} = e^{ikj\Delta x} \begin{pmatrix} U \\ V \end{pmatrix}$$

In this case, there are two growth factors, one for each variable, corresponding to the **eigenvalues** of the 2×2 matrix $\boldsymbol{M}$:

$$\begin{pmatrix} \tilde{u}_j^1 \\ \tilde{v}_j^1 \end{pmatrix} = \boldsymbol{M} \begin{pmatrix} \tilde{u}_j^0 \\ \tilde{v}_j^0 \end{pmatrix} = G(k\Delta x) \begin{pmatrix} \tilde{u}_j^0 \\ \tilde{v}_j^0 \end{pmatrix}$$

We will now derive the growth factor matrix $G(k\Delta x)$:

$$\begin{aligned} \tilde{u}_j^1 &= e^{ikj\Delta x}U + \Delta t e^{ikj\Delta x}V \\ &= e^{ikj\Delta x}(U + \Delta t V) \\ \tilde{v}_j^1 &= e^{ikj\Delta x}V + \frac{\Delta t c^2}{(\Delta x)^2} e^{ikj\Delta x}U(e^{-ik\Delta x} + e^{ik\Delta x} - 2) \\ &= e^{ikj\Delta x}\left(V + U\frac{\Delta t c^2}{(\Delta x)^2}(2\cos(k\Delta x) - 2)\right) \end{aligned}$$

Writing this in matrix form, we get:

$$\begin{pmatrix} \tilde{u}_j^1 \\ \tilde{v}_j^1 \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ \frac{2\Delta t c^2}{(\Delta x)^2}(\cos(k\Delta x) - 1) & 1 \end{pmatrix} \begin{pmatrix} \tilde{u}_j^0 \\ \tilde{v}_j^0 \end{pmatrix}$$

Thus we have found the matrix $\boldsymbol{M}$, the eigenvalues of which are the growth factors G_1 and G_2. Stability requires that the magnitude of *both* growth factors be less than one.

Let's now find the eigenvalues $G(k\Delta x)$ of the matrix $\boldsymbol{M}$. For simplicity, let's say $G = G(k\Delta x)$:

$$0 = \begin{pmatrix} 1-G & \Delta t \\ \frac{2\Delta t c^2}{(\Delta x)^2}(\cos(k\Delta x) - 1) & 1-G \end{pmatrix} \begin{pmatrix} \tilde{u}_j^0 \\ \tilde{v}_j^0 \end{pmatrix}$$

For this matrix equation to have a nontrivial solution ($\tilde{u}_j^0$, $\tilde{v}_j^0$ not both equal to zero), the determinant of the matrix must be zero.

Thus we have

$$\begin{aligned} (1-G)^2 - 2\left(\frac{c\Delta t}{\Delta x}\right)^2(\cos(k\Delta x) - 1) &= 0 \\ G^2 - 2G + \left[1 - 2\left(\frac{c\Delta t}{\Delta x}\right)^2(\cos(k\Delta x) - 1)\right] &= 0 \\ G^2 - 2G + \left[1 - 2b^2(\cos(k\Delta x) - 1)\right] &= 0, \end{aligned}$$

where we define $b = \frac{c\Delta t}{\Delta x}$. Solving this quadratic equation for G, we get

$$\begin{aligned} G &= \frac{2 \pm \sqrt{4 - 4[1 - 2b^2(\cos(k\Delta x) - 1)]}}{2} \\ &= 1 \pm \sqrt{-[-2b^2(\cos(k\Delta x) - 1)]} \\ &= 1 \pm \sqrt{2b^2(\cos(k\Delta x) - 1)} \end{aligned}$$

We know that $\cos(x) = 1 - 2\sin^2(x/2)$, so $\cos(x) - 1 = -2\sin^2(x/2)$

$$\begin{aligned} G &= 1 \pm \sqrt{2b^2(-2\sin^2(k\Delta x/2))} \\ &= 1 \pm 2bi\sin(k\Delta x/2) \end{aligned}$$

Thus the stipulation that $|G|^2 \leq 1$ for a stable numerical scheme yields

$$\begin{aligned} 0 \leq \quad & 1 + [2b\sin(k\Delta x/2)]^2 \quad \leq 1 \\ -1 \leq \quad & 4b^2\sin^2(k\Delta x/2) \quad \leq 0 \end{aligned}$$

$4b^2 > 0$, so the only value of $k\Delta x/2$ that would satisfy the inequality would be $k\Delta x/2 = 0$, corresponding to the constant solution. All other FTCS scheme for the usual wave equation is unstable, just as it was for the one-directional wave equation.

10.5 von Neumann Stability Analysis: Practice Problems

von Neumann Exercises

These exercises are designed to familiarize you with the mathematical reasoning that goes into the final stages of a von Neumann stability analysis. Find all ranges of w where each inequality is satisfied. Each range must be written in the form

$$\alpha \leq w \leq \beta,$$

where α and β are constant parameters that you must find.
As for the functions of $k\Delta x$, you are to assume that these inequalities were generated at the end of a von Neumann stability analysis. Considering *all* possible values of $k\Delta x$ resolvable on your grid—not just one or two values—you must find and justify the single value of $k\Delta x$ that is applicable to the von Neumann stability analysis. Write all steps in your reasoning.
Negative values of w are permitted, but imaginary values are not. All trigonometric identities must be fully justified. You are expected to solve these problems without the help of plotting packages or computational aides.

1. $-10 \leq (-w^2 - w + 6)(\cos(k\Delta x) - 1) \leq 0$
2. $-2 \leq (-w^2 - 5w - 6)(1 - \cos^2(k\Delta x/2)) \leq 0$
3. $0 \leq (-w^2 + w + 6)\sin^2(k\Delta x/4)\cos^2(k\Delta x/4) \leq \frac{7}{4}$
4. $0 \leq (-w^4 + w^2 + 6)\frac{1+\sin(k\Delta x/2)-\sin^2(k\Delta x/2)-\sin^3(k\Delta x/2)}{1+\sin(k\Delta x/2)} \leq 5$

Solutions:

1. $-10 \leq (-w^2 - w + 6)(\cos(k\Delta x) - 1) \leq 0$

   We know that $\cos(x) - 1 = -2\sin^2(x/2)$, so we have

$$\begin{aligned} -10 &\leq (-w^2 - w + 6)(\cos(k\Delta x) - 1) \leq 0 \\ \implies -10 &\leq (-w^2 - w + 6)(-2)\sin^2(k\Delta x/2) \leq 0 \\ \implies -10 &\leq (w^2 + w - 6)(2)\sin^2(k\Delta x/2) \leq 0 \\ \implies -5 &\leq (w^2 + w - 6)\sin^2(k\Delta x/2) \leq 0 \end{aligned}$$

   $\sin(k\Delta x/2)^2 = 0$ at $k\Delta x = 0$, which trivially satisfies the inequality. Otherwise, it is just some positive constant factor multiplying the polynomial in the resolvable range $0 < k\Delta x \leq \pi$. Call this constant factor Γ. Then dividing through by Γ, the inequality becomes

$$-\frac{5}{\Gamma} \leq (w^2 + w - 6) \leq 0$$

   Clearly the value for $\Gamma \in (0, 1]$ providing the most restrictive inequality is $\Gamma = 1$, corresponding to $k\Delta x/2 = \pi/2$, so the inequality of interest is:

$$-5 \leq w^2 + w - 6 \leq 0.$$

   Next we analyze the polynomial

$$f(w) = w^2 + w - 6.$$

   Notice that $f(w)$ is a concave up parabola, with a minimum at

$$f'(w) = 2w + 1 = 0 \implies w = -\frac{1}{2}.$$

   At this minimum, $f(w)$ evaluates to $1/4 - 1/2 - 6 = -25/4 = -6.25$, which is less than the lower bound of the inequality! Thus we need to find regions where $f(w)$ passes between zero and -5.

   From solving the quadratic equation, we find that $f(w)$ crosses zero at $w = -3$ and $w = 2$. We find out where it crosses -5 by solving the equation

$$w^2 + w - 6 = -5,$$

   which according to the quadratic formula is satisfied at

$$w = \frac{-1 \pm \sqrt{5}}{2}.$$

   Thus the ranges for w that satisfy the inequality are:

$$-3 \leq w \leq \frac{-1 - \sqrt{5}}{2}, \text{ and } \frac{-1 + \sqrt{5}}{2} \leq w \leq 2.$$

2. $-2 \leq (-w^2 - 5w - 6)(1 - \cos^2(k\Delta x/2)) \leq 0$

   $1 - \cos^2(k\Delta x/2) = \sin^2(k\Delta x/2)$, so we can rewrite the inequality as

$$-2 \leq (-w^2 - 5w - 6)\sin^2(k\Delta x/2) \leq 0$$

   $\sin(k\Delta x/2)^2 = 0$ at $k\Delta x = 0$, which trivially satisfies the inequality. Otherwise, it is just some positive constant factor multiplying the polynomial in the resolvable range $0 < k\Delta x \leq \pi$. Call this constant factor Γ. Then dividing through by Γ, the inequality becomes

$$-\frac{2}{\Gamma} \leq (-w^2 - 5w - 6) \leq 0$$

Clearly the value for $\Gamma \in (0, 1]$ providing the most restrictive inequality is is $\Gamma = 1$, corresponding to $k\Delta x/2 = \pi/2$.

So the most restrictive inequality is

$$-2 \leq -w^2 - 5w - 6 \leq 0.$$

Let's analyze the polynomial

$$f(w) = -w^2 - 5w - 6.$$

Notice that $f(w)$ is a concave down parabola ($f''(w) < 0$), with a maximum at

$$f'(w) = -2w - 5 = 0 \implies w = -\frac{5}{2}.$$

At this maximum, $f(w)$ evaluates to $1/4$, which is outside the inequality. But again, this is a *maximum* value for $f(w)$, so to find the regions where the inequality is satisfied, we must find where $f(w) = 0$ and where $f(w) = -2$.

From solving the quadratic equation, we find that $f(w)$ crosses zero at $w = -3$ and $w = -2$. We find out where it crosses -2 by solving another quadratic equation:

$$-w^2 - 5w - 6 = -2 \implies w^2 + 5w + 4 = 0 = (w + 4)(w + 1).$$

So $f(w)$ passes through the range indicated by the inequality when

$$-4 \leq w \leq -3, \quad -2 \leq w \leq -1,$$

which indicates the stable range of w in the von Neumann stability analysis.

3. $0 \leq (-w^2 + w + 6)\sin^2(k\Delta x/4)\cos^2(k\Delta x/4) \leq \frac{7}{4}$

The half-angle trigonometric identity

$$4\sin^2(x/4)\cos^2(x/4) = \sin^2(x/2)$$

is useful here, as it immediately simplifies the trigonometric function in the inequality:

$$\begin{aligned} \sin^2(k\Delta x/4)\cos^2(k\Delta x/4) &= \frac{\sin^2(k\Delta x/2)}{4} \\ \implies 0 &\leq (-w^2 + w + 6)\frac{\sin^2(k\Delta x/2)}{4} \leq \frac{7}{4} \\ \implies 0 &\leq (-w^2 + w + 6)\sin^2(k\Delta x/2) \leq 7. \end{aligned}$$

Again, $\sin(k\Delta x/2)^2 = 0$ at $k\Delta x = 0$, which trivially satisfies the inequality. Otherwise, it is just some positive constant factor multiplying the polynomial in the resolvable range $0 < k\Delta x \leq \pi$. Call this constant factor Γ. Then dividing through by Γ, the inequality becomes

$$0 \leq -w^2 + w + 6 \leq \frac{7}{\Gamma}.$$

Clearly the value for $\Gamma \in (0, 1]$ providing the most restrictive inequality is is $\Gamma = 1$, corresponding to $k\Delta x/2 = \pi/2$.

So the most restrictive inequality is

$$0 \leq -w^2 + w + 6 \leq 7.$$

$f(w) = -w^2 + w + 6$ is a concave-down function, with a maximum at

$$f'(w) = -2w + 1 = 0 \implies w = \frac{1}{2}.$$

The value of the function at maximum is 6.25, which lies in the stable range! Thus the points w such that $f(w) = 0$ will yield the range of w that satisfies the inequality:

$$f(w) = 0 = -w^2 + w + 6 = -(w - 3)(w + 2) \implies w = 3, \quad w = -2.$$

We conclude that the stable range of w will occur when $-2 \leq w \leq 3$.

4. $0 \le (-w^4 + w^2 + 6)\frac{1+\sin(k\Delta x/2)-\sin^2(k\Delta x/2)-\sin^3(k\Delta x/2)}{1+\sin(k\Delta x/2)} \le 5$

First we note that the complicated sine expression factorizes nicely:

$$\begin{aligned}\frac{1+\sin(k\Delta x/2)-\sin^2(k\Delta x/2)-\sin^3(k\Delta x/2)}{1+\sin(k\Delta x/2)} &= \frac{[1+\sin(k\Delta x/2)]-\sin^2(k\Delta x/2)\,[1+\sin(k\Delta x/2)]}{1+\sin(k\Delta x/2)} \\ &= \frac{[1+\sin(k\Delta x/2)]\left[1-\sin^2(k\Delta x/2)\right]}{1+\sin(k\Delta x/2)} \\ &= \left[1-\sin^2(k\Delta x/2)\right] \\ &= \cos^2(k\Delta x/2),\end{aligned}$$

so our inequality becomes

$$0 \le (-w^4 + w^2 + 6)\cos^2(k\Delta x/2) \le 5.$$

Next note that $\cos^2(k\Delta x/2)$ monotonically decreases from $k\Delta x = 0$ to $k\Delta x = \pi$, from 1 to 0. Obviously the zero value satisfies the inequality, so results in a stable numerical scheme. Then what value of $\cos^2(k\Delta x/2)$ generates the most restrictive range for the inequality?

Call $\Gamma = \cos^2(k\Delta x/2) \neq 0$ and divide the entire inequality by Γ:

$$0 \le (-w^4 + w^2 + 6) \le \frac{5}{\Gamma}.$$

Thus the Γ producing the most restrictive inequality will be $\Gamma = 1$, corresponding to $k\Delta x = 0$, the constant solution!

Then the inequality we are trying to solve is:

$$0 \le (-w^4 + w^2 + 6) \le 5$$

To simplify, define $y = w^2$. Then

$$f(y) = -y^2 + y + 6.$$

This is concave down, with a maximum at

$$f'(y) - 2y + 1 = 0 \implies y = 1/2.$$

At $y = 1/2$,

$$f(y) = -1/4 + 1/2 + 6 > 5,$$

which is clearly in the unstable range for this inequality, so there will be two ranges of stability for $f(y)$. First, $f(y) = 5$ at

$$-y^2 + y + 6 = 5 \implies y = \frac{1 \pm \sqrt{5}}{2}.$$

Next, $f(y) = 0$ at

$$-y^2 + y + 6 = -(y-3)(y+2) = 0 \implies y = 3,\ -2.$$

Thus the ranges for y that satisfy the inequality are

$$-2 \le y \le \frac{1-\sqrt{5}}{2},\ \text{and}\ \ \frac{1+\sqrt{5}}{2} \le y \le 3$$

However, remember that $y = w^2$, and w must be real, so the negative range is irrelevant.

Thus we have

$$\frac{1+\sqrt{5}}{2} \le w^2 \le 3$$

Then, taking the root of both sides we get two ranges:

$$\begin{aligned}\sqrt{\frac{1+\sqrt{5}}{2}} &\le w \le \sqrt{3}, \\ -\sqrt{3} &\le w \le -\sqrt{\frac{1+\sqrt{5}}{2}}.\end{aligned}$$

MATH 522, Spring 2019 Notes

Chapter 11: Pseudocode for Solving One-Directional Wave Equation in One Spatial Dimension

Consider again the one-directional wave equation (also known as the *advection equation*) in one spatial dimension:

$$\partial_t u(x,t) = -v\partial_x u(x,t), \quad u(x,0) = g(x).$$

In the previous section, we started with the forward-time, centered-space representation of this equation (FTCS method):

The FTCS Method for solving the advection equation

$u_j^0 = g(j, x0, DeltaX)$ is given as the initial condition. Then all subsequent iterations are as follows:

$$u_j^{n+1} = u_j^n - \frac{v\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n). \tag{11.1}$$

11.1 Naïve (incorrect) approach for implementing FTCS Method

Pseudocode: the (incorrect and inefficient) FTCS Method for solving the advection equation

Warning: Do not use! The below pseudocode has severe bugs and is extremely inefficient, as described below.

Here is a naïve strategy for implementing the above equation, **which**, as we'll find, is inefficient and **will ultimately fail**:

Assuming $t_n = n\Delta t$ for $n \in [0, N]$ and $x_j = x_0 + j\Delta x$ for $j \in [0, J]$, where $\Delta t =$`Dt` and $\Delta x =$`DeltaX` have been set, our pseudocode is as follows:

```
# We define the 2D array u[n][j], where the first index, n,
#    corresponds to a discrete point in time and the second
#    index, j, corresponds to a discrete location in space.

# First set the initial data
do j=0,J # Iterate over all spatial points
  u[0][j] = g(j,x0,DeltaX) # Call function that sets initial data
end do

# FTCS Method finds solution at later times u[n+1][j], where i>0:
do n=0,N-1  # Iterate forward in time, where n=1 corresponds to time dt
  do j=0,J # Iterate over all spatial points
    u[n+1][j] = u[n][j] - (v*Dt)/(2.0*DeltaX)*(u[n][j+1] - u[n][j-1])
  end do
end do
```

Since `u[0][j]` is known, the above gives the basic strategy for solving this PDE. But notice there is a major problem. Notice in the second `j` loop, `j` runs from `0` to `J`, so `j-1` *is undefined* at `j=0` and `j+1` is undefined at `j=J`. To fix this, consider what would happen if we adjusted the bounds on `j`:

Pseudocode v2.0 (still inefficient and incorrect): FTCS Method for solving the advection equation

Warning: Do not use! This pseudocode has severe bugs, as described below.

```
# ... (To this point, pseudocode is same as previous example) ...

# Next implement the FTCS Method for solving the advection equation:
do i=0,N-1  # Iterate forward in time, where n=1 corresponds to time dt
  do j=1,J-1 # Iterate over all but the boundary spatial points
    u[n+1][j] = u[n][j] - (v*Dt)/(2.0*DeltaX)*(u[n][j+1] - u[n][j-1])
  end do
end do
```

There is still a problem: we do not have data at $j = 0$ and $j = J$ at the end of iteration $n = 1$! This means that as we iterate, we lose one point from both $x_{\min}$ and $x_{\max}$ boundaries at each iteration.

11.2 Correct, but inefficient approach for implementing FTCS Method

Pseudocode v3.0: Correct, but *inefficient* approach for implementing FTCS Method

To fix the problem with our naïve approach, we *apply boundary conditions at the end of each* `i` *iteration*:

```
# We define the 2D array u[n][j], where the first index, n,
#    corresponds to a discrete point in time and the second
#    index, j, corresponds to a discrete location in space.

# First set the initial data
do j=0,J # Iterate over all spatial points
  u[0][j] = g(j,x0,DeltaX) # Call function that sets initial data
end do

# Next implement the FTCS Method for solving the advection equation:
do n=0,N-1
  do J=1,J-1
    u[n+1][j] = u[n][j] - (v*Dt)/(2*DeltaX)*(u[n][j+1] - u[n][j-1])
  end do
  apply_bcs(u)
end do
```

Again, we assume $t_i = i\Delta t$ for $i \in [0, N]$ and $x_j = x_0 + j\Delta x$ for $j \in [0, J]$, where $\Delta t =$`Dt` and $\Delta x =$`DeltaX` have been set.
Further, `apply_bcs(u)` could refer to boundary conditions such as Dirichlet, Newman, Robin, periodic, etc. **It is quite inefficient though**, as it stores the solution $u(x, t)$ at all $J + 1$ points in space and all $N + 1$ points in time. Thus if we wish to double our numerical sampling in space and time, it will require a total of *four times as much memory*. This is extremely inefficient.

11.3 Correct and efficient approach for implementing FTCS Method

Pseudocode v4.0: Correct and efficient approach for implementing FTCS Method

Notice that u^{n+1} at any given spatial point $j \in [1, J-1]$ depends only on u^n at spatial points $j \in [0, J]$, so there is no need to store the solution at all times. Instead let's just store two *grid functions* u_j^n=`un[j]` and u_j^{n+1}=`unp1[j]`. In this way if we wanted to double our numerical sampling in space and time, the memory requirements would only *double*:

```
# We define the 1D arrays un[j] and unp1[j], where un=u^n stores,
# the solution at time iteration n, unp1=u^{n+1} stores the solution,
# at time iteration n+1, and the index j corresponds to a discrete
# location in space.

# First set the initial data
do j=0,J # Iterate over all spatial points
  un[j] = g(j,x0,DeltaX) # Call function that sets initial data
end do

# Next implement the FTCS Method for solving the advection equation:
print_solution(un)
do n=0,N-1
  do J=1,J-1
    unp1[j] = un[j] - (v*Dt)/(2*DeltaX)*(un[j+1] - un[j-1])
  end do
  apply_bcs(unp1)
  if(n+1 is a multiple of print_every):
    print_solution(unp1)
  end if
end do
```

Again, `apply_bcs(u)` could refer to boundary conditions such as Dirichlet, Newman, Robin, periodic, etc. Also, we assume $t_i = i\Delta t$ for $i \in [0, N]$ and $x_j = x_0 + j\Delta x$ for $j \in [0, J]$, where Δt =`Dt` and Δx =`DeltaX` have been set.
`apply_bcs(u)` could refer to boundary conditions such as Dirichlet, Newman, Robin, periodic, etc.
The function `print_solution()` outputs the solution to a file or to the screen every `print_every` iterations in n, so we can analyze the output from our numerical PDE solver. For example, set `print_every=2` to output at iterations 0,2,4,6,...
This is the basic structure of the numerical scheme for computing the solution to the 1D, 1-directional wave equation. Note that other hyperbolic and parabolic PDEs can be solved using a similar strategy. **You are expected to implement this basic approach when solving hyperbolic PDEs numerically in this class, unless otherwise requested.**

MATH 522, Spring 2019 Notes

Chapter 12: Convergence to the Exact Solution

12.1 Verifying that Truncation Error Dominates; Code Validation Check

We have written a finite difference algorithm and want to verify that our programming is correct. How do we determine that our convergence order is consistent with what we expect? In particular, how do we do this if we do not know the exact value?

Suppose we have a second-order-in-space error, which dominates our solution. Then the exact value of a function F at some gridpoint may be written in terms of the numerical approximation F_a at that point as follows:

$$F \approx F_a + K(\Delta x)^2,$$

where K is some constant that depends on a high-order derivative of the function at that gridpoint.

So F_a and Δx are known, but K and F are unknown. Thus we can solve for F by computing values of F_a with two values of Δx.

You know that for smooth functions your solution must converge to the exact value as the interval spacing $(\Delta x)^2$. When solving mathematical problems on the computer, it is always our task to perform *code validation* tests, to demonstrate to ourselves and our colleagues that our codes do not have bugs, and in fact we have programmed the algorithms correctly. Thus it is your solemn duty to verify that indeed your error is very nearly proportional to $(\Delta x)^2$. If our code satisfies this test, we can say that "our code converged to second order in the interval spacing (Δx)".

After programming a routine that must converge to second order, we are obligated to solve

$$F = F_a + K'(\Delta x)^n$$

for unknowns F, K', and n. Remember that this equality is only approximate, as it assumes higher-order errors are negligible, and that truncation error dominates. To solve for these three unknowns, we must solve the PDE three times to get known values F_a at three known Δx values. Let's solve for these three, using resolutions Δx_1, Δx_2, and Δx_3. Suppose also that, as in our example, $\Delta x_2 = \Delta x_1/2$ and $\Delta x_3 = \Delta x_1/4$:

$$\begin{aligned} F &= F_{a,1} + K'(\Delta x_1)^n && (12.1)\\ F &= F_{a,2} + K'(\Delta x_1/2)^n && (12.2)\\ F &= F_{a,3} + K'(\Delta x_1/4)^n && (12.3) \end{aligned}$$

First let's solve for n. Subtract Eq. 12.2 from Eq. 12.1:

$$\begin{aligned} 0 &= F_{a,1} - F_{a,2} + K'[(\Delta x_1)^n - (\Delta x_1/2)^n] \\ \implies F_{a,1} - F_{a,2} &= -K'\Delta x_1^n[1 - (1/2)^n] \end{aligned}$$

Similarly, subtracting Eq. 12.3 from Eq. 12.2:

$$\begin{aligned} 0 &= F_{a,2} - F_{a,3} + K'[(\Delta x_1/2)^n - (\Delta x_1/4)^n] \\ \implies F_{a,2} - F_{a,3} &= -K'\Delta x_1^n[(1/2)^n - (1/4)^n]. \end{aligned}$$

Combining these expressions, we find

$$\begin{aligned} \frac{F_{a,1} - F_{a,2}}{F_{a,2} - F_{a,3}} &= \frac{1 - (1/2)^n}{(1/2)^n - (1/4)^n} && (12.4)\\ &= \frac{1 - (1/2)^n}{(1/2)^n(1 - (1/2)^n)} = 2^n. && (12.5) \end{aligned}$$

This equation holds regardless of the order of the approximation (i.e., the order of the finite difference derivative). It says that, so long as the grid spacing is doubled between $F_{a,1}$ and $F_{a,2}$, and doubled again between $F_{a,2}$ and $F_{a,3}$, the gap between subsequent doublings shrinks exponentially with the approximation order n, as 2^n.

So if $n = 2$, Eq. 12.5 implies that

$$\frac{F_{a,1} - F_{a,2}}{F_{a,2} - F_{a,3}} = 2^2 = 4.$$

I.e., as we continue to double the resolution, the distance between subsequent approximations drops by a factor of 4, meaning that convergence accelerates greatly with an increase in resolution, where the exponential rate of acceleration is set by the approximation order n.

We will use this value as a check on the consistency of our method. If this ratio is found to be far from 4, there are at least three possible reasons:

1. There may be a bug in our code,
2. We are under-sampling the function (aliasing error), or
3. We chose a Δx so tiny that, e.g., $F_{a,1} - F_{a,2}$ results in a catastrophic cancellation (roundoff error/loss of significance).

All of the above can be easily checked via careful analysis of the code and a back-of-the-envelope estimate.

12.2 Richardson Extrapolation

Next, let's assume that our method is consistent with $n = 2$ and proceed to solve for F, the "exact" value of the function:

Equation 12.1 gives:

$$K \approx \frac{F - F_{a,1}}{(\Delta x_1)^2}$$

Plugging this into Eq. 12.2 we get:

$$\begin{aligned} F &\approx F_{a,2} + (F - F_{a,1})\frac{(\Delta x_1/2)^2}{(\Delta x_1)^2} \\ &\approx F_{a,2} + (F - F_{a,1})/4 \\ \implies \frac{3}{4}F &\approx F_{a,2} - F_{a,1}/4 \\ \implies F &\approx \frac{1}{3}(4F_{a,2} - F_{a,1}) \end{aligned}$$

It turns out that this estimate for the exact value of the function will be more accurate than either of our numerical estimates $F_{a,1}$ and $F_{a,2}$, and since this estimate, attributed to Richardson, is effectively based on extrapolates to infinite resolution, we call it "Richardson extrapolation".

Of course the value of the function will not be exact because there are higher-order terms for which we are not accounting, like $\mathcal{O}((\Delta x)^3)$. So Richardson extrapolation effectively gives us an estimate of the exact solution to a higher order in Δx, or equivalently, it cancels out the second-order-accurate term.

MATH 522, Spring 2019 Notes

Chapter 13: Higher-Order Timestepping Algorithms, The Method of Lines

Recall in Chapters 10 and 11, we constructed constructed step-by-step approaches (algorithms) for solving PDEs in which the solution at some initial time is provided. Given the solution at the initial time, the algorithm provides the solution at any later time. In those chapters, we invariably used Euler's method for stepping forward in time.

It is a rare circumstance that Euler's method should *ever* be used when solving a PDE or an ODE on the computer, as it is extremely inefficient. To understand the inefficiency, consider the time-dependent PDE:

$$\partial_t u(t,x,y,z,...) = f(u,u_x,u_{xx},...,t,x,y,z,...).$$

Since u is a function only of independent variables $t,x,y,z,...$, f does not depend on any time derivatives of u, and we are only interested in representing the time derivative $\partial_t u$. In this case, standard strategies for solving the ODE

$$\frac{dy(t)}{dt} = y'(t) = f(y,t)$$

can be applied. Applying ODE solution methods to solve PDEs in this way is known as **the method of lines**.

For example, recall the forward-time, first-order derivative, which can be represented as:

$$f(u,u_x,u_{xx},...t,x,y,z,...) = \frac{u(t+\Delta t,x,y,z,...) - u(t,x,y,z,...)}{\Delta t} + \mathcal{O}(\Delta t).$$

This can be derived by rearranging into the form of a Taylor expansion:

$$u(t+\Delta t,x,y,z,...) = u(t,x,y,z,...) + \Delta t f(u,u_x,u_{xx},...t,x,y,z,...) + \mathcal{O}\left((\Delta t)^2\right).$$

This is called the *forward-in-time* or *Euler's* method. Note that the spatial partial derivatives of u, like u_x and u_{xx}, on the right-hand side are simply evaluated at time t in Euler's method. We'll find that when using higher-order explicit timestepping methods (i.e., methods with truncation error that scales as a higher power of Δt), these spatial partial derivatives will also be evaluated at a past time. In *implicit* methods, f might also be evaluated at the future time $t+\Delta t$. We will review implicit methods in the next chapter.

Algorithmically, Euler's method takes the same form when solving the ODE

$$\frac{dy(t)}{dt} = y'(t) = f(y,t) : y(t+\Delta t) = y(t) + \Delta t f(y,t) + \mathcal{O}\left((\Delta t)^2\right).$$

Notice that the dominant truncation error in a single step (also known as the **local truncation error**) is proportional to $(\Delta t)^2$. Yet we say that Euler's/the forward-in-time method has dominant truncation error proportional to Δt, which makes Euler's method incredibly inefficient. Why do we say this?

Local truncation error is not particularly useful, as we normally need to solve the PDE from t_0 to some *fixed* T, which will be many iterations ("steps") away from t_0. To reach $t=T$ at a given step size Δt, we will need to step forward a total of $(T-t_0)/(\Delta t)$ iterations. Thus the **total accumulated error** associated with Euler's method after $(T-t_0)/(\Delta t)$ is simply the error in a single step, $(\Delta t)^2$, multiplied by the total number of steps, which is proportional to $1/(\Delta t)$. Thus the total accumulated error is proportional to Δt, which is why we typically refer to Euler's method as a *first-order method*.

Example: Suppose after N iterations and fixed time $t_f = N\Delta t$, we measure the error in Euler's method to be E. How many iterations M would be necessary to drop the error at time t_f by a factor of 100? What would the timestep $\Delta t'$ need to be? *Answer:* $M = 100N$, $\Delta t' = \Delta t/100$. Thus the computational cost to significantly drop our truncation errors at some time t_f grows quite rapidly.

