Anteckningar - MM5016

Sebastijan Babic

22 May 2025

Abstract

The purpose of this document is to provide a concise summary of the course MM5016 - Numerical Analysis I VT25. Some notation may be inconsistent.

Contents

1. Lecture - Introduction to the numerical analysis: practical examples in numerical analysis. Beginning of numerical integration

The main objective of this course is to learn how to solve mathematical formulations numerically. This is because many real-life problems that arise in applied mathematics especially do not have an analytical solution.

1.1. General notation

- Denote the length of any interval as Δx or h. This meaning if you have an interval $[a, b]$ the length of this entire interval would be precisely $b - a$.
- Denote the left point x-value as x_l, similarly the right point x-value as x_r.

1.2. Some simple rules

Left point rule: The left point rule states that

$$\int_a^b f(x)dx = \Delta x f(x_0) + \Delta x f(x_1) + ... + \Delta x f(x_n).$$

The name may come from the fact that we are starting at x_0, ie. the left-most point in the interval.

Error: This method has a global error of $O(h)$.

Right point rule: Similar to lef point rule we can start with index 1. Meaning that

$$\int_b^a f(x)dx = \Delta x f(x_1) + \Delta x f(x_2) + ... + \Delta x f(x_n + 1).$$

Error: Global error is $O(h)$.

Mid point rule: A combination of left point and right point rule is the mid point rule, here, we will instead use the formulation

$$\Delta x f(x_m)$$

Error: Global error is $O(h^2)$.

Note: These methods are often obsolete compared to the next method due to the error they produce in comparison to the Trapezoid rule.

Trapezoid rule: The trapezoid rule states that (if equidistant intervals)

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N} \Delta x \frac{f(x_N) - f(x_{N+1})}{2}$$

Alternatively, if not equidistant intervals then

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N} (x_{k+1} - x_k) \frac{f(x_k) - f(x_{k+1})}{2}$$

Do note that the method used most commonly while actually applying this method is the following:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2} \left(f(x_0) + 2 \sum_{k=1}^{N-1} f(x_k) + f(x_N) \right)$$

Error: The global error is $O(h^2)$.

2. Lecture - Numerical integration

2.1. Continuation of the trapezoid rule and it's errors

Say we are in the subinterval $[x_k, x_{k+1}]$, the error will be the area under or over the theoretical function $f(x)$. The approximate initegral on this subinterval is

$$T(\Delta x) = \Delta x \frac{f(x_k) + f(x_{k+1})}{2}$$

Where Δx or h is the width of the interval. The exact integral is

$$F(x_{k+1}) - F(x_k)$$

Using Taylor seris around x_k we get

$$F(x_{k+1}) = F(x_k) + F'(x_k)h + F'' \frac{x_k}{2} h^2 + O(h^3)$$

now using Taylor seris on $F(x_k)$ around x_{k+1} we get

$$F(x_k) = F(x_{k+1}) - F'(x_{k+1})h + F'' \frac{x_{k+1}}{2} h^2 + O(h^3)$$

Subtracting the Taylor expansions now gives by the mean value theorem

$$f'(x_k) - f'(x_{k+1}) = f''(\xi)h$$

where $\xi \in [x_k, x_{k+1}]$ meaning that the error is globally

$$O(h^3)$$

2.2. Simpson's rule

A method of higher order that utilizes second degree polynomials. Therefore, we need a minimum of three points to construct this approximation, ie $\{x_{k-1}, x_k, x_{k+1}\}$. We get

Simpson's rule:

$$\int_a^b f(x)dx = \frac{h}{3}\left(f(x_1) + 4\sum_{k=2}^{\frac{N}{2}} f(x_{2k}) + 2\sum_{k=2}^{\frac{N}{2}-1} f(x_{2k-1}) + f(x_{N+1}) \right)$$

Given that N is an even number.

Error: This method is $O(h^4)$ globally.

3. Lecture - Errors: Truncation and roundoff errors, error propagation, cancellation

3.1. Error types

3.1.1. Truncation error

Computer's cannot work with concepts containing infinities. This error that comes from computation while trying to avoid infinity is called truncation error. We can minimize this truncation error through more floating point operations per second (FLOP). Higher order method can decrease the truncation error but require more FLOP.

3.1.2. Rounding error

Definition: Real numbers are floating point numbers to computers in the shape of

$$(-1)^S \cdot 1.M \cdot 2^{E-127}$$

where S, M, E are bits, ie. 1 or 0. S represents whether positive or negative number. M determines how many decimals are represented. E determines how big or small a number can be.

3.1.3. Machine epsilon

This is a number that determines how close two numbers need to be in order to be considered the same number. `print(0.1+0.1+0.1==0.3)` will print false because the rounding error is larger than $\varepsilon_{\text{machine}}$.

3.1.4. Measuring errors

If f is the exact value and $\tilde{f}$ the numerical approximation. Then

Definition:

$$\left| f - \tilde{f} \right|$$

is the absolute error.

Definition:

$$\frac{\left| f - \tilde{f} \right|}{|f|}$$

is the relative error.

It is often easier to judge error through the relative error, bad however if $f \approx 0$. The smallest relative error you can get is $\varepsilon_{\text{machine}}$ which means that truncation errors are smaller than the floating point rounding errors.

3.2. IEEE floating point numbers

These are numbers using the IEEE standard. This system is composed of two methods: single precision (32 bits) and double precisiong (64 bits).

3.3. Error propagation

How does an error propagate/spread through computations? This can be measured through calculaitons but we skip the detailed analysis in this coures.

4. Lecture - Root finding

We know from lecture one that there are methods that solve difficult problems by converting said problems into many smaller problems. This conversion can be done through this subdivison of the domain or through iterations. We will go through the latter in this lecture.

We are generally solving

$$f(x) = 0 \quad x \in [a, b]$$

So, how do we solve these in practice?

1. Have some inital guess x_0, most likely $f(x_0) \neq 0$
2. Use some method to improve the intitial guess, ie. $x_0 \to x_1$
3. Repeat process
4. Stop when $|f(x_k)| < \varepsilon$ where ε is some sort of tolerance.

We are going to do this through various ways. In this course we go through

- Bisection method
- Fixed point iteration
- Newton's method

4.1. Bisection method

1. While $|b - a| > \varepsilon$
2. $c = \frac{a+b}{2}$
3. If $\text{sign}(f(c)) \neq \text{sign}(f(a))$ then assign $b = c$
4. Otherwise assign $a = c$

Do note that we require a somewhat nice function that should be either monotonically increasing or decreasing. This method is often a last resort.

- *Pro*: Guaranteed to converge.
- *Con*: Slow, ie. many iterations needed.

  Example: Say $f(x) = x^3 - x - 1$ where $x \in [0, 2]$ and say we choose tolerance $\varepsilon = 0.3$.

  $a_0 = 2, b_0 = 2, c_0 = \frac{0+2}{2} = 1$, so we have

  $$f(a_0) = -1$$
  $$f(b_0) = 5$$
  $$f(c_0) = -1$$

  now checking signs we see that the signs of a_0, c_0 are the same, ie negative. We do see however that the signs of b_0, c_0 differ and we therefore get the new interval $[a_0, b_0]$. Checking tolerance $|c_0 - b_0| = 1 > \varepsilon = 0.3$ so continue. Continue this iteration until tolerance reached.

4.2. Fixed point iteration

This method is about rewriting the equation $f(x) = 0$ as $x = g(x)$.

1. Let $x_0 =$ guess
2. While $|f(x_k)| > \varepsilon$ then let $x_{k+1} = g(x_k)$

   Example: Same function as above. Can rewrite this as

   $$\sqrt[3]{x + 1} = x$$

   Where the left hand side is our $g(x)$.

   Say we guess $x_0 = 1.5$. Then $x_1 = g(x_0) \approx 1.35721\ldots$. Continue since value of $f(x_1) > \varepsilon$. Then let $x_2 = g(x_1) \approx 1.33086\ldots$. Continue this until the threshold has been reached.

4.2.1. Convergence of the fixed point iteration

Convergence formula is

$$|x_k - x| < \frac{c^k}{1 - c}|x_1 - x_0|$$

where c is given by the condition $|g'(x)| < c, \quad x \in [a, b]$. So if $c \geq 1$ then do not go to zero as $k \to \infty$.

4.3. Newton's method

Definition: This method is about using Taylor expansions on f and expand to rewrite $f(x) = 0$ as $g(x) = x$. The iteration is

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

Common strategy is to start with a fixed point iteration or via bisection.

5. Lecture - Interpolation

Interpolation is about constructing data (estimating) between known data-points. Many ways to do this. We will go through just a few.

- Polynomial interpolation
  - Linear interpolation (piecewise)
  - Lagrange interpolation
- Spline interpolation

5.1. Linear interpolation (piecewise)

Definition: This method is about drawing straight lines between known data points.

$$y = y_k + \frac{(y_{k+1} - y_k)(x - x_k)}{x_{k+1} - x_k} \quad \text{at } (x, y)$$

5.2. Lagrange interpolation

Method for finding formulas for higher order polynomial interpolation.

Definition:

$$L(x) := \sum_{k=0}^{N} \left(f(x_k) \prod_{\substack{0 \le m \le N \\ m \neq k}} \frac{x - x_m}{x_k - x_m} \right)$$

The lagrange polynomial is the lowest degree polynomial going through each point.

Example: Say we have $x_0 = 0, x_1 = 0.5, x_2 = 1$ and $f(x_i) = 0, 2, 1$ respectively. So we will have, more simply written

$$f(x_0)l_0 = f(x_0)\frac{x - x_1}{x_0 - x_1} \cdot \frac{x - x_2}{x_0 - x_2}$$

next we have

$$f(x_1)l_1 = f(x_1)\frac{x - x_0}{x_1 - x_2} \cdot \frac{x - x_2}{x_1 - x_2}$$

and finally

$$f(x_2)l_2 = f(x_2)\frac{x - x_0}{x_2 - x_0} \cdot \frac{x - x_1}{x_2 - x_1}$$

plugging in gives us

$$\begin{cases} 0\frac{x-0.5}{0-0.5} \cdot \frac{x-1}{0-1} = 0 \\ 2\frac{x-0}{0.5-0} \cdot \frac{x-1}{0.5-1} = -2\frac{x(x-1)}{0.25} \\ 1\frac{x-0}{1-0} \cdot \frac{x-0.5}{1-0.5} = \frac{x(x-0.5)}{0.5} \end{cases}$$

so we get

$$L(x) = 0 - 8x(x-1) + 2x(x-0.5) = -8x^2 + 8x + 2x^2 - x = -6x^2 + 7x$$

5.3. Newton polynomial

Basically an extension of Lagrange interpolation where we are instead reusing lower order results.

Definition:

$$N(X) = \sum_{k=0}^{N} a_k \prod_{i=0}^{k-1} (x - x_i)$$

Where a_k depends $f(x_k)$. This will give same polynomial as Lagrange but is just a more computationally efficient method.

5.4. Splines (cubic)

Say we have 6 points, $x_0, x_1, ..., x_5$, we would fit 5 cubic polynomials in each interval, ie. fit $ax^3 + bx^2 + cx + d = y$ in each interval.

5.5. Runge's phenomenon

High degree polynomial interpolation leads to oscillations. this happens close to the edges. We can get the error of a polynomial of degree n as

$$|f(x) - p_n(x)| = \frac{f^{n+1}(\xi)}{(n+1)!} \prod_{k=0}^{n} (x - x_k)$$

where $\xi \in [x, x_k]$. It is the product that causes a very high error as we get further from x. In order to avoid this problem we can use piecewise interpolation since we will as such avoid high order methods, alternatively use something such as splines (e.g. cubic).

Note: This is caused by equidistant intervals. So, to avoid this we can use something like Chebyshev's nodes but this is not something you need to learn.

6. Lecture - Approximation of derivatives, basics of ODE's

The order of an ODE is decided by the order of the highest derivative so if we have

$$d^2\frac{y}{dx^2} + x\frac{dy}{dx} = 0$$

is of order 2 since the highest derivative is of order 2.

6.1. Boundary types

6.1.1. IVP ODE's

We need to set some conditions in order to solve an ODE by approximation. We can set an initial value condition and a boundary value condition, see BVP's later. Do note that if you put too many conditions a solution might not exist since every condition eliminates some solutions.

Example: Say we have

$$\frac{dM}{dt} = -kM$$

and say $M(0) = C$, so at time 0 it is equal to C. This gives us the system

$$\begin{cases} \frac{dM}{dt} = -kM \\ M(0) = C \end{cases}$$

We know the analytical solution is $Ae^{kt} = M$, using our initial solution gives us then

$$Ae^{-k0} = A = C$$

so

$$M(t) = Ce^{kt}$$

6.1.2. BVP ODE's

Can be set to other points than 0 so we usually don't think of the independant variable x as time.

Example: Say we have the system

$$\begin{cases} y''(x) + y(x) = 0 \\ y(0) = 1 \\ y'(\pi) = 5 \end{cases}$$

so we have limited $x \in [0, \pi]$. The general form of this ODE is

$$\begin{cases} y = C_1\cos(x) + C_2\sin(x) \\ y' = -C_1\sin(x) + C_2\cos(x) \\ y'' = -C_1\cos(x) - C_2\sin(x) \end{cases}$$

plugging in our conditions we get

$$\begin{cases} y(0) = C_1 = 1 \\ y'(\pi) = -C_2 = 5 \Rightarrow C_2 = -5 \end{cases}$$

which then gives us $y = \cos(x) - 5\sin(x)$

6.2. Numerically solving ODE's

In an IVP the function evolves through time. Given some initial condition. We divide the function in intervals and estimate the function piece by piece. This method is therefore part of the methodology of subdividing a larger problem into many smaller problems. Each next interval will have a rising error due to no longer having perfect conditions anymore since we have approximated them.

We need to therefore aprpoximate/describe the derivatives in the ODE using the information in discrete points. We have the classical definition of the derivative

$$\lim_{h\to 0}\frac{y(t+h) - y(t)}{h} = y'(t)$$

we are using this without the limits since computers cannot work with limits since that would require infinite computations. Giving us y

Definition:

$$y'(t_n) \approx \frac{y(t_n + h) - y(t_n)}{h}$$

We call this the right sided finite difference.

Error: This method is through Taylor expansion of linear order, ie. $O(h)$.

Definition:

$$y'(t_n) \approx \frac{y(t_n) - y(t_{n-1})}{h}$$

We call this the left sided finite difference.

Error: This method is through Taylor expansion of linear order, ie. $O(h)$.

Definition: Similarly, the centered finite difference is

$$y'(t_n) \approx \frac{y(t_n + h) - y(t_n - h)}{2h}$$

Error: This method is instead $O(h^2)$.

A problem is that derivatives might not sense a crazy numerical function such as if its a very oscillating function then we might skip some values.

6.2.1. Approximations of second-order derivatives

Definition: Using the left sided finite difference and right sided finite difference we can get the second order central difference through

$$y''(t_n) = \frac{y(t_n + h) - 2y(t_n) + y(t_n - h)}{h^2}$$

We will use these formulas in order to disretize BVP's into systems of linear equations $A\boldsymbol{f} = \boldsymbol{f}$. Need to be able to subtitute these into the given ODE to derive the matrix A and vector f.

7. Lecture - Higher order ODE methods

7.1. Runge-Kutta 2

The Midpoint Rule/RK2: Say we have two slopes $k_1 = f(t_n, y_n)$ and $k_2 = f\big(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\big)$. Using this we can compute

$$y_{n+1} = y_n + hk_2$$

Error: This method has a global error of $O(h^2)$.

Note: Since each value is calculated explicitly using only previously known values, and we never need to solve an equation that contains $y_{\{n+1\}}$ on both sides it is an explicit method. No iteration is required.

Example: Say $y' = \lambda y$ for $t \in [0, T]$. Applying RK2 gives us then in the first step:

- $k_1 = \lambda y_n$
- $k_2 = \lambda\big(y_n + \frac{h}{2}k_1\big) = \lambda\big(y_n + \frac{h}{2}(\lambda y_n)\big) = \lambda y_n\big(1 + \frac{h\lambda}{2}\big)$
- $y_{n+1} = y_n + hk_2$

Comparing this with Euler Forward, we immediately get: $y_{n+1} = y_n + hf(t_n, y_n) = y_n + h\lambda y_n$.

- How to choose between RK2 and Euler Forward?