This leads to the question, why not take a time derivative that is accurate to the next higher-order in the timestep? Then the error, $E_{\text{second-order}}$, at $t = t_f$ would be given by:

$$E_{\text{second-order}} \approx k'(\Delta t)^2,$$

where k' is some constant.

Example: So to drop error in timestepping for this second-order method by a factor of 100, we would only need to decrease Δt by a factor of 10. So if our numerical errors are dominated by truncation error in time, moving to a higher-order timestepping scheme may drop the cost of our numerical scheme by up to *a factor of 10*!

This is why Euler's method is almost never used in serious numerical work.

So how can we move to a higher order method in time? Recall that to get a higher-order method in spatial derivatives, we simply moved to a higher finite-differencing order. This idea can be applied to time derivatives as well, as demonstrated in Sec. 13.1 below with the "Staggered Leapfrog" method, but this method introduces new complications having to do with the fact that starting the algorithm requires data at *two* initial times. This is inconvenient and poses significant challenges when solving, e.g., the advection equation as we will find.

In Sec. 13.2 in this chapter, we will explore higher-order methods for stepping forward in time based on the method of lines. These methods are superior in that they require data only be provided at an initial time, and can provide solutions with higher-order accuracy in time than Euler's method. Again, the method of lines applies higher-order algorithms originally developed for numerically solving ODE initial value problems, to serve as the foundation for stepping *PDE* initial value problems forward in time.

13.1 Staggered Leapfrog

The most obvious idea to increase the timestepping order would be to simply use the second-order expression for first derivative we have already derived for spatial derivatives, namely:

$$\partial_t y(t) = \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t} + \mathcal{O}\left((\Delta t)^2\right).$$

Let's apply this timestepping strategy to the one-directional wave equation:

$$\partial_t u(x,t) = -v\partial_x u(x,t),$$

using second-order for the spatial derivative as well. Then our second-order-accurate numerical representation of this PDE becomes:

$$\frac{u_j^{n+1} - u_j^{n-1}}{2\Delta t} = -v\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}$$

which implies

$$u_j^{n+1} - u_j^{n-1} = -\frac{v\Delta t}{\Delta x}(u_{j+1}^n - u_{j-1}^n)$$

Let's now apply the von Neumann stability analysis at iteration n, where

$$\tilde{u}_j^n = G(k\Delta x)^n \tilde{u}_j^0 = G(k\Delta x)^n e^{ikj\Delta x}$$

to get

$$\left[G(k\Delta x)^{n+1} - G(k\Delta x)^{n-1}\right]\tilde{u}_j^0 = -\frac{v\Delta t}{\Delta x}G(k\Delta x)^n\left(\tilde{u}_{j+1}^0 - \tilde{u}_{j-1}^0\right)$$

Next multiply both sides by $G(k\Delta x)^{1-n}$:

$$\left(G(k\Delta x)^2 - 1\right)\tilde{u}_j^0 \;\; = \;\; -G(k\Delta x)\frac{v\Delta t}{\Delta x}\left(\tilde{u}_{j+1}^0 - \tilde{u}_{j-1}^0\right).$$

But note that

$$\tilde{u}_{j\pm1}^0 = e^{ik(j\pm1)\Delta x} = e^{\pm ik\Delta x}\tilde{u}_j^0.$$

So

$$\tilde{u}_{j+1}^0 - \tilde{u}_{j-1}^0 = (e^{ik\Delta x} - e^{-ik\Delta x})\tilde{u}_j^0,$$

and we have

$$\begin{aligned}
\left(G(k\Delta x)^2 - 1\right)\tilde{u}_j^0 &= -G(k\Delta x)\frac{v\Delta t}{\Delta x}\left(\tilde{u}_{j+1}^0 - \tilde{u}_{j-1}^0\right) \\
&= -G(k\Delta x)\frac{v\Delta t}{\Delta x}\left(e^{ik\Delta x} - e^{-ik\Delta x}\right)\tilde{u}_j^0 \\
&= -2iG(k\Delta x)\frac{v\Delta t}{\Delta x}\sin(k\Delta x)\tilde{u}_j^0 \\
\implies 0 &= G(k\Delta x)^2 + 2iG(k\Delta x)\frac{v\Delta t}{\Delta x}\sin(k\Delta x) - 1 \\
0 &= G(k\Delta x)^2 + 2iG(k\Delta x)a\sin(k\Delta x) - 1,
\end{aligned}$$

where $a = \frac{v\Delta t}{\Delta x}$.

This is a quadratic equation for $G(k\Delta x)$, which we can quickly solve:

$$\begin{aligned}
G(k\Delta x) &= \frac{-2ia\sin(k\Delta x) \pm \sqrt{-4a^2\sin^2(k\Delta x) + 4}}{2} \\
&= -ia\sin(k\Delta x) \pm \sqrt{1 - a^2\sin^2(k\Delta x)}.
\end{aligned}$$

Let's define $\beta = a\sin(k\Delta x)$. Then we get

$$G(k\Delta x) = -i\beta \pm \sqrt{1-\beta^2}.$$

Notice that $\sin(k\Delta x)$ varies between 0 to 1 in the range $0 \le k\Delta x \le \pi$, and $a > 0$. The $\sqrt{1-\beta^2}$ indicates that there are three cases here: $\beta = 1$, $\beta > 1$, and $\beta < 1$; in the $\beta > 1$ case the square root term is purely imaginary and in the $\beta < 1$ case, it is purely real:

1. $\beta = 1$. Then $|G|^2 = |-i|^2 = 1 \implies$ stable!
2. $\beta > 1$. Then $\sqrt{\beta^2 - 1}$ is real and we can write $\sqrt{1-\beta^2} = i\sqrt{\beta^2-1}$, a purely imaginary number.

$$\begin{aligned}
|G(k\Delta x)|^2 &= \left|i\left(-\beta \pm \sqrt{\beta^2-1}\right)\right|^2 \\
&= \left[-\beta \pm \sqrt{\beta^2-1}\right]^2 \\
&= \beta^2 \mp 2\beta\sqrt{\beta^2-1} + (\beta^2 - 1) \\
&= \beta^2 \mp 2\beta\sqrt{\beta^2\left(1 - \frac{1}{\beta^2}\right)} + (\beta^2 - 1) \\
&= \beta^2 \mp 2\beta^2\sqrt{1 - \frac{1}{\beta^2}} + (\beta^2 - 1) \\
&= 2\beta^2 \mp 2\beta^2\sqrt{1 - \frac{1}{\beta^2}} - 1 \\
&= 2\beta^2\left[1 \mp \sqrt{1 - \frac{1}{\beta^2}}\right] - 1.
\end{aligned}$$

Plugging in to the inequality $0 \le |G(k\Delta x)|^2 \le 1$, we get

$$\begin{aligned}
\implies 0 &\le 2\beta^2\left[1 \mp \sqrt{1 - \frac{1}{\beta^2}}\right] - 1 \le 1 \\
\implies 1 &\le 2\beta^2\left[1 \mp \sqrt{1 - \frac{1}{\beta^2}}\right] \le 2 \\
\implies \frac{1}{2} &\le \beta^2\left[1 \mp \sqrt{1 - \frac{1}{\beta^2}}\right] \le 1 \\
\implies \frac{1}{2} &\le (\text{some number} > 1)\left[1 \mp \sqrt{\text{some number between 0 and 1}}\right] \le 1 \\
\implies \frac{1}{2} &\le (\text{some number} > 1)\left[1 \mp (\text{some number between 0 and 1})\right] \le 1.
\end{aligned}$$

Both of the roots ($\mp$) are valid, and the positive root doesn't fit the inequality, so this scheme is unstable for $\beta > 1$. (You will find that the inequality is properly satisfied for the negative root, with $\beta \to \infty$ corresponding to the lower bound of the inequality, and $\beta = 1$ corresponding to the upper bound. However, this is a moot point since both roots are valid.)

3. $\beta < 1$. Then $\sqrt{1-\beta^2}$ is a real number and we have

$$\begin{aligned} |G(k\Delta x)|^2 &= \left|\left(-i\beta \pm \sqrt{1-\beta^2}\right)\right|^2 \\ &= \beta^2 + (1-\beta^2) \\ &= 1. \end{aligned}$$

Clearly this growth factor is between 0 and 1, so the numerical scheme is stable if $\beta < 1$.

We conclude that the Staggered Leapfrog scheme is *stable* so long as $\beta = a\sin(k\Delta x) \leq 1$. Further, since $a > 0$ and $\sin(k\Delta x \geq 0$ (in the resolvable range $0 \leq k\Delta x \leq \pi$), the CFL criterion can be derived from

$$0 \leq a\sin(k\Delta x) \leq 1,$$

where $\sin(k\Delta x) = 0$ at $k\Delta x = 0, \pi$. Clearly the *most restrictive* inequality comes when $\sin(k\Delta x) = 1$ at $k\Delta x = \pi/2$, so the CFL criterion is

$$a = \frac{v\Delta t}{\Delta x} \leq 1.$$

What makes the Staggered Leapfrog scheme stand out is the fact that where it is stable, $|G| = 1$, meaning that the solution's amplitude is *not diminished* from one iteration to the next due to the growth factor.

The Staggered Leapfrog Method: Implementation Considerations

Mesh-drifting/checkerboard instability: Imagine the grid in space & time superposed on a chessboard. The finite differencing is set up so that the black squares can never communicate with the white squares. This does not typically affect the stability of simple linear PDEs. However, for nonlinear PDEs with large gradients, this can lead to the so-called mesh-drifting, or checkerboard instability. To fix it, one simply adds a small parabolic diffusive term, which breaks this decoupling.

Filling in $n = -1$: Notice that at timestep $n = 0$ we need solution values at $n = -1$ and $n = 0$. Generally, we can solve this by a simple timeshift: $t \to t + \Delta t$, so that $n = 0$ corresponds to time $t = \Delta t$. Then we can use the Euler timestepping (a.k.a., Forward-Time timestepping, where $(\partial_t f)^n$ is represented by the first-order accurate $(f^{n+1} - f^n)/\Delta t)$ to get from $t = 0$ to $t = \Delta t$, taking as many sub-timesteps as needed to get our desired accuracy. At this point, we then have data at $t = 0$ and $t = \Delta t$, and can then proceed with the staggered leapfrog method.

13.2 Explicit Runge-Kutta Second-Order Methods

Source: `http://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node5.html`

Runge-Kutta Second-Order methods are a type of second-order **predictor-corrector** method. Such methods perform an estimate step from iteration i to $i+1$, using e.g., Euler's method, to get a prediction of the solution at step $i+1$. This is the "predictor" step. Then it uses this prediction to perform at least one more "corrector" step, which increases the accuracy of the solution.

Let's return to the simple ODE:

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

A general two-step Runge-Kutta method is as follows,

$$\begin{aligned} k_1 &= \Delta t f(y_i, t_i) \\ k_2 &= \Delta t f(y_i + \beta k_1, t_i + \alpha\Delta t) \end{aligned}$$

where k_1 is the predicted value of the solution (simply the Euler step), k_2 is the corrected value of the solution, and α and β are unknowns that must be chosen such that the method is locally third-order convergent in Δt. Then to get our solution at the next iteration, we simply sum a linear combination of the k_1 and k_2:

$$y_{i+1} = y_i + ak_1 + bk_2 + \mathcal{O}\left((\Delta t)^3\right),$$

where a and b are additional unknowns that must be chosen so that the method is locally third-order convergent in Δt.

Our goal now is to find these values of a, b, α, and β so that our scheme possesses local truncation error that converges to zero at third order in Δt (i.e., local truncation error proportional to $(\Delta t)^3$). We proceed by relating $y(t_i)$, the solution at some step t_i, to the solution at the next step $t_{i+1} = t_i + \Delta t$ using the Taylor expansion of $y(t_i + \Delta t)$ to third order in Δt:

$$y(t_i + \Delta t) = y(t_i) + \Delta t y'(t_i) + \frac{(\Delta t)^2}{2!} y''(t_i) + \mathcal{O}\left((\Delta t)^3\right)$$

Note that

$$y''(t_i) = \frac{df(y_i, t_i)}{dt}$$

Using the chain rule for partial derivatives, we get

$$\begin{aligned} y''(t_i) &= \frac{\partial f(y_i, t_i)}{\partial t} + \frac{\partial f(y_i, t_i)}{\partial y}\frac{dy}{dt} \\ &= \partial_t f(y_i, t_i) + f(y_i, t_i)\partial_y f(y_i, t_i) \end{aligned}$$

Thus we get:

$$y(t_i + \Delta t) = y(t_i) + \Delta t y'(t_i) + \frac{(\Delta t)^2}{2!}(\partial_t f(y_i, t_i) + f(y_i, t_i)\partial_y f(y_i, t_i)) + \mathcal{O}\left((\Delta t)^3\right).$$

Let's simplify the notation a bit

$$y_{i+1} = y_i + \Delta t f_i + \frac{(\Delta t)^2}{2!}([\partial_t f]_i + f_i[\partial_y f]_i) + \mathcal{O}\left((\Delta t)^3\right),$$

where we have defined $f_i = f(y_i, t_i)$, and the partial derivatives of f with respect to variable z at $(y, t) = (y_i, t_i)$ as $[\partial_z f]_i = [\partial_z f]_{y=y_i,\ t=t_i}$.

Let's return to the definition of k_2, which can be Taylor expanded as well (using the standard Taylor series expression for a function of two variables), at iteration n:

$$\begin{aligned} k_2 &= \Delta t f(y_i + \beta k_1, t_i + \alpha \Delta t) \\ &= \Delta t\left([f(y,t)]_i + \alpha \Delta t[\partial_t f]_i + \beta k_1[\partial_y f]_i\right) + \mathcal{O}\left((\Delta t)^3\right) \end{aligned}$$

Let's substitute all of this back into the expression for y_{i+1}. For notational simplicity, let's agree that $f = f(y_i, t_i)$:

$$\begin{aligned} y_{i+1} &= y_i + ak_1 + bk_2 + \mathcal{O}\left((\Delta t)^3\right) \\ &= y_i + a\Delta t f + b\Delta t\left(f + \alpha \Delta t \partial_t f + \beta k_1 \partial_y f\right) + \mathcal{O}\left((\Delta t)^3\right) \\ &= y_i + a\Delta t f + b\Delta t\left(f + \alpha \Delta t \partial_t f + \beta \Delta t f \partial_y f\right) + \mathcal{O}\left((\Delta t)^3\right) \\ &= y_i + (a+b)\Delta t f + b(\Delta t)^2\left(\alpha \partial_t f + \beta f \partial_y f\right) + \mathcal{O}\left((\Delta t)^3\right) \end{aligned}$$

We can compare this to the Taylor expansion of y_{i+1} to obtain values for our coefficients:

$$\begin{aligned} a + b &= 1 \\ \alpha b &= \frac{1}{2} \\ \beta b &= \frac{1}{2} \end{aligned}$$

Notice we have 3 equations and 4 unknowns! This means there are infinitely many ways we could write this stepping to give us a second-order accurate scheme, so let's choose to write the most general scheme with free parameter α:

$$\begin{aligned} \alpha b = \frac{1}{2} \text{ and } \beta b = \frac{1}{2} \quad &\implies \quad \alpha = \beta \\ &\implies \quad b = \frac{1}{2\alpha} \quad (\alpha \neq 0) \\ a + b = 1 \quad &\implies \quad a = 1 - \frac{1}{2\alpha}. \end{aligned}$$

Thus the generic, second-order Runge-Kutta method may be written as

$$\begin{aligned} k_1 &= \Delta t f(y_i, t_i) \\ k_2 &= \Delta t f(y_i + \alpha k_1, t_i + \alpha \Delta t) \\ y_{i+1} &= y_i + \left(1 - \frac{1}{2\alpha}\right) k_1 + \frac{1}{2\alpha} k_2 + \mathcal{O}\left((\Delta t)^3\right), \end{aligned}$$

for any $\alpha \neq 0$. *Open exercise: What does it mean if $\alpha < 0$?*

Picking the following specific values for α results in the corresponding methods listed below:

- $\alpha = \frac{1}{2}$: The **midpoint method**.
- $\alpha = 1$: **Heun's method**.
- $\alpha = \frac{2}{3}$: The **Ralston method**. This method was derived with a goal of minimizing the error associated with the $\mathcal{O}\left((\Delta t)^3\right)$ term. We will explore the impact of this choice in an example below.

13.2.1 Moving from t_0 to t_1: Worked Examples of Heun's Method

Example #1

Consider the ODE

$$y'(t) = -ty(t), \quad y(0) = y(t_0) = 1.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$ to fifth order in Δt.
- Next solve this ODE using the second-order-accurate Heun's method *by hand* with a step size of Δt to find $y(\Delta t)$. Confirm that the solution obtained when using Heun's method has an error term that is at worst $\mathcal{O}\left((\Delta t)^3\right)$. If the dominant error is proportional to a higher power of Δt, explain the discrepancy.

This ODE is separable, and has the solution of a normal, Gaussian distribution:

$$y(t) = ce^{-t^2/2}$$

Consider initial conditions $y(0) = 1$ Then the solution is

$$y(t) = e^{-t^2/2}$$

We know $y(0) = y(t_0) = y_0 = 1$, so let's now find $y_1 = y(\Delta t)$ using Heun's method (i.e., the generic RK2 method with $\alpha = 1$):

$$\begin{aligned} k_1 &= \Delta t f(y_n, t_n) \\ k_2 &= \Delta t f(y_n + k_1, t_n + \Delta t)) \\ y_{n+1} &= y_n + \frac{1}{2}(k_1 + k_2) + \mathcal{O}\left((\Delta t)^3\right). \end{aligned}$$

Here, $f(y,t) = -ty$, so

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) \\ &= \Delta t \times 0 \\ &= 0 \\ k_2 &= \Delta t f(y_0 + k_1, 0 + \Delta t) \\ &= \Delta t f(y_0 + 0, 0 + \Delta t) \\ &= \Delta t (y_0)(-\Delta t) \\ &= -\Delta t^2 \\ y(\Delta t) &= y(0) + \frac{1}{2}(k_1 + k_2) \\ &= 1 - (\Delta t)^2/2 \end{aligned}$$

Notice that the exact solution is

$$y(\Delta t) = e^{-(\Delta t)^2/2} = 1 - (\Delta t)^2/2 + (\Delta t)^4/8 + \mathcal{O}\left((\Delta t)^6\right)$$

Thus Heun's method yields the correct solution at the end of the first iteration to *fourth order* in the error, in this case. This is in part because Heun's method preserves the even-ness of the function (i.e., $f(-t) = f(t)$). In general you should only expect the solution at the end of the first iteration to be accurate to third-order in the error.

Example #2

Consider the ODE

$$y' = y - 2te^{-2t}, \quad y(0) = y(t_0) = 0.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$ to fifth order in Δt.
- Next solve this ODE using Heun's method *by hand* with a step size of Δt to find $y(\Delta t)$. Confirm that the solution obtained when using Heun's method has an error term that is at worst $\mathcal{O}\left((\Delta t)^3\right)$. If the dominant error is proportional to a higher power of Δt, explain the discrepancy.
- Finally solve this ODE using the Ralston method *by hand* with a step size of Δt to find $y(\Delta t)$. Is the coefficient on the dominant error term closer to the exact solution than Heun's method?

We can solve this equation via the method of integrating factors, which states that ODEs of the form:

$$y'(t) + p(t)y(t) = g(t)$$

are solved via

$$y(t) = \frac{1}{\mu(t)}\left[\int \mu(s)g(s)ds + c\right],$$

where the integrating factor $\mu(t)$ is given by

$$\mu(t) = \exp\left(\int p(t)dt\right)$$

Here, $p(t) = -1$ and $g(t) = -2te^{-2t}$. Then

$$\mu(t) = \exp\left(-\int dt\right) = e^{-t+c} = ke^{-t}$$

and

$$\begin{aligned} y(t) &= e^t/k\left[\int ke^{-s}(-2se^{-2s})ds + c\right] = -2e^t\left[\int se^{-3s}ds + c'\right] \\ &= -2e^t\left[e^{-3t}\left(-\frac{t}{3}-\frac{1}{9}\right)+c'\right] = -2e^{-2t}\left(-\frac{t}{3}-\frac{1}{9}\right) - 2c'e^t \\ &= e^{-2t}\left(2\frac{t}{3}+\frac{2}{9}\right) + c''e^t \end{aligned}$$

If $y(0) = 0$ then we can compute the integration constant c'', and $y(t)$ becomes

$$y(t) = \frac{2}{9}e^{-2t}\left(3t + 1 - e^{3t}\right).$$

The Taylor Series expansion of the exact solution about $t = 0$ evaluated at $y(\Delta t)$ yields

$$y(\Delta t) = -(\Delta t)^2 + (\Delta t)^3 - \frac{3(\Delta t)^4}{4} + \frac{23(\Delta t)^5}{60} - \frac{19(\Delta t)^6}{120} + O\left((\Delta t)^7\right).$$

Next we evaluate $y(\Delta t)$ using Heun's method. We know $y(0) = y_0 = 0$ and $f(y,t) = y - 2te^{-2t}$, so

$$\begin{aligned} k_1 &= \Delta t f(y(0), 0) \\ &= \Delta t \times 0 \\ &= 0 \\ k_2 &= \Delta t f(y(0) + k_1, 0 + \Delta t) \\ &= \Delta t f(y(0) + 0, 0 + \Delta t) \\ &= \Delta t(-2\Delta t e^{-2\Delta t}) \\ &= -2(\Delta t)^2 e^{-2\Delta t} \\ y(\Delta t) &= y_0 + \frac{1}{2}(k_1 + k_2) + \mathcal{O}\left((\Delta t)^3\right) \\ &= 0 - (\Delta t)^2 e^{-2\Delta t} \\ &= -(\Delta t)^2(1 - 2\Delta t + 2(\Delta t)^2 + ...) \\ &= -(\Delta t)^2 + 2(\Delta t)^3 + \mathcal{O}\left((\Delta t)^4\right). \end{aligned}$$

Thus the coefficient on the $(\Delta t)^3$ term is wrong, but this is completely consistent with the fact that our stepping scheme is only third-order accurate in Δt.

Let's see if we can improve them with the Ralston method:

$$\begin{aligned} k_1 &= \Delta t f(y_n, t_n) \\ k_2 &= \Delta t f(y_n + 2k_1/3, t_n + 2\Delta t/3)) \\ y_{n+1} &= y_n + \left(\frac{1}{4}k_1 + \frac{3}{4}k_2\right) + \mathcal{O}\left((\Delta t)^3\right). \end{aligned}$$

Applying the Ralston method to our ODE, we get:

$$\begin{aligned}
k_1 &= \Delta t f(y(0), 0) \\
&= \Delta t \times 0 \\
&= 0 \\
k_2 &= \Delta t f(y(0) + 2k_1/3, 0 + 2\Delta t/3) \\
&= \Delta t f(y(0) + 0, 0 + 2\Delta t/3) \\
&= \Delta t(-2(2\Delta t/3)e^{-4\Delta t/3}) \\
&= -4/3(\Delta t)^2 e^{-4\Delta t/3} \\
y(\Delta t) &= y_0 + \frac{1}{4}k_1 + \frac{3}{4}k_2 + \mathcal{O}\left((\Delta t)^3\right) \\
&= 0 - (\Delta t)^2 e^{-4\Delta t/3} \\
&= -(\Delta t)^2\left(1 - \frac{4\Delta t}{3} + \frac{1}{2!}\left(\frac{4\Delta t}{3}\right)^2 + ...\right) \\
&= -(\Delta t)^2 + 4/3(\Delta t)^3 + \mathcal{O}\left((\Delta t)^4\right)
\end{aligned}$$

Notice that 4/3 is closer to the exact Taylor Series expansion coefficient of 1 than the 2 we found with Heun's method. Recall that α in the Ralston method was chosen to minimize truncation error, and indeed the Ralson method possesses lower truncation error than Heun's method in this case.

13.2.2 Demonstration that Total Accumulated Error is $\mathcal{O}\left((\Delta t)^2\right)$ with Heun's Method

Example #3

Consider the simple ODE:

$$y' = -y, \quad y(0) = y_0 = 1.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$, $y(t = 2\Delta t)$, and $y(t = 3\Delta t)$ to third order in Δt (i.e., up to and including the $(\Delta t)^3$ term).
- Next solve this ODE using Heun's method *by hand* with a step size of Δt to find $y(\Delta t)$, $y(2\Delta t)$, and $y(3\Delta t)$.
- Compute the absolute error between the exact solution and the numerical solution for the $(\Delta t)^3$ term. Notice that the constant coefficient increases linearly with the number of steps taken.
- Next suppose we wish to solve the ODE from $t_0 = 0$ to some final t, t_f, taking N steps. Derive an expression relating t_f, N, and Δt to prove that after N steps, the dominant error is proportional to $(\Delta t)^2$ and not $(\Delta t)^3$.

This is an exact equation, with solution $y = e^{-t}$. The Taylor expansion about $t = \Delta t$ is given by

$$y(\Delta t) = y_1 = 1 - \Delta t + \frac{\Delta t^2}{2!} - \frac{\Delta t^3}{3!} + \mathcal{O}\left((\Delta t)^4\right)$$

Similarly, the exact solution at $t = 2\Delta t$ is given by the Taylor expansion:

$$\begin{aligned}
y(2\Delta t) &= 1 - 2\Delta t + \frac{(2\Delta t)^2}{2!} - \frac{(2\Delta t)^3}{3!} + ... \\
&= 1 - 2\Delta t + 2(\Delta t)^2 - \frac{8}{6}(\Delta t)^3 + ...
\end{aligned}$$

Finally, the exact solution at $t = 3\Delta t$ is given by the Taylor expansion:

$$\begin{aligned}
y(3\Delta t) &= 1 - 3\Delta t + \frac{(3\Delta t)^2}{2!} - \frac{(3\Delta t)^3}{3!} + ... \\
&= 1 - 3\Delta t + \frac{3}{4}(\Delta t)^2 - \frac{9}{2}(\Delta t)^3 + ...
\end{aligned}$$

Next we focus on Heun's method. For $y_1 = y(\Delta t)$, this method yields:

$$\begin{aligned}
k_1 &= \Delta t f(y_0, t_0) \\
&= -\Delta t \times 1 \\
&= -\Delta t \\
k_2 &= \Delta t f(y_0 + k_1, t_0 + \Delta t) \\
&= \Delta t f(1 - \Delta t, 0 + \Delta t) \\
&= -\Delta t(1 - \Delta t) = -\Delta t + (\Delta t)^2 \\
y_1 &= y_0 + \frac{1}{2}(k_1 + k_2) + \mathcal{O}\left((\Delta t)^3\right) \\
&= 1 + \frac{1}{2}(-\Delta t - \Delta t + (\Delta t)^2) \\
&= 1 - \Delta t + \frac{(\Delta t)^2}{2},
\end{aligned}$$

which indeed matches the exact solution up to and including the second-order term in the Taylor expansion. Thus the absolute error of the dominant error term at the end of iteration 1 is

$$E_1^{\text{dom}} = |\text{num} - \text{exact}| = \frac{1}{6}(\Delta t)^3.$$

Let's now compute y_2:

$$\begin{aligned}
k_1 &= \Delta t f(y_1, t_1) \\
&= -\Delta t \left(1 - \Delta t + \frac{(\Delta t)^2}{2}\right) \\
&= -\Delta t + (\Delta t)^2 - \frac{(\Delta t)^3}{2} \\
k_2 &= \Delta t f(y_1 + k_1, t_1 + \Delta t) \\
&= -\Delta t \left(1 - \Delta t + \frac{(\Delta t)^2}{2} - \Delta t + (\Delta t)^2 - \frac{(\Delta t)^3}{2}\right) \\
&= -\Delta t + 2(\Delta t)^2 - \frac{3}{2}(\Delta t)^3 + \frac{1}{2}(\Delta t)^4 \\
y_2 &= y_1 + \frac{1}{2}(k_1 + k_2) \\
&= 1 - \Delta t + \frac{(\Delta t)^2}{2} + \frac{1}{2}\left(-\Delta t + (\Delta t)^2 - \frac{(\Delta t)^3}{2} - \Delta t + 2(\Delta t)^2 - \frac{3}{2}(\Delta t)^3 + \frac{1}{2}(\Delta t)^4\right) \\
&= 1 - 2\Delta t + 2(\Delta t)^2 - (\Delta t)^3 + \frac{1}{4}(\Delta t)^4.
\end{aligned}$$

Thus the dominant error at iteration 2, which occurs in the 3rd-order term, is

$$E_2^{\text{dom}} = |\text{num} - \text{exact}| = \left|-1 + \frac{8}{6}\right|(\Delta t)^3 = \frac{2}{6}(\Delta t)^3.$$

If you proceed and compute y_3, y_4, etc., you will find that the error in the dominant-order (i.e., 3rd-order in Δt) term at iteration N follows the pattern

$$E_i^{\text{dom}} = \frac{N}{6}(\Delta t)^3.$$

Thus the dominant error is indeed proportional to $N(\Delta t)^3$, so that if we wish to measure the dominant error at some fixed $t = T = N\Delta t$, it will be given by

$$E_i^{\text{dom}}(t) = \frac{1}{6}N(\Delta t)^3 = \frac{T}{6}\frac{(\Delta t)^3}{\Delta t} \propto (\Delta t)^2.$$

So if we increased our sampling rate, going from Δt to $\Delta t/2$, the number of iterations required to evaluate the solution at $t = T$ would double. But we have just shown that the dominant error term at $t = T$ would double as well, meaning that at some fixed $t = T = N\Delta t$, the error E_i^{dom} should grow proportionally to $(\Delta t)^3/(\Delta t) = (\Delta t)^2$.

13.2.3 Additional ODE Example: Heun versus Ralston

Example #4

Consider the simple ODE:

$$y' = y + t, \quad y(0) = y_0 = 1.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$, $y(t = 2\Delta t)$, and $y(t = 3\Delta t)$ to third order in Δt (i.e., up to and including the $(\Delta t)^3$ term).
- Next solve this ODE using Heun's method *by hand* with a step size of Δt to find $y(\Delta t)$.
- Finally solve this ODE using the Ralston method *by hand* with a step size of Δt to find $y(\Delta t)$.
- Explain the similarities or differences between Heun's method and the Ralston method when computing $y(\Delta t)$.