Comparing RK2 and Euler Forward we have to note a thing. When to choose RK2 over Euler Forward since if we look at the FLOP required for RK2 in the previous example is 7 while only 3 FLOP is required for Euler Forward. So, Euler is only more efficient (computationally cheaper) if a very large error tolerance (low accuracy) is acceptable.

- When to choose Euler?

1. Low accuracy requirement (tolerance $\gtrsim 0.18$).
2. Simplicity of implementation is prioritized over computational cost for higher accuracy.

- When do we want higher accuracy?

We want higher accuracy in systems that simulate real world scenarios that require accurate predictions. We also want higher accuracy when a system is sensitive to errors, some examples being chaotic systems, long-term integrations, systems near bifurcations etc. A system nearing bifurcation is one that's approaching a critical threshold where its qualitative behavior changes dramatically in response to a small change in parameters.

7.2. Explicit Runge-Kutta methods

RK2 that was mentioned in RK2 is an explicit method. General form:

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i$$

where s is the number of stages. The intermediate slopes are calculated as

$$\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + c_3 h, y_n + h a_{21} k_1) \\
k_3 &= f(t_n + c_3 h, y_n + h(a_{31} k_1 + a_{32} k_2)) \\
&\vdots \\
k_s &= f\left(t_n + c_s h, y_n + h\sum_{j=1}^{s-1} a_{sj} k_j\right)
\end{aligned}$$

Do note that for $j \geq i$ we get $a_{ij} = 0$.

7.3. Butcher Tableau

A specific RK method is presented in a Butcher Tableu. Represented by the coefficients s, a_{ij}, b_i, c_j where $c_i = \sum_{j=1}^{i-1} a_{ij}$.

c_1	|	a_{11}	a_{12}	...	a_{1s}
c_2	|	a_{21}	a_{22}	...	a_{2s}
...	|	...	...	...	...
c_s	|	a_{s1}	a_{s2}	...	a_{ss}
	|	b_1	b_2	...	b_s

For explicit methods, the matrix $A = [a_{ij}]$ is strictly lower triangular ($a_{ij} = 0$ for $j \geq i$), and $c_1 = 0$. The tableau is often written omitting the zeros:

$\mathbf{0}$	|				
$\boldsymbol{c_2}$	|	$\boldsymbol{a_{21}}$			
$\boldsymbol{c_3}$	|	$\boldsymbol{a_{31}}$	$\boldsymbol{a_{32}}$		
$\boldsymbol{...}$	|	$\boldsymbol{...}$	$\boldsymbol{...}$	$\boldsymbol{...}$	
$\boldsymbol{c_s}$	|	$\boldsymbol{a_{s1}}$	$\boldsymbol{a_{s2}}$	$\boldsymbol{...}$	$\boldsymbol{a_{s,s-1}}$
	|	$\boldsymbol{b_1}$	$\boldsymbol{b_2}$	$\boldsymbol{...}$	$\boldsymbol{b_s}$

Midpoint rule (RK2) Tableau: $\left(s = 2, c_1 = 0, c_2 = \frac{1}{2}, a_{21} = \frac{1}{2}, b_1 = 0, b_2 = 1\right)$

$\mathbf{0}$	|		
$\frac{\mathbf{1}}{\mathbf{2}}$	|	$\frac{\mathbf{1}}{\mathbf{2}}$	
	|	$\mathbf{0}$	$\mathbf{1}$

7.4. Runge-Kutta 4

Butcher Tableau for the "classical" RK4: ($s = 4$)

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

The numbers to the left, ie. the first column is composed of the coefficients of h. The numbers in the bottom row are the coefficients of k in the expression for the value of y at the next step. The numbers in the middle of the array are the coefficients of the ks in second argument of f. Because this is an explicit method, each k only depends on the previous k, and so the table of coefficients has a triangular form.

- Accuracy: $\mathcal{O}(h^4)$
- (Often related to Simpson's rule for numerical integration due to its weights and evaluation points).

7.5. Implicit Methods

Implicit methods require the next step in the formula, such as in implicit Euler forward

7.5.1. Crank-Nicolson (Implicit Trapezoidal Rule)

Crank-Nicolson:

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n)) + f(t_{n+1}, y_{n+1})$$

Accuracy: Has an accuracy of $O(h^2)$.

8. Lecture - Boundary Value Problems (BVP)

Example:

$$y''(x) + y(x) = 0 \quad x \in \left[0, \frac{\pi}{2}\right]$$

The difference here is that we have defined this specific boundary compared to IVPs.

Let $y(0) = 1, y(\frac{\pi}{2}) = a$ These are out boundary values.

General solution is then

$$\begin{cases} y(x) = C_1 \cos(x) + C_2 \sin(x) \\ y'(x) = -C_1 \sin(x) + C_2 \cos(x) \\ y''(x) = -C_1 \cos(x) - C_2 \sin(x) \end{cases}$$

- What is C_1, C_2?

$y(0) = 1$ gives $C_1 \cdot 1 + C_2 \cdot 0 = 1$ so $C_1 = 1$. We also have $y(\frac{\pi}{2}) = a$ giving us $C_1 \cdot 0 + C_2 \cdot 1 = C_2 = a$. This gives the solution

$$y(x) = \cos(x) + a \sin(x)$$

Example: What happens if $x \in [0, \pi]$, same boundary values. Same ODE posed.

Same general solutions. We now instead have $C_1 = 1, -C_1 = a$ so this gives us that $a = -1$. No condition on C_2 so infinite solutions. This problem is therefore badly posed, ie. there is no unique solution.

Heat equation: This is a partial differential equation, ie PDE. Let

$$\frac{\partial T}{\partial t} + \frac{\partial^2 T}{\partial x^2} = f(x)$$

Here, T is the temperature. Called the heat equation because it explains heat diffusion, classical example a metal rod between two walls, temperature of left wall being α=left boundary condition, respectively right wall being β=right BC. Let there be a heat source in the middle of the metal rod, this is our right hand side in the PDE.

To simplify the problem. let us make this into an ODE. Assume we have reached a steady state. Ie. enough time has passed so that the system is in balance so no change in the solution with respect to time. So:

$$\frac{\partial T}{\partial t} = 0$$

so this gives us then

$$\frac{d^2 T}{dx^2} = f(x)$$

since we are only handling one variable since we assumed stead state. This is now an ODE. We have

$$\begin{cases} T(a) = \alpha \\ T(b) = \beta \end{cases}$$

on the space interval $[a, b]$. Solve this using finite difference methods.

Recall the finite difference approximation of second derivative, so if

$$\frac{d^2 T}{dx^2} = f(x)$$

then

$$\frac{d^2 T}{dx^2} = \frac{T_{i-1} - 2T_i + T_{i+1}}{h^2}$$

So, inserting in the equation gives us then

$$f(x_i) = \frac{T_{i-1} - 2T_i - T_{i+1}}{h^2}$$

Note: IVP strategy of starting at $x = a$ and stop to $x = b$ will not work since we won't fulfil the right boundary condition, ie. $T(b) = \beta$.

Let us use 0-indexing and say

$$\begin{cases} T_0 = \alpha \\ T_{N+1} = \beta \end{cases}$$

We need to solve a system of equations for each i. Let this system be

$$A(T) = F$$

where (T) is a vector containing all T_i:s. So we get

$$\frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \dots \\ 1 & -2 & 1 & 0 & \dots \\ 0 & 1 & -2 & 1 & \dots \\ 0 & 0 & 1 & -2 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}\begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ \vdots \\ T_N \end{pmatrix} = \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_N) \end{pmatrix} + \begin{pmatrix} -\frac{\alpha}{h^2} \\ 0 \\ \vdots \\ 0 \\ -\frac{\beta}{h^2} \end{pmatrix}$$

First matrix has diagonal element: -2 and the adjoint elements are 1.

Get the first matrix elements by putting in $i = 1, 2, 3, ..., N$.

Get the last matrix by $i = 1$, gives

$$\frac{-2T_1 + T_2}{h^2} = f(x_1) - \frac{T_0}{h^2} = f(x_1) - \frac{\alpha}{h^2}$$

. Same goes for $i = N$. This gives us through

$$\frac{T_{N-1} - 2T_N}{h^2} = f(x_N) - \frac{\beta}{h^2}$$

Example: Let

$$\begin{cases} u''(x) + u'(x) = f(x) \\ u(0) = \alpha \\ u(1) = \beta \end{cases}$$

Note that the approximations of the derivative are

$$\begin{cases} u''(x) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \\ u'(x) \approx \frac{u_{i+1} - u_i}{h} \end{cases}$$

for $i \in \{1, 2, 3, ...\}$ (for the first derivative, you can choose between left, right and midpoint).

Say we have left boundary at x_0, right at x_{N+1} of which the intervals are equidistant. This means that

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \frac{u_{i+1} - u_i}{h} = f(x)$$

We have that $u_0 = \alpha, u_{N+1} = \beta$. So, writing in a system now. We have

$$\left(\frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \dots \\ 1 & -2 & 1 & 0 & \dots \\ 0 & 1 & -2 & 1 & \dots \\ 0 & 0 & 1 & -2 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} + \frac{1}{h}\begin{pmatrix} -1 & 1 & 0 & 0 & \dots \\ 0 & -1 & 1 & 0 & \dots \\ 0 & 0 & -1 & 1 & \dots \\ 0 & 0 & 0 & -1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}\right)\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{pmatrix} + \begin{pmatrix} -\frac{\alpha}{h^2} \\ 0 \\ 0 \\ 0 \\ -\frac{u_{N+1}}{h} - \frac{1}{h^2}u_{N+1} \end{pmatrix}$$

So we have separated the matrices into two since we already know the matrix for the 2nd derivative. Get the second matrix by putting in the different i-values into the approximation.

Using central (thats the notes online) we get

$$\frac{1}{2h}\begin{pmatrix} 0 & 0 & 0 & 0 & \dots \\ -1 & 0 & 1 & 0 & \dots \\ 0 & -1 & 0 & 1 & \dots \\ 0 & 0 & -1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

For the left finite difference we get

$$\frac{1}{h}\begin{pmatrix} 1 & 0 & 0 & 0 & \dots \\ -1 & 1 & 0 & 0 & \dots \\ 0 & -1 & 1 & 0 & \dots \\ 0 & 0 & -1 & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

8.1. Errors

- Local truncation error here is quite different from IVP. Here, it is

$$\text{LTE} \sim \text{error from finite difference approximation}$$

- Global truncation error:

$$\mathrm{E} = U - \hat{U}$$

here, the first term is the is the approximation, second is the true solution. These are vectors. Ie.

$$U = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{pmatrix}, \tilde{U} = \begin{pmatrix} u(x_1) \\ u(x_2) \\ u(x_3) \\ \vdots \\ u(x_N) \end{pmatrix}$$

This leads to the error matrix:

$$E = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_N \end{pmatrix}$$

Ie. $e_i = u_i - u(x_i)$. Here we call each element in E a local error. Not to be confused with LTE from previous point.
Take some norm of E,

$$\|E\|_1 = h\sum_{i=1}^{N}|e_i|$$
$$\|E\|_2 = \left(h^2\sum_{i=1}^{N}|e_i|\right)^{\frac{1}{2}}$$
$$\|E\|_\infty = \max(e_i)$$

Heat equation - LTE: We have

$$\frac{1}{h^2}(T_{i-1} - 2T_i + T_{i+1}) - f(x_i) = \tau_i$$

Get the error via Taylor expansion of τ_i, the residual truncation error.

$$\frac{1}{h^2}(T(x-h) - 2T(x) + T(x+h)) - f(x_i) = \tau_i$$

Trying to fit the true solution into the finite difference scheme. This gives us via Taylor

$$\frac{1}{h^2}\Bigg(T(x) - hT'(x) + \frac{h^2}{2}T''(x) - \frac{h^3}{3!}T'''(x) + O(h^4)$$
$$-2T(x)$$
$$+T(x) + hT'(x) + \frac{h^2}{2}T''(x) + \frac{h^3}{3}T'''(x) + O(h^4)\Bigg)$$
$$-f(x_i)$$

This gives us then

$$= \frac{1}{h^2}(h^2T''(x) + O^2(h^4)) - f(x_i)$$
$$= T''(x) + O(h^2) - f(x_i)$$

do note that $T''(x) - f(x_i) = 0$ from the beginning, so we get

$$= O(h^2) = \tau_i$$

This is the local truncation residual error e_i.

Heat equation - Global residual error:

$$\begin{pmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \\ \vdots \\ \tau_N \end{pmatrix}$$

which means that

$$A\hat{U} = F + \tau$$

we also know that the true solution

$$AU = F \quad \text{(numerical)}$$

Subtracting these two gives us

$$A\underbrace{\left(U-\hat{U}\right)}_{E}=\tau$$

So

$$AE=-\tau$$

this could be seen as a discretization of

$$e''(x)=-\tau(x)$$

ie, second derivative of the error on the life hand side and the residual on the right hand side. We also have some boundary conditions

$$e(a)=0=e(b)$$

The equation $e''(x)=-\tau(x)$ is easy to solve by hand. Integrate twice and write truncation error $\tau(x)\sim Ch^4$, we then get

$$\implies e(x)=Dh^2$$

So, conclusion, global truncation error is $O(h^2)$.

8.2. Stability for BVP:s

- Different from IVP: definition since we are missing time-evolution.

We want a well-behaved system $AU=F$, ie. so that E does not blow up; $AE=-\tau$. Want to control E, say

$$\|E\|\leq\left\|A^{-1}\tau\right\|\leq\left\|A^{-1}\right\|\|\tau\|$$

Dont want the inverse here to have a big norm.

9. Lecture - Gaussian elimination and LU decomposition

9.1. How do we solve ODE-BVP numerically?

Remember the general system:

$$Ax = B$$

for some matrices A, B and a vector $x = (x_1, ..., x_n)$

There are a few ways, in this lesson we will go through so called direct methods. Some key points:
- More exact
- Slow
- Robust

Another way is via iterative methods (future lecture)
- Fast
- Solves up to use defined accuracy tolerance

9.2. Direct methods

Based on Gaussian elimination. Ie, the classical and perhaps most common way of solving the general system. How many FLOP does this method involve? How does FLOP depend on the dimension n?

In general FLOP required for Gaussian elimination is as follows.
- Row 1: $(n+1)(n-1)$ multiplications
- Row 2: $(n)(n-2)$ -||-
- Row 3: $(n-1)(n-3)$ -||-
- Row $n-1$: $(3)(n-(n-1))$

we can even more comprehensively say that creating an upper triangular matrix is $O(n^3)$ and backward substition (ie. getting actual values) is $O(n^2)$. We can however avoid the $O(n^3)$ part through LU-factorization.

9.2.1. LU-factorization

Definition: Find lower triangular matrix L

$$\begin{pmatrix} x & 0 & 0 \\ x & x & 0 \\ x & x & x \end{pmatrix}$$

and upper triangular matrix U

$$\begin{pmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & x \end{pmatrix}$$

such that

$$A = LU$$

Example: Let

$$A = \begin{pmatrix} 2 & -1 & 1 \\ 3 & 2 & 2 \\ 4 & 4 & -4 \end{pmatrix}$$

and let

$$U = \begin{pmatrix} 2 & -1 & 1 \\ 3 & 2 & 2 \\ 4 & 4 & -4 \end{pmatrix}$$

and the other factor

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

adding to row 2 by row $1 \cdot \frac{3}{2}$ and row 3 by row $1 \cdot \frac{4}{2}$ in U then gives

$$U = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3.5 & 0.5 \\ 0 & 6 & -6 \end{pmatrix}$$

in L, we get instead

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{3}{2} & 1 & 0 \\ \frac{4}{2} & 0 & 1 \end{pmatrix}$$

and then adding to row 3 by row 2$\cdot \frac{6}{3.5}$ finally gives us

$$U = \begin{pmatrix} 2 & -1 & 1 & \\ 0 & 3.5 & 0 & 5 \\ 0 & 0 & -\frac{48}{7} & \end{pmatrix} \quad L \begin{pmatrix} 1 & 0 & 0 \\ \frac{3}{2} & 1 & 0 \\ \frac{4}{2} & \frac{6}{3.5} & 1 \end{pmatrix}$$

this completes the LU-factorization.