The ODE takes the general form

$$y'(t) + p(t)y(t) = g(t),$$

which is solved via the Method of Integrating Factors:

$$y(t) = \frac{1}{\mu(t)} \int \mu(s)g(s)ds,$$

where the integrating factor $\mu(t)$ is given by

$$\mu(t) = \exp\left(\int p(t)dt\right).$$

Here, $p(t) = -1$ and $g(t) = t$. Then

$$\mu(t) = \exp\left(-\int dt'\right) = e^{-t+c} = ke^{-t}$$

and

$$y(t) = e^t/k \int^t kse^{-s}ds = e^t \int^t se^{-s}ds.$$

Integration by parts is based on the product rule:

$$(u(s)v(s))' = u'v + v'u \implies \int u'vds = \int (uv)'ds - \int v'uds,$$

so defining $u' = e^{-s}$ and $v = s$, we get $u = -e^{-s}$ and $v' = 1$. Thus the solution becomes

$$\begin{aligned} y(t) &= e^t \int^t se^{-s}ds = e^t \left[-te^{-t} - \int^t e^{-s} + C_0\right] \\ &= e^t \left[-te^{-t} - e^{-t} + C\right] = Ce^t - t - 1. \end{aligned}$$

Plugging in our initial condition, $y(t = 0) = 1$, we get $y(0) = 1 = C - 1$, so $C = 2$, and our solution is given by

$$y(t) = 2e^t - t - 1.$$

Taylor expanding this about $t = 0$, we get

$$y(\Delta t) = 2\left(1 + \Delta t + \frac{(\Delta t)^2}{2!} + \frac{(\Delta t)^3}{3!} + ...\right) - \Delta t - 1 = 1 + \Delta t + (\Delta t)^2 + \frac{(\Delta t)^3}{3}.$$

Next we apply Heun's method. Recall our ODE implies that $f(y, t) = y + t$, so Heun's method for the zeroth iteration ($i = 0$) gives:

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) = \Delta t(y_0 + t_0) = \Delta t \\ k_2 &= \Delta t f(y_0 + k_1, t_0 + \Delta t) = \Delta t\left((y_0 + k_1) + (t_0 + \Delta t)\right) = \Delta t\left((1 + \Delta t) + (0 + \Delta t)\right) = \Delta t + 2(\Delta t)^2 \\ y(\Delta t) = y_1 &= y_0 + \frac{1}{2}(k_1 + k_2) = 1 + \frac{1}{2}\left(\Delta t + \Delta t + 2(\Delta t)^2\right) = 1 + \Delta t + (\Delta t)^2, \end{aligned}$$

which is consistent with the exact solution up to and including the second-order term in powers of Δt!

Next we apply the Ralston method:

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) = \Delta t(y_0 + t_0) = \Delta t \\ k_2 &= \Delta t f\left(y_0 + \frac{2}{3}k_1, t_0 + \frac{2}{3}\Delta t\right) = \Delta t\left(\left(y_0 + \frac{2}{3}k_1\right) + \left(t_0 + \frac{2}{3}\Delta t\right)\right) = \Delta t\left(\left(1 + \frac{2}{3}\Delta t\right) + \left(0 + \frac{2}{3}\Delta t\right)\right) \\ &= \Delta t + \frac{4}{3}(\Delta t)^2 \\ y(\Delta t) = y_1 &= y_0 + \frac{1}{4}k_1 + \frac{3}{4}k_2 = 1 + \frac{1}{4}\Delta t + \frac{3}{4}\left(\Delta t + \frac{4}{3}(\Delta t)^2\right) = 1 + \Delta t + (\Delta t)^2, \end{aligned}$$

which is consistent with the exact solution up to and including the second-order term in powers of Δt!

In fact, we have found that for this ODE, both the Ralston and Heun's method yield exactly the same result for $y(\Delta t)$. This contrasts with the earlier example, largely because of terms like $e^{-2\Delta t}$ that appeared due to the explicit e^{-2t} in $f(y,t)$, which we needed to Taylor expand in that example. *Open question: Will both methods be equivalent for all $y(t)$ for $t > \Delta t$ as well?*

13.3 Applying the Method of Lines with Heun's Method to Solve the Advection Equation

Let's return to the one-directional wave equation (i.e., the advection equation), as usual, posed as an initial value problem:

$$\partial_t u(x,t) = -v\partial_x u(x,t), \quad u(x,0) = g(x).$$

Recall in Chapter 11, we derived the following pseudocode for stepping these initial data forward in time using the FTCS method:

Pseudocode v4.0: Correct and efficient approach for implementing FTCS Method

```
# We define the 1D arrays un[j] and unp1[j], where un=u^n stores,
# the solution at time iteration n, unp1=u^{n+1} stores the solution,
# at time iteration n+1, and the index j corresponds to a discrete
# location in space.

# First set the initial data
do j=0,J # Iterate over all spatial points
  un[j] = g(j,x0,DeltaX) # Call function that sets initial data
end do

# Next implement the FTCS Method for solving the advection equation:
print_solution(un)
do n=0,N-1
  do J=1,J-1
    unp1[j] = un[j] - (v*Dt)/(2*DeltaX)*(un[j+1] - un[j-1])
  end do
  apply_bcs(unp1)
  if(n+1 is a multiple of print_every):
    print_solution(unp1)
  end if
end do
```

Recall that

- `apply_bcs(u)` could refer to boundary conditions such as Dirichlet, Newman, Robin, periodic, etc. Also, we assume $t_i = i\Delta t$ for $i \in [0, N]$ and $x_j = x_0 + j\Delta x$ for $j \in [0, J]$, where $\Delta t =$`Dt` and $\Delta x =$`DeltaX` have been set.
- `apply_bcs(u)` could refer to boundary conditions such as Dirichlet, Newman, Robin, periodic, etc.

- The function `print_solution()` outputs the solution to a file or to the screen every `print_every` iterations in n, so we can analyze the output from our numerical PDE solver. For example, set `print_every=2` to output at iterations 0,2,4,6,...

The above example implements Euler's method, which can be written as

$$\begin{aligned} k_1 &= \Delta t f(y_n, t_n) \\ y_{n+1} &= y_n + k_1 + \mathcal{O}(\Delta t^2). \end{aligned}$$

Thus we can rewrite the example in a slightly different way, by defining gridfunctions `k1[]`, `un[]`, and `unp1[]` instead. This algorithm is less efficient in terms of memory usage because it involves 3 gridfunctions instead of 2, but it is easier to connect with Euler's method as written above and more importantly with the generic Method of Lines approach we will adopt:

Pseudocode v5.0: FTCS Method using Method of Lines

```
# We define the 1D arrays un[j] and unp1[j], where un=u^n stores,
# the solution at time iteration n, unp1=u^{n+1} stores the solution,
# at time iteration n+1, and the index j corresponds to a discrete
# location in space.

# First set the initial data
do j=0,J # Iterate over all spatial points
  un[j] = g(j,x0,DeltaX) # Call function that sets initial data
end do

# Next implement the FTCS Method for solving the advection equation:
print_solution(un)
do n=0,N-1
  do J=1,J-1
    k1[j] = - (v*Dt)/(2*DeltaX)*(un[j+1] - un[j-1])
  end do
  apply_bcs(k1)

  do J=0,J
    unp1[j] = un[j] + k1[j]
  end do

  if(n+1 is a multiple of print_every):
    print_solution(unp1)
  end if
end do
```

Extending this idea to Heun's Method requires that we extend the above algorithm to set `k1` and `k2`, in the following way.

We are given initial data $u(x,0) = g(x)$. We then set $k_1^{n=0}$ *at all spatial points* according to:

$$k_1^{n=0} = \Delta t f(u, u_x, u_{xx}, ..., t, x, y, z, ...)^{n=0},$$

(see discussion at the start of this chapter), where in this case $f(u, u_x, u_{xx}, ..., t, x, y, z, ...)^{n=0} = -v\partial_x u(x,0) = -v\partial_x g(x)$, and $\partial_x u(x,0)$ should be written using a finite-difference approximation.

$k_1^{n=0}$ can be interpreted as the Euler's method predictor step for $u(x, t = \Delta t)$. As shown above in the Lax method pseudocode (and fully described in Chapter 11), to fully implement the Euler step, we must apply the boundary conditions to $k_1^{n=0}$, and since $k_1^{n=0}$ has the same units as $u(x,t)$, we may simply apply the boundary conditions for u to $k_1^{n=0}$.

Next, we need to evaluate $k_2^{n=0}$ at all spatial points according to the same basic prescription we would have used for solving the ODE, but this time for the PDE:

$$k_2^{n=0} = \Delta t f\left((u^0 + k_1^{n=0}), (u^0 + k_1^{n=0})_x, (u^0 + k_1^{n=0})_{xx}, ..., t, x, y, z, ...\right),$$

Again after evaluating $k_2^{n=0}$ at all but the boundary points, we must apply boundary conditions so that $k_2^{n=0}$ is defined at all spatial points.

Now that $k_1^{n=0}$ and $k_2^{n=0}$ are known at all spatial points, $u(x, \Delta t)$ is simply given by

$$u(x, \Delta t) = u_j^1 = \frac{1}{2}\left(k_{1,j}^{n=0} + k_{2,j}^{n=0}\right).$$

To evaluate the solution at any later time and all spatial locations u_j^n, we simply iterate the above algorithm as needed.

13.4 Beyond Second-Order: Runge-Kutta Fourth Order (RK4)

Thus far, we have explored first- and second-order Runge-Kutta (RK) methods. These methods guarantee that the total accumulated error will be proportional to $(\Delta t)^1$ and $(\Delta t)^2$, respectively. Recall that the first-order (Euler's) RK method can be derived directly from keeping terms up to and including $(\Delta t)^2$ in the Taylor series expansion of $y(t + \Delta t)$. The generic second-order RK method was then derived by expanding the Taylor series to the next higher order.

A similar procedure can be applied to derive higher-order RK methods. Like the generic second-order RK method, these higher-order methods are not unique.

The most widely known Runge-Kutta technique guarantees total accumulated truncation error proportional to $(\Delta t)^4$ for smooth functions $y(t)$. This technique is known as RK4, or sometimes **the Runge-Kutta method**. RK4 won popularity by being both highly robust and rapidly convergent.

Definition of RK4: Given the generic ODE

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

The RK4 method obtains the solution $y(t + \Delta t) = y_{i+1}$ at time t_{i+1} from y_i and t_i via:

$$\begin{aligned}
k_1 &= \Delta t f(y_i, t_i), \\
k_2 &= \Delta t f(y_i + \tfrac{1}{2}k_1, t_i + \tfrac{\Delta t}{2}), \\
k_3 &= \Delta t f(y_i + \tfrac{1}{2}k_2, t_i + \tfrac{\Delta t}{2}), \\
k_4 &= \Delta t f(y_i + k_3, t_i + \Delta t), \\
y_{i+1} &= y_i + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) + \mathcal{O}\left((\Delta t)^5\right).
\end{aligned}$$

Notice that for each timestep, RK4 requires four evaluations of the RHS of the equation. Recall second-order RK methods only required two such evaluations. Thus RK4 will only be superior to second-order methods if you are also able to double your timestep while maintaining at worst the same level of error. This depends on the problem, but usually RK4 is superior to second-order methods in this sense, which is why so many people depend on it. The bottom line is, a higher-order scheme often, but *does not always* yield smaller errors more efficiently than a lower-order scheme.

Now let's return to the first example:

Example #1, Revisited: Validating RK4

Consider the ODE

$$y'(t) = -ty(t) \quad y(0) = 1$$

Again, this ODE is separable, and has the solution of a normal, Gaussian distribution:

$$y(t) = e^{-t^2/2}.$$

- Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$ up to and including the seventh-order term in Δt.
- Next solve this ODE using RK4 *by hand* with a step size of Δt to find $y(\Delta t)$. Verify that the solution obtained when using RK4 has an error term that is at worst $\mathcal{O}\left((\Delta t)^5\right)$.

The exact solution of $y(\Delta t)$ up to and including the $\mathcal{O}\left(\Delta t^7\right)$ term is given by

$$y(\Delta t) = e^{-(\Delta t)^2/2} = 1 - (\Delta t)^2/2 + (\Delta t)^4/8 - (\Delta t)^6/48 + \mathcal{O}\left(\Delta t^8\right).$$

(Notice that since this is an even function, the $\mathcal{O}\left(\Delta t^7\right)$ is zero.)

We know $y(0) = y(t_0) = y_0 = 1$, so $y(\Delta t)$ using RK4 is as follows. Note that $f(y,t) = -ty$, so

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) = \Delta t f(1, 0) \\ &= \Delta t \times (-0 \times y_0) \\ &= 0, \end{aligned}$$

$$\begin{aligned} k_2 &= \Delta t f\left(y_0 + \frac{k_1}{2}, t_0 + \frac{\Delta t}{2}\right) \\ &= \Delta t f(y_0 + 0, t_0 + \Delta t/2) \\ &= \Delta t(y_0)(-\Delta t/2) \\ &= -(\Delta t)^2/2, \end{aligned}$$

$$\begin{aligned} k_3 &= \Delta t f\left(y_0 + \frac{k_2}{2}, t_0 + \frac{\Delta t}{2}\right) \\ &= \Delta t f(y_0 + (-(\Delta t)^2/2)/2, 0 + \Delta t/2) \\ &= \Delta t f(y = 1 - (\Delta t)^2/4, t = \Delta t/2) \\ &= \Delta t \left[-(1 - (\Delta t)^2/4)(\Delta t/2)\right] \\ &= -(\Delta t)^2/2 + (\Delta t)^4/8, \end{aligned}$$

and

$$\begin{aligned} k_4 &= \Delta t f(y_0 + k_3 \Delta t, t_0 + \Delta t) \\ &= \Delta t f(1 + (-(\Delta t)^2/2 + (\Delta t)^4/8)\Delta t, 0 + \Delta t) \\ &= \Delta t \left[-(1 + (-(\Delta t)^2/2 + (\Delta t)^4/8)\Delta t)(\Delta t)\right] \\ &= (\Delta t)^2 \left[-(1 + (-(\Delta t)^2/2 + (\Delta t)^4/8)\Delta t)\right] \\ &= -(\Delta t)^2 + (\Delta t)^4/2 - (\Delta t)^6/8, \end{aligned}$$

so that

$$\begin{aligned} y_1 &= y_0 + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ &= 1 + \tfrac{1}{6}\left[0 + 2(-(\Delta t)^2/2) + 2(-(\Delta t)^2/2 + (\Delta t)^4/8) + (-(\Delta t)^2 + (\Delta t)^4/2 - (\Delta t)^6/8)\right] \\ &= 1 + \tfrac{1}{6}\left[-3(\Delta t)^2 + 2((\Delta t)^4/8) + ((\Delta t)^4/2 - (\Delta t)^6/8)\right] \\ &= 1 + \tfrac{1}{6}\left[-3(\Delta t)^2 + 3(\Delta t)^4/4 - (\Delta t)^6/8\right] \\ &= 1 - \tfrac{1}{2}(\Delta t)^2 + (\Delta t)^4/8 - (\Delta t)^6/48 \end{aligned}$$

Comparing this to the exact solution, we find that although RK4 can only be expected to yield local truncation error proportional to $(\Delta t)^5$, RK4 manages to get the next order term exactly correct in this case!

13.5 The Family of Explicit Runge-Kutta-Like Schemes

"Explicit" Runge-Kutta methods, including all the RK methods discussed in this chapter, can be written:

$$y_{i+1} = y_i + \sum_{\ell=1}^{s} b_\ell k_\ell,$$

where

$$\begin{aligned}
k_1 &= \Delta t f(y_i, t_i) \\
k_2 &= \Delta t f(y_i + [a_{21}k_1], t_i + c_2\Delta t) \\
k_3 &= \Delta t f(y_i + [a_{31}k_1 + a_{32}k_2], t_i + c_3\Delta t) \\
&\vdots \\
k_s &= \Delta t f(y_i + [a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1}], t_i + c_s\Delta t).
\end{aligned}$$

Whereas "explicit" methods generally relate y_{i+1} to possibly complicated functions of y_i, "implicit" methods generally relate y_i to more complicated functions of y_{i+1} requiring a matrix inversion to evaluate y_{i+1}.

Both explicit and implicit methods are usually communicated in the form of so-called "Butcher tableaus", or "Butcher tables". A Butcher table has the general form:

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}$$

The Butcher table for Heun's method is:

$$\begin{array}{c|cc}
0 & & \\
1 & 1 & \\
\hline
 & \frac{1}{2} & \frac{1}{2}
\end{array}$$

We say that a Runge-Kutta method is consistent if

$$\sum_{j=1}^{i-1} a_{ij} = c_i \text{ for } i = 2, \ldots, s$$

Similarly, the Butcher table for the Ralston method is:

$$\begin{array}{c|cc}
0 & & \\
2/3 & 2/3 & \\
\hline
 & 1/4 & 3/4
\end{array}$$

Finally, the RK4 scheme's Butcher table is given by:

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

13.6 Higher-Order-in-Time ODEs/PDEs

We can solve ODEs or PDEs that have higher than first-order time derivatives by applying a trick to reduce them to a set of coupled, first-order ODEs/PDEs in time. Consider the ODE

$$y''(t) - ty'(t) + \cos(t)y(t) = 0, \qquad \text{where } y'(0) = y(0) = 0.$$

Let's define a variable $v(t) = y'(t)$. Then we can rewrite this second order equation as the first-order equations:

$$\begin{aligned} y'(t) &= v(t), \text{where } y(0) = 0, \\ v'(t) &= tv(t) - \cos(t)y(t), \text{where } v(0) = y'(0) = 0. \end{aligned}$$

We can then solve this equation using Euler's method via:

$$\begin{aligned} y_{i+1} &= y_i + \Delta t v_i \\ v_{i+1} &= v_i + \Delta t(t_i v_i - \cos(t_i)y_i) \end{aligned}$$

Notice that Euler's method generalizes quite simply to coupled first-order ODEs.

Heun's method also easily generalizes:

$$\begin{aligned} k_1^y &= \Delta t v_i \\ k_1^v &= \Delta t(t_i v_i - \cos(t_i)y_i) \\ k_2^y &= \Delta t(v_i + k_1^v) \\ k_2^v &= \Delta t((t_i + \Delta t)(v_i + k_1^v) - \cos(t_i + \Delta t)(y_i + k_1^y) \\ y_{i+1} &= y_i + \frac{1}{2}(k_1^y + k_2^y) \\ v_{i+1} &= v_i + \frac{1}{2}(k_1^v + k_2^v). \end{aligned}$$

MATH 522, Spring 2019 Notes

Chapter 14: Implicit Timestepping Schemes; Solving Tridiagonal Matrix Equations

To date, we have focused entirely on timestepping schemes that can be written

$$u_{n+1} = f(u_n, u_{n-1}, ...)$$

I.e., the solution at iteration $n+1$ depends only on the solution at n or previous iterations.

Recall we have demonstrated that the FTCS representation of the heat equation $[\partial_t u(x,t) = \alpha^2 \partial_x^2 u(x,t)]$:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \alpha^2 \left[\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} \right]$$

yields a CFL criterion:

$$\beta = \frac{\alpha^2 \Delta t}{(\Delta x)^2} \leq \frac{1}{2}.$$

Let's turn our attention to the "fully implicit" scheme:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \alpha^2 \left[\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2} \right]$$

This is also called a "backward time" scheme, as it can be rewritten as (defining $\beta = \frac{\alpha^2 \Delta t}{(\Delta x)^2}$):

$$u_j^n = -\beta u_{j-1}^{n+1} + (1+2\beta) u_j^{n+1} - \beta u_{j+1}^{n+1}$$

What happens to the CFL criterion in this fully implicit scheme? To find out, we apply the von Neumann stability analysis.

Recall the analysis starts by injecting a "wiggle"—with unit magnitude and arbitrary, but resolvable wavelength λ corresponding to wavenumber k on our uniform numerical grid with spatial resolution Δx—into the PDE:

$$\tilde{u}_j^0 = e^{ikj\Delta x},$$

so that after one iteration, we can derive

$$\tilde{u}_j^1 = G(k\Delta x) e^{ikj\Delta x} = G(k\Delta x) \tilde{u}_j^0,$$

where $G(k\Delta x)$ is the growth factor.

Based on these definitions, we can immediately write a few expressions relating $\tilde{u}_j^n$ to neighboring j and n:

$$\begin{aligned}
\tilde{u}_j^n &= G(k\Delta x)^n \tilde{u}_j^0 \\
\tilde{u}_j^{n+1} &= G(k\Delta x) \tilde{u}_j^n \\
\tilde{u}_{j\pm1}^n &= e^{\pm ik\Delta x} \tilde{u}_j^n \\
\tilde{u}_{j\pm1}^{n+1} &= G(k\Delta x) e^{\pm ik\Delta x} \tilde{u}_j^n,
\end{aligned}$$

where the third follows as a corollary of the first two relations. You can use these relations to more quickly derive CFL conditions.

Thus

$$\tilde{u}_j^n = -\beta G(k\Delta x)e^{-ik\Delta x}\tilde{u}_j^n + (1+2\beta)G(k\Delta x)\tilde{u}_j^n - \beta G(k\Delta x)e^{ik\Delta x}\tilde{u}_j^n.$$

Dividing both sides by $\tilde{u}_j^n$, we get

$$\begin{aligned} 1 &= -\beta G(k\Delta x)e^{-ik\Delta x} + (1+2\beta)G(k\Delta x) - \beta G(k\Delta x)e^{ik\Delta x} \\ &= -2\beta G(k\Delta x)(\cos(k\Delta x) - 1) + G(k\Delta x) \\ &= -G(k\Delta x)\left[2\beta\left(\cos(k\Delta x) - 1\right) - 1\right] \end{aligned}$$

Solving for $G(k\Delta x)$, we find:

$$G(k\Delta x) = \frac{1}{1 - 2\beta(\cos(k\Delta x) - 1)}.$$

Next we apply the trigonometric identity:

$$2\sin^2 t = (1 - \cos(2t)) \implies 2\sin^2\frac{t}{2} = (1 - \cos t) \implies -2(\cos t - 1) = 4\sin^2\frac{t}{2},$$

we get

$$G(k\Delta x) = \frac{1}{1 + 4\beta\sin^2(k\Delta x/2)}$$

Since $\sin^2(k\Delta x/2) \leq 1$ for all k, $|G(k\Delta x)| \leq 1 \implies$ stability for all Δt!

Although the von Neumann analysis seems to indicate excellent stability properties for this scheme, the difficulty here comes in the numerical implementation.

14.1 Numerical Implementation of Implicit Timestepping

Let's rewrite our implicit scheme for solving the heat equation with all the $n+1$ terms moved to the left-hand side of the equation:

$$\beta u_{j-1}^{n+1} - (1+2\beta)u_j^{n+1} + \beta u_{j+1}^{n+1} = -u_j^n$$

β and u_j^n are known, and based on these known variables we must solve for unknowns u_j^{n+1} and $u_{j\pm1}^{n+1}$, but how?

Let's define our known quantities as $d_j = -u_j^n$, and **drop the $n+1$ superscript on the unknown quantities** u_j^{n+1}. Then when we expand the above expression for a few values of j, we get

$$\begin{aligned} j = 1 \quad & \beta u_0 - (1+2\beta)u_1 + \beta u_2 = d_1 \\ j = 2 \quad & \beta u_1 - (1+2\beta)u_2 + \beta u_3 = d_2 \\ j = 3 \quad & \beta u_2 - (1+2\beta)u_3 + \beta u_4 = d_3 \\ \vdots \quad & \vdots \\ j = J-2 \quad & \beta u_{J-3} - (1+2\beta)u_{J-2} + \beta u_{J-1} = d_{J-2} \\ j = J-1 \quad & \beta u_{J-2} - (1+2\beta)u_{J-1} + \beta u_J = d_{J-1}. \end{aligned}$$

Let's first assume that the values of u_0 and u_J are known and are fixed for all iterations in time n, so that we have Dirichlet boundary conditions. Then we can write the $j = 1$ term as

$$-(1+2\beta)u_1 + \beta u_2 = d_1 - \beta u_0$$

and the $j = J-1$ term as

$$\beta u_{J-2} - (1+2\beta)u_{J-1} = d_{J-1} - \beta u_J$$

Then the matrix takes the form:

$$\begin{bmatrix} -(1+2\beta) & \beta & 0 & \dots & & & & \\ \beta & -(1+2\beta) & \beta & 0 & \dots & & & \\ 0 & \beta & -(1+2\beta) & \beta & 0 & \dots & & \\ \vdots & 0 & & & \ddots & & & 0 \\ & & & & 0 & \beta & -(1+2\beta) & \beta \\ & & & & 0 & 0 & \beta & -(1+2\beta) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{J-2} \\ u_{J-1} \end{bmatrix} = \begin{bmatrix} d_1 - \beta u_0 \\ d_2 \\ d_3 \\ \vdots \\ d_{J-2} \\ d_{J-1} - \beta u_J \end{bmatrix}$$

So to compute the solution at iteration $n+1$ from iteration n, we must solve this *tridiagonal* matrix equation. It turns out there are very efficient means to solve such equations on the computer, and we will next explore the most common strategy, called the *Thomas Algorithm*.

14.1.1 Tridiagonal Matrix Solution Strategy: The Thomas Algorithm

Recall that the fully implicit version of the heat equation we had derived is as follows:

$$\beta u_{j-1}^{n+1} - (1+2\beta)u_j^{n+1} + \beta u_{j+1}^{n+1} = -u_j^n$$

Consider an *even more* generic form of this equation, allowing for the thermal diffusivity to vary in *both space and time*:

$$a_j^n u_{j-1}^{n+1} + b_j^n u_j^{n+1} + c_j^n u_{j+1}^{n+1} = d_j^n.$$

a_j^n, b_j^n, c_j^n, and $d_j^n = -u_j^n$ are known, and we are to solve for u_j^{n+1} for all j. Why write it in this form?

- First, it is more general: allowing for a, b, and c to vary at each gridpoint enables us to use a non-uniformly-spaced grid, as well as a diffusion coefficient that varies over space and time. If we were to use the usual coefficients, i.e.,

  $$[\partial_x^2 u]_j^n = \left[\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}\right],$$

  with a nonuniform grid, we would not get a derivative that was second order in the gridspacing. In fact, we might not get a convergent solution at all.

- Second, it enables us to construct a general solution strategy that works for all tridiagonal matrices, which can appear in contexts outside of numerical solutions of PDEs.

At a given iteration n, a_j^n, b_j^n, c_j^n, d_j^n, and u_j^{n+1} depend only on j, so we drop the n and $n+1$ superscripts. Then the components of the unknown vector $\boldsymbol{u}$ are $u_1, \dots, u_{J-1}$, and the equations to be solved are

$$\begin{aligned} b_1 u_1 + c_1 u_2 &= d_1; & j &= 1 \\ a_2 u_1 + b_2 u_2 + c_2 u_3 &= d_2; & j &= 2 \\ a_3 u_2 + b_3 u_3 + c_3 u_4 &= d_3; & j &= 3 \\ a_i u_{i-1} + b_i u_i + c_i u_{i+1} &= d_i; & j &= i, \dots, J-2 \\ a_{J-1} u_{J-2} + b_{J-1} u_{J-1} &= d_{J-1}; & j &= J-1, \end{aligned}$$

where the Dirichlet boundary conditions u_0 =[constant] and u_J=[constant] have been absorbed into the d_1 and d_{J-1} terms, respectively.

We will next apply Gaussian elimination to convert this matrix into an upper-triangular form. Let's first eliminate the u_1 term in the second ($j=2$) equation:

$$(\text{equation } 2)' = (\text{equation } 2) \cdot b_1 - (\text{equation } 1) \cdot a_2,$$

which yields

$$\begin{aligned} (\text{equation } 2)' = \quad & (a_2 u_1 + b_2 u_2 + c_2 u_3) b_1 - (b_1 u_1 + c_1 u_2) a_2 & &= d_2 b_1 - d_1 a_2 \\ (\text{equation } 2)' = \quad & (b_2 b_1 - c_1 a_2) u_2 + c_2 b_1 u_3 & &= d_2 b_1 - d_1 a_2 \\ (\text{equation } 2)' = \quad & b_2' u_2 + c_2' u_3 & &= d_2', \end{aligned}$$

where we have defined

$$\begin{aligned} b_2' &= b_2 b_1 - c_1 a_2 \\ c_2' &= c_2 b_1 \\ d_2' &= d_2 b_1 - d_1 a_2. \end{aligned}$$

Notice that if we define

$$\begin{aligned} b_1' &= b_1 \\ c_1' &= c_1 \\ d_1' &= d_1, \end{aligned}$$

then

$$\begin{aligned} b_2' &= b_2 b_1' - c_1' a_2 \\ c_2' &= c_2 b_1' \\ d_2' &= d_2 b_1' - d_1' a_2. \end{aligned}$$

We will find this quite useful in our next step.

Let's next eliminate u_2 in the third equation:

$$\begin{aligned} (\text{equation } 3)' &= (\text{equation } 3) \cdot b_2' - (\text{equation } 2)' \cdot a_3 : \\ &= (a_3 u_2 + b_3 u_3 + c_3 u_4) \cdot b_2' - (b_2' u_2 + c_2' u_3)' \cdot a_3 \\ &= u_2(a_3 b_2' - b_2' a_3) + u_3(b_3 b_2' - a_3 c_2') + u_4 b_2 c_3 \\ &= u_3(b_3 b_2' - a_3 c_2') + u_4 b_2 c_3 = d_3 b_2' - d_2' a_3. \end{aligned}$$

Comparing the bottom equation with the expressions for b_2', c_2', and d_2', you'll notice that if we define

$$\begin{aligned} b_3' &= b_3 b_2' - c_2' a_3 \\ c_3' &= c_3 b_2' \\ d_3' &= d_3 b_2' - d_2' a_3, \end{aligned}$$

then

$$\begin{aligned} (\text{equation } 3)' &= u_3(b_3 b_2' - a_3 c_2') + u_4 b_2 c_3 = d_3 b_2' - d_2' a_3 \\ &= u_3 b_3' + u_4 c_3' = d_3'. \end{aligned}$$

Following this pattern, we can generate the upper-triangle matrix for $j > 2$ (the top row of our matrix, $j = 1$, has no lower-triangle element) by first defining

$$\begin{aligned} b_1' &= b_1 \\ c_1' &= c_1 \\ d_1' &= d_1, \end{aligned}$$

and then, starting with $j = 2$, making the replacements

$$\begin{aligned} a_j &\rightarrow a_j' = 0 \\ b_j &\rightarrow b_j' = b_j b_{j-1}' - c_{j-1}' a_j \\ c_j &\rightarrow c_j' = c_j b_{j-1}' \\ d_j &\rightarrow d_j' = d_j b_{j-1}' - d_{j-1}' a_j. \end{aligned}$$

With these simple transformations, the matrix can be immediately written in upper-triangular form, where all the a_j's are zero. The matrix can be simplified further when all b'_j's are nonzero. In that case, we may divide both sides of the equation by b'_j and define double-primed quantities:

$$\begin{aligned} a''_j &= a'_j/b'_j = 0/b'_j = 0, \\ b''_j &= b'_j/b'_j = 1, \end{aligned}$$

$$\begin{aligned} c''_j &= \frac{c'_j}{b'_j} = \frac{c_j b'_{j-1}}{b'_j} \\ &= \frac{c_j b'_{j-1}}{b_j b'_{j-1} - c'_{j-1} a_j} \\ &= \frac{c_j}{b_j - a_j c'_{j-1}/b'_{j-1}} \\ &= \frac{c_j}{b_j - c''_{j-1} a_j}, \end{aligned}$$

$$\begin{aligned} d''_j &= \frac{d'_j}{b'_j} = \frac{d_j b'_{j-1} - d'_{j-1} a_j}{b'_j} \\ &= \frac{d_j b'_{j-1}}{b'_j} - \frac{d'_{j-1} a_j}{b'_j} \\ &= \frac{d_j}{b_j - c''_{j-1} a_j} - \frac{d'_{j-1} a_j}{b_j b'_{j-1} - c'_{j-1} a_j} \\ &= \frac{d_j}{b_j - c''_{j-1} a_j} - \frac{a_j d'_{j-1}/b'_{j-1}}{b_j - a_j c'_{j-1}/b'_{j-1}} \\ &= \frac{d_j}{b_j - c''_{j-1} a_j} - \frac{a_j d''_{j-1}}{b_j - a_j c''_{j-1}} \\ &= \frac{d_j - a_j d''_{j-1}}{b_j - c''_{j-1} a_j}. \end{aligned}$$

To summarize:

Thomas Algorithm: Obtaining The Upper-Triangle Matrix

Provided $b_j \neq 0$, the elements of the upper-triangle matrix are given by

$$\begin{aligned} a''_j &= 0 & b''_j &= 1 \\ c''_1 &= \frac{c_1}{b_1} & c''_j &= \frac{c_j}{b_j - c''_{j-1} a_j} \\ d''_1 &= \frac{d_1}{b_1} & d''_j &= \frac{d_j - a_j d''_{j-1}}{b_j - c''_{j-1} a_j}. \end{aligned}$$

What about the back-substitution step to compute the u_j's? Consider the 7×7 tridiagonal matrix equation (i.e., $J = 8$):

$$\boldsymbol{Au} = \begin{pmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 & 0 & 0 \\ 0 & 0 & 0 & a_5 & b_5 & c_5 & 0 \\ 0 & 0 & 0 & 0 & a_6 & b_6 & c_6 \\ 0 & 0 & 0 & 0 & 0 & a_7 & b_7 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{pmatrix}$$

$$\Longrightarrow \begin{pmatrix} 1 & c''_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & c''_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & c''_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & c''_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & c''_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & c''_6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{pmatrix} = \begin{pmatrix} d''_1 \\ d''_2 \\ d''_3 \\ d''_4 \\ d''_5 \\ d''_6 \\ d''_7 \end{pmatrix}$$

So our strategy when using the Thomas Algorithm is to start from the upper row of the matrix equation, computing c''_1 and d''_1, and then proceeding through c''_2, ..., c''_{J-1} and d''_2, ..., d''_{J-1}. Once we have d''_{J-1} we have the trivial equation

for u_{J-1}:

$$u_{J-1} = d''_{J-1}$$

With u_{J-1} known, we simply go to the next row up:

$$u_{J-2} + c''_{J-2}u_{J-1} = d''_{J-2}.$$

Notice we only have one unknown, u_{J-2}. We solve for it and continue this "back-substitution" until we have, finally, all u_j's.

In summary:

Thomas Algorithm: Solution via Back-Substitution

Once we have found the upper-triangle matrix, we apply back-substitution to find the solution vector u_j. That is to say, we know that $u_{J-1} = d''_{J-1}$, so next we solve for u_{J-2} using known quantities c''_{J-2}, u_{J-1}, and d''_{J-2}.

$$u_{J-2} + c''_{J-2}u_{J-1} = d''_{J-2},$$

Then we solve for

$$u_{J-3} + c''_{J-3}u_{J-2} = d''_{J-3},$$

using known quantities c''_{J-3}, u_{J-2}, and d''_{J-3}, and so forth until we have solved for all the components of the u_j vector.

14.1.1.1 Worked Example of Thomas Algorithm

Consider the matrix equation:

$$\begin{pmatrix} -3 & 1 & 0 & 0 & 0 \\ 1 & -3 & 1 & 0 & 0 \\ 0 & 1 & -3 & 1 & 0 \\ 0 & 0 & 1 & -3 & 1 \\ 0 & 0 & 0 & 1 & -3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$$

First we compute c''_1 and d''_1:

$$\begin{aligned} c''_1 &= \frac{c_1}{b_1} = \frac{1}{-3} \\ d''_1 &= \frac{d_1}{b_1} = \frac{1}{-3} \end{aligned}$$

Then we compute c''_j and d''_j for $j > 1$:

$$\begin{aligned}
c''_2 &= \frac{c_2}{b_2 - c''_1 a_2} \\
&= \frac{1}{-3 - \frac{1}{3} \times 1} \\
&= \frac{1}{-3 - \frac{1}{3} \times 1} \\
&= -0.375 \\
c''_3 &= \frac{c_3}{b_3 - c''_2 a_3} \\
&= \frac{1}{-3 - (-0.375) \times 1} \\
&= -0.380952 \\
c''_4 &= \frac{c_4}{b_4 - c''_3 a_4} \\
&= \frac{1}{-3 - (-0.380952) \times 1} \\
&= -0.381818,
\end{aligned}$$

$$\begin{aligned}
d''_2 &= \frac{d_2 - a_2 d''_1}{b_2 - c''_1 a_2} \\
&= \frac{2 - 1(-1/3)}{-3 - (-1/3) \times 1} \\
&= -0.875 \\
d''_3 &= \frac{d_3 - a_3 d''_2}{b_3 - c''_2 a_3} \\
&= \frac{3 - (-0.875)}{-3 - (-0.375) \times 1} \\
&= -1.47619 \\
d''_4 &= \frac{d_4 - a_4 d''_3}{b_4 - c''_3 a_4} \\
&= \frac{4 - (-1.47619)}{-3 - (-0.380952) \times 1} \\
&= -2.09091 \\
d''_5 &= \frac{d_5 - a_5 d''_4}{b_5 - c''_4 a_5} \\
&= \frac{5 - (-2.09091)}{-3 - (-0.381818) \times 1} \\
&= -2.70833
\end{aligned}$$

Thus we have the new matrix equation:

$$\begin{pmatrix} 1 & -0.333333 & 0 & 0 & 0 \\ 0 & 1 & -0.375 & 0 & 0 \\ 0 & 0 & 1 & -0.380952 & 0 \\ 0 & 0 & 0 & 1 & -0.381818 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} -0.333333 \\ -0.875 \\ -1.47619 \\ -2.09091 \\ -2.70833 \end{pmatrix}$$

Thus we get, through back substitution:

$$\begin{aligned}
x_5 &= -2.70833 \\
x_4 + -2.70833 \times -0.381818 &= -2.09091 \implies x_4 = -3.125 \\
x_3 + -0.380952 \times -3.125 &= -1.47619 \implies x_3 = -2.66667 \\
x_2 + -0.375 \times -2.66667 &= -0.875 \implies x_2 = -1.875 \\
x_1 + -0.333333 \times -1.875 &= -0.333333 \implies x_1 = -0.958333
\end{aligned}$$

MATH 522, Spring 2019 Notes

Chapter 15: Computational Cost and Computational Complexity (Big-$\mathcal{O}$ Notation)

15.1 Computational Cost: Floating Point Operations (FLOPs)

Often, when solving mathematical problems on the computer for research problems, we must be careful not to overwhelm the computational resources available to us. The exact amount of time needed to solve a problem will depend on how many **FL**oating point **OP**erations (FLOPs) are required, and the number of FLOPs per second (FLOPS) our computer can evaluate. Being able to count FLOPs in an algorithm enables us to reliably estimate the computational cost, and provides us insights into how we might make our algorithms more computationally efficient.

A single floating point operation (FLOP) can be defined as a variable assignment ($=$), multiplication ($\times$), addition ($+$), or subtraction ($-$). On modern CPUs, A single division ($\div$) typically requires between 3–5 FLOPs (`http://tinyurl.com/estimateFLOPS`).

Applying this definition, for floating point numbers a, b, c, d, and F, the expression

$$F = a * b - c * d$$

requires a total of 4 FLOPs (one assignment, multiplication on $a * b$, multiplication on $c * d$, and subtraction of the result).

As another example, for floating point numbers $a, b, c, ..., h$ and G, the expression

$$G = -(a * b - c * d)/(e * f + g * h)$$

requires a total of 8 floating point operations to evaluate all but the division (one variable assignment, the overall minus sign (`-1*`) is a multiplication, 3 in the numerator, and 3 in the denominator). The division could add between 3–5 FLOPs, yielding a total cost of between 11–13 FLOPs.

15.2 Common Subexpression Elimination (CSE)

Consider the following expression:

```
x = b * sin(2*a) + c / sin(2*a).
```

Computation of this expression requires one assignment, three multiplications, one division, two `sin()` function calls, and one addition. Multiplications, additions, and subtractions typically require one FLOP on a modern-day CPU, and divisions can require roughly 3–5 FLOPs each. Transcendental functions are far worse—each computation of a sine or cosine can require about 20 FLOPs on a modern-day CPU.

Our goal in writing efficient codes should be to minimize the number of FLOPs in our mathematical expressions. One way of doing this is to implement **common subexpression elimination**, or CSE for short.

Automatic CSE algorithms do exist. Generally they search for common patterns within expressions and declare them as new variables, so they need not be computed again.

Consider the following simplification of the above expression:

```
tmp = sin(2*a)
x   = b * tmp + c / tmp
```

For the cost of only one additional assignment, we potentially saved about 21 FLOPs by not computing `sin(2*a)` twice. *Special note: Compiled languages like C++ or FORTRAN sometimes implement decent automatic CSE algorithms, so optimizing code by hand like this might not yield the expected performance gains, as the compiler may attempt its own CSE.*

Words of caution

Since division on most modern-day computers is significantly more expensive than addition, multiplication, subtraction, and variable assignment, when rearranging expressions to make your computer program run faster, be sure that minimizing the number of divisions is a high priority!

Modern CPUs possess multiple memory caches, typically with lower-numbered memory caches being closer to the CPU (e.g., memory in the L1 cache on Intel and AMD CPUs can be accessed *far* more quickly than L2 or L3 caches; i.e., accesses from L2 or L3 cache can waste up to hundreds of CPU cycles). When it comes to CPU cache, accessing memory in higher-numbered caches takes far longer than in low-numbered caches. Sometimes this means that declaring large numbers of temporary variables in a CSE optimization might overload the L1 cache, causing a significant slowdown. Therefore one must be generally careful to consider CPU cache sizes. Note also that RAM (outside CPU cache) is even more slowly accessed, and hard disk reading/writing can be slower than RAM by *orders of magnitude*.

Finally, most compiled languages like C, FORTRAN, or C++ will make use of (with suitable compiler options) Single Instruction-Multiple Data CPU instructions (e.g., SSE2 or AVX instruction sets) when compiling mathematical expressions. These instructions enable multiple additions, subtractions, multiplications, or fused-multiply-adds to be performed *each CPU clock cycle*! As a result, you may find when using these languages that your code runs far faster than you might otherwise expect. Writing codes with this—and the behavior of CPU caches—in mind can be very powerful ammunition to improving the performance of your codes *by orders of magnitude*!

15.3 Computational Complexity (Big-$\mathcal{O}$ Notation)

Some algorithms are more efficient than others, and big-$\mathcal{O}$ notation can be used as a measure of an algorithm's efficiency.

Imagine we have a cluster of N stars. Suppose we wished to calculate the gravitational force $\boldsymbol{F}_i$ on a given star i. The force would be equal to

$$\boldsymbol{F}_i = \sum_{i \neq j}^{N} \boldsymbol{F}_{i,j},$$

where $\boldsymbol{F}_{i,j}$ is simply the force on star i from some other star j. Clearly to calculate the gravitational pull on star i, we need to add a total of $N-1$ forces. In order to model the motions of all the stars in this star cluster, we must compute $\boldsymbol{F}_i$ for all N stars in the cluster. As you know, addition is an operation allowed by floating point numbers, so to add up all the forces on this star cluster will require $N(N-1) = N^2 - N$ floating point operations. Written in pseudocode, this translates to a nested loop:

```
do i=1,N
  F[i]=0
  do j=1,N
    if(i not equal to j) F[i] = F[i] + F(i,j)
  end do
end do
```

When you see nested loops, be careful: the number of nested loops adds directly to the *exponent* of the number of operations. And the largest exponent is what typically goes into the Big-$\mathcal{O}$ notation. Basically, Big-$\mathcal{O}$ notation is used to denote the *computational complexity* of a given algorithm, and we say that the algorithm just described is an $\mathcal{O}(N^2)$ operation.

Suppose we have chosen an algorithm that is $\mathcal{O}(N^3)$ due to a nested loop. If we need to run the $\mathcal{O}(N^3)$ loop in this algorithm three times as well as a $\mathcal{O}(N^2)$ loop once, many numerical analysis texts will write the computational complexity as $\mathcal{O}(3N^3 + N^2)$. Unless we are comparing the complexity of two algorithms with the same exponent

on N (e.g., to find the most efficient algorithm) or are interested in the small-N limit, the constant coefficient on N^3 and the N^2 term will rarely be useful.

Thus we typically choose to write an algorithm that has computational complexity $\mathcal{O}(3N^3 + N^2)$ as $\mathcal{O}(N^3)$ because Big-$\mathcal{O}$ notation is typically used to estimate *how long* it will take a given algorithm to run on our computer if N (assumed $\gg 1$) is varied.

In particular, the time T required for our computer to evaluate an $\mathcal{O}(N^3)$ algorithm in the large-N limit will be given by

$$T \propto N^3,$$

which can be written

$$T = aN^3,$$

where a is a proportionality constant. a will not only depend on the fact that there are three loops, but also the speed of our computer! Different computers have different intrinsic speeds, so assessing the computational complexity of our algorithm will provide us an estimate of computational cost, modulo a proportionality constant that accounts for the speed of our computer.

Question:

If an $\mathcal{O}(N^2)$ algorithm requires 10 minutes to complete, how long will it take the algorithm to complete if N is doubled? Assume N is large.

Answer:

The cost in time of an $\mathcal{O}(N^2)$ algorithm will grow quadratically (as N^2) with N. We are given than $a(N_0)^2 = 10$ minutes for some value $N = N_0$, so the proportionality constant is 10 minutes / $(N_0)^2$. Therefore if we choose $N_1 = 2N_0$, then the cost in time is $a(N_1)^2 = a(2N_0)^2 = 4 \times a(N_0)^2 = 4 \times 10$ minutes $= 40$ minutes.

Definition: An algorithm with computational complexity $\mathcal{O}(1)$ will complete in a time *independent* of some size factor N. For example, an algorithm that evaluates the single arithmetic expression

```
x = N * 2 + 4
```

has computational complexity $\mathcal{O}(1)$, despite the fact that `x` depends on `N`, since as `N` increases, the computational cost remains the same.

15.4 Example: More Efficient Algorithms May Exist!

Suppose we play the game "higher or lower". In this game between two people – Person A and Person B, Person A picks a secret number (an integer) between 0 and N, and Person B attempts to guess the secret number. If the guess is correct, the game ends. After each incorrect guess, Person A must say either "higher" or "lower" if the number Person B guessed is less than or greater than the secret number, respectively. The goal of Person B is to end the game as quickly as possible.

One algorithm Person B might choose would be to start at 0 and guess 1, then 2, then 3, etc., while Person A keeps saying "higher", until Person B reaches the secret number. This algorithm for guessing the correct number would be $\mathcal{O}(N)$, since Person B would need to make on average $N/2$ guesses before reaching the correct number.

Consider the following alternative strategy for Person B. As the first guess, Person B chooses $N/2$. Provided this is not the correct number and Person A said

- "higher", then Person B would pick the closest number to $3N/4$, or
- "lower", then Person B would pick the closest number to $1N/4$.

By continuing this pattern of bisecting the remaining interval on each guess, Person B would rule out fully *half* of the remaining possible secret numbers at each guess. You will find that (*exercise to student*) if $N = 4$, Person B would need at most 3 guesses. If $N = 8$, Person B would need at most 4 guesses. If $N = 64$, Person B would need

at most 7 guesses. Continuing the pattern, Person B would need at most $1 + \log_2 N$ guesses for any N. Thus this "bisection" algorithm for finding the secret number has computational complexity of $\mathcal{O}(\log_2 N)$.

Now consider a very large N... say $N = 2^{100}$. Then the cost of the first algorithm would be of order $2^{100} \approx 1.3 \times 10^{30}$ guesses, while the second algorithm would require only 101 guesses at most... a reduction in the number of required guesses of about *28 orders of magnitude*.

Therefore as a general tip, it is very important to seek out algorithms with lower computational complexity in the large-N limit if they exist — following this tip might save you years of waiting in completing research projects!

15.5 Big-$\mathcal{O}$ Notation, and Computational Cost Exercises

Exercises:

First write the computational complexity of each of the following algorithms, and then evaluate the total computational cost in FLOPs if $N = 10$. Use Big-$\mathcal{O}$ notation for computational complexity and assume variable assignments, multiplications, additions, subtractions, and divisions all require a single floating point operation.

1.
```
x = 2*N*N*N - 4*N*N + 5*N + 3
```
2.
```
do i=1,N
  F[i]=0
end do
do i=1,N
  do j=1,N
    F[i] = F[i] + j
  end do
end do
```
3.
```
do i=1,N*N
  F[i]=0
  do j=1,N*N
    F[i] = F[i] + i*0.3 + j*0.1
  end do
end do
```
4.
```
do i=1,N*N*N
  F[i]=i+2
end do
do i=1,N
  do j=1,N*N
    F[i] = F[i] + j
  end do
end do
```

Solutions:

1. $\mathcal{O}(1)$. There are no loops, and only a single expression to evaluate. The expression itself, $x = 2 * N * N - 4 * N * N + 5 * N + 3$, requires 10 FLOPs to compute: 6 multiplications, one subtraction, two additions, and one assignment.
2. The second loop is an $\mathcal{O}(N^2)$ operation. FLOPs for $N = 10$: each innermost loop requires 2 FLOPs per iteration, so we get 2×10^2 FLOPs for the innermost loop. Add this to the first loop of 10 FLOPs, and we get 210 FLOPs.
3. $\mathcal{O}(N^4)$, because there are two nested N^2 loops. FLOPs for $N = 10$: each innermost loop iteration requires 5 FLOPs, so a total of $5N^4 = 5 \times 10^4$ FLOPs are needed for the innermost loop. Add this to the N^2 assignments needed to initialize $F[i] = 0$, and we get 50,100 FLOPs.
4. $\mathcal{O}(N^3)$; both the `do i=1,N*N*N` and `do i=1,N ; do j=1,N*N` loops have this computational complexity. FLOPs for $N = 10$: All loops require 2 FLOPs per iteration, and there are 10^3 evaluations per loop. Therefore we find the algorithm requires 4,000 FLOPs.

MATH 522, Spring 2019 Notes

Chapter 16: Gaussian Elimination on the Computer and the LU Decomposition

In numerical analysis, we are often required to solve the matrix equation

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},$$

where $\boldsymbol{A}$ is a square matrix, $\boldsymbol{b}$ is a known vector, and $\boldsymbol{x}$ is the unknown vector representing our solution. How do we solve for $\boldsymbol{x}$?

16.1 Gaussian Elimination

Gaussian Elimination is the strategy we are first taught to solve sets of linear equations. Consider the following set of equations:

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

We first rewrite these in the form $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ as follows:

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} 5 & -2 & 1 \\ 4 & -4 & 3 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \\ -1 \end{bmatrix} = \boldsymbol{b}$$

When performing Gaussian elimination we first write the **augmented matrix** corresponding to this linear system of equations:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 4 & -4 & 3 & 1 \\ 1 & 1 & -1 & -1 \end{array}\right].$$

Recall the goal now is to transform this matrix into echelon or **upper-triangle** form, where all elements below the diagonal are zero. The trick is to notice that the linear set of equations remain unchanged if

1. any row is multiplied by any nonzero, non-infinite constant,
2. rows are interchanged, or
3. any two rows can be added together, with the result replacing one of the rows.

By careful application of these rules, we can "zero-out" all elements below the diagonal, accomplishing our goal of rewriting the matrix in upper triangle form.

We know how to do this process by hand, but how can we teach a computer to do it? This is a question at the core of numerical analysis. Typically when solving the linear system of equations by hand, we try to avoid fractions in favor of integers.

For example, suppose we wish to zero-out the first column of the second row. We could start by multiplying the top row by -4 and then adding it to 5 times the second row, then replacing row 2 with the result.

In shorthand, $-4R_1 + 5R_2 \to R_2$:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ -20+20 & 8-20 & -4+15 & 8+5 \\ 1 & 1 & -1 & -1 \end{array}\right] = \left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 0 & -12 & 11 & 13 \\ 1 & 1 & -1 & -1 \end{array}\right]$$

Our computer is capable of performing floating-point arithmetic to many significant digits, and teaching our computer to perform only integer arithmetic in these cases would greatly complicate our algorithm for Gaussian elimination. In addition, the fact that we are dealing with an *integer* matrix is very artificial; normally the matrix elements will be populated by finite precision, floating point numbers.

So to zero-out the first column of the second row in a more automatic way, let's simply multiply the first column by $-\frac{4}{5}$, add the result to the second row, and then replace the second row.

In shorthand notation we have, $-\frac{4}{5}R_1 + R_2 \to R_2$:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 5\times-\frac{4}{5}+4 & -2\times-\frac{4}{5}-4 & 1\times-\frac{4}{5}+3 & -2\times-\frac{4}{5}+1 \\ 1 & 1 & -1 & -1 \end{array}\right] = \left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 0 & -\frac{12}{5} & \frac{11}{5} & \frac{13}{5} \\ 1 & 1 & -1 & -1 \end{array}\right]$$

Applying this strategy, we then multiply the first row by $-\frac{1}{5}$ and add it to the third row, replacing the third row. This will zero-out the lower-left component of the matrix. In shorthand notation: $-\frac{1}{5}R_1 + R_3 \to R_3$.

Analyzing our approach for zeroing-out the element in row i of the first column, we see the following pattern:

$$-c_i R_1 + R_i \to R_i, \quad \text{where} \quad c_i = \frac{\text{row } i\text{, column 1}}{\text{row 1, column 1}}. \tag{16.1}$$

Let's standardize and simplify the notation a bit:

Notation:

For a matrix $\boldsymbol{A}$, $a_{i,j}$ will henceforth refer to the element at the ith row and the jth column.

With this notation, we can now write the steps necessary for zeroing-out all elements below the diagonal in the first column (i.e., zeroing out all elements $a_{i,1}$ for $i > 1$): Multiply row 1 (R_1) by $-a_{i,1}/a_{1,1}$, add the result to row i (R_i), and replace row i (R_i) with the result. In short-hand notation:

$$-\frac{a_{i,1}}{a_{1,1}}R_1 + R_i \to R_i.$$

Note that each row in the augmented matrix has a total of $N+1$ columns. This means that zeroing out each element possesses $\mathcal{O}(N)$ complexity. The corresponding pseudo-code, using the 2-dimensional array `a[i][j]`$=a_{i,j}$, is as follows (**Warning: There is a serious bug in this algorithm. See Exercise below**):

```
do j=1,N+1
   a[i][j] = -a[i][1]/a[1][1] * a[1][j]  + a[i][j]
end do
```

Exercise:

Find the serious bug in the above "zeroing-out all rows below the diagonal in the first column" algorithm. *Hint: Add a* `print i,j,a[i][j]` *before the* `end do` *and carefully analyze whether the output is as intended.*

Solution:

After the first iteration of the inner loop, `a[i][1]` has been set to zero. As a result, for `j>1`, `a[i][1]/a[1][1]=0`, meaning that the rescaling information c_i (Eq. 16.1) is lost after the first column of the row. To fix this, we store c_i before launching into the `j` loop:

```
c_i = a[i][1]/a[1][1]
do j=1,N+1
   a[i][j] = -c_i * a[1][j]  + a[i][j]
end do
```

This might be called a *data-dependency* bug, in that the buggy version modifies data that terms `j>1` assume are unchanged.

Based on the Exercise above, the bug-fixed algorithm for zeroing out all elements in the first column below the diagonal is as follows:

```
c_i = a[i][1]/a[1][1]
do j=1,N+1
   a[i][j] = -c_i * a[1][j]  + a[i][j]
end do
```

To fill in *all* rows $i > 1$ in this way requires that we repeat the above bug-fixed loop for all other rows:

```
do i=2,N
   c_i = a[i][1]/a[1][1]
   do j=1,N+1
      a[i][j] = -c_i * a[1][j]  + a[i][j]
   end do
end do
```

This algorithm will zero-out all elements in the *first* column below the matrix diagonal. However, we need to zero-out elements below the diagonal in all columns to get the matrix in echelon (upper-triangle) form!

Once we have zeroed-out all elements below the diagonal in the first column, the pattern for the second column follows immediately:

```
do i=3,N
   c_i = a[i][2]/a[2][2]
   do j=2,N+1
      a[i][j] = -c_i * a[2][j]  + a[i][j]
   end do
end do
```

Noticing the pattern, we can write the entire algorithm for decomposing the matrix into an upper-triangle (echelon form) matrix using Gaussian elimination as follows:

```
do k=1,N-1
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=k,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
```

This algorithm will output the following echelon or upper-triangle form of $\boldsymbol{A}|\boldsymbol{b}$, to double precision:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 0 & -12/5 & 11/5 & 13/5 \\ 0 & 0 & 1/12 & 11/12 \end{array}\right] \tag{16.2}$$

With the matrix in echelon form, x_1, x_2, and x_3 can be solved by **back substitution**. Let's work out the algorithm. The last row implies that

$$1/12x_3 = 11/12 \implies x_3 = 11.$$

In general the last row can be written in the form

$$\begin{aligned} & a_{N,N}x_N = a_{N,N+1} \\ \implies \qquad & x_N = a_{N,N+1}/a_{N,N}. \end{aligned}$$

The second row then gives us

$$(-12/5)x_2 + (11/5)x_3(=11) = 13/5 \implies -12x_2 = 13 - 121 = -108 \implies x_2 = 108/12 = 54/6 = 9.$$

The general form of the second-to-last row is

$$\begin{aligned} & a_{N-1,N-1}x_{N-1} + a_{N-1,N}x_N = a_{N-1,N+1} \\ \implies \qquad & x_{N-1} = \left(a_{N-1,N+1} - a_{N-1,N}x_N\right)/a_{N-1,N-1}, \end{aligned}$$

where all the terms on the right-hand side are known.

Then we apply back substitution on the first row:

$$5x_1 - 2x_2(=9) + x_3(=11) = -2 \implies 5x_1 = 2*9 - 11 - 2 = 18 - 13 = 5 \implies x_1 = 1.$$

The general form of the third-to-last row is

$$\begin{aligned} & a_{N-2,N-2}x_{N-2} + a_{N-2,N-1}x_{N-1} + a_{N-2,N}x_N = a_{N-2,N+1} \\ \implies \qquad & x_{N-2} = \left(a_{N-2,N+1} - a_{N-2,N-1}x_{N-1} - a_{N-2,N}x_N\right)/a_{N-2,N-2}. \end{aligned}$$

Looking at the general forms of these expressions, we see that in the process of back-substitution, x_i will depend on $x_N, x_{N-1}, ..., x_{i-1}$, as well as the matrix elements $a_{i,i}, a_{i,i+1}, ..., a_{i,N}, a_{i,N+1}$.

Upon careful examination, we find that the expression for an arbitrary x_i in the process of back substitution will be given by

$$x_i = \frac{a_{i,N+1} - \sum_{j=i+1}^{N} a_{i,j}x_j}{a_{i,i}},$$

where all x_js on the right-hand side of this equation have already been computed. Note that for $i = N$, the sum becomes a sum from $N+1$ to N—no elements of the sum are computed, and the expression is consistent with the x_N expression above.

In pseudo-code, the back substitution algorithm takes the form

```
do i=N,1,-1 # Loop from N to 1 inclusive in increments of -1 (opposite normal loop order)
   x[i] = a[i][N+1]/a[i][i]
   do j=i+1,N
      x[i] = x[i] - a[i][j]*x[j]/a[i][i]
   end do
end do
```

Here is a summary of the Gaussian Elimination algorithm we have devised:

Summary: Simple Gaussian Elimination Algorithm

Warning: This algorithm is not very robust. See next section for details!

```
do k=1,N-1
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=k,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
do i=N,1,-1 # Loop from N to 1 inclusive in increments of -1 (opposite normal loop order)
   x[i] = a[i][N+1]/a[i][i]
   do j=i+1,N
      x[i] = x[i] - a[i][j]*x[j]/a[i][i]
   end do
end do
```

Exercises:

1. What is the computational complexity of the Gaussian elimination algorithm (including back substitution)?
2. What is the computational complexity of the back substitution algorithm?
3. Write the pseudocode for computing $\boldsymbol{AB}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are $N \times N$ matrices. Store the result in matrix $\boldsymbol{C} = \boldsymbol{AB}$. What is the computational complexity? If our computer can multiply two 100×100 matrices in 1 second, how long can we expect it will need to multiply two $1,000 \times 1,000$ matrices? (Ignore cache effects.)

Solutions:

1. $\mathcal{O}(N^3)$
2. $\mathcal{O}(N^2)$—far less expensive than generating the upper triangle matrix!
3. The computational complexity of multiplication of two $N \times N$ matrices is $\mathcal{O}(N^3)$. If our computer can multiply two 100^2 matrices in 1 second, it will therefore need $(1,000/100)^3$ seconds to multiply two $1,000^2$ matrices, or 1,000 seconds.

```
do i=1,N
   do j=1,N
      sum = 0
      do k=1,N
         sum = sum + a[i][k]*b[k][j]
      end do
      c[i][j] = sum
   end do
end do
```

16.1.1 Toward a More Robust Gaussian Elimination Algorithm

In this section, we will explore cases in which the Gaussian elimination algorithm we have devised will fail, and work to make our algorithm more robust.

16.1.1.1 Zeroes Appearing along the Diagonal of A during Gaussian Elimination

Take a look at the line of code from our Gaussian elimination algorithm:

```
c_i = a[i][k]/a[k][k]
```

Notice that the algorithm will fail if `a[k][k]=0`. That is to say, if a zero appears along a diagonal of our matrix $\boldsymbol{A}$, then the algorithm will divide by zero, yielding `NaN`s (Not A Number) for our solution vector $\boldsymbol{x}$.

Consider the following set of linear equations:

$$\boldsymbol{A}|\boldsymbol{b} = \left[\begin{array}{ccc|c} 0 & -12 & 11 & 13 \\ 5 & -2 & 1 & -2 \\ 0 & 0 & 1 & 11 \end{array}\right]$$

Notice that this is equivalent to

$$\begin{aligned} -12x_2 + 11x_3 &= 13 \\ 5x_1 - 2x_2 + x_3 &= -2 \\ x_3 &= 11, \end{aligned}$$

which is precisely the same system of equations we found in the previous section (Eq. 16.2), just before back-substitution – all we did here was interchange the top two equations. We know the solution is $x_1 = 1$, $x_2 = 9$, and $x_3 = 11$, but if we were to plug this system of equations into our Gaussian elimination algorithm, it would fail because the upper-left diagonal component is zero, resulting in a division by zero in `c_i` when `k=1` and `i=1`.