9.3. Partial pivoting

This method is about changing rows so that pivot elements is always the element in its column with largest absolute value. How do we keep track of row switches? We use the permutation matrix P such that

$$PA = LU$$

If we have

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad U = \begin{pmatrix} 2 & -1 & 1 \\ 3 & 2 & 2 \\ 4 & 4 & -4 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ & 1 & 0 \\ & & 1 \end{pmatrix}$$

here we see that in U the largest absolute element in column 1,2,3 is 4 so we switch

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad U = \begin{pmatrix} 4 & 4 & -4 \\ 3 & 2 & 2 \\ 2 & -1 & 1 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ & 1 & 0 \\ & & 1 \end{pmatrix}$$

do note that L we never switch at this part. Through elimination now we can get

$$P \text{ the same} \quad U = \begin{pmatrix} 4 & 4 & -4 \\ 0 & -1 & 5 \\ 0 & -3 & 3 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{4} & 1 & 0 \\ \frac{3}{4} & & 1 \end{pmatrix}$$

and final elimination gives

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad U = \begin{pmatrix} 4 & 4 & -4 \\ 0 & -3 & 3 \\ 0 & 0 & 4 \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{4} & 1 & 0 \\ \frac{3}{4} & \frac{1}{3} & 1 \end{pmatrix}$$

this satisifes the condition $PA = LU$.

10. Lecture - Iterative methods for solving linear systems

Motivation: Direct methods (LU decomposition) are slow and computationally expensive since it is $O(N^3)$ so since A is of dimension $N \times N$ this grows quickly. For example 3D PDE simulations, large ODE-BVP problems, machine learning etc.

- Bit less robust and less accurate
- Faster

10.1. Idea - Iterative methods

- Have some initial guess, x_0.
- Have a loop, eg

```
while error in x > tolerance
    xk = f(A,xbefore, b)
    k ++
```

10.2. Basic iterative methods for solving linear systems

- Idea: do some smart splitting of A, such that $A = M - K$ (depends on method), here M should be an invertible matrix and K, called the remainder.

Say we have $Ax = b$, ie.

$$\begin{aligned}(M-K)x &= b \Rightarrow \\ Mx &= Kx + b \Rightarrow \\ x &= M^{-1}Kx + M^{-1}b\end{aligned}$$

Use as iteration formula:

$$x_k = M^{-1}Kx_{k-1} + M^{-1}b$$

Note: No point if it as hard to compute M^{-1} as A^{-1}. Not any splitting $A = M - K$ is ok, need to prove convergence.

10.2.1. Jacobi iterations

Let $M = D$ where D is the diagonal of A. So we pick out the diagonal of A and let that be the diagonal of D. Let $K = L + U$ (not same as LU-decomposition). $-L, -U$ strictly lower and upper triangular part (meaning dont include the diagonal). Can then write

$$x_k = D^{-1}(L+U)x_k + D^{-1}b$$

Example: Let

$$A = \begin{pmatrix} 2 & 1 \\ 5 & 7 \end{pmatrix}$$

So $N = 2$ and let

$$b = \begin{pmatrix} 11 \\ 13 \end{pmatrix}$$

Computing the true solution x'

$$x' = \begin{pmatrix} 7.11... \\ -3.22... \end{pmatrix}$$

So

$$D = \begin{pmatrix} 2 & 0 \\ 0 & 7 \end{pmatrix}, L = \begin{pmatrix} 0 & 0 \\ -5 & 0 \end{pmatrix}, U = \begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix}$$

Note the sign switch here.

$$D^{-1} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{7} \end{pmatrix}$$

in general we simply divide like this for a diagonal matrix so it is quite easy to compute inverse of diagonal matrix.

$$D^{-1}(L+U) = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{7} \end{pmatrix} \left[\begin{pmatrix} 0 & 0 \\ -5 & 0 \end{pmatrix} + \begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix} \right] = \begin{pmatrix} 0 & -\frac{1}{2} \\ -\frac{5}{7} & 0 \end{pmatrix}$$

So our iteration becomes

$$x_k = \begin{pmatrix} 0 & -\frac{1}{2} \\ -\frac{5}{7} & 0 \end{pmatrix} x_{k-1} + \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{7} \end{pmatrix} \begin{pmatrix} 11 \\ 13 \end{pmatrix}$$

Starting said iteration gives us with the guess $x_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ the following:

$$k=1 \Rightarrow x_1 = \begin{pmatrix} 0 & -\frac{1}{2} \\ -\frac{5}{7} & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} \frac{11}{2} \\ \frac{13}{7} \end{pmatrix} = \begin{pmatrix} 5 \\ \frac{8}{7} \end{pmatrix}$$

$$k=2 \Rightarrow x_2 = \begin{pmatrix} 0 & -\frac{1}{2} \\ -\frac{5}{7} & 0 \end{pmatrix} \begin{pmatrix} 5 \\ \frac{8}{7} \end{pmatrix} + \begin{pmatrix} \frac{11}{2} \\ \frac{13}{7} \end{pmatrix} = \begin{pmatrix} \frac{69}{14} \\ -\frac{12}{7} \end{pmatrix}$$

$$\vdots$$

which will converge to the true solution x' mentioned above. We will prove this convergence for some types of A later.

10.2.2. Gauss-Seidel method

Also based on a splitting but here we use $M = D - L$ with same properties as before. We let $K = U$. We have

$$x_k = (D-L)^{-1}Ux_k + (D-L)^{-1}b$$

$D - L$ is lower triangular and we therefore have a forward substition at these steps. Meaning it is easy to compute.

Example: Same example as previously. Let

$$A = \begin{pmatrix} 2 & 1 \\ 5 & 7 \end{pmatrix}$$

So $N = 2$ and let

$$b = \begin{pmatrix} 11 \\ 13 \end{pmatrix}$$

Computing the true solution x'

$$x' = \begin{pmatrix} 7.11... \\ -3.22... \end{pmatrix}$$

So

$$D = \begin{pmatrix} 2 & 0 \\ 0 & 7 \end{pmatrix}, L = \begin{pmatrix} 0 & 0 \\ -5 & 0 \end{pmatrix}, U = \begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix}$$

We know that

$$M = D - L = \begin{pmatrix} 2 & 0 \\ 0 & 7 \end{pmatrix} - \begin{pmatrix} 0 & 0 \\ -5 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 5 & 7 \end{pmatrix}$$

we see that it is lower triangular.

$$K = \begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix}$$

Instead of explicitly computing the inverse now, we can solve

$$Mx_k = Kx_{k-1} + b$$

instead. Ie. multiply other side by M. Solve this using forward substition which is $O(N^2)$. So we have

$$\begin{pmatrix} 2 & 0 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} x_k^1 \\ x_k^2 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x_{k-1}^1 \\ x_{k-1}^2 \end{pmatrix} + \begin{pmatrix} 11 \\ 13 \end{pmatrix}$$

So computing this for the same initial guess $x_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ we have

$$\begin{aligned} k=1 \Rightarrow \begin{pmatrix} 2 & 0 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} x_1^1 \\ x_1^2 \end{pmatrix} &= \begin{pmatrix} 0 & -1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 11 \\ 13 \end{pmatrix} \\ &\Rightarrow 2x_1^1 = -1 + 11 \\ &\Rightarrow x_1^1 = 5 \\ &\Rightarrow 5x_1^1 + 7x_1^2 = 13 \\ &\Rightarrow x_1^2 = \frac{13 - 5 \cdot 5}{7} = -\frac{12}{7} \end{aligned}$$

giving us

$$x_1 = \begin{pmatrix} 5 \\ -\frac{12}{7} \end{pmatrix}$$

Note: x_0^1 is not used to compute the first component of the new solution x_1^1. Likewise x_0^2 not used to compute the second component of the new solution x_0^2 or x_1^1 because $D - L$ is lower triangular, this always happens in such cases. This means that if you have one and only one array in your code to store x_k, x_{k-1}, ie.

$$\begin{pmatrix} x_k^0 \\ x_k^1 \\ x_{k-1}^2 \\ \vdots \\ x_{k-1}^N \end{pmatrix}$$

we call this the Gauss-Seidel Sweep.

10.2.3. Convergence of basic method

In general,

$$x_{k+1} = \underbrace{M^{-1}K}_{R} x_k + \underbrace{M^{-1}b}_{c}$$

Say $M^{-1}K = R$, call this the relaxation matrix or the iterations matrix. We have the error

$$e_k = x_k - x$$

where x is the true solution. For the true solution we have

$$x = Rx + c$$

If we want to know the error at $k+1$ we have

$$\begin{aligned} e_{k+1} &= x_{k+1} - x \\ &= Rx_k + c - (Rx + c) \\ &= R(x_k - x) \\ &= Re_k \\ &= R(Re_{k-1}) \\ &= R(R(Re_{k-2})) \\ &= ... \\ &= R^{k+1}e_0 \end{aligned}$$

If R does not amplify vectors, then we have that

$$\|e_{k+1}\| < \|e_k\|$$

and $\|e_k\| \to 0$ as $k \to \infty$, this is what we mean by convergence. Here the norm is some vector norm.

By R not amplifying the vector means that

$$\|R\| < 1 \Leftrightarrow \rho(R) < 1$$

(matrix norms next time). Taking all the eigenvalues using Spectral and looking at the maximum absolute value. So $Rx = \lambda x$.

10.2.3.1. Jacobi convergence

We have

$$R = D^{-1}(L + U)$$

note that D is diagonal and the paranthesis becomes everything outside the diagonal.

$$R_{ij} = \frac{-A_{ij}}{A_{ii}} \quad i \neq j$$

we the stuff outside of the diagonal divided by the diagonal.

$$R_{ij} = 0 \quad i \neq j$$

These are the elements of the relaxation matrix. The maximum norm is

$$\|R\|_\infty = \max_i \left(\sum_j |R_{ij}| \right)$$

so we take the biggest element by taking the sum over columns. We then have

$$= \max_i \left(\sum_j \left| \frac{A_{ij}}{A_{ii}} \right| \right) < 1$$

by condition said above. So we want the diagonal elements to be big in relation to the non-diagonal elements. Ie this is true if

$$\sum_{j, j \neq i} |A_{ij}| < |A_{ii}|$$

we call this strictly diagonally dominant.

Example: Back to the example from before where we had

$$A = \begin{pmatrix} 2 & 5 \\ 1 & 7 \end{pmatrix}$$

the bottom row gives $7 > 1$ meaning that the diagonal dominates, but not for the upper row. So the condition is actually not fulfilled for this example, this does not mean that it doesn't converge, just a faulty choice of method?

Jacobi converges for strictly diagonally dominant matrices (sufficiently not necessary condition).

10.2.3.2. Gauss-Seidel convergence

Converges for strictly diagonally dominant matrices or SPD (symmetric positive definite). If we don't have SPD, we can do the following trick on $Ax = b$:

$$A^T Ax = A^T b$$

which is SPD.

10.2.4. How to know when to stop iteration of error (stopping criteria)

We had

$$e_k = x_k - x$$

which is hard to evalute since true solution is unknown. One thing to look at is the absolute

$$\left\| x_{k+1} - x_k \right\| < \text{tolerance}$$

If don't know how big the solution may be, use the relative instead

$$\frac{\left\| x_{k+1} - x_k \right\|}{\|x_k\|} < \text{tolerance}$$

Another thing to look at is how well does x_k satisfy the linear system $Ax = b$.

$$\|r_k\| = \|Ax_k - b\| < \text{tolerance}$$

which should be 0 in a perfect world, ie. for the true solution. Call this the residual. Maybe here too don't size to expect, so look at the relative instead

$$\frac{\|r_k\|}{\|b\|}$$

11. Lecture - Some matrix properties related to accuracy & efficiency

The speed of convergence for iterative methods depends on the properties of the matrix A.

In this lecture we are going to talk about
- Condition number
- Sparsity, SPD

11.1. Prerequisites: Vector & matrix norms

11.1.1. Vector norms

11.1.1.1. L2 and higher norm

Have some

$$\|x\|_2 = \sqrt{\sum_{i=1}^{N} x_i^2}$$

This is called the $l2$-norm, $L2$ if in continous system.

$$\|x\|_p = \sqrt[p]{\sum_{i=1}^{N} x_i^p}$$

11.1.1.2. The maximum norm

$$\|x\|_\infty = \max_i |x_i|$$

This norm is the most sensitive to errors since if other errors are 0 but one is a high error then it is the high error that will be taken into consideration only. The other two sort of average out the errors.

11.1.1.3. Equivalence of norms

Any two norms $\|\cdot\|_a$ and $\|\cdot\|_b$ on a finite dimensional space, ie. $\{x_i\}$ for $i \leq N$, say the vector space V are equivalent, meaning that there exists some constants c, C so that

$$c\|x\|_a \leq \|x\|_b \leq C\|x\|_a \quad \forall x \in V$$

Useful for bounding errors.

11.1.2. Matrix norms (constructed from vector norms)

Given two vector norms $\|\cdot\|_a, \|\cdot\|_b$, we can then create

$$\|A\|_{a,b} = \sup_x \{\|Ax\|_b\}$$

with $\|x\|_a = 1$. More common to write as

$$\|A\|_{a,b} = \sup_x \left\{ \frac{\|Ax\|_b}{\|x\|_a}, \quad x \neq 0 \right\}$$

- In general we will have $a = b = p$. So

$$\|A\|_{p,p} =: \|A\|_p = \sup_{x \neq 0} \left\{ \frac{\|Ax\|_p}{\|x\|_p} \right\}$$

- $a = b = 1$

$$\|A\|_{1,1} =: \|A\|_1 = \max_j \sum_i |a_{ij}|$$

So maximum column sum.

- $a = b = \infty$

$$\|A\|_{\infty,\infty} =: \|A\|_\infty = \max_i \sum_j |a_{ij}|$$

So maximum row sum.

- $a = b = 2$

$$\|A\|_{2,2} =: \|A\|_2 = \sqrt{\rho(A^t A)}$$

$\rho(B)$ is the largest eigennvalue. Call this the spectral radius such that $Bx = \lambda x$.

In order to be able to bound some errors we need some inequaliaties.

11.1.3. Inequalities satisifed by vector induced matrix norms

$$\|Ax\|_\gamma \leq \|A\|_{\gamma,\gamma} \|x\|_\gamma$$

$$\|ABx\|_\gamma \leq \|A\|_{\gamma,\gamma} \|B\|_{\gamma,\gamma} \|x\|_\gamma$$

$$\|AB\|_{\gamma,\gamma} \leq \|A\|_{\gamma,\gamma}\|B\|_{\gamma,\gamma}$$

11.2. Last lecture (on Jacobi iteration)

We needed

$$\|R\| \leq 1$$

This relaxation matrix came from

$$x_{k+1} = Rx_k + c$$

This lead us to

$$e_{k+1} = R^{k+1}e_0$$

If we want to bound the error, know how big the error is

$$\left\|e_{k+1}\right\| = \left\|R^{k+1}e_0\right\|$$

Using the first inequality on the right hand side we have

$$\left\|R^{k+1}\right\|\|e_0\|$$

Want

$$\left\|R^{k+1}\right\| < 1$$

(preferably much smaller than 1 if k is large).

$$\|RRRR...\| < 1$$

let us call the first R its own matrix, others a collection.

$$\|R\|\left\|R^k\right\| < 1$$

continuing this process will give us

$$\|R\|^{k+1} < 1$$

Okay if

$$\|R\| < 1$$

The type of norm we choose depends completely on the choice at the beggining for $\left\|e_{k+1}\right\|$.

We chose $\|R\|_\infty$ before, it is the easiest to show convergence for. We had a formula

$$\max_i \sum_j |R_{ij}| = \max_i \sum_{j,j\neq i} \frac{|A_{ij}|}{|A_{ii}|} < 1$$

this means that

$$\sum_{i,i\neq j} |A_{ij}| < |A_{ii}|$$

11.3. Condition number

Definition:

$$\kappa(A) = \left\|A^{-1}\right\|\|A\|$$

Call this kappa the condition number.

Best case scenario is this equals 1 meaning that convergence is likely and quick. If you have a large condition number you also get large errors in x and your iterative solver is slow.

Sidenote: A bad condition number of $Ax = b$ can be fixed through some preconditioner $PAx = Pb$. So your new preconditioned matrix is PA. Then we hope that $\kappa(PA) \ll \kappa(A)$.

11.3.1. Why do we get high errors if the condition number is large?

Imagine error ε in right hand side in

$$Ax = b + \varepsilon$$

Could be rounding erros, measuring etc. How does this ε affect the solution x? Let's invert

$$x = A^{-1}(b + \varepsilon) = A^{-1}b + A^{-1}\varepsilon$$

So we see that the error depends on how the inverse of A behaves.