If we flip the top two rows again, we get:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 0 & -12 & 11 & 13 \\ 0 & 0 & 1 & 11 \end{array}\right],$$

which does not contain a zero along the diagonal, and our standard Gaussian elimination algorithm will effectively do nothing until back-substitution, since the matrix is already in upper-triangle (i.e., echelon) form.

Definition: Flipping rows in this way is called **zero-avoidance pivoting**.

Let us now modify our Gaussian elimination algorithm to incorporate this simple zero-avoidance pivoting to avoid failures:

```
do k=1,N-1
   if(a[k][k] is 0)
      (flip row k with the next row down that does not have a zero in column k)
   end if
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=k,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
```

16.1.1.2 Addressing Roundoff Error Issues with Partial Pivoting

Even after installing the above zero-avoiding pivot algorithm, our Gaussian elimination algorithm can still fail. Consider the following system of equations:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} = \left[\begin{array}{cc} 10^{-20} & 1 \\ 1 & 1 \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \end{array}\right] = \left[\begin{array}{c} 1 \\ 0 \end{array}\right].$$

The exact solution to this equation is

$$\begin{aligned} x_1 &= -\frac{1}{1-10^{-20}} \\ x_2 &= \frac{1}{1-10^{-20}}. \end{aligned}$$

So in double precision, we might expect to get the answers ± 1, which are consistent with the exact results to about 19 significant digits. But this is not the case when using our Gaussian elimination algorithm!

Our Gaussian elimination algorithm will multiply the first row by -10^{20}, add the result to the bottom row, and replace the bottom row with this result. In shorthand notation: $-10^{20}R_1 + R_2 \to R_2$:

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20}+1 & -10^{20} \end{array}\right]$$

Notice the loss of significance in the bottom row: $-10^{20} + 1 = -10^{20}$ in double precision. So to double precision, the augmented matrix in echelon (upper-triangle) form is written

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20} & -10^{20} \end{array}\right]$$

Next the algorithm will solve for $\boldsymbol{x}$ using back substitution:

$$\begin{aligned} x_2 = &\quad a_{23}/a_{22} = (-10^{20})/(-10^{20}) &= 1 \\ x_1 = &\quad a_{13}/a_{11} - x_2(a_{12}/a_{11}) = 1/10^{-20} - 1(1/10^{-20}) = 10^{20} - 10^{20} &= 0 \end{aligned}$$

We conclude that back-substitution in double precision yields $x_2 = 1$, which is consistent with the exact result to about 19 significant digits. However, it also finds $x_1 = 0$, which is not even close to the correct answer! What happened here?

Recall our discussions of machine epsilon. It turns out that any $|x_1| \lesssim 10^4$ will satisfy the equation $10^{-20}x_1 + 1 = 1$ in double precision, since machine epsilon is $\epsilon_m \sim 10^{-16}$ in double precision. Notice that the correct value $x_1 = -1$ is in this inequality, but because the $\boldsymbol{U}$ matrix involves an equation that multiplies x_1 by a number 20 orders of magnitude smaller in magnitude than x_2, we have observed a catastrophic cancellation.

Definitions: To combat this loss of significance, we simply adjust our zero-pivoting algorithm to, instead of avoiding zeroes along the diagonal, to instead *maximize the magnitude of the elements along the diagonal*. This strategy is called **partial** or **row pivoting**. Usually partial pivoting is enough to minimize roundoff errors, but sometimes **complete pivoting** is necessary. Complete pivoting flips both columns and rows to maximize the diagonal elements. Note that unlike row pivoting, column pivoting changes the ordering of the components of the solution vector $\boldsymbol{x}$, which requires that one keep track of the order of the columns, usually by use of a **permutation matrix** $\boldsymbol{P}$. The permutation matrix is the identity matrix $\boldsymbol{I}$, but with the columns flipped consistent to the reordered solution vector $\boldsymbol{x}$. In this way, $\boldsymbol{Px}$ is the reordered solution vector.

The partial pivoting algorithm is as follows:

- As column j of the upper triangle (echelon) form of the matrix is computed, first find the largest magnitude element in this column at or below the main diagonal.
- Interchange that element's row with row j.

in pseudocode, the Gaussian elimination algorithm with partial pivoting is given by:

```
do k=1,N
   row_of_maximum_element = k
   do i=k+1,N
      if(|a[i][k]| > |a[row_of_maximum_element][k]|) then
         maximum_element_row = i
      end if
   end do
   (flip k and row_of_maximum_element rows)
```

```
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=1,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
```

Note that partial pivoting not only acts to minimize roundoff error, but also replaces zero elements along the diagonal. Thus Gaussian elimination with partial pivoting is *superior* to the original zero-avoiding pivot algorithm described above.

Let's apply partial pivoting strategy to our simple example:

$$\boldsymbol{A}|\boldsymbol{b} = \left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 1 & 0 \end{array}\right]$$

Partial pivoting first scans the first column for the row with the largest magnitude element in this column. Once it finds this element, it flips the rows. Clearly $|1| > |10^{-20}|$, so we must flip rows:

$$\left[\begin{array}{cc|c} 1 & 1 & 0 \\ 10^{-20} & 1 & 1 \end{array}\right]$$

Gaussian elimination then proceeds using our algorithm via $-10^{-20}R_1 + R_2 \to R_2$:

$$\left[\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1-10^{-20} & 1-10^{-20} \end{array}\right] = \left[\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1 & 1 \end{array}\right]$$

Back substitution yields

$$\begin{aligned} x_2 = a_{23}/a_{22} = 1/1 &= 1 \\ x_1 = a_{13}/a_{11} - x_2(a_{12}/a_{11}) = 0/1 - 1(1/1) &= -1, \end{aligned}$$

which is consistent with the exact solution to about 19 significant digits!

16.1.1.3 When Partial Pivoting Fails: Rescale maximum element in each row to unit magnitude

Consider the same example as in the previous section, except with the top row multiplied by 10^{20}:

$$\boldsymbol{A}|\boldsymbol{b} = \left[\begin{array}{cc|c} 1 & 10^{20} & 10^{20} \\ 1 & 1 & 0 \end{array}\right].$$

In this case, our row/partial pivoting strategy will be ineffective, because the elements in the first column under the upper-left diagonal are equal. The result will again be $x_1 = 0$ and $x_2 = 1$. To fix this, we only need to multiply each row by the inverse of the largest magnitude element, prior to partial pivoting. This will have computational complexity of $\mathcal{O}(N^2)$—again a small fraction of the total cost of Gaussian elimination in the large-N limit. An alternative to this approach is to apply a **scaled pivoting** algorithm, whereby rows are flipped based on their largest magnitude. This will have computational complexity of $\mathcal{O}(N^2)$ as well because all elements of the matrix must be analyzed.

16.1.1.4 No Solution

Sometimes we might encounter a case in which there exists zero or infinitely many solutions to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. This will happen if and only if $\det(\boldsymbol{A}) = 0$. In this case, a solution $\boldsymbol{x}$ obtained through Gaussian elimination is guaranteed to fail. To prevent this from happening, one could compute the determinant of $\boldsymbol{A}$, but this is also a $\mathcal{O}(N^3)$ operation using standard approach (cofactor, or "Laplace" expansion)! So if our output contains `NaN`s, then and only then do we check for a zero determinant.

16.1.1.5 Summary: A Robust Gaussian Elimination Algorithm

Caution: There are examples of matrices for which complete pivoting is necessary to guarantee a solution.

Summary: Gaussian Elimination Algorithm with Partial Pivoting and $\det(\boldsymbol{A}) = 0$ Checking

```
do i=1,N
   # First normalize maximum magnitude in each row to 1
   max_magnitude_in_row_i = 0
   do j=1,N+1
      if(|a[i][j]| > max_magnitude_in_row_i)
         max_magnitude_in_row_i = |a[i][j]|
      end if
   end do
   do j=1,N+1
      a[i][j] = a[i][j] / max_magnitude_in_row_i
   end do
end do
do k=1,N
   # Partial pivoting algorithm:
   row_of_maximum_element = k
   do i=k+1,N
      if(|a[i][k]| > |a[row_of_maximum_element][k]|) then
         maximum_element_row = i
      end if
   end do
   (flip k and row_of_maximum_element rows)
   # Algorithm for computing upper-triangle form of A|b
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=1,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
# Back-substitution algorithm:
do i=N,1,-1 # Loop from N to 1 inclusive in increments of -1 (opposite normal loop order)
   x[i] = a[i][N+1]/a[i][i]
   do j=i+1,N
      x[i] = x[i] - a[i][j]*x[j]/a[i][i]
   end do
end do
# Check for NaNs in solution vector. If found, print appropriate error message.
do i=1,N
   if(x[i] == NaN) then
      detA = (determinant of matrix A)
      if(detA == 0) then
         print("Error: Matrix A is singular. No solution can be found.")
      else
         print("Error: Matrix A is non-singular, yet no solution found.")
         print(" You may have chosen a matrix that requires complete pivoting")
      end if
   end if
end do
```

16.2 The LU Decomposition

The LU decomposition first converts the square matrix $\boldsymbol{A}$ into an upper-triangular matrix $\boldsymbol{U}$, using Gaussian elimination. Simultaneously, as we apply the strategy of Gaussian elimination to compute $\boldsymbol{U}$, we build a lower-triangular matrix $\boldsymbol{L}$, such that

$$\boldsymbol{A} = \boldsymbol{LU}.$$

Definition: When a matrix $\boldsymbol{A}$ can be written as a product of simpler matrices, we call it a **decomposition** of $\boldsymbol{A}$, so since $\boldsymbol{A} = \boldsymbol{LU}$ we call this the approach the $\boldsymbol{LU}$ decomposition of $\boldsymbol{A}$.

With this definition for $\boldsymbol{A}$, the matrix equation becomes

$$\begin{aligned} \boldsymbol{b} &= \boldsymbol{Ax} \\ &= \boldsymbol{LUx} \\ &= \boldsymbol{L}(\boldsymbol{Ux}) \end{aligned}$$

Define

$$\boldsymbol{y} = \boldsymbol{Ux}.$$

Notice that both $\boldsymbol{y}$ and $\boldsymbol{x}$ are unknowns. The original equation can then be written in terms of $\boldsymbol{y}$ as follows:

$$\begin{aligned} \boldsymbol{b} &= \boldsymbol{Ax} \\ &= \boldsymbol{LUx} \\ &= \boldsymbol{L}(\boldsymbol{Ux}) \\ &= \boldsymbol{Ly} \end{aligned}$$

we can quickly solve for $\boldsymbol{y}$ through back substitution, since $\boldsymbol{L}$ is a lower-triangle matrix. Once we have solved for $\boldsymbol{y}$, we can then solve for $\boldsymbol{x}$ from the definition of $\boldsymbol{y}$:

$$\boldsymbol{y} = \boldsymbol{Ux}.$$

Again, this matrix equation is easy to solve via back-substitution, as $\boldsymbol{U}$ is an upper-triangle matrix.

The LU decomposition has a number of advantages over Gaussian elimination. The generation of the triangular matrices is an $\mathcal{O}(N^3)$ operation (which is the same as for Gaussian elimination), while the solution stage for the LU decomposition for a given $\boldsymbol{b}$ can be performed in $\mathcal{O}(N^2)$.

This makes the LU decomposition quite useful for many engineering/scientific applications in which $\boldsymbol{A}$ stays fixed, while we must solve for many different $\boldsymbol{b}$. Suppose we have r different $\boldsymbol{b}$'s. Then the total cost of LU is $\mathcal{O}(N^3) + \mathcal{O}(rN^2)$, where the cost for Gaussian elimination would be $\mathcal{O}(r(N^3 + N^2))$.

16.2.1 Worked Example of LU Decomposition

Understanding the advantages of decomposing the $N \times N$ matrix $\boldsymbol{A}$ into $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices, we present here by example the basic strategy for LU decomposition. We start with the following 3×3 matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 & -2 & 4 \\ 2 & -6 & 10 \\ 1 & 2 & -1 \end{bmatrix}$$

The idea is, $\boldsymbol{L}$ and $\boldsymbol{U}$ encode all the information necessary to perform a Gaussian elimination solution of $\boldsymbol{Ax} = \boldsymbol{b}$. This will be most obvious as we build the $\boldsymbol{L}$ matrix. We start by writing $\boldsymbol{L}$ as follows:

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 \\ - & 1 & 0 \\ - & - & 1 \end{bmatrix},$$

where the dashes indicate empty matrix elements that will be filled in with the Gaussian elimination information necessary to construct the upper-triangle matrix $\boldsymbol{U}$.

We now proceed with our Gaussian elimination on $\boldsymbol{A}$: $-2R_1 + R_2 \to R_2$

$$\boldsymbol{A_1} = \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 1 & 2 & -1 \end{bmatrix}$$

The idea is, we record this Gaussian elimination step by inserting the opposite of the multiplication factor we used to get rid of $a_{2,1}$, to the $l_{2,1}$ element of our matrix $\boldsymbol{L}$. The multiplication factor was -2, so we insert a $+2$ at element $l_{2,1}$ in the $\boldsymbol{L}$ matrix:

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ - & - & 1 \end{bmatrix},$$

We continue by zeroing the first column in the next row via $-R_1 + R_3 \to R_3$, and record the result in the lower-left corner of the $\boldsymbol{L}$ matrix:

$$\boldsymbol{A_2} = \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 0 & 4 & -5 \end{bmatrix}, \qquad \boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & - & 1 \end{bmatrix},$$

Next, to eliminate the 4 in the bottom row, we apply $2R_2 + R_3 \to R_3$. This completes not only our $\boldsymbol{U}$ matrix, but also our $\boldsymbol{L}$ matrix as well:

$$\boldsymbol{A_3} = \boldsymbol{U} = \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 0 & 0 & -1 \end{bmatrix}, \qquad \boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix}$$

We can confirm that $\boldsymbol{A} = \boldsymbol{LU}$:

$$\boldsymbol{LU} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 0 & 0 & -1 \end{bmatrix} = \boldsymbol{A} = \begin{bmatrix} 1 & -2 & 4 \\ 2 & -6 & 10 \\ 1 & 2 & -1 \end{bmatrix}$$

Let us now apply the LU decomposition technique to solve for $\boldsymbol{x}$. Recall that our first step is to define $\boldsymbol{y} = \boldsymbol{Ux}$, so that

$$\boldsymbol{Ax} = \boldsymbol{LUx} = \boldsymbol{L}(\boldsymbol{Ux}) = \boldsymbol{Ly} = \boldsymbol{b}.$$

Then we use back-substitution to quickly solve for $\boldsymbol{y}$. Let us suppose we have $\boldsymbol{b}^T = (1, 2, 4)$. Then

$$\boldsymbol{Ly} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}.$$

Then clearly, $y_1 = 1$, $2y_1 + y_2 = 2 \implies y_2 = 0$, and $y_1 - 2y_2 + y_3 = 1 + y_3 = 4 \implies y_3 = 3$.

Next we use the relation

$$\boldsymbol{Ux} = y = \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix}.$$

Thus we immediately get $x_3 = -3$, $-2x_2 + 2x_3 = -2x_2 - 6 = 0 \implies x_2 = -3$, and $x_1 - 2x_2 + 4x_3 = x_1 + 6 - 12 = 1 \implies x_1 = 7$.

Again, the power of the LU decomposition is that if we needed the solution $\boldsymbol{x}$ from $\boldsymbol{Ax} = \boldsymbol{b}$ now for a *different* $\boldsymbol{b}$, we could apply the LU decomposition of $\boldsymbol{A}$ we have already computed, along with some trivial back-substitution to get the new $\boldsymbol{x}$.

MATH 522, Spring 2019 Notes

Chapter 17: Iteratively Solving $N \times N$ Matrix Equations on the Computer

17.1 Jacobi's Method

The LU decomposition with partial pivoting is very powerful, as it is generally applicable to any $N \times N$ matrix equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. However, decomposing $\boldsymbol{A}$ into $\boldsymbol{L}$ and $\boldsymbol{U}$ does require $\mathcal{O}(N^3)$ operations. Might there be a better way?

Consider the fact that we're solving

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}.$$

Notice that multiplying a vector times a matrix ($\boldsymbol{A}\boldsymbol{x}$) is an $\mathcal{O}(N^2)$ operation, since for each of the N rows, we must perform N multiplications and $N-1$ additions (the additions and multiplications in a given row is thus an $\mathcal{O}(N)$ operation). So if we guessed a value for $\boldsymbol{x}$, perhaps through multiple (hopefully $< N$) matrix multiplications we could *iterate* to find the solution $\boldsymbol{x}$ to some precision?

This idea is at the heart of the Jacobi's Method.

Note that $\boldsymbol{A}$ can be written as the sum of its diagonal $\boldsymbol{D}$ and remainder $\boldsymbol{R}$:

Source: `https://en.wikipedia.org/wiki/Jacobi_method`

$$\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{R} \qquad \text{where} \qquad \boldsymbol{D} = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \text{ and } \boldsymbol{R} = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}.$$

Let's rewrite the equation as

$$\begin{aligned} \boldsymbol{b} &= \boldsymbol{A}\boldsymbol{x} = \boldsymbol{D}\boldsymbol{x} + \boldsymbol{R}\boldsymbol{x} \\ \implies \boldsymbol{D}\boldsymbol{x} &= \boldsymbol{b} - \boldsymbol{R}\boldsymbol{x} \\ \implies \boldsymbol{x} &= \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}) \end{aligned}$$

So suppose we have a guess for the solution $\boldsymbol{x}^{(0)}$. Let's plug this guess into the right-hand side (RHS) of the equation. Unless our guess is exactly correct, we know that the left-hand side (LHS) will not be the same $\boldsymbol{x}^{(0)}$, so we will call the LHS $\boldsymbol{x}^{(1)}$. We can continue this process by plugging $\boldsymbol{x}^{(1)}$ into the RHS and getting $\boldsymbol{x}^{(2)}$. So to obtain the $(k+1)^{\text{st}}$ iteration from iteration (k), we have:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}^{(k)}).$$

Defining $x_i^{(k+1)}$ to be the i'th element of $\boldsymbol{x}^{(k+1)}$ (i.e., the i'th row), we can rewrite the above equation in **index notation** as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

Index notation is particularly useful when we need to write a numerical code involving vectors, matrices, or tensors, as it provides a more explicit form of the algorithmic structure the code must take than in the, e.g., matrix-vector form. For example, b_i can be read as the i'th row of array `b[i]` in our code.

Convergence considerations

What guarantee do we have that this method will converge to the exact solution? It turns out that the Jacobi method is proven to converge if $\boldsymbol{A}$ is strictly or irreducibly diagonally dominant. Note that even if the matrix is not strictly of irreducibly diagonally dominant, the Jacobi method may still converge.

So the Jacobi method will converge if $\boldsymbol{A}$ is strictly or irreducibly diagonally dominant, but what do these terms mean? Let's first review **strict diagonal dominance**:

Strict diagonal dominance, also known as **strict row diagonal dominance** means that $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$ for all i holds, where i denotes the row and j denotes the column of matrix $\boldsymbol{A}$.

Example

Often we will be given a problem $\boldsymbol{Ax} = \boldsymbol{b}$ in which we can prove all matrices $\boldsymbol{A}$ have some specific form. For example, consider the class of matrices

$$\boldsymbol{A} = \begin{bmatrix} \alpha & 4 & 10 \\ \beta^2 & 5 & \beta^4 \\ -1 & 0 & 2 \end{bmatrix}$$

Under what conditions will this matrix be strictly diagonally dominant? Assume that α and β are real. Your answer must be in the form of a list of inequalities for α and β (i.e., an inequalities in terms of β^2 or β^3 *are not allowed*). In other words, you must write inequalities in the form $a_i < \alpha < b_i$ and $c_i < \beta < d_i$ that result in strict diagonal dominance for the following matrices, where a_i, b_i, c_i, d_i are constants you must find. Note that there may be multiple inequalities that α and β must satisfy, and you must find all of them.

Solution

The definition of strict diagonal dominance is that for any given row, the magnitude of the diagonal element is greater than the sums of magnitudes of all other elements:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad \text{for all } i.$$

The third row is clearly satisfied: $2 > |-1| + 0$.
The first row of this matrix is slightly less trivial, but still easy:

$$\begin{aligned} |\alpha| &> 4 + 10 \\ |\alpha| &> 14 \\ \implies -\infty &< \alpha < -14, \quad \text{and} \\ 14 &< \alpha < \infty \end{aligned}$$

Next, we attempt the second row:

$$\begin{aligned} & & \beta^4 + \beta^2 &< 5 \\ \gamma = \beta^2 & \implies & \gamma^2 + \gamma - 5 &< 0 \end{aligned}$$

It is useful to draw $f(\gamma) = \gamma^2 + \gamma - 5$. Notice that it is a concave-up parabola since $f''(\gamma) = 2 > 0$. The question is, does the parabola's minimum lie above or below the $f(\gamma) = 0$ line? If it lies above, then there are no γ that satisfy $f(\gamma) < 0$. If below, then there is a range of γ such that $f(\gamma) < 0$, and the boundaries of this range will be given by the locations where $f(\gamma) \equiv 0$.
Let's solve this quadratic equation for $f(\gamma)$ and see if there do exist real γ such that $f(\gamma) = 0$:

$$\begin{aligned} \gamma^2 + \gamma - 5 &= 0 \\ \implies \gamma &= \frac{-1 \pm \sqrt{1 + 4 * 5}}{2} \\ &= \frac{-1 \pm \sqrt{21}}{2} \end{aligned}$$

Check $\gamma = 0$: satisfies the inequality, so

$$\begin{aligned} \frac{-1-\sqrt{21}}{2} &< \gamma < \frac{-1+\sqrt{21}}{2} \\ \frac{-1-\sqrt{21}}{2} &< \beta^2 < \frac{-1+\sqrt{21}}{2}, \end{aligned}$$

which implies that $\beta^4 + \beta^2 - 5 = 0$ at four points:

$$\beta = \pm\sqrt{\frac{-1 \pm \sqrt{21}}{2}}$$

We demand β to be real, and $-1 - \sqrt{21} < 0$, so there are only two real roots of the equation:

$$\beta = \pm\sqrt{\frac{-1 + \sqrt{21}}{2}}$$

Now let's stop and think about $f(\beta) = \beta^4 + \beta^2 - 5$. What does the plot $f(\beta)$ look like? Clearly $f''(\beta) = 12\beta^2 + 2 > 0$ for all real β, so we have a quartic equation that is concave-up everywhere.
Thus there are a total of three possible intervals where the inequality $\beta^4 + \beta^2 - 5 < 0$ might be satisfied:

$$\begin{aligned} \sqrt{\frac{-1+\sqrt{21}}{2}} &< \beta < \infty \\ -\sqrt{\frac{-1+\sqrt{21}}{2}} &< \beta < \sqrt{\frac{-1+\sqrt{21}}{2}} \\ -\infty &< \beta < \sqrt{\frac{-1+\sqrt{21}}{2}} \end{aligned}$$

but which intervals actually work? Choose the limit where β is large and either positive or negative. Then $\beta^4 + \beta^2 - 5$ is clearly greater than zero, which violates the inequality. Next choose the limit where β is zero: $\beta^4 + \beta^2 - 5 = -5 < 0$, which satisfies the inequality. Thus we find

$$-\sqrt{\frac{-1+\sqrt{21}}{2}} < \beta < \sqrt{\frac{-1+\sqrt{21}}{2}}$$

We could have also determined this by just thinking about the implication of a concave-up quartic, which implies that there is only one minimum. Thus the middle interval must be the valid one.

Next let's review **irreducible diagonal dominance**.

Irreducible diagonal dominance requires that the matrix first be irreducible:

- i.e., there exists no permutation matrix $\boldsymbol{P}$ such that $\boldsymbol{PAP}^{-1}$ is a block upper triangle matrix. Recall that the permutation matrix is a square matrix in which each row and column contains a single entry of 1, and all other elements are zero. The inverse of a permutation matrix is equal to its transpose.
- An equivalent definition is that for all i and j, there exists a natural number m such that $(\boldsymbol{A}^m)_{i,j} > 0$, where $\boldsymbol{A}^m$ is $\boldsymbol{A}$ multiplied by itself m times.

For example, a square matrix with no zero elements is obviously irreducible.

If the matrix is irreducible, and weak diagonal dominance $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$ holds for all i, with at least one strictly diagonally-dominant row, then we say we have irreducible diagonal dominance.

The converse is not true; Jacobi's method *might* converge for non-diagonally-dominant matrices, but this is not guaranteed, and convergence may be slow.

Jacobi's method is particularly useful for engineers, scientists, and applied mathematicians because we often must solve $\boldsymbol{Ax} = \boldsymbol{b}$ for diagonally-dominant $\boldsymbol{A}$ when solving PDEs.

17.2 Gauss-Seidel Method

The Gauss-Seidel method improves on Jacobi's method when solving $\boldsymbol{Ax} = \boldsymbol{b}$, as we will see. Although the Gauss-Seidel method does represent an improvement, Jacobi's method is more readily parallelizable, so is still in wide use today.

The basic strategy for the Gauss-Seidel method is as follows

Source: `http://mech.utah.edu/~pardyjak/me2040/Lect10_IterativeSolvers.pdf`

Consider the matrix equation

$$\boldsymbol{Ax} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Then we can solve the top row for x_1, the second row for x_2, and the third row for x_3:

$$\begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \\ x_2 &= \frac{b_2 - a_{21}x_1 - a_{23}x_3}{a_{22}} \\ x_3 &= \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}}. \end{aligned}$$

Let's solve for $\boldsymbol{x}$ iteratively, so that for element j in this vector at iteration k, we have $x_j^{(k)}$. Then the Gauss-Seidel algorithm for a generic 3×3 matrix is as follows:

1. generate $x_1^{(1)}$ by plugging in guessed values for $x_2^{(0)}$ and $x_3^{(0)}$ into the top equation.
2. generate $x_2^{(1)}$ by plugging in the guessed value $x_3^{(0)}$ and the updated value $x_1^{(1)}$.
3. generate $x_3^{(1)}$ by plugging in the updated values $x_1^{(1)}$ and $x_2^{(1)}$.
4. If the relative error between steps

$$E_{\text{rel}}^{(k)} = \sum_{j=0}^{N} \frac{\left| x_j^{(k)} - x_j^{(k-1)} \right|}{\left(\left| x_j^{(k)} \right| + \left| x_j^{(k-1)} \right| \right) / 2}$$

falls below some threshold (e.g., $10^{-14}\sqrt{N}$ in double precision), exit the algorithm. Otherwise repeat the above, replacing $(k) \rightarrow (k+1)$.

The generic form for a Gauss-Seidel step is as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right),$$

Like Jacobi's method, Gauss-Seidel is also guaranteed to converge if $\boldsymbol{A}$ is strictly or irreducibly diagonally dominant, but in addition will converge if $\boldsymbol{A}$ is symmetric positive-definite (i.e., if $\boldsymbol{A}$ is both symmetric and positive definite).

A matrix $\boldsymbol{A}$ is symmetric if $a_{ij} = a_{ji}$ for all elements a_{ij}.

Further, a matrix $\boldsymbol{A}$ is positive-definite if and only if for all nonzero vectors $\boldsymbol{x}$,

$$\boldsymbol{x}^T \boldsymbol{Ax} > 0$$

The notion of a positive-definite matrix is analogous to that of a positive real number. Note that the identity matrix $\boldsymbol{I}$ is positive definite, because for nonzero vector $\boldsymbol{x} = \{a_1, a_2, a_3, a_4, ..., a_n\}$,

$$\boldsymbol{x}^T \boldsymbol{Ix} = a_1^2 + a_2^2 + a_3^2 + ... + a_n^2 > 0$$

Similarly,

Source: `https://en.wikipedia.org/wiki/Positive-definite_matrix`

$$M = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

is positive definite since for any non-zero column vector "z" with entries "a", "b" and "c", we have

$$\begin{aligned} z^{\mathrm{T}}Mz = (z^{\mathrm{T}}M)z &= \begin{bmatrix} (2a-b) & (-a+2b-c) & (-b+2c) \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \\ &= 2a^2 - 2ab + 2b^2 - 2bc + 2c^2 \\ &= a^2 + (a-b)^2 + (b-c)^2 + c^2 \end{aligned}$$

Again, we may obtain convergence even if these convergence guarantees are not met. I.e., positive-definite and diagonally-dominant are *sufficient* but *not necessary* conditions for convergence.

Exercise:

Source: `http://www.math.utah.edu/~zwick/Classes/Fall2012_2270/Lectures/Lecture33_with_Examples.pdf`

Applying the technique just described for determining whether a matrix is positive definite, prove that $\boldsymbol{A} = \begin{bmatrix} a & b \\ 0 & c \end{bmatrix}$ is positive definite if $a > 0$, $c > 0$, and $|b| < 2\sqrt{ac}$. All quantities are real. *Hint:* You'll end up with a quadratic equation for x. First prove that the equation is never equal to zero unless $x = y = 0$. Then concavity (second derivative) will prove that it must be positive.

Solution:

Let $\boldsymbol{z} = \{x, y\}^T$. Then $\boldsymbol{z}^T\boldsymbol{A}\boldsymbol{z} = ax^2 + byx + cy^2$. We prove that $ax^2 + byx + cy^2 > 0$ by first showing that $ax^2 + byx + cy^2 = 0$ only if $x = y = 0$. WOLOG, choose y fixed. Then we have a function of one variable $f(x)$, and we simply solve the quadratic equation to find when $f(x) = 0$. This yields $x = y\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. $b^2 - 4ac < 0$ so since x, y, a, b, c real, the only way this can hold is if $x = y = 0$. Thus we conclude that $ax^2 + byx + cy^2 > 0$ or < 0, strictly. Next we can check for concavity. Note that $\frac{d^2}{dx^2}f(x) = 2a > 0$, so this function is concave up with respect to x, and the only way a function that never hits zero can be concave up is if it is strictly positive for all x.

There are other, more straightforward ways of proving positive definiteness when the matrix is symmetric. One way is to compute all eigenvalues and show they're all positive, but this can be expensive if the matrix is huge. Another way of proving positive definiteness when the matrix $\boldsymbol{A}$ is symmetric is to compute its upper triangle form $\boldsymbol{U}$. If any of the elements along the diagonal of $\boldsymbol{U}$ (called the matrix's *pivots*) is negative, then $\boldsymbol{A}$ is not positive definite. Otherwise $\boldsymbol{A}$ is positive definite.

17.3 Successive Over-Relaxation

Source: `http://mech.utah.edu/~pardyjak/me2040/Lect10_IterativeSolvers.pdf`

As we converge to a solution when using the Gauss-Seidel method, the solution vector $\boldsymbol{x}^{(k)}$ often asymptotes to the exact solution. So how about instead of keeping $\boldsymbol{x}^{(k+1)}$ provided to us by Gauss-Seidel, we linearly extrapolate the solution farther along the asymptote, accelerating our convergence?

This idea is at the heart of successive over-relaxation (SOR)[1].

Let's first perform a Gauss-Seidel step from $\boldsymbol{x}^{(k)}$ to obtain a new guess, which we'll call $\boldsymbol{x}^{(*)}$. Instead of keeping this value as our next iteration, let's instead draw a line connecting the two values for $\boldsymbol{x}$ and choose $\boldsymbol{x}^{(k+1)}$ to lie

[1] SOR is an extension to the Gauss-Seidel method to accelerate convergence. A similar approach, called Scheduled Relaxation Jacobi (SRJ) is an analogue to SOR, but for Jacobi's method.

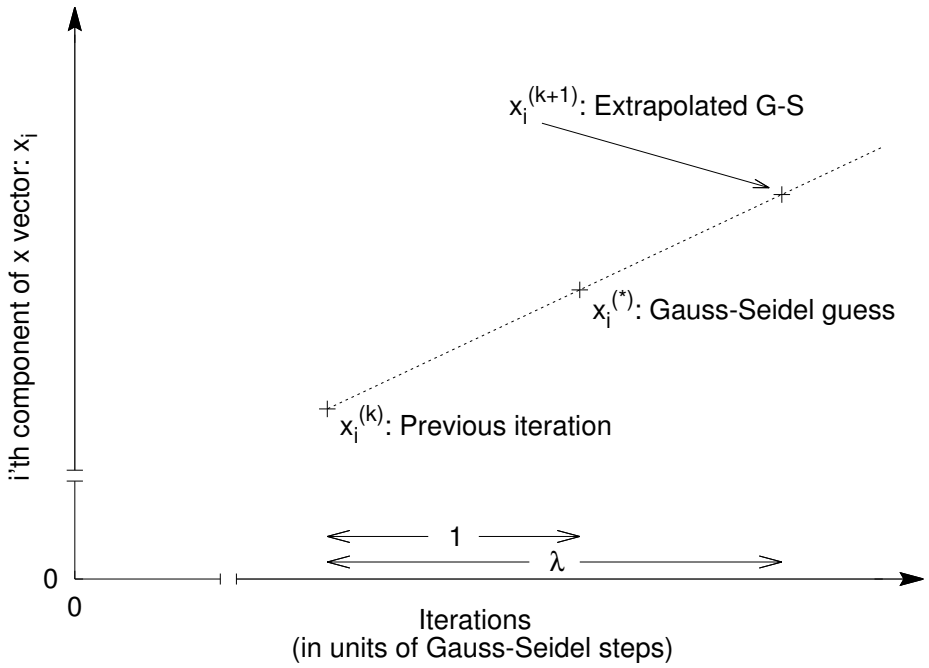

Figure 17.1: Extrapolating the Gauss-Seidel solution for faster convergence. The value for the ith component of the solution vector at iteration $(k+1)$, $x_i^{(k+1)}$, is obtained by first performing the Gauss-Seidel step, which carries the solution vector from $\boldsymbol{x}^{(k)}$ to $\boldsymbol{x}^{(*)}$, and then linearly extrapolating the result by a distance $\lambda > 1$ in units of Gauss-Seidel iterations. This is known as over-relaxation. Under-relaxation occurs for $0 < \lambda < 1$, and Gauss-Seidel (i.e., $\boldsymbol{x}^{(*)} = \boldsymbol{x}^{(k+1)}$) is restored when $\lambda = 1$.

along that line, at distance $\lambda > 1$ from $\boldsymbol{x}^{(k)}$, where $\lambda = 1$ would correspond to the step size of a single Gauss-Seidel iteration.

Figure 17.3 shows how this extrapolation works, plotting $\boldsymbol{x}$ as a function of iteration (k). Since $\boldsymbol{x}^{(k+1)}$ is collinear with $\boldsymbol{x}^{(k)}$ and $\boldsymbol{x}^{(*)}$, the slopes must be the same:

$$\begin{aligned} \frac{\boldsymbol{x}^{(*)} - \boldsymbol{x}^{(k)}}{1} &= \frac{\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}}{\lambda} \\ \implies \boldsymbol{x}^{(k+1)} &= \boldsymbol{x}^{(k)} + \lambda\left(\boldsymbol{x}^{(*)} - \boldsymbol{x}^{(k)}\right) \\ &= \lambda\boldsymbol{x}^{(*)} + (1-\lambda)\boldsymbol{x}^{(k)} \end{aligned}$$

We call λ the "relaxation coefficient".

One iteration of Gauss-Seidel yields $\boldsymbol{x}^{(*)}$:

$$x_i^{(*)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(*)} - \sum_{j>i} a_{ij}x_j^{(k)}\right),$$

so according to the prescription above, we have

$$x_i^{(k+1)} = (1-\lambda)x_i^{(k)} + \frac{\lambda}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(*)} - \sum_{j>i} a_{ij}x_j^{(k)}\right).$$

However, recall that the Gauss-Seidel method is based on the notion that we fully reuse the solution vector as it is updated. So instead of using $x_j^{(*)}$, we could accelerate convergence further by using the already-computed $x_j^{(k+1)}$ instead. This final substitution completes our construction of the method of SOR:

$$x_i^{(k+1)} = (1-\lambda)x_i^{(k)} + \frac{\lambda}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right). \tag{17.1}$$

SOR enables us to obtain faster convergence than Gauss-Seidel, but how should we set λ? In short, choice of an optimal λ depends on the matrix equation we wish to solve, and may require tuning through trial and error. Here is the basic approach:

- Underrelaxation: $0 < \lambda < 1$. Provides a weighted average of current & previous result, with extra emphasis on the previous result. In short, we choose not to take an entire Gauss-Seidel step.
  - If the Gauss-Seidel method is either not converging or oscillating around a solution, choosing λ in this range may yield a convergent, less oscillatory solution.
- No relaxation: $\lambda = 1 \implies$ Just use the plain Gauss-Seidel Method!
- Overrelaxation: $1 < \lambda < 2$. Extra weight is placed on present value. In short, we choose to try and speed up convergence by extrapolating beyond a Gauss-Seidel step.
  - If the Gauss-Seidel method is convergent when using the Gauss-Seidel method, this *over-relaxation* will speed up convergence to the solution.

In summary, we have found a strategy whereby a matrix that is already convergent when using the Gauss-Seidel method can be solved even faster using successive over-relaxation, by setting $\lambda > 1$. The acceleration in convergence grows with λ, but why must $\lambda < 2$? The next section answers this question.

17.4 Convergence of an Iterative Method

A generic iterative method for solving $\boldsymbol{Ax} = \boldsymbol{b}$ may be written

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{M}\boldsymbol{x}^{(k)} + \boldsymbol{c}$$

Definition: The spectral radius $\rho(\boldsymbol{M})$ for a generic iterative method is given by the maximum magnitude eigenvalue $|L_i|$ in the matrix $\boldsymbol{M}$.

Interesting note: If L is an eigenvalue of $\boldsymbol{M}$, then L^n is an eigenvalue of $\boldsymbol{M}^n$, $n = 1, 2, 3....$ Thus $\rho(\boldsymbol{M}^n) = \rho(\boldsymbol{M})^n$.

Theorem: The iterative method $\boldsymbol{x}^{(k+1)} = \boldsymbol{M}\boldsymbol{x}^{(k)} + \boldsymbol{c}$ converges for all starting points iff (i.e., if and only if) $\rho(\boldsymbol{M}) < 1$.

To understand why this theorem is true, consider that for an iterative method, we get ($l.o.t \equiv$ lower order terms; terms with lower powers of $\boldsymbol{M}$):

$$\begin{aligned}
\boldsymbol{x}^{(1)} &= \boldsymbol{M}\boldsymbol{x}^{(0)} + \boldsymbol{c} \\
\boldsymbol{x}^{(2)} &= \boldsymbol{M}\boldsymbol{x}^{(1)} + \boldsymbol{c} \\
&= \boldsymbol{M}(\boldsymbol{M}\boldsymbol{x}^{(0)} + \boldsymbol{c}) + \boldsymbol{c} \\
&= \boldsymbol{M}^2\boldsymbol{x}^{(0)} + l.o.t \\
&\vdots \quad\quad \vdots \\
\boldsymbol{x}^{(n)} &= \boldsymbol{M}^n\boldsymbol{x}^{(0)} + l.o.t
\end{aligned}$$

Notice that if we start with $\boldsymbol{x}^{(0)} = \boldsymbol{0}$ then the highest order nonzero term will be $\boldsymbol{M}^{n-1}\boldsymbol{c}$ [what would need to happen for this to be zero?]. Thus in the limit $n \to \infty$, $\boldsymbol{M}^n$ had better not diverge!

Source: `https://en.wikipedia.org/wiki/Spectral_radius`

Consider L to be an eigenvalue of $\boldsymbol{M}$ and $\boldsymbol{v}$ to be the corresponding eigenvector. From the definition of eigenvalue and eigenvector, we know that

$$\boldsymbol{M}\boldsymbol{v} = L\boldsymbol{v}$$

Also, from above we know

$$\boldsymbol{M}^k\boldsymbol{v} = L^k\boldsymbol{v}$$

Thus

$$\lim_{k\to\infty} \boldsymbol{M}^k\boldsymbol{v} = \lim_{k\to\infty} L^k\boldsymbol{v} = \boldsymbol{v} \lim_{k\to\infty} L^k$$

which will tend toward zero if and only if $|L| < 1$. Since this must hold for all eigenvalues, $\rho(\boldsymbol{M}) < 1$ guarantees convergence.

We will next show that for SOR, the spectral radius satisfies $\rho(\boldsymbol{M}) \geq |\lambda - 1|$. So if $\rho(\boldsymbol{M}) < 1$ is required for convergence, we have

$$\begin{aligned} |\lambda - 1| &\leq \rho(\boldsymbol{M}) < 1 \\ \implies -1 &< \lambda - 1 < 1 \\ \implies 0 &< \lambda < 2. \end{aligned}$$

I.e., when $\rho(\boldsymbol{M}) < 1$, SOR will converge if and only if $0 < \lambda < 2$, which is exactly what we said before.

Source: `http://people.maths.ox.ac.uk/wendland/teaching/mt201011/part5.pdf`

Proof: SOR can be written in matrix form as follows. We are solving $\boldsymbol{Ax} = \boldsymbol{b}$. Write $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U}$. Then

$$\begin{aligned} \boldsymbol{Ax} = (\boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U})\boldsymbol{x} &= \boldsymbol{b} \\ \implies (\boldsymbol{D} + \lambda\boldsymbol{L})\boldsymbol{x} + (\lambda\boldsymbol{U} + (\lambda - 1)\boldsymbol{D})\boldsymbol{x} &= \lambda\boldsymbol{b} \\ \implies (\boldsymbol{D} + \lambda\boldsymbol{L})\boldsymbol{x} &= \lambda\boldsymbol{b} - (\lambda\boldsymbol{U} + (\lambda - 1)\boldsymbol{D})\boldsymbol{x} \\ \implies \boldsymbol{x} &= (\boldsymbol{D} + \lambda\boldsymbol{L})^{-1}[\lambda\boldsymbol{b} - (\lambda\boldsymbol{U} + (\lambda - 1)\boldsymbol{D})\boldsymbol{x}] \end{aligned}$$

SOR generates an iterative method by plugging in a guess $\boldsymbol{x}^{(k)}$ in the RHS. Unless $\boldsymbol{Ax}^{(k)} \equiv \boldsymbol{b}$ LHS≠RHS and we get a new estimate for the solution $\boldsymbol{x}$, called $\boldsymbol{x}^{(k+1)}$:

$$\boldsymbol{x}^{(k+1)} = (\boldsymbol{D} + \lambda\boldsymbol{L})^{-1}\left[\lambda\boldsymbol{b} - (\lambda\boldsymbol{U} + (\lambda - 1)\boldsymbol{D})\boldsymbol{x}^{(k)}\right]. \tag{17.2}$$

Exercise:

Starting with Eq. 17.1, derive Eq. 17.2 to prove the two expressions are identical.

Solution:

Eq. 17.1 states that

$$x_i^{(k+1)} = (1-\lambda)x_i^{(k)} + \frac{\lambda}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right).$$

Moving the $x_i^{(k+1)}$ terms to the left-hand side of the equation yields

$$x_i^{(k+1)} + \frac{\lambda}{a_{ii}}\sum_{j<i} a_{ij}x_j^{(k+1)} = (1-\lambda)x_i^{(k)} + \frac{\lambda}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)}\right). \tag{17.3}$$

Next notice that $\sum_{j<i} a_{ij}x_j^{(k+1)}$ multiplies the lower-triangle portion of $\boldsymbol{A}$ with the components of $\boldsymbol{x}$, thus is equal to $\boldsymbol{Lx}$. Similarly, $\frac{\lambda}{a_{ii}}$ is simply $\lambda\boldsymbol{D}^{-1}$. Therefore,

$$\begin{aligned} x_i^{(k+1)} + \frac{\lambda}{a_{ii}}\sum_{j<i} a_{ij}x_j^{(k+1)} &= x_i^{(k+1)} + \lambda\left[\boldsymbol{D}^{-1}\boldsymbol{L}\boldsymbol{x}^{(k+1)}\right]_i \\ &= \left[\left(1+\lambda\boldsymbol{D}^{-1}\boldsymbol{L}\right)\boldsymbol{x}^{(k+1)}\right]_i \\ &= \left[\boldsymbol{D}^{-1}\left(\boldsymbol{D}+\lambda\boldsymbol{L}\right)\boldsymbol{x}^{(k+1)}\right]_i. \end{aligned}$$

The right-hand side of Eq. 17.3 can be rewritten in matrix notation in a similar way:

$$\begin{aligned} (1-\lambda)x_i^{(k)} + \frac{\lambda}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)}\right) &= \frac{1}{a_{ii}}\left(\lambda b_i - \left[\lambda\sum_{j>i} a_{ij} + a_{ii}(\lambda-1)\right]x_i^{(k)}\right) \\ &= \left[\boldsymbol{D}^{-1}\left(\lambda\boldsymbol{b} - [\lambda\boldsymbol{U} + \boldsymbol{D}(\lambda-1)]\,\boldsymbol{x}^{(k)}\right)\right]_i. \end{aligned}$$

Thus we have shown

$$\begin{aligned} \left[\boldsymbol{D}^{-1}\left(\boldsymbol{D}+\lambda\boldsymbol{L}\right)\boldsymbol{x}^{(k+1)}\right]_i &= \left[\boldsymbol{D}^{-1}\left(\lambda\boldsymbol{b} - [\lambda\boldsymbol{U} + \boldsymbol{D}(\lambda-1)]\,\boldsymbol{x}^{(k)}\right)\right]_i \\ \implies \boldsymbol{D}^{-1}\left(\boldsymbol{D}+\lambda\boldsymbol{L}\right)\boldsymbol{x}^{(k+1)} &= \boldsymbol{D}^{-1}\left(\lambda\boldsymbol{b} - [\lambda\boldsymbol{U} + \boldsymbol{D}(\lambda-1)]\,\boldsymbol{x}^{(k)}\right). \end{aligned}$$

Left multiply both sides of the above equation by $\left(\boldsymbol{D}+\lambda\boldsymbol{L}\right)^{-1}\boldsymbol{D}$ to obtain

$$\boldsymbol{x}^{(k+1)} = \left(\boldsymbol{D}+\lambda\boldsymbol{L}\right)^{-1}\left(\lambda\boldsymbol{b} - [\lambda\boldsymbol{U} + \boldsymbol{D}(\lambda-1)]\,\boldsymbol{x}^{(k)}\right),$$

which is precisely Eq. 17.2.

Thus we now have SOR written in the generic iterative form:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{M}\boldsymbol{x}^{(k)} + \boldsymbol{c},$$

where (since $(\boldsymbol{AB})^{-1} = \boldsymbol{B}^{-1}\boldsymbol{A}^{-1}$):

$$\begin{aligned} \boldsymbol{M} &= (\boldsymbol{D}+\lambda\boldsymbol{L})^{-1}((1-\lambda)\boldsymbol{D} - \lambda\boldsymbol{U}) \\ &= (\boldsymbol{D}(\boldsymbol{I}+\lambda\boldsymbol{D}^{-1}\boldsymbol{L}))^{-1}((1-\lambda)\boldsymbol{D} - \lambda\boldsymbol{U}) \\ &= (\boldsymbol{I}+\lambda\boldsymbol{D}^{-1}\boldsymbol{L})^{-1}\boldsymbol{D}^{-1}((1-\lambda)\boldsymbol{D} - \lambda\boldsymbol{U}) \\ &= (\boldsymbol{I}+\lambda\boldsymbol{D}^{-1}\boldsymbol{L})^{-1}((1-\lambda)\boldsymbol{I} - \lambda\boldsymbol{D}^{-1}\boldsymbol{U}) \end{aligned}$$

So $\boldsymbol{M}$ is the product of two matrices:

1. $(\boldsymbol{I} + \lambda \boldsymbol{D}^{-1}\boldsymbol{L})^{-1}$ is the inverse of a lower triangle matrix with ones along the diagonal. To understand why, note that $\boldsymbol{D}^{-1}$ has nonzero elements only along the diagonal, just like the unit matrix $\boldsymbol{I}$. So the multiplication $\boldsymbol{D}^{-1}\boldsymbol{L}$ does not result in a matrix with nonzero elements along the diagonal and is therefore simply a lower-triangle matrix with zeroes along the diagonal.
2. $((1-\lambda)\boldsymbol{I} - \lambda \boldsymbol{D}^{-1}\boldsymbol{U})$ is an upper triangle matrix with $(1-\lambda)$'s along the diagonal, using similar reasoning as above.

But what are the eigenvalues of $\boldsymbol{M}$ which will yield information about the spectral radius? To make progress, we need to apply a few properties of determinants:

Source: `https://en.wikipedia.org/wiki/Triangular_matrix`

1. The determinant of a triangular matrix is the product of entries on the diagonal.
2. $\det(\boldsymbol{A}^{-1}) = 1/\det(\boldsymbol{A})$
3. $\det(\boldsymbol{A}\boldsymbol{B}) = \det(\boldsymbol{A})\det(\boldsymbol{B})$
4. $\det(\boldsymbol{A})$ is equal to the product of eigenvalues of $\boldsymbol{A}$.

Source: `http://www.hesse-kerstin.de/NLA_Lecture_Notes.pdf` Clearly, since $\det((\boldsymbol{I} + \lambda \boldsymbol{D}^{-1}\boldsymbol{L}))$ is lower triangular and has ones along the diagonal:

$$\det((\boldsymbol{I} + \lambda \boldsymbol{D}^{-1}\boldsymbol{L})^{-1}) = 1/\det((\boldsymbol{I} + \lambda \boldsymbol{D}^{-1}\boldsymbol{L})) = 1$$

Also, from determinant property (1) and (4),

$$|\det((1-\lambda)\boldsymbol{I} - \lambda \boldsymbol{D}^{-1}\boldsymbol{U})| = |(1-\lambda)|^n = |\ell_1\ell_2\ell_3...\ell_n|$$

for n eigenvalues ℓ, since we are dealing with $n \times n$ matrices.

$|(1-\lambda)|^n = |\ell_1\ell_2\ell_3...\ell_n|$ implies that there must be at least one eigenvalue ℓ_i of $\boldsymbol{M}$ such that $|\ell_i| \geq |1-\lambda|$. Thus since

$$\rho(\boldsymbol{M}) = \max(|\ell(\boldsymbol{M})|),$$

we have

$$\rho(\boldsymbol{M}) = \max(|\ell(\boldsymbol{M})|) \geq |\ell_i| \geq |1-\lambda|.$$

Notice that this inequality sets a limit on λ based on the spectral radius of $\boldsymbol{M}$. In order for SOR to be convergent, we require the additional property that $\rho(\boldsymbol{M}) < 1$, which as shown above implies $0 < \lambda < 2$. The properties of $\boldsymbol{A}$ that guarantee convergence of SOR (e.g., $\boldsymbol{A}$ is symmetric positive-definite) are therefore properties that have been proven to always yield $\rho(\boldsymbol{M}) < 1$. Clearly convergence of SOR may be observed for other types of matrices, but proofs for what $\boldsymbol{A}$'s satisfy $\rho(\boldsymbol{M}) < 1$ are less than straightforward.

MATH 522, Spring 2019 Notes

Chapter 18: Hydrodynamics of Barotropic Fluids and the Method of Characteristics

Based on/for further reading: `http://www.mpia.de/homes/dullemon/lectures/fluiddynamics/Chapter_1.pdf`

Hydrodynamics is the study of gasses in which the fluid description is valid. I.e., the mean free path between collisions of particles is far far smaller than the length scales we study the gas dynamics or flows.

Let's now apply what we have learned to the equations that govern fluid flow. On a numerical grid, we model the fluid's properties at every grid point, for example the pressure, density, velocity, magnetic field, internal energy, etc. We call each grid cell in which the fluid resides a **fluid element**.

For simplicity, we will consider in this course the dynamics of ideal, or perfect fluids. A perfect fluid, as you might imagine, is an idealization of a fluid, in which for example no viscosities are present. It's a valid approximation when densities are low enough and temperatures are high enough, and the fluid behaves as an ideal gas.

We will further restrict ourselves to barotropic fluids. I.e., for fluids in which the fluid pressure is a function of the fluid density only. Such an approximation is valid, for example, in the isothermal limit; when the heating or cooling timescale of the gas is much shorter than the fluid flow timescales. In this case, the internal energy ϵ, i.e., the energy per unit fluid element, is constant.

We will therefore consider fluids with only pressure, velocity, and density defined. The CGS/SI units for each is: P =dyne/cm^2 or N/m^2, ρ =g/cm^3 or kg/m^3, and velocity cm/s or m/s. Thus pressure is the force per unit area, density is the mass per unit volume, and velocity is the speed per unit distance in a given direction.

18.1 Euler's equations for an ideal, barotropic fluid

Based on/for further reading: `http://www.mpia.de/homes/dullemon/lectures/fluiddynamics/Chapter_1.pdf`

Conservation of mass: Consider an arbitrary volume V in the space in which our gas flow takes place. Its surface we denote as $\partial V \equiv S$ with the (outward pointing) normal unit vector at each location on the surface denoted as $\boldsymbol{n}$ and differential surface element as dS (Fig. 1.1). The conservation of mass says that the variation of the mass in the volume must be entirely due to the in- or outflow of mass through ∂V.

Since the total mass is given by

$$M = \int_{\mathcal{V}} \rho dV,$$

the change in mass over time within the volume is simply

$$\partial_t M^{\rm in} = \partial_t \int_{\mathcal{V}} \rho dV.$$

But we know that by mass conservation, mass can be neither created nor destroyed. Therefore any increase or decrease in total mass within the volume must pass through the surface. We introduce a quantity $\rho\boldsymbol{u}$ called "mass flux", which gives the change in mass per unit area per unit time in the direction of the velocity $\boldsymbol{u}$. Thus the mass flux across a point on the surface of the volume will be given by $\rho\boldsymbol{u}\cdot\boldsymbol{n}$, where again $\boldsymbol{n}$ is the unit vector normal to the surface of the volume. So the total mass transfer out of the volume per unit time is given by

$$\partial_t M^{\rm out} = \int_{\partial\mathcal{V}} \rho\boldsymbol{u}\cdot\boldsymbol{n} dS.$$

For mass conservation, we must have $\partial_t M^{\rm in} = -\partial_t M^{\rm out}$, so we get

$$\begin{aligned}\partial_t \int_{\mathcal{V}} \rho dV &= -\int_{\partial\mathcal{V}} \rho \boldsymbol{u} \cdot \boldsymbol{n} dS \\ &= -\int_{\mathcal{V}} \nabla \cdot (\rho \boldsymbol{u}) dV,\end{aligned}$$

where in the last equality we used Gauss' law.

Now let's think about the "mass flux" quantity. We know that momentum is defined as $\boldsymbol{p} = m\boldsymbol{u}$, so $\rho\boldsymbol{u}$ is simply the momentum per unit volume. Thus the total momentum of the fluid contained within the volume is given by $\int_{\mathcal{V}} \rho \boldsymbol{u} dV$. Similar to the argument we just applied for mass conservation, momentum conservation requires that the change in momentum in the volume must be equal to the momentum flux across the surface. So we introduce a quantity analogous to the mass flux $\rho\boldsymbol{u}$ called the "momentum flux": $\rho\boldsymbol{u}\boldsymbol{u}$. Notice the momentum flux is a bit strange; it's not a vector. It is a tensor! Similar to the mass flux, if you dot this tensor into the unit normal to the volume $\boldsymbol{n}$, you get the total momentum per unit area per unit time exiting the volume over a patch of the surface. Thus

$$\partial_t \boldsymbol{p}^{\text{out}} = \int_{\partial\mathcal{V}} \rho \boldsymbol{u} (\boldsymbol{u} \cdot \boldsymbol{n}) dS.$$

But the time-rate-of-change in momentum on the surface is equivalent to the net force over the surface. In fluids, force can manifest from a momentum flux across a surface and from *pressure* along the surface.

Pressure is force per unit area, so the force exerted by an internal pressure on an infinitesimal surface element on the volume is simply $P\boldsymbol{n}dS$. Again, force is simply the time-rate-of-change in momentum, so the total momentum per unit time exiting the surface is

$$\partial_t \boldsymbol{p}^{\text{out}} = \int_{\partial\mathcal{V}} \left(\rho \boldsymbol{u} (\boldsymbol{u} \cdot \boldsymbol{n}) + P\boldsymbol{n}\right) dS$$

We would like to apply Gauss' law to this relation, so that the RHS may be written entirely in terms of a volume integral. But Gauss' law requires that something be dotted in to the normal to the surface. The first term is clearly $\rho\boldsymbol{u}\boldsymbol{u}$ dotted into the normal to the surface, so no problems there. What about the P term? Well, how about we introduce the unit tensor I. With I, $P\boldsymbol{n}$ can be written $P\boldsymbol{I} \cdot \boldsymbol{n}$.

So Gauss' law gives us

$$\partial_t \boldsymbol{p}^{\text{out}} = \int_{\mathcal{V}} \nabla \cdot \left(\rho \boldsymbol{u}\boldsymbol{u} + P\boldsymbol{I}\right) dV$$

Again,

$$\begin{aligned}\partial_t \boldsymbol{p}^{\text{in}} &= -\partial_t \boldsymbol{p}^{\text{out}} \\ \partial_t \int_{\mathcal{V}} \rho \boldsymbol{u} dV &= -\int_{\mathcal{V}} \nabla \cdot \left(\rho \boldsymbol{u}\boldsymbol{u} + P\boldsymbol{I}\right) dV.\end{aligned}$$

Because the ∂_t operator can be absorbed into the integral, we can now immediately write this and the mass conservation equations in differential form:

$$\begin{aligned}\partial_t(\rho) + \nabla \cdot (\rho \boldsymbol{u}) &= 0 \\ \partial_t(\rho \boldsymbol{u}) + \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u} + P\boldsymbol{I}) &= 0\end{aligned}$$

These go by the name of **Euler's equations in the barotropic limit**. The first equation is called the mass conservation or continuity equation, and the second is called the momentum or Euler's equation.

18.1.1 Sound Waves in an Isothermal Fluid

With Euler's equations in the barotropic limit, we have a total of four equations for five unknowns: P, ρ, and $\boldsymbol{u}$. How do we close this system of equations? The answer is: the equation of state of the fluid.

The general equation of state of an ideal gas is

$$P = (\gamma - 1)\epsilon\rho$$

where γ is the adiabatic index of the gas, and ϵ is the internal specific energy. For barotropic fluids, we assume that P can be written in terms of ρ only. For an *isothermal* gas, the specific energy, or "intrinsic kinetic energy" of the gas is assumed constant, so we have

$$P = a\rho$$

where a is a constant.

Let's now try to uncover the nature of this constant and Euler's equations by considering the Euler's equations we just derived:

$$\begin{aligned} \partial_t \rho(x,t) + \partial_x[\rho(x,t)u(x,t)] &= 0 \\ \partial_t[\rho(x,t)u(x,t)] + \partial_x[\rho(x,t)u(x,t)^2 + P(x,t)] &= 0 \end{aligned}$$

Let's analyze this equation in the context of an infinitesimal density perturbation in one spatial and one temporal dimension, about a constant density ρ_0 and zero velocity $u_0 = 0$ background:

$$\begin{aligned} \rho(x,t) &= \rho_0 + \rho_1(x,t) \\ u(x,t) &= u_1(x,t). \end{aligned}$$

Effectively, we are considering the motion of a *sound wave*.

We assume that u_1 and ρ_1 are both vanishingly small and vary over large lengthscales and timescales, so that when they or their derivatives are multiplied with each other or with themselves, the result is not significant enough to contribute to our PDE.

Proceeding, let's apply these assumptions to the continuity equation in the barotropic limit, in one spatial dimension:

$$\begin{aligned} \partial_t(\rho_0 + \rho_1) + \partial_x[(\rho_0 + \rho_1)u_1] &= 0 \\ \implies \partial_t\rho_1 + u_1\partial_x\rho_1 + (\rho_0 + \rho_1)\partial_x u_1 &= 0. \end{aligned}$$

Dropping tiny terms multiplied together as prescribed, we get:

$$\partial_t\rho_1 + \rho_0\partial_x u_1 = 0.$$

Next we apply the assumptions to the momentum equation:

$$\partial_t[(\rho_0 + \rho_1)(u_1)] + \partial_x[(\rho_0 + \rho_1)u_1^2 + P(x,t)] = 0.$$

Note that $P = a\rho$ in the isothermal limit, so

$$\begin{aligned} \partial_t[(\rho_0 + \rho_1)(u_1)] + \partial_x[(\rho_0 + \rho_1)u_1^2 + a(\rho_0 + \rho_1)] &= 0 \\ \partial_t[(\rho_0)(u_1)] + \partial_x[a(\rho_0 + \rho_1)] &= 0 \\ \rho_0\partial_t u_1 + a\partial_x\rho_1 &= 0. \end{aligned}$$

Thus Euler's equations in the isothermal limit can be written

$$\begin{aligned} \partial_t\rho_1 + \rho_0\partial_x u_1 &= 0 \\ \rho_0\partial_t u_1 + a\partial_x\rho_1 &= 0. \end{aligned}$$

Take the time derivative of the top equation:

$$\begin{aligned} \partial_t^2\rho_1 + \rho_0\partial_t\partial_x u_1 &= 0 \\ \partial_t^2\rho_1 + \rho_0\partial_x\partial_t u_1 &= 0 \\ \partial_t^2\rho_1 + \rho_0\partial_x[-\frac{a}{\rho_0}\partial_x\rho_1] &= 0 \\ \partial_t^2\rho_1 - a\partial_x^2\rho_1 &= 0 \\ \partial_t^2\rho_1 &= a\partial_x^2\rho_1 \end{aligned}$$

We conclude that the perturbation ρ_1 satisfies the wave equation, with a speed of propagation given by the **sound speed**[1] $c_s = \sqrt{a}$ (note that $a > 0$ since fluid pressures and densities cannot be negative). Thus we can rewrite the equation relating pressure and density as

$$P = c_s^2 \rho.$$

18.2 The Method of Characteristics

Based on/for further reading: Alice Quillen's AST242 lecture notes, **"Method of Characteristics"** section, starting on Eq 36 `http://astro.pas.rochester.edu/~aquillen/ast242/lecturenotes2.pdf`.