- What is the ratio between the error in x and b?

$$\frac{\text{Relative error in } x}{\text{Relative error in } b} = \frac{\frac{\|A^{-1}\|}{\|x'\|}}{\frac{\|\varepsilon\|}{\|b\|}}$$

with x' being the true solution. This can be rewriten as

$$\frac{\|A^{-1}\varepsilon\|}{\|\varepsilon\|} \cdot \frac{\|b\|}{\|x'\|}$$

What is x'? We know it satisifes

$$Ax' = b$$

so $x' = A^{-1}b$. Meaning

$$\frac{\|A^{-1}\varepsilon\|}{\|\varepsilon\|} \cdot \frac{\|b\|}{\|A^{-1}b\|}$$

- worst case:

$$\max_{\varepsilon, b\neq 0}\left(\frac{\|A^{-1}\varepsilon\|}{\|\varepsilon\|} \cdot \frac{\|b\|}{\|A^{-1}b\|}\right) = \max_{\varepsilon}\left(\frac{\|A^{-1}\varepsilon\|}{\|\varepsilon\|}\right) \max_{b\neq 0}\left(\frac{\|b\|}{\|A^{-1}b\|}\right)$$

now we get

$$\max_{\varepsilon}\left(\frac{\|A^{-1}\varepsilon\|}{\|\varepsilon\|}\right) \max_{x}\left(\frac{\|Ax\|}{\|x\|}\right)$$

Through the definition of the norm we then have

$$\|A^{-1}\|\|A\| \coloneqq \kappa(A)$$

Large condition number means that x is sensitive to changes in b.

- Best case scenario $\kappa(A) = 1$
- If $\kappa(A) \gg 1$ then we call it ill-conditioned.

  Example: Have $Ax = b$ where

$$A = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}, \quad b = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}$$

  The true solution

$$x' = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

  . Introducing a little error

$$\varepsilon = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$$

  so

$$\tilde{b} = b + \varepsilon = \begin{pmatrix} 32.1 \\ 23.1 \\ 33.1 \\ 31.1 \end{pmatrix}$$

  Solving this system then

$$Ax = \tilde{b}$$

  gives completely different solutions

$$x = \begin{pmatrix} 9.2 \\ -12.6 \\ 4.5 \\ -1.1 \end{pmatrix}$$

  because this matrix is ill-conditioned we have very high error, see lab 6.

11.3.2. Condition number a general concept

This number says something about the sensitity of the problem, can be used in the context of functions.

Example: Ill-conditioned problem: Find the intersection between two lines. Easy if far from parallel. Just a simple error in the inclination then the intersection point will be vastly different. So a small error in description of lines leads to a large error in the intersection of the lines.

11.4. More properties

Sparsity: Sparse matrix is a matrix with mostly zeros. Mostly is upto interpretation. Often we get sparse matrices when discretazing. The opposite of sparsity is dense.

- Pros:
  - Less computation
  - Less storage

It is possible to ruin the sparsity of some matrix, call this "fill-in", ie. when a zero becomes non-zero by some algorithm. An example is LU-factorization, it causes fill-in. This can be avoided by switching rows and columns in a clever way, compare to partial pivoting.

Symmetry: A matrix with

$$A = A^T$$

There is a wider selection of iterative solvers for symmetrical matrices. Can also compute the condition number in a nicer way if symmetry is achieved, as follows:

$$\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$$

where λ are the eigenvalues.

Positive definiteness: A is positive definite if

$$x^T Ax > 0 \quad \forall x \neq 0$$

Negative definiteness: A is positive definite if

$$x^T Ax < 0 \quad \forall x \neq 0$$

Note that we much prefer positive definite, negative definite can often be converted into positive definiteness.

SPD: Stands for symmetric posivite definite.

Are often the easiest to handle for most iterative solvers. Will have positive eigenvalues. If you have an SPD matrix you can do Cholesky decomposition where

$$A = LL^T$$

where L is lower triangular. This is faster to compute.

12. Lecture - Iterative methods for solving linear systems

12.1. Projection methods for solving linear systems

- Faster than the basic methods mentioned in the previous lecture, Jacobi and Gauss-Seidel.

Idea is to look for solution in a smaller space, ie. $K \subset \mathbb{R}^n$. So we *project* to a smaller space. The true solution resides in $\mathbb{R}^n$ so we want to find an approximation $\tilde{x} \subset K$. The closer K is to the real space the better the approximation but more computationally expensive.

Look for this approximation by

$$\tilde{x} = x_0 + \delta$$

here δ is some improvement. This is the difficult part, ie. how to find the update δ.

True solution fullfils

$$r = b - Ax = 0$$

where r is the residual. Let's instead require

$$rw^T = 0 \quad \forall w \in L$$

this is essentially a scalar product. This also means that the residual is orthogonal. This has to do with the fact that we are projecting to K.

Notation:
- L is spanned by arrays (vectors) w_i stored in a matrix W.

$$W = \begin{pmatrix} \vdots & \vdots & \dots & \vdots \\ w_1 & w_2 & \dots & w_m \\ \vdots & \vdots & \dots & \vdots \end{pmatrix}$$

- K is spanned by v_i stored in V

$$V = \begin{pmatrix} \vdots & \vdots & \dots & \vdots \\ v_1 & v_2 & \dots & v_m \\ \vdots & \vdots & \dots & \vdots \end{pmatrix}$$

Note: We skip the derivation of the update δ, not required to know. Some old exams may have this question but just skip it.
- Given V, W and $rw^T = 0$ we can get

$$\delta = V(W^TAV)^{-1}W^Tr$$

All projection methods can be written as

$$x_{k+1} = x_k + \delta$$

so every method depends fully on the choice of δ.

Requirements:
- The inverse of W^TAV must exist, make sure of this through two different ways.
  - One way is to choose $V = W$ so that we get

    $$W^TAV = V^TAV$$

    Do note that A must be symmetric positive definite. Ie. $x^TAx > 0 \forall x \neq 0$.
  - If A not SPD then choose $W = AV$. So we get

    $$W^TAV = (AV)^TAV = V^TA^TAV$$

    A^TA will end up giving an SPD matrix.

12.1.1. Steepest descent / Gradient descent

- Idea of this method is to find minimum of some things. Often used in machine learning and optimization.

We choose SPD matrix A and let

$$V = W = \begin{pmatrix} \vdots \\ r_k \\ \vdots \end{pmatrix}$$

so consists of only residuals. The iteration of this method looks as the following

$$\begin{aligned} x_{k+1} &= x_k + V(W^TAV)^{-1}W^Tr_k \\ &= x_k + r_k \underbrace{(r_k^TAr)^{-1}r_k^Tr_k}_{\text{scalar } \alpha_k} \end{aligned}$$

Is $r_{k+1}w^T$ fulfilled? We have

$$\begin{aligned} r_{k+1}w^T &= r_{k+1}r_k^T \\ &= -(Ax_{k+1} - b)r_k^T \\ &= -(A(x_k + \alpha_k r_k) - b)r_k^T \\ &= -(-r_k + \alpha_k A r_k)r_k^T \\ &= r_k r_k^T - \frac{r_k^T r_k}{r_k^T A r_k} A r_k r_k^T = 0 \end{aligned}$$

(we essentially change space every iteration). So should converge since it fulfills the condition from before if SPD.

Example: Let

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

True solution $x' = \begin{pmatrix} 1 & 1 \end{pmatrix}^T$. We don't know this so have an initial guess $x^T = \begin{pmatrix} 1 & 0 \end{pmatrix}^T$. We need the residaul

$$\begin{aligned} r_0 &= \begin{pmatrix} 3 \\ 3 \end{pmatrix} - \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 3 \\ 3 \end{pmatrix} - \begin{pmatrix} 2 \\ 1 \end{pmatrix} \end{aligned}$$

also have

$$\begin{aligned} \alpha_0 = (r_0^T A r_0)^{-1} r_0^T r_0 &= \left(\begin{pmatrix} 1 & 2 \end{pmatrix}\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} 1 \\ 2 \end{pmatrix} \right)^{-1} \begin{pmatrix} 1 & 2 \end{pmatrix}\begin{pmatrix} 1 \\ 2 \end{pmatrix} \\ &= \left(\begin{pmatrix} 1 & 2 \end{pmatrix}\begin{pmatrix} 2+2 \\ 1+4 \end{pmatrix} \right)^{-1} \cdot 5 \\ &= \frac{1}{14} \cdot 5 = \frac{5}{14} \end{aligned}$$

so the most expensive part in this computation is the matrix vector multiplication. The work per iteration will be dependant on this multiplication. If sparse matrix then we have $O(n)$. Using our update formula now to get a hopefully better approximation of the solution.

$$x_1 = x_0 + \alpha_0 r_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \frac{5}{14}\begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} \frac{19}{14} \\ \frac{5}{7} \end{pmatrix}$$

this is our first approximation. Remember the true solution is $\begin{pmatrix} 1 & 1 \end{pmatrix}$. Continuining the next iteration then.

$$r_1 = \begin{pmatrix} 3 \\ 3 \end{pmatrix} - \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} \frac{19}{14} \\ \frac{5}{7} \end{pmatrix} = \begin{pmatrix} -\frac{6}{14} \\ \frac{3}{14} \end{pmatrix}$$

note that the residual is now smaller, which is good since we are approaching 0 for the residual, etc. etc.