The Method of Characteristics finds a set of curves, called *characteristic curves* along which a PDE can be represented as an ODE. Physically speaking, these curves are related to the propagation of information implied by a given PDE.

Suppose we have coordinates x and t which represent space and time, respectively. Let us write x and t in terms of a parameter s, so that

$$\begin{aligned} x &= x(s) \\ t &= t(s) \end{aligned}$$

Next consider a generic first-order PDE:

$$a(x,t)\partial_x u + b(x,t)\partial_t u + c(x,t)u = 0$$

If $x = x(s)$ and $t = t(s)$, then $a(x,t) = a(x(s),t(s))$. Similarly for b and c. Now notice that if we define

$$\begin{aligned} x'(s) &= a(x(s),t(s)) \\ t'(s) &= b(x(s),t(s)), \end{aligned}$$

called the *characteristic equations*, we can write our first-order PDE as:

$$\frac{dx}{ds}\partial_x u + \frac{dt}{ds}\partial_t u + c(x,t)u = 0$$

By the chain rule of partial differentiation, we know that

$$\frac{du(x(s),t(s))}{ds} = \frac{dx}{ds}\partial_x u + \frac{dt}{ds}\partial_t u$$

Therefore the first-order PDE can be written as an ODE along the parameter s:

$$\frac{du(x(s),t(s))}{ds} + c(x(s),t(s))u = 0,$$

with solution:

$$\begin{aligned} \frac{du}{u} &= -c(s)ds \\ d\ln(u) &= -c(s)ds \end{aligned}$$

Since

$$\begin{aligned} x'(s) &= a(x(s),t(s)) \\ t'(s) &= b(x(s),t(s)), \end{aligned}$$

[1] Note that $c_s^2 \equiv \frac{\partial P}{\partial \rho}$ is actually the *definition* of sound speed c_s in a fluid.

the speed of information propagation in our first-order PDE (i.e., the slope of characteristic curves) at any point in the $x-t$ plane is given by

$$\frac{dx}{dt} = \frac{dx/ds}{dt/ds} = \frac{x'(s)}{t'(s)} = \frac{a}{b}.$$

Generally speaking, the method of characteristics exploits this parameterization of the independent variables x and t to either solve the PDE or at least yield deeper insights into the PDE's solution.

The general strategy for solving the PDE with initial condition $u(x,0) = f(x)$ is as follows:

1. Solve the **characteristic equations**

$$\begin{aligned} x'(s) &= a(x(s),t(s)) \\ t'(s) &= b(x(s),t(s)), \end{aligned}$$

   for $x(s)$ and $t(s)$.

2. To specify the constants of integration for t, assume $t(s=0) = 0$.

3. Choose the constant of integration for x to be $x(s=0) = x_0$. We will interpret this constant later.

4. Invert the solutions found for $x(s)$ and $t(s)$ to find how s and x_0 are related to x and t if possible.

5. Solve

$$\frac{du(x(s),t(s))}{ds} + c(x(s),t(s))u = 0,$$

   for $u(s)$, applying the initial condition $u(x(s=0),0) = f(x_0)$.

6. Substitute $s(x,t)$ into the solution found for $u(s)$, subject to the integration constant $x(s=0) = x_0$ to find $u(x,t)$.

Here is an example: Let $a(x,t) = C$, a constant, $b = 1$ and $c = 0$. Then the first-order PDE becomes:

$$\begin{aligned} C\partial_x u + \partial_t u &= 0 \\ u(x, t=0) &= f(x) \end{aligned}$$

Notice that this is simply the one-directional wave equation, with C being the speed of the wave! We will now find the generic solution to this equation by applying the method of characteristics.

The characteristic equations are

$$\begin{aligned} t'(s) &= b(x(s),t(s)) = 1 \implies t = s + c_1 \\ x'(s) &= a(x(s),t(s)) = C \implies x = Cs + c_2 \end{aligned}$$

Since $t(s=0) = 0$, we have $c_1 = 0$. Similarly, since $x(s=0) = x_0$, we can solve for c_2: $x(s=0) = C*0 + c_2 = x_0$. Thus

$$\begin{aligned} t(s) &= s \\ x(s) &= x_0 + Cs \end{aligned}$$

We already have s in terms of t, so next we solve for x_0 in terms of x and t:

$$\begin{aligned} x(s) &= x_0 + Cs \\ \implies x_0 &= x(s) + Cs \\ \implies x_0 &= x - Ct. \end{aligned}$$

Notice all of the expressions above behave as desired in the limit $s \to 0$.

Next we solve the ODE:

$$\begin{aligned} \frac{du}{ds} &= = 0 \\ \implies u(s) &= c_3. \end{aligned}$$

Thus the solution remains constant along characteristic curves!

Our initial condition is $u(x,0) = f(x)$, which implies $u(s=0) = f(x(s=0)) = f(x_0)$. Thus $c_3 = f(x_0)$, and our solution for u, assuming that we start at an arbitrary point $x(s=0) = x_0$ is $u(s) = f(x_0)$, where $x_0 = x - Ct$ as shown above. Thus

$$u(s) = f(x_0) = f(x - Ct) = u(x,t).$$

Thus our solution u of the one-directional wave equation simply translates the initial condition to the left at speed C, explaining why we call it the one-directional wave equation!

Now let's return to the notion of *characteristic curves*. If you interpret this solution, which is a wave, as the motion of a collection of particles, the characteristic curves will trace the motion of the particles in the $x-t$ plane.

18.2.1 Solving Euler's Hydrodynamics Equations for Barotropic Fluids in One Spatial Dimension, Using the Method of Characteristics

Based on/for further reading: Michael Stone & Paul Goldbart text **Mathematics for Physics**, Section 7.3.1: Sound in air, `http://www.goldbart.gatech.edu/PostScript/MS_PG_book/bookmaster.pdf`; Alice Quillen's AST242 lecture notes, **"Method of Characteristics"** section `http://astro.pas.rochester.edu/~aquillen/ast242/lecturenotes2.pdf`.

Next, let's derive the characteristic equations for Euler's equations in the barotropic limit in one spatial dimension: In this limit, Euler's equations are given by:

$$\begin{aligned} \partial_t \rho + \partial_x(\rho u) &= 0 \\ \partial_t(\rho u) + \partial_x(\rho u^2 + P) &= 0. \end{aligned}$$

In the above expressions, we have two equations and three unknowns. We can close this set of equations if we assume we are dealing with a *barotropic* fluid; i.e., a fluid in which P is a function of ρ only. In this case, we can rewrite the above equations entirely in terms of ρ. Note that the isothermal limit, $P = a\rho$ (where a is a constant) is just a special case of a barotropic fluid.

Our goal is to rewrite the above equations in a form more amenable to the Method of Characteristics.

Let's start by trying to rewrite the momentum equation such that only derivatives of u appear on one side of the equation, and derivatives of P and ρ appear on the other side:

$$\begin{aligned} 0 &= \partial_t(\rho u) + \partial_x(\rho u^2 + P) \\ &= u\partial_t\rho + \rho\partial_t u + u^2\partial_x\rho + 2\rho u\partial_x u + \partial_x P \\ &= (u\partial_t\rho + u^2\partial_x\rho) + \rho\partial_t u + 2\rho u\partial_x u + \partial_x P. \end{aligned}$$

Notice that the two terms $u\partial_t\rho + u^2\partial_x\rho$ can be written:

$$\begin{aligned} u\partial_t\rho + u^2\partial_x\rho &= u(\partial_t\rho + u\partial_x\rho) \\ &= u(\partial_t\rho + [\partial_x(\rho u) - \rho\partial_x u]) \\ &= u(\partial_t\rho + \partial_x(\rho u)) - \rho u\partial_x u \\ &= -\rho u\partial_x u, \end{aligned}$$

where in the last step we applied the continuity equation.

Substituting this back into the momentum equation and simplifying further, we get:

$$\begin{aligned} 0 &= (-\rho u\partial_x u) + \rho\partial_t u + 2\rho u\partial_x u + \partial_x P \\ &= \rho\partial_t u + \rho u\partial_x u + \partial_x P \\ \implies \rho(\partial_t u + u\partial_x u) &= -\partial_x P \\ \implies \partial_t u + u\partial_x u &= -\frac{\partial_x P}{\rho}. \end{aligned}$$

We have rewritten the momentum equation such that all the u terms appear on the left-hand-side, and all the terms involving P and ρ appear on the right-hand side. Bernhard Riemann, one of the greatest mathematicians of all time, studied this problem. In his derivations, he found it convenient to define a new quantity $\pi(P)$ via

$$\begin{aligned} \frac{d\pi}{dP} &= \frac{1}{\rho c_s}, \quad \text{so that} \\ \pi(P) &= \int_{P_0}^{P} \frac{1}{\rho(P')c_s(P')} dP', \end{aligned}$$

where P_0 is the background pressure, atop which we wish to model the propagation of waves. Notice that since $P(\rho)$ is known, and

$$\frac{\partial P}{\partial \rho} = \frac{dP}{d\rho} = c_s^2,$$

c_s is also just a function of ρ. Also note that we have made the assumption that $P(\rho)$ is a *one-to-one* function, meaning that the inverse function $\rho(P)$ exists (i.e., $\rho(P)$ is not multi-valued) in the region of interest for ρ and P, so that $c_s(P(\rho)) = c_s(\rho)$ also exists in this region.

Since π can be written as a function of P only (where P is a function of x and t), we can apply the chain rule to compute $\partial_x \pi$:

$$\begin{aligned} \partial_x \pi(P) &= \frac{d\pi}{dP} \partial_x P \\ &= \frac{1}{\rho c_s} \partial_x P. \end{aligned}$$

Similarly, $\partial_t \pi$ is given by

$$\begin{aligned} \partial_t \pi(P) &= \frac{d\pi}{dP} \partial_t P \\ &= \frac{1}{\rho c_s} \partial_t P. \end{aligned}$$

Thus we have found that

$$\begin{aligned} \partial_t P &= \rho c_s \partial_t \pi \\ \partial_x P &= \rho c_s \partial_x \pi. \end{aligned}$$

Then the momentum equation becomes:

$$\begin{aligned} \partial_t u + u\partial_x u &= -\frac{\partial_x P}{\rho} \\ &= -\frac{\rho c_s \partial_x \pi}{\rho} \\ &= -c_s \partial_x \pi \\ \implies \partial_t u + u\partial_x u + c_s \partial_x \pi &= 0 \end{aligned}$$

Next we will attempt to rewrite the continuity equation in terms of π, c_s, and u as well:

$$\begin{aligned} 0 &= \partial_t \rho + \partial_x(\rho u) \\ &= \partial_t \rho + u\partial_x \rho + \rho \partial_x u. \end{aligned}$$

From the definition of $\pi(P)$ and the fact that P is a function of ρ, we can write

$$\begin{aligned} \partial_x \pi(P) &= \frac{d\pi}{dP} \partial_x P(\rho) \\ &= \frac{d\pi}{dP} \frac{dP}{d\rho} \partial_x \rho \\ &= \frac{1}{\rho c_s} c_s^2 \partial_x \rho \\ &= \frac{c_s}{\rho} \partial_x \rho \\ \implies \partial_x \rho &= \frac{\rho}{c_s} \partial_x \pi \end{aligned}$$

Similarly,

$$\partial_t P = \frac{\rho}{c_s}\partial_t \pi.$$

Using these facts, we can rewrite the continuity equation in terms of π:

$$\begin{aligned} 0 &= \partial_t \rho + \partial_x(\rho u) \\ &= \partial_t \rho + u\partial_x \rho + \rho \partial_x u \\ &= \frac{\rho}{c_s}\partial_t \pi + u\frac{\rho}{c_s}\partial_x \pi + \rho \partial_x u \\ &= \frac{\rho}{c_s}\left(\partial_t \pi + u\partial_x \pi + c_s \partial_x u\right) \end{aligned}$$

We are not interested in the trivial solution $\rho = 0$, so the term in parentheses must be zero. Thus we have rewritten Euler's equations in the barotropic limit in terms of two equations and two unknowns:

$$\begin{aligned} \partial_t \pi + u\partial_x \pi + c_s \partial_x u &= 0 \\ \partial_t u + u\partial_x u + c_s \partial_x \pi &= 0 \end{aligned}$$

Remember our goal is to understand the nature of these equations using the method of characteristics. So let's rewrite these equations in a more suggestive form. First, adding the two equations we get:

$$\partial_t[u + \pi] + (u + c_s)\partial_x[u + \pi] = 0$$

Then subtracting the two equations, we get:

$$\partial_t[u - \pi] + (u - c_s)\partial_x[u - \pi] = 0$$

The characteristic equations are given by

$$\begin{aligned} \frac{dx}{ds} &= u \pm c_s \\ \frac{dt}{ds} &= 1 \end{aligned}$$

Since u on the RHS of the $x'(s)$ equation itself depends on s, we cannot immediately integrate these equations using the strategy outlined previously. However, the method of characteristics does give us some physical insights into this equation.

First, the method of characteristics gives us the speed of propagation of signals:

$$\frac{dx}{dt} = \frac{\frac{dx}{ds}}{\frac{dt}{ds}} = u \pm c_s.$$

Note that u here is the local fluid velocity. So this tells us that signals in the fluid travel with repect to the fluid motion at the sound speed c_s... not a particularly surprising result, but a nice physical insight nonetheless.

The method of characteristics also tells us that the solution to each PDE remains constant along the characteristic curves. I.e.,

$$\frac{d[u \pm \pi]}{ds} = -c(x(s), t(s))[u \pm \pi](s) = 0 \implies [u \pm \pi](s) = \text{constant}.$$

So the value of $u + \pi$ is constant along the characteristic curve, which itself is given by the solution to the equation

$$\frac{dx}{dt} = u + c_s$$

and the value of $u - \pi$ is constant along the characteristic curve given by

$$\frac{dx}{dt} = u - c_s.$$

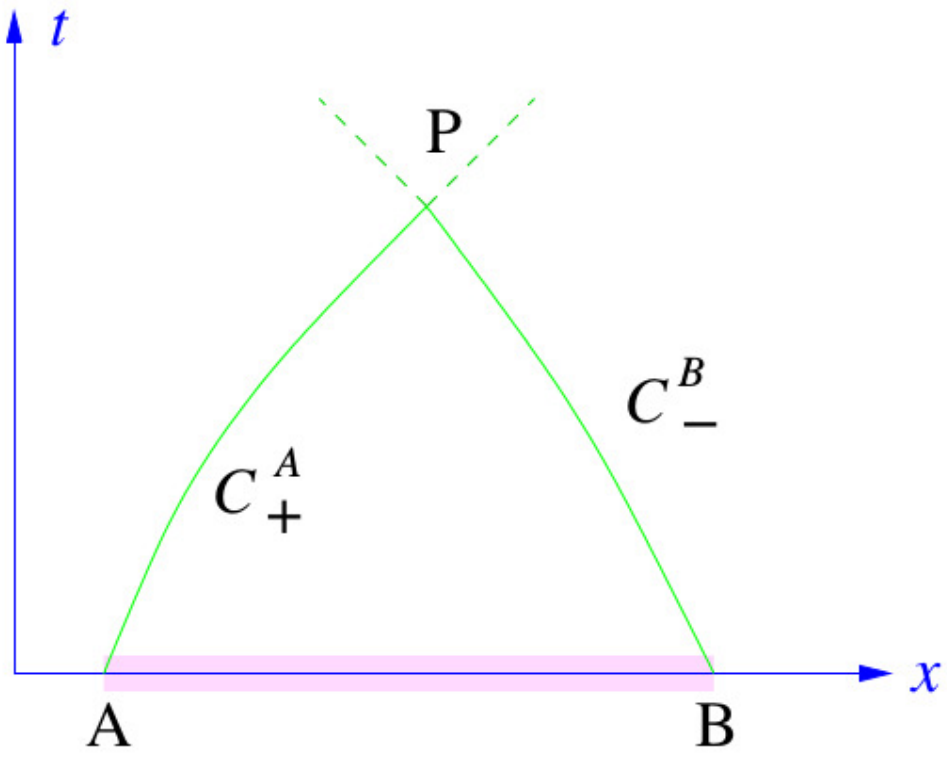

Figure 7.9: *Non-linear characteristic curves.*

Figure 18.1: Characteristic curves of Euler's equations of hydrodynamics for a barotropic fluid, in one spatial dimension. From Figure 7.9 of Michael Stone & Paul Goldbart text **Mathematics for Physics**, Section 7.3.1: Sound in air, `http://www.goldbart.gatech.edu/PostScript/MS_PG_book/bookmaster.pdf`.

Looking at Fig. 18.1, the solution π and u at any point in space and time $\mathcal{P}$ can be found if we know the initial values of $u + \pi$ at the point A and $u - \pi$ at the point B.

However, if we are given $\mathcal{P}$, determining the precise values $x = A$ and $x = B$ is challenging, because the slope of the characteristic curves $\mathcal{C}_+$ and $\mathcal{C}_-$ will depend on the time-dependent values of u and c_s *at every point along the curves*.

Thus constructing the characteristic curves from point $\mathcal{P}$ backwards is the real challenge, requiring that we consider the interaction of all other characteristic curves that influence our characteristic curves. Luckily, information cannot travel faster than the speed $u \pm c_s$, so the solution at $\mathcal{P}$ may only depend on the initial conditions between $x = A$ and $x = B$. This region along the $x-$axis is known as the *domain of dependence*, and draws parallels to the Courant-Friedrichs-Lewy (CFL) condition, which sets a ceiling on the largest allowed timestep in certain timestepping schemes.

Note that the domain of dependence is limited by the local fluid velocity u and sound speed c_s at each point (x, t), such that no information can travel faster than $u \pm c_s$ at any point. In this way, fluids with larger (smaller) c_s possess a wider (narrower) domain of dependence.

If we are able to find $u + \pi$ and $u - \pi$ at $\mathcal{P}$ from their values at A and B, respectively, we can then solve for the pressure P by inverting

$$\pi = \int_{P_0}^{P} \frac{1}{\rho(P')c_s(P')} dP',$$

which in practice may require numerical integration and/or numerical root-finding.

18.3 Shocks

Shocks are what happens when characteristic curves intersect. In general, the initial conditions at the starting point of the intersecting characteristic curves will have different values. So at the point of a characteristic crossing, the function will need to be *multi-valued*! This is precisely what happens, violating our usual intuition that differential equations must have smooth solutions.

Shocks appear in many physical scenarios, including bomb blasts, the breaking of ocean waves against the shore, and sonic booms, just to name a few. Numerical modeling of shocks is an entire field in and of itself. When shocks arise, numerical models typically rely one of two algorithms for handling the discontinuity.

1. *High-resolution shock capturing (HRSC)* schemes search for the development of shocks and treat fluxes on the "left" and "right" sides of the shocks separately, solving the *Riemann problem* across the shock itself, which is equivalent to ensuring that mass, momentum, and energy are all conserved across the shock. Generally HRSC schemes require that fluid dynamic equations be written in a *conservative* form. For example, in the case of a fluid without external driving forces, hydrodynamic equations can usually be written in the form

   $$\partial_t \boldsymbol{U} + \nabla \cdot \boldsymbol{F} = \boldsymbol{0}$$

   where $\boldsymbol{F}$ is the flux vector and $\boldsymbol{U}$ contains the *evolved* quantities (i.e., the quantities that are propagated forward in time using a time stepping algorithm).

   In general we can rewrite the divergence of the flux term in the form

   $$\boldsymbol{A} \nabla \cdot \boldsymbol{U},$$

   where $\boldsymbol{A}$ is a matrix. As we'll find in an upcoming section, the eigenvalues of this matrix correspond to the characteristic speeds. Solving the Riemann problem amounts to an analysis of the characteristic speeds, coupled to the assumptions of mass, momentum, and energy conservation, to ensure that all quantities and fluxes are conserved across the shock. The important thing to note is that this deviates from our usual strategy for solving PDEs, in which we may assume that the underlying solution is differentiable. There are different strategies for solving the Riemann problem, and for very large systems of nonlinear PDEs, the Riemann problem may require the eigenvalues of a large matrix be computed, which could be very expensive. For such systems, classes of *approximate* Riemann solvers have been developed and shown to generate results that agree well qualitatively as compared to more sophisticated solvers.

2. *Artificial viscosity* schemes take a cue from our earlier strategy for stabilizing the forward-time, centered-space scheme, in that a *parabolic* term is added to the right-hand side of all PDEs. For example, consider Euler's equations in the limit where $c_s \to 0$ and $\pi = 0$:

$$\partial_t u + u\partial_x u = 0$$

would become

$$\partial_t u + u\partial_x u = \gamma \partial_x^2 u.$$

This parabolic term acts as a filter that eliminates most strongly the shortest wavelengths. Effectively this results in a "smoothing over" of the developing discontinuity, ensuring that a true discontinuity never develops. Physically speaking, this is equivalent to adding an artificial viscosity, hence the name. Artificial viscosity is by far less algorithmically sophisticated than HRSC schemes that rely on solutions to the Riemann problem, and will produce a less reliable solution in general, as the filtering acts to violate mass, momentum, and energy conservation.

18.4 Method of Characteristics for a Non-Homogeneous PDE, with Shocks: the Inviscid Burgers' Equation

Based on/for further reading: Leon van Dommelen's online lecture notes `https://www.eng.fsu.edu/~dommelen/pdes/style_a/burgers.html`, Stanford MATH 220a online handout `http://web.stanford.edu/class/math220a/handouts/firstorder.pdf`, and Scott Sarra's lecture notes `https://services.math.duke.edu/education/joma/sarra/sarra3.html`.

When we take the limit of Euler's equations as $c_s \to 0$ and $\pi = 0$, we get the inviscid Burgers' equation:

$$\partial_t u + u\partial_x u = 0,$$

which is one of the simplest *nonlinear* time-dependent PDEs that one might construct. As a result, it is widely studied by mathematicians and is used in other fields to better understand nonlinear gas flows (as might arise from Euler's equations) and even *traffic* flows. It has a nasty habit of generating hydrodynamic shocks, or in the case of traffic flows, traffic accidents.

Let's analyze this PDE under the initial conditions

$$u(x,0) = \phi(x)$$

using the Method of Characteristics.

Recall that the Method of Characteristics can be applied to PDEs of the form

$$a(x,t)\partial_x u + b(x,t)\partial_t u + c(x,t)u = 0,$$

so that

$$\begin{aligned} \frac{dx}{ds}\partial_x u + \frac{dt}{ds}\partial_t u + cu &= 0 \\ \implies \frac{du\left(x(s),t(s)\right)}{ds} &= -cu, \end{aligned}$$

if s is defined according to the characteristic equations, which for this case are given by

$$\begin{aligned} \frac{dx}{ds} &= a = u \\ \frac{dt}{ds} &= b = 1 \\ \frac{du}{ds} &= -cu = 0. \end{aligned}$$

How do we solve the top equation, since in general u depends on x and t, which each depend on s! The trick is to look at the third equation, which implies that u *is constant along characteristics*, so if we write $u(x,t) = u(x(s),t(s))$, u will remain constant if we only vary s. Then we can rewrite the top ODE as:

$$\frac{dx}{ds} = a = u(s) = u(s=0)$$

the solution is

$$\begin{aligned} x(s) &= u(s=0)s + c_1 \\ t(s) &= s + c_2 \\ u(s) &= c_3 \end{aligned}$$

Applying the initial conditions $x(s=0) = x_0$ and $t(s=0) = 0$, we get

$$\begin{aligned} x(s) &= u(s=0)s + x_0 \\ t(s) &= s. \end{aligned}$$

So effectively we have found that u is constant along s, and that $t = s$. This means that as we trace along a characteristic curve, u remains constant with respect to s, while the characteristic curve parameter s grows linearly with t.

Next notice that $u(s=0) = \phi(x_0)$, so

$$\begin{aligned} x(s) &= \phi(x_0)s + x_0 \\ t(s) &= s \\ \implies x &= \phi(x_0)t + x_0 \\ \implies x_0 &= x - \phi(x_0)t. \end{aligned}$$

We've shown that u is constant along characteristic curves: $u(s) = c_3 = \phi(x_0)$, so

$$u(s) = u(s=0) = u(t=0) = \phi(x_0) = \phi(x - \phi(x_0)t),$$

which is an implicit solution, which could be found using a numerical root solver, for example.

Question: If $t = s$, doesn't $u'(s) = 0$ mean that $\partial_s u = \partial_t u = 0$? The key to understanding why this is *not* the case is by remembering that x and t are *both* functions of s. So the correct way of writing $\partial_s u$ is as follows

$$\partial_s u = \partial_s u(x(s),t(s)) = \frac{d}{ds}u = x'(s)\partial_x u + t'(s)\partial_t u \neq \partial_t u$$

consistent with our earlier definition. Thus as we march our solution u forward in time, it remains constant *only along* a characteristic curve (parameterized by s), which will have a slope that depends on our choice of $x(s=0) = x_0$:

$$\frac{dx}{dt} = \frac{\frac{dx}{ds}}{\frac{dt}{ds}} = u(s=0) = \phi(x_0).$$

So in other words, as s increases, both t *and* x will vary, so $u'(s) = 0$ does not imply that $\partial_t u = 0$ even if $t = s$.

Let's analyze a simple initial data case: $\phi(x) = \cos(x)$. The slope of characteristics at $s = t = 0$ is given by

$$\frac{dx}{dt} = \frac{\frac{dx}{ds}}{\frac{dt}{ds}} = u(s=0) = \phi(x_0) = \cos(x),$$

so notice that the speed of the characteristic curve is larger at $x = 0$ and zero speed at $x = \pi/2$. We will find that the faster wave will cause the solution to steepen with time, until finally the characteristics cross, leading to a shock!

18.5 Systematic Approach for Applying Method of Characteristics to Coupled, First-Order PDEs

Recall our original approach for finding characteristics for Euler's equations in the barotropic limit required clever variable definitions and substitutions. The goal of this section is to introduce a more systematic approach to find characteristic velocities for coupled and even nonlinear PDEs.

You may recall from your class on ordinary differential equations that the basic strategy in reducing a high-order PDE or ODE to a set of coupled differential equations is to *define new variables*.

For example, consider the ODE

$$u'' + \frac{1}{2}u' + u = 0.$$

Rewrite this as a set of coupled, *first-order* ODEs in the form

$$\begin{aligned} x_1' &= f(x_1, x_2) \\ x_2' &= g(x_1, x_2). \end{aligned}$$

Solution:
Define $x_1 = u$ and $x_2 = x_1' = u'$. Then

$$\begin{aligned} x_1' &= u' = x_2 \\ x_2' &= u'' \end{aligned}$$

But according to the ODE,

$$u'' = x_2' = -\frac{1}{2}u' - u = -\frac{1}{2}x_2 - x_1.$$

Notice we rewrote u' as x_2 and not x_1' so that our ODE would satisfy the form required in the problem statement:

$$\begin{aligned} x_1' &= f(x_1, x_2) \\ x_2' &= g(x_1, x_2). \end{aligned}$$

Thus we have found our pair of coupled, first-order ODEs:

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= -\frac{1}{2}x_2 - x_1. \end{aligned}$$

Then for an nth-order ODE/PDE, you will need to define x_1, $x_2 = x_1'$, ..., $x_n = x_{n-1}'$, and follow the same basic strategy. For example, if we are given

$$u''' + 3u'' + 2u' + u = 0,$$

we define

$$\begin{aligned} x_1' &= x_2 = u' \\ x_2' &= x_3 = u'' \\ \implies x_3' &= u''' = -(3u'' + 2u' + u) = -(3x_3 + 2x_2 + x_1). \end{aligned}$$

The basic approach for finding the characteristics of PDEs with functions of independent variables x and t is as follows. First write the PDEs in the form

$$\partial_t \boldsymbol{U} + \boldsymbol{A}\partial_x \boldsymbol{U} = \boldsymbol{0},$$

where $\boldsymbol{U}$ is the solution vector. Analogous to the Method of Characteristics for a single equation, $\boldsymbol{A}$ contains information about the slope of the characteristic curves dx/dt. In particular, the eigenvalues of $\boldsymbol{A}$ correspond to the characteristic curve slopes. Given the units on these slopes are distance per unit time, we generally call the slopes the *characteristic speeds*.

Let's next apply this strategy for Euler's equations in one spatial dimension in the barotropic limit:

$$\begin{aligned} \partial_t \rho + \partial_x[\rho u] &= 0 \\ \partial_t[\rho u] + \partial_x[\rho u^2 + P] &= 0. \end{aligned}$$

The top equation can be immediately written in the needed form, by just expanding the x-derivative term:

$$\partial_t \rho + u\partial_x \rho + \rho \partial_x u = 0.$$

The momentum equation on the other hand is clearly not in the form we need if we use ρ and u as the solution variables, and this particularly obvious if you consider that it contains time derivatives of both ρ and u. So let's expand the second equation and insert the first equation for the $\partial_t \rho$ term:

$$\begin{aligned} 0 &= \partial_t[\rho u] + \partial_x[\rho u^2 + P] \\ &= u\partial_t \rho + \rho \partial_t u + \partial_x[\rho u^2 + P] \\ &= -u\partial_x[\rho u] + \rho \partial_t u + \partial_x[\rho u^2 + P] \\ &= -u^2 \partial_x \rho - \rho u \partial_x u + \rho \partial_t u + u^2 \partial_x \rho + 2\rho u \partial_x u + \partial_x P \\ &= \underbrace{-u^2 \partial_x \rho} \overbrace{-\rho u \partial_x u} + \rho \partial_t u \underbrace{+u^2 \partial_x \rho} \overbrace{+2\rho u \partial_x u} + \partial_x P. \end{aligned}$$

Canceling/combining terms with the underbraces and overbraces, we obtain

$$0 = \rho \partial_t u + \rho u \partial_x u + \partial_x P.$$

Again, we're dealing with barotropic fluids, where $P(\rho)$, so

$$\partial_x P = \frac{dP}{d\rho} \partial_x \rho = c_s^2 \partial_x \rho.$$

Substituting this expression and dividing through by ρ, we then have the momentum equation in the form we need (i.e., a row of the matrix equation $\partial_t \boldsymbol{U} + \boldsymbol{A}\partial_x \boldsymbol{U} = \boldsymbol{0}$):

$$\partial_t u + u\partial_x u + \frac{c_s^2}{\rho} \partial_x \rho = 0.$$

In summary, we have written the continuity and momentum equations in the form

$$\begin{aligned} \partial_t \rho + u\partial_x \rho + \rho \partial_x u &= 0 \\ \partial_t u + u\partial_x u + \frac{c_s^2}{\rho} \partial_x \rho &= 0, \end{aligned}$$

which can be rewritten in the required matrix form as

$$\partial_t \begin{pmatrix} \rho \\ u \end{pmatrix} + \begin{pmatrix} u & \rho \\ \frac{c_s^2}{\rho} & u \end{pmatrix} \partial_x \begin{pmatrix} \rho \\ u \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Again, the characteristic speeds are given by the eigenvalues of the 2×2 matrix, which can be quickly computed:

$$\begin{aligned} 0 &= \det(\boldsymbol{A} - \lambda \boldsymbol{I}) \\ &= \begin{vmatrix} u - \lambda & \rho \\ \frac{c_s^2}{\rho} & u - \lambda \end{vmatrix} \\ &= (u - \lambda)^2 - c_s^2 \\ \implies 0 &= \lambda^2 - 2\lambda u + (u^2 - c_s^2). \end{aligned}$$

The eigenvalues can be immediately computed from the quadratic equation:

$$\begin{aligned} \lambda &= \frac{2u \pm \sqrt{4u^2 - 4(u^2 - c_s^2)}}{2} \\ &= u \pm \sqrt{c_s^2} \\ &= u \pm c_s, \end{aligned}$$

consistent with what we found previously!