Note: In reality we would have some while loop checking if residual $>$ tolerance.

12.1.1.1. Will it converge?

Is

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

SPD?

Symmetric? Yes since $A^T = A$. Is it PD? Then we must have

$$\begin{aligned} x^T A x &= \begin{pmatrix} x_1 & x_2 \end{pmatrix}\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ &= \begin{pmatrix} x_1 & x_2 \end{pmatrix}\begin{pmatrix} 2x_1 + x_2 \\ x_1 + 2x_2 \end{pmatrix} \\ &= x_1(2x_1 + x_2) + x_2(x_1 + 2x_2) \\ &= 2x_1^2 + x_1 x_2 x_2 x_1 + 2x_2^2 \end{aligned}$$

rewriting to guarantee that we have PD.

$$(x_1 + x_2)^2 + x_1^2 + x_2^2$$

this is always positive unless $x = 0$. So now is SPD. It should then converge.

12.1.1.2. How fast does it converge?

$$\|e_k\| \leq \left(1 - \frac{2}{\kappa(A)}\right)^k \|e_0\| \quad \text{for large} \quad \kappa$$

Large condition number will cause very slow convergence. Ie. convergence depends on the condition number. This isn't super great. It is infact quite slow for large condition numbers. It is however simple and is therefore easy to implement. In general it is faster than Jacobi iterations, bit hard to compare due to different conditions but remember that speed of convergence for Jacobi is $\rho(R) < 1$ for some relaxation matrix R.

A large condition number will cause zig-zagging in finding the path to the minimum points.

12.1.2. Conjugate gradient | CG

- This algorithm fixes the problem for large condition numbers.
- Idea is quite similar to steepest descent but just has a small fix for large conditions numbers.
- Krylov method, $L = K = K_m(A; r_0)$ where $K_m(A; r_0 =)$ is called the Krylov space. Where

$$K_{m(A;v)} = \text{span}\{v, Av, A^2v, A^3v, ..., A^{m-1}v\}$$

so

$$V = \begin{pmatrix} \vdots & \vdots & ... & \vdots \\ r_0 & Ar_0 & ... & A^{m-1}r_0 \\ \vdots & \vdots & ... & \vdots \end{pmatrix}$$

so every iteration we expand the space we are operating in. Ie. column vectors are added for each iteration. To make sure these elements are less dependant we need to do Gram-Schmidt.

The update formula is quite similar to steepest descent:

$$x_{k+1} + x_k + \alpha_k p_k$$

so the direction we are headed to in the descent isnt just the residual.

$$\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$$

where in the denominator we have the new p_k instead.

$$p_k = r_k + \beta_{k-1} p_{k-1}$$

where

$$\beta_{k-1} = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}}$$

It is the term $\beta_{k-1} p_{k-1}$ that has memory of old-steps to prevent zig-zagging.

12.1.3. The algorithm for the lab

```
x0 = initial guess
r0 = b - Ax0
p0 = r0
WHILE norm(residual) > tolerance:
    alpha_k = (r_k^T r_k)/(p_k^T A p_k)
    xnext = xk + alpha_k pk
    rnext = rk - alpha_k A pk
    betak = (rnext^T rnext)/(rk^T rk^T)
    pnext = rnext + betak pk
    k++
```

12.1.3.1. Convergence

$$\|e_k\| \leq \left(1 - \frac{2}{\sqrt{\kappa(A)}}\right)^k \|e_0\|$$

so only difference is the square root. So, this is faster than steepest descent for big condition numbers.

- *Note*: Convergence is guaranteed in at least n iterations if A is $n \times n$. Often in much fewer.
- *Note*: The work per iteration depends largely on the matrix vector multiplication. It is $O(n^2)$ for dense matrices, $O(n)$ for sparse matrices.
- *Note*: CG is the go-to method for SPD matrices while some other Krylov methods are better for non-SPD matrices. Also, Jacobi and Gauss-Seidel mostly used for preconditioning. Ie

$$PAx = Pb$$

with some preconditioner matrix P. The point is that $\kappa(PA) \ll \kappa(A)$ but P is still relatively easy to find.

13. Lecture - Methods for eigenvalue problems

Have

$$Ax = \lambda x$$

There can be many solutions here. So, written more correctly

$$Ax_i = \lambda_i x_i$$

where $x_i =$ eigenvector and λ_i = eigenvalues. Can order through assumption

$$|\lambda_1| \geq |\lambda_2| \geq ... \geq |\lambda_n|$$

simply for more easily saying the smallest or the biggest eigenvalues. So we say $|\lambda_1| =$ largest eigenvalue. The algorithm that finds the largest eigenvalue is power iterations.

- Note: Power iteration requires us to have $|\lambda_1| > ... > |\lambda_n|$.

13.1. Power iteration

Assume A is diagonizable, meaning

$$A = PDP^{-1}$$

where $D = \text{diag}(\lambda_1, \lambda_2, ..., \lambda_n))$ and P has eigenvectors as columns. Do note that if A is SPD then A is also diagonizable.

$$A^k = (PDP^{-1})^k = (PDP^{-1})(PDP^{-1})...(PDP^{-1})$$

k times. This becomes

$$A = PD^kP^{-1}$$

through $PP^{-1} = I$. Multiplying by P from the right we get

$$A^kP = PD^k$$

getting rid of the last P^{-1}. Writing a vector

$$x = P\tilde{x}$$

using this in the problem from above we get

$$A^kx = A^kP\tilde{x} = PD^k\tilde{x} = \sum_{j=1}^{n} p_j\lambda_j^k\tilde{x}_j$$

which further simplifies to

$$\lambda_1^k \sum_{j=1}^{n} p_j \left(\frac{\lambda_j}{\lambda_1}\right)^k \tilde{x}_j$$

as $k \to \infty$ we will get only the maximum left, ie. the term outside the sum. Since λ_1 is the largest eigenvalue the fraction will go to 0 as $k \to \infty$ for $j \neq 1$. This means

$$A^kx \approx \lambda_1^k p_1 \tilde{x}_1$$

Normalization to get a vector pointing in the same direction as the eigenvector. Let that vector be

$$x' = \frac{A^kx}{\|A^kx\|}$$

As iteration:

```
x_0 = initial guess
WHILE error > tolerance DO:
    xnext = (A^k xnow)/(norm(A^k xnow))
    k++
```

How to find λ_1 from x_k? We have

$$\lambda_1 = \frac{x_k^TAx_k}{x_k^Tx_k}$$

Assumptions are as follows

- A diagonizable, eg. SPD matrix
- $|\lambda_1| > |\lambda_2|$
- Your guess $\tilde{x}_1 \neq 0$ as this will simply lead to a nullification of the entire process.

13.2. QR-algorithm

- If not just interested in largest eigenvalue, then do this algorithm. This algorithm gives us all the eigenvalues and is as such more expensive.

QR decomposition:

$$A = QR$$

where R is upper-triangular.Q is orthogonal, ie. $Q^TQ = I$. Do this via Gram-Schmidt.

Useful for finding eigenvalues:

- Least square problems

The algorithm of QR algorithm is as follows

1. $A_k = Q_kR_k$
2. $A_{k+1} = R_kQ_k$
3. Repeat 1.

$$A_{k+1} = R_kQ_k = Q_k^{-1}Q_kR_kQ_k = Q_k^TQ_kR_kQ_k = Q_k^TA_kQ_k$$

since an orthogonal matrix then A, A_{k+1} will have same eigenvalues but different structure by some rule. What happens is that you get a matrix with eigenvalues on the diagonal which you can simply just pick out.