18.6 Method of Characteristics for a Non-Homogeneous PDE

Based on: Tobias von Petersdorff's MATH 462 Lecture Notes, Spring 2012: Section 1.3 Method of Characteristics example `http://terpconnect.umd.edu/~petersd/462/charact.pdf`.

Using the Method of Characteristics, solve the PDE

$$\partial_t u + x\partial_x u + u = 3x, \qquad u(x,0) = \sin(x).$$

Recall that in our original formulation of the Method of Characteristics, we defined the characteristic equations

$$\begin{aligned} \frac{dx}{ds} &= a \\ \frac{dt}{ds} &= b \end{aligned}$$

so that we can rewrite the generic first-order PDE

$$a(x,t)\partial_x u + b(x,t)\partial_t u + c(x,t)u = 0$$

in the form

$$\frac{dx}{ds}\partial_x u + \frac{dt}{ds}\partial_t u + c(x,t)u = \frac{du(x(s),t(s))}{ds} + c(x(s),t(s))u = 0.$$

Notice that for the PDE of this example, we have a *function of* x on the right-hand side; otherwise it matches the pattern perfectly. Thus the equation for $u(s)$ is given by

$$\frac{du(x(s),t(s))}{ds} + c(x(s),t(s))u = 3x,$$

where $c = 1$. Thus instead of solving the separable ODE $u'(s) + cu = 0$, we must instead solve the ODE

$$\frac{du(x(s),t(s))}{ds} + u = 3x(s),$$

which can be solved so long as we can find an explicit formula for $x(s)$. To that end, the characteristic equations and their general solutions are given by

$$\begin{aligned} \frac{dt}{ds} &= 1 \implies t = s + c_1 \\ \frac{dx}{ds} &= x(s) \implies d\ln(x) = ds \implies x(s) = ke^s. \end{aligned}$$

Next we apply the initial conditions $t(s=0) = 0$, $x(s=0) = x_0$ to get

$$\begin{aligned} t(s=0) = 0 + c_1 = 0 &\implies t = s \\ x(s=0) = ke^0 = x_0 &\implies x = x_0 e^s. \end{aligned}$$

Thus the ODE for $u(s)$ is then given by

$$\frac{du(x(s),t(s))}{ds} + u = 3x(s) = 3x_0 e^s.$$

This is easily solved using the Method of Integrating factors. Recall that this method is used to solve generic, first-order, nonhomogeneous linear ODEs of the form:

$$y'(t) + p(t)y(t) = g(t).$$

We define the *integrating factor* as

$$\mu(t) = \exp\left(\int p(t')dt'\right).$$

Knowing this, the solution is given by

$$y(t) = \frac{1}{\mu(t)} \left(\int \mu(t')g(t')dt' \right).$$

So in this case, $p(s) = 1$ and $g(s) = 3x(s) = 3x_0 e^{-s}$, so

$$\mu(s) = \exp\left(\int ds' \right) = Ke^s,$$

and

$$\begin{aligned} u(s) &= \frac{1}{Ke^s} \left(\int K \exp(s') \left[3x_0 \exp(s') \right] ds' \right) \\ &= \frac{1}{Ke^s} \left(\int 3x_0 K \exp(2s') ds' \right) \\ &= \frac{1}{Ke^s} \left(\frac{3x_0 K}{2} e^{2s} + C \right) \\ &= \frac{3x_0}{2} e^s + ce^{-s}. \end{aligned}$$

Applying our initial condition of $u(x, 0) = \sin(x)$, we get

$$\sin(x(s = 0)) = \sin(x_0) = \frac{3x_0}{2} + c \implies c = \sin(x_0) - \frac{3x_0}{2}$$

Thus our solution along characteristic curves $u(s)$ is given by

$$u(s) = \frac{3x_0}{2} e^s + \left(\sin(x_0) - \frac{3x_0}{2} \right) e^{-s}.$$

But we want the solution $u(x, t)$, which requires that we first invert $x(s)$ and $t(s)$ to obtain expressions for s and x_0 in terms of x and t:

$$\begin{aligned} s &= t \\ x &= x_0 e^s \implies x_0 = xe^{-s} = xe^{-t}. \end{aligned}$$

Then $u(x, t)$ is given by

$$\begin{aligned} u(x, t) &= \frac{3xe^{-t}}{2} e^t + \left(\sin\left(xe^{-t}\right) - \frac{3xe^{-t}}{2} \right) e^{-t} \\ &= \frac{3x}{2} + e^{-t} \sin\left(xe^{-t}\right) - \frac{3xe^{-2t}}{2}. \end{aligned}$$

Let's confirm this solution by plugging back in to the original expression:

$$\begin{aligned} \partial_t u &= -e^{-t} \sin\left(xe^{-t}\right) - e^{-t}(xe^{-t}) \cos\left(xe^{-t}\right) - (-2)\frac{3xe^{-2t}}{2} \\ &= -e^{-t} \sin\left(xe^{-t}\right) - xe^{-2t} \cos\left(xe^{-t}\right) + 3xe^{-2t} \\ x\partial_x u &= x \left[\frac{3}{2} + e^{-t}(e^{-t}) \cos\left(xe^{-t}\right) - \frac{3e^{-2t}}{2} \right] \\ &= \frac{3x}{2} + xe^{-2t} \cos\left(xe^{-t}\right) - \frac{3xe^{-2t}}{2} \\ u &= \frac{3x}{2} + e^{-t} \sin\left(xe^{-t}\right) - \frac{3xe^{-2t}}{2} \end{aligned}$$

Let's add these three equations and identify common terms:

$$\begin{aligned}
\partial_t u &= \overbrace{-e^{-t}\sin\left(xe^{-t}\right)}\underbrace{-xe^{-2t}\cos\left(xe^{-t}\right)}\underbrace{+3xe^{-2t}} \\
x\partial_x u &= \frac{3x}{2}\underbrace{+xe^{-2t}\cos\left(xe^{-t}\right)}\underbrace{-\frac{3xe^{-2t}}{2}} \\
u &= \frac{3x}{2}\overbrace{+e^{-t}\sin\left(xe^{-t}\right)}+\underbrace{-\frac{3xe^{-2t}}{2}} \\
\implies \partial_t u + x\partial_x u + u &= \frac{3x}{2}+\frac{3x}{2} = 3x.
\end{aligned}$$

Adding the three expressions for u and its derivatives indeed yields $3x$. Next we check the initial condition

$$u(x,0) = \frac{3x}{2} + \sin(x) - \frac{3x}{2} = \sin(x).$$

In conclusion, I hope you agree the complexity of the solution for $u(x,t)$ is rather remarkable, demonstrating the beauty and power of the Method of Characteristics.

MATH 522, Spring 2019 Notes

Chapter 19: Final Exam Review Problems

The following are just a sampling of the types of problems you may encounter on the Final Exam. In preparing for the Final Exam, you will need to review and understand all material in your homework assignments and midterms. In addition, you will need to review your notes on the Method of Characteristics, as there will definitely be at least one problem on the final in which you will need to use this method.

19.1 Number Storage and Arithmetic on the Computer

In addition to Midterm Exams, homeworks, and notes, you may find the following exercises useful.

Double Precision Arithmetic Exercises

When the following expressions are evaluated by the computer, **to how many significant decimal digits will the numerical result agree with the exact result?** Your answer will consist of a single integer (∞ is an acceptable answer). If the integer is finite and nonzero, your answer will be accepted if it is within 1 decimal digit of the exact answer.
We use computer scientific notation, such that, e.g., `5.63e22` $\equiv 5.63 \times 10^{22}$. For the purpose of this problem, apply IEEE 754 standard-compliant **double precision** arithmetic, assuming all given *integers* represented in integer or floating point format between -2^{53} and 2^{53} (i.e., integers between $\approx -9 \times 10^{15}$ and $\approx 9 \times 10^{15}$ inclusive) are **exact** (for example, `2.01e2` is exact), as well as powers of $\frac{1}{2}$. Otherwise, assume that in double precision the number is only known to 16 significant digits, and that machine epsilon is `4e-16`.
Answers will receive full credit so long as they are correct within 1 decimal digit. You will not receive credit for showing work.

1. `7 + 2.4e-230/5e280`
2. `7 + 2.4e230/5e280`
3. `1/100`
4. `log10(2e-230) - 2e-200`
5. `(3.1415-3.14)/(3.1415-3.14)`
6. `1/1024`
7. `3.2e-200/5e-202 + 2`

Solutions

1. `7 + 2.4e-230/5e280`: `7` is exactly representable, and the expression `2.4e-230/5e280` will yield an underflow in double precision, giving a double precision result of `7` *exactly*. The exact result is `7` plus a perturbation at the 511th digit. Thus the double precision and exact results agree to about `511` significant digits.
2. `7 + 2.4e230/5e280`: `7` is exactly representable, and the expression `2.4e230/5e280` will yield a number that is of order `1e-51`. Recall that machine epsilon is defined as the largest ϵ such that $1 + \epsilon = 1$. In double precision machine epsilon is approximately `4e-16`, so `7+1e-51 ~ 7 (1 + 1e-52)` is `7` exactly. Thus the double precision and exact results agree to about 51 significant digits.
3. `1/100`: The division is not a power of `2`, so the answer is 16 significant digits of agreement between double precision and the exact answer.
4. `log10(2e-230) - 2e-200`: `log10(2e-230)` is computed via Taylor series, thus will yield a double precision result that matches the exact result to 16 significant digits. The `-2e-200` will not affect the result. The answer is 16.
5. `(3.1415-3.14)/(3.1415-3.14)`: The numerator and denominator will evaluate to exactly the same bits, despite being consistent with the exact result to 16 significant digits. Thus the answer in double precision is 1 exactly, matching the exact result to ∞ significant digits.
6. `1/1024`: $1024 = 2^{10}$, so the expression is 2^{-10}, an exactly representable number in double precision that matches the exact result to ∞ significant digits.
7. `3.2e-200/5e-202 + 2`: `3.2e-200/5e-202` will be evaluated to a precision of 16 significant digits and is of order `1e2`. Thus `2` will perturb it at the second significant digit. The result is *not* a number exactly representable in double precision, so will be consistent with the exact solution to 16 significant digits.

19.2 Determining the Scale of the Solution

Be able to combine estimates in "word problem" format to estimate the scale of a solution to a problem. Be sure to study relevant course notes and homeworks. Such problems on the Final Exam will not require that you base any estimates on your own experiences. Rather, all numbers required to solve problems like these will be provided; your task will be to synthesize any given numbers into a final answer.

19.3 von Neumann review

This problem is designed to test your familiarity with the mathematical reasoning that goes into the final stages of a von Neumann stability analysis. Find all ranges of w where the following inequality

$$-5 \leq (w^2 - 6w + 2)\sin^4(k\Delta x) \leq 0$$

is satisfied. Each range must be written in the form

$$\alpha \leq w \leq \beta,$$

where α and β are constants that you must find.

As for the functions of $k\Delta x$, you are to assume that this inequality was generated at the end of a von Neumann stability analysis. Considering *all* possible values of $k\Delta x$ resolvable on your grid—not just one or two values—what single value of $k\Delta x$ is applicable to the von Neumann stability analysis? Why?

Assume the grid has infinite extent: $-\infty < x < \infty$. Write all steps in your reasoning; if there are gaps in your argument, you will lose points.

Negative values of w are permitted, but imaginary values are not.

We know that the range of $k\Delta x$ resolvable on our grid is $0 \leq k\Delta x \leq \pi$. We also know that $\sin(k\Delta x)$ ranges from zero to 1 in this range, with a maximum of 1 at $(k\Delta x) = \pi/2$. Of course, $\sin(k\Delta x) = \sin^4(k\Delta x) = 1$ at

$(k\Delta x) = \pi/2$. Since $k\Delta x = \pi/2$ yields the most restrictive inequality, we must choose the value $(k\Delta x) = \pi/2$ when doing our von Neumann stability analysis.

Then we have

$$-5 \leq (w^2 - 6w + 2) \leq 0$$

Define $f(w) = (w^2 - 6w + 2)$. $f''(w) = 2 > 0$, so $f(w)$ is concave up, with one minimum. The minimum occurs at $f'(w) = 0 = 2w - 6 \implies w = 3$. Then $f(3) = 9 - 18 + 2 < -5$, which is below the lower bound of our range for $f(w)$. However, this is the minimum, so there will be two ranges of w that satisfy the inequality. To find these ranges, we simply find the values of w where $f(w)$ reaches the bounds on the inequality: 0 and -5:

$$f(w) = (w^2 - 6w + 2) = 0 \implies w = \frac{6 \pm \sqrt{36 - 8}}{2} = 3 \pm \sqrt{7}.$$

$$f(w) = (w^2 - 6w + 2) = -5 \implies (w^2 - 6w + 7) = 0 \implies w = \frac{6 \pm \sqrt{36 - 28}}{2} = 3 \pm \sqrt{2}.$$

Going from left to right on the plot of $f(w)$, $f(w) = 0$ at $w = 3 - \sqrt{7}$, then $f(w) = -5$ at $w = 3 - \sqrt{2}$, and then $f(w) = 0$ again at $w = 3 + \sqrt{7}$. Thus our first range for w will be at $3 - \sqrt{7} \leq w \leq 3 - \sqrt{2}$. Similarly, our second range for w will be at $3 + \sqrt{2} \leq w \leq 3 + \sqrt{7}$.

19.4 Finite difference coefficient review

Compute the second-order-accurate finite difference coefficients for an off-centered stencil on an evenly-spaced grid. I.e., find a, b, and c to make the following expression second-order accurate in Δx:

$$f'(x) = \frac{af(x + 2\Delta x) + bf(x + \Delta x) + cf(x)}{\Delta x}$$

Solution Strategy #1:

First we note that, for this to be accurate to second order in Δx, the derivative operator must be able to compute derivatives of polynomials up to second degree with zero error. As we will see, this fact makes it straightforward to compute a, b, and c, which we will find to be given by:

$$\begin{aligned} a &= -1/2 \\ b &= 2 \\ c &= -3/2 \end{aligned}$$

- Step 1: If $f(x) = C$, $f'(x) = 0$:

$$\begin{aligned} 0 &= \frac{aC + bC + cC}{\Delta x} \\ 0 &= a + b + c \end{aligned}$$

- Step 2: If $f(x) = Bx + C$, $f'(x) = B$:

$$\begin{aligned} B &= \frac{a[B(x + 2\Delta x) + C] + b[B(x + \Delta x) + C] + c[Bx + C]}{\Delta x} \\ B\Delta x &= C(a + b + c) + Bx(a + b + c) + B(2a\Delta x + b\Delta x) \\ B\Delta x &= B(2a\Delta x + b\Delta x) \\ \Delta x &= (2a\Delta x + b\Delta x) \\ 1 &= (2a + b) \\ 2a + b &= 1 \end{aligned}$$

- Step 3: If $f(x) = Ax^2 + Bx + C$, $f'(x) = 2Ax + B$:

$$2Ax + B = \frac{a[A(x + 2\Delta x)^2 + B(x + 2\Delta x) + C] + b[A(x + \Delta x) + B(x + \Delta x) + C] + c[Ax^2 + Bx + C]}{\Delta x}$$

This equation must hold for all A, B, and C, so it must also hold for $B = C = 0$. Alternatively, we can simply group together the terms involving B and C. The result will be the previous equalities, which cancel out of the equation.

Thus we are left with only terms involving A:

$$\begin{aligned} 2Ax &= \frac{a[A(x + 2\Delta x)^2] + b[A(x + \Delta x)^2] + c[Ax^2]}{\Delta x} \\ 2Ax\Delta x &= a[A(x + 2\Delta x)^2] + b[A(x + \Delta x)^2] + cAx^2 \\ 2x\Delta x &= a(x + 2\Delta x)^2 + b(x + \Delta x)^2 + cx^2 \end{aligned}$$

This expression can be greatly simplified by realizing it must hold for *all* x and Δx. So it is our task then to come up with an x and Δx that generates a unique relationship between a, b, and c. Let's try $x = \Delta x = 1$. Then we get

$$2 = 9a + 4b + c$$

$$\begin{aligned} 0 &= a + b + c \\ \Longrightarrow c &= -a - b \\ 2 &= 9a + 4b + (-a - b) \\ 2 &= 8a + 3b \\ \Longrightarrow b &= (2 - 8a)/3 \\ 2a + (2 - 8a)/3 &= 1 \\ -2/3a + 2/3 &= 1 \\ -2/3a &= 1/3 \\ a &= -1/2 \\ b &= (2 - 8a)/3 \\ b &= (2 + 4)/3 = 2 \\ 0 &= a + b + c \\ 0 &= -1/2 + 2 + c \\ 0 &= 3/2 + c \\ c &= -3/2 \end{aligned}$$

Solution Strategy #2:
Application of Taylor series will also yield the answer, and with even less algebra:

$$\begin{aligned} f(x + 2\Delta x) &= f(x) + (2\Delta x)\frac{f'(x)}{1!} + (2\Delta x)^2\frac{f''(x)}{2!} + (2\Delta x)^3\frac{f'''(x)}{3!} + ... \\ \Longrightarrow f'(x) &= \frac{f(x + 2\Delta x) - f(x)}{2\Delta x} + (2\Delta x)\frac{f''(x)}{2!} + (2\Delta x)^2\frac{f'''(x)}{3!} + ... \\ f(x + \Delta x) &= f(x) + (\Delta x)\frac{f'(x)}{1!} + (\Delta x)^2\frac{f''(x)}{2!} + (\Delta x)^3\frac{f'''(x)}{3!} + ... \\ \Longrightarrow f'(x) &= \frac{f(x + \Delta x) - f(x)}{\Delta x} + (\Delta x)\frac{f''(x)}{2!} + (\Delta x)^2\frac{f'''(x)}{3!} + ... \end{aligned}$$

Taking the second equation and subtracting twice the bottom equation, we get:

$$\begin{aligned} f'(x) - 2f'(x) &= \frac{f(x+2\Delta x) - f(x)}{2\Delta x} - 2\frac{f(x+\Delta x) - f(x)}{\Delta x} + \mathcal{O}(\Delta x^2) \\ &= \frac{1}{\Delta x}\left(\frac{1}{2}f(x+2\Delta x) - 2f(x+\Delta x) + \frac{3}{2}f(x)\right) + \mathcal{O}(\Delta x^2) \\ \implies f'(x) &= \frac{1}{\Delta x}\left(-\frac{1}{2}f(x+2\Delta x) + 2f(x+\Delta x) - \frac{3}{2}f(x)\right) + \mathcal{O}(\Delta x^2), \end{aligned}$$

which is consistent with our result using **Solution Strategy #1**.

19.5 Eigenvalues and Eigenvectors review

Being able to compute Eigenvalues/Eigenvectors is essential when solving coupled sets of PDEs on the computer, whether it be in the context of a von Neumann stability analysis to determine the maximum magnitude growth factor, or finding the characteristic speeds. Regarding the latter, often we need to rewrite the PDE in *conservative* form so that we can employ a high-resolution shock-capturing scheme. Typically writing in conservative form requires that we rewrite the equations in terms of "conservative variables", which themselves are expressions in terms of the given variables. Eigenvectors can be used to find the conservative variables.

Basic Review:

It turns out that for every $n \times n$ matrix $\boldsymbol{A}$, there exists some set of vectors $\boldsymbol{x}$ such that $\boldsymbol{A}$ transforms $\boldsymbol{x}$ into *a multiple of itself.* I.e.,

$$\boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{x}.$$

In this case, $\boldsymbol{x}$ is known as an **eigenvector** of $\boldsymbol{A}$ and λ an **eigenvalue**. Eigenvectors and eigenvalues of $\boldsymbol{A}$ are, therefore, a reflection of some deep property of $\boldsymbol{A}$; the name "eigen" is German, meaning "self" or "own", So an eigenvector of $\boldsymbol{A}$ reflects $\boldsymbol{A}$'s "own" vector. An $n \times n$ matrix $\boldsymbol{A}$ will have n eigenvalues and n eigenvectors.

So how do we compute eigenvalues and eigenvectors? Notice first that

$$\boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{x} \implies \boldsymbol{A}(c\boldsymbol{x}) = \lambda(c\boldsymbol{x}),$$

for any constant c! Thus eigenvectors can only be known up to a constant. In other words, a multiple of an eigenvector for $\boldsymbol{A}$ is also an eigenvector of $\boldsymbol{A}$.

Let's rewrite the definition for eigenvalues and eigenvectors:

$$\boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{x} \implies (\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{x} = \boldsymbol{0},$$

where $\boldsymbol{I}$ is the $n \times n$ identity matrix (with 1's along the diagonal and zeroes everywhere else). Again, this equation will be satisfied for an infinite number of $\boldsymbol{x}$, which are the same as the eigenvector $\boldsymbol{x}$, but multiplied by some constant. You'll recall that $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ has a unique solution $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$ if and only if $\det \boldsymbol{A} \neq 0$.

In this case, $\boldsymbol{b} = \boldsymbol{0}$, so the only way we could have a *unique* solution would be if $\det(\boldsymbol{A}) = 0$ and $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{0} = \boldsymbol{0}$. But we are not interested in the solution $\boldsymbol{x} = \boldsymbol{0}$, and in fact *we are not looking for a unique solution* $\boldsymbol{x}$!

So the only way $(\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{x} = \boldsymbol{0}$ will have nonzero solutions for $\boldsymbol{x}$ is if the determinant of $(\boldsymbol{A} - \lambda \boldsymbol{I})$ *is zero*:

$$\det(\boldsymbol{A} - \lambda \boldsymbol{I}) = 0.$$

This is called the *characteristic equation* for $\boldsymbol{A}$, and we'll see it is the same as the characteristic equation we've used when solving high-order ODEs with constant coefficients.

Then the **nonzero** $\boldsymbol{x}$ satisfying

$$(\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{x} = \boldsymbol{0}$$

for a given λ is the eigenvector corresponding to that λ. Remember that for an $n \times n$ matrix $\boldsymbol{A}$ there will be n eigenvalues, with one eigenvector corresponding to each eigenvalue.

Problem that will be on the Final. There will be one additional eigenvalue problem (not necessarily like this one) on the Final.

Find the eigenvalues of the following 3×3 matrix:

Matrix:

$$\begin{pmatrix} -10 & 9 & 9 \\ 0 & -2 & -2 \\ -6 & 7 & 7 \end{pmatrix}$$

Solution:
To find the eigenvalues, we are to solve the equation

$$\det\left(\boldsymbol{A} - \lambda \boldsymbol{I}\right) = 0.$$

We get

$$\begin{aligned} 0 &= \begin{vmatrix} -10-\lambda & 9 & 9 \\ 0 & -2-\lambda & -2 \\ -6 & 7 & 7-\lambda \end{vmatrix} \\ &= (-10-\lambda)\left[(-2-\lambda)(7-\lambda)+14\right] - 9(0-12) + 9(0-(-6)(-2-\lambda)) \\ &= (-10-\lambda)\left[(-2-\lambda)(7-\lambda)+14\right] + 9(12) - 9(12) - 54\lambda \\ &= (-10-\lambda)\left[-14-5\lambda+\lambda^2+14\right] - 54\lambda \\ &= (-10-\lambda)\left[-5\lambda+\lambda^2\right] - 54\lambda \\ &= \lambda(-10-\lambda)(-5+\lambda) - 54\lambda \\ &= \lambda(50-5\lambda-\lambda^2) - 54\lambda \\ &= -\lambda^3 - 5\lambda^2 - 4\lambda \\ &= -\lambda(\lambda+4)(\lambda+1) \\ \implies \lambda &= \{0,-4,-1\} \end{aligned}$$

Summary:
Characteristic polynomial:

$$-\lambda^3 - 5\lambda^2 - 4\lambda$$

Eigenvalues:

$$\{-4,-1,0\}$$

19.6 Method of Characteristics review

Method of Characteristics: Example #1

From: `http://exampleproblems.com/wiki/index.php/PDEMOC4`:
Using the Method of Characteristics, solve the following PDE:

$$u_x + 2u_t = u^2, \;\; u(x,0) = f(x).$$

$$
\begin{aligned}
x &= s + x_0 \\
t &= 2s \\
u &= -1/(s - 1/f(x_0)) \\
&= -1/[t/2 - 1/f(x - t/2)] \\
&= -f(x - t/2)/[t/2f(x - t/2) - 1]
\end{aligned}
$$

Now we have

$$
\begin{aligned}
u(t = 0) &= -1/(c_3) = f(x_0) \\
\implies c_3 &= -1/f(x_0)
\end{aligned}
$$

At $t = 0$, $s = 0$ and $u(x, 0) = f(x)$, so

$$
\begin{aligned}
\frac{dt}{ds} &= 2 \implies t = 2s + c_1 \implies t = 2s \\
\frac{dx}{ds} &= 1 \implies x = s + c_2 = s + x_0 \quad [x(s = 0) = x_0] \\
\frac{du}{ds} &= u^2 \implies \frac{du}{u^2} = ds \implies -1/u = s + c_3 \implies u = -1/(c_3 + s)
\end{aligned}
$$

The characteristic equations are written

Solution

Method of Characteristics: Example #2

Based on: Alice Quillen's AST242 lecture notes, **"Method of Characteristics"** section, starting on Eq 36 `http://astro.pas.rochester.edu/~aquillen/ast242/lecturenotes2.pdf`.
Using the Method of Characteristics, solve the following PDE:

$$
\begin{aligned}
2xt\partial_x u + \partial_t u &= u \\
u(x, 0) &= x
\end{aligned}
$$

Solution

The characteristic equations are

$$\begin{aligned}\frac{dt}{ds} &= 1 \implies t = s + c_1 \\ \frac{dx}{ds} &= 2x(s)t(s) \implies d\ln(x) = 2t(s)ds\end{aligned}$$

Since $t(s=0) = 0$, we get $c_1 = 0$ and $t = s$. Then the second characteristic equation yields

$$\begin{aligned}d\ln(x) &= 2sds \\ \implies \ln(x) &= s^2 + c_2 \\ \implies x &= Ke^{s^2}\end{aligned}$$

At $t = s = 0$, $x = x_0$, so $K = x_0$ and we get

$$x = x_0 e^{s^2} = x_0 e^{t^2}$$

We now solve for x_0 in terms of x and t:

$$x_0(x,t) = xe^{-t^2}$$

Next we return to the ODE for u:

$$\frac{du}{ds} = u,$$

which has solution $u = K_2 e^s$. Our initial condition gives $u(x, t=0) = u(s=0) = x = x_0$, so $K_2 = x_0$, and the solution for u becomes

$$\begin{aligned}u &= x_0 e^s \\ &= (xe^{-t^2})e^t \\ &= xe^{-t^2+t}\end{aligned}$$

Along each characteristic, u exponentially increases with s. We can easily draw characteristic curves by writing $t(x)$ via inversion of the equation $x_0(x,t) = xe^{-t^2}$:

$$t(x) = \sqrt{\ln(x/x_0)}$$

So the characteristic curves each look logarithmic on the $x - t$ plane with increasing x, as shown in Fig. 19.1.

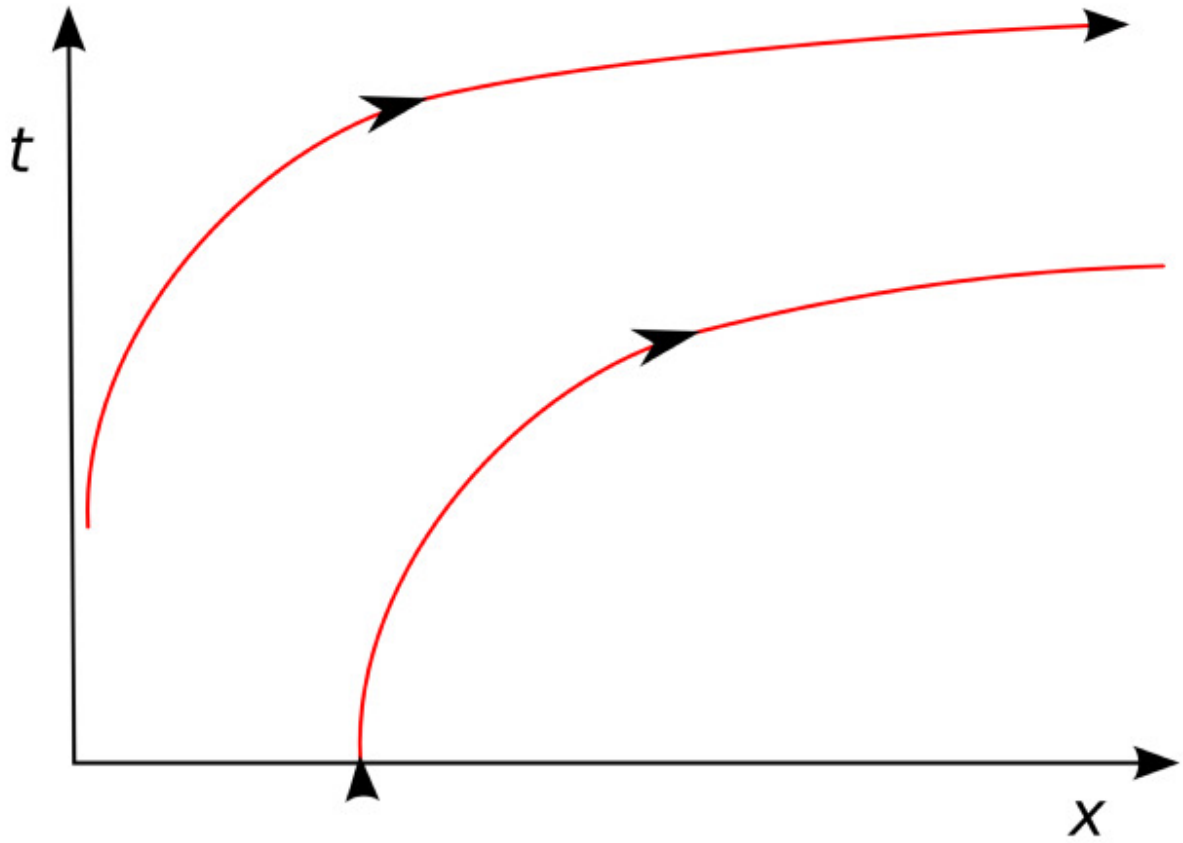

Figure 19.1: Characteristic curves for $t(x) = \sqrt{\ln(x/x_0)}$. From Figure 10 of Alice Quillen's AST242 lecture notes, **"Method of Characteristics"** section `http://astro.pas.rochester.edu/~aquillen/ast242/lecturenotes2.pdf`.