Lecture 1

Objective : Design efficient Algorithms, numerical techniques.

> MATLAB will be use usually for assignments !

Topics :

- Computer numbers and arithmetic
- **MATLAB** (*Interactive environment. Easy to learn, code and debut !*)
    1. **MATLAB** has many toolboxes so in that respect it's better than Python
    2. No types !!
    3. You can use **MATLAB** apparently ?
- Solving linear systems of operations
- Solving nonlinear equations
- Solving ODES.

We want to solve $Ax = b$, where A is a matrix

$$A = \begin{bmatrix} a_1 1 & a_1 2 & \dots a_1 n \\ a_1 1 & a_1 2 & \dots a_1 n \\ \vdots & & \\ a_1 1 & a_1 2 & \dots a_1 n \end{bmatrix} \in \mathbb{R}^{n \times n}$$

and b is a column vector . *Note that n can be very large, thence we need numerical techniques.*

> If A is nonsingular then $Ax = b$ has a unique solution. which is $x = A^{-1}b$

Motivation to solve the matrix

- This system of equations come sin many applications (e.g. *PDEs*)

$f(x) = 0$, where x is a scalar an df is a scalar valued function.

For instance we want to solve

$e^x - \sin x = 0$, here we have a non linear function ! We have some techniques to solve that. Respectively, they have advantages and disvantages.

> We will use iterations iterative methods for solution of a local solution
>
> - Some may converge faster than others so we must do some analysis.

In data analysis , we have

$y_i = f(x_i),\ i = 0, 1, \dots, n$

we don't know this function but we know $\exists$ a relationship between (x_i, y_i). If we want to estimate f we use polynomial fitting to get the x values. We want to use the smallest polynomial function possible.

WE can then use this polynomial to evaluate the x_is.

> we can actually use polynomial interpolation (*Sprine ?*).

We can also use interpolation. It does not make sense to fit all the points exactly since the y_is have inherent errors ! In this case ewe use A Least-Squares Approximation.

[Definition]= GPS.

- We have a signal on the earth and some satellite (SV_i and $$). We can measure the *signal travelling time* from the reciever, from which we can deduce the distance d_{i}.

Given

$$\text{Receiver on the ground } \begin{pmatrix} x_r \\ y_r \\ z_r \end{pmatrix}$$

$$\text{and the satelite in the sky at } \begin{pmatrix} xS^i \\ yS^i \\ zS^i \end{pmatrix},$$

$$\text{WE have the equation } d_i = \sqrt{(x_r - x^{S_i})^2 + \ldots}, \text{with } i = 1, \ldots n$$

we need actually n satellites for the best estimation. Here we solve a nonlinear lest squares solution

$$\text{we take } \min \sum_{i=1}^{n} (d_i - \sqrt{\ldots})^2$$

thence, we minimize the distance. We *square* the expression since it's a least square, and *Least* because we minimize !

[Note] We need at least 4 satellites for accuracy ? Not sure why.

$$d_i = \sqrt{(x_r - x^{S_i})^2 + \ldots} + cdt, \text{where } c \text{ is the speed of light}$$

because of relativity, so we have

> $$\text{we take } \min \sum_{i=1}^{n} (d_i - \sqrt{\ldots} - cdt)^2$$

> This is important in research because we need very accurate measurements for GPS. SO we solve this model : **Integer Least Squres Problem**.

We will also do Numerical Integration

> $$\int_a^b f(x)\, dx$$

, in practice we cannot solve all the integrals by hand, or find their antiderivatives. For instance in statistics a density function of a population ? Basically can't be don analytically.

Last topic is ODEs

we consider a simple case

> $$f(x) = \begin{cases} x'(t) = f(t), x(t)) \\ x(t_0) = x_0 \end{cases}$$

, here we cannot find $f(t)$ analytically. Instead we will find approximations for $x(t_i)$ for $i = 0, 1, \ldots n$. We divide in intervals $[a, b]$.

Course Outline

- The main textbook is Numerical and Mathematical and Computing form McGill's website or store.
- WE have available course notes online

Grading

- No late submissions are accepted.
- Any requests have to be made within 10 days.
- 6 assignments on **Crowdmark** for 60%
- Midterm 20%
    1. Thursday Oct 14 6-8 pm
- Final 60%

Note, both exams will be in person and closed book.

Introduction to this course

Numerical computing is as old as civilization itself. E.g..., prediction of eclipses.

- Modern NC began with Isaac Newton, his calculus invetion motivation was to solve numerical problems.
- Until 20th century we use pen and paper only. The abacus appeared in the east
- IN the first half of the 20th century we found the slide rule which made multplication easier, but gave only *3 digits of accuracy*.
- IN 1950, invention of the electronic computer in the 40s, brought a new era of NC.

> Many mundane things such as a Google search, involves numerical computing

- In science and atmospherics (*PDEs*), we use a lot of NC
- Also important in many areas of computer science: animation, data mining, machine learning, simulations, robotics, computer vision, etc..

Lecture 2

> - Assignment 1 has been posted on **Crowdmark**
> - TAs zoom links has been posted on MyCourses
> - 2 TAs are available for each office hours

Error: Unknown environment 'pamtrix'

Computer Numbers and Arithmetics

Today:

1. binary and decimal representation of real numbers2.
2. computer representation of integers and non-integral real numbers
3. **IEEE** floating point representaiton

$$\text{real numbers} = \begin{cases} \text{rational number} \begin{cases} \text{integers} \\ \text{nonintegral fractions} \end{cases} \\ \text{irrational numbers} \end{cases}$$

***Rational numbers* : all the real nubmers which consist of a ratio of two integers e.g** $2/1, 1/3, \ldots$

There are two basic systems :

- the **decimal** (base 10)
- The **binary** (base 2)

- **Integers**

  The decimal and binary representation of integers requires anexpansion of nonenegative powers of the base

  $$(71)_{10} = 7\times 10 + 1$$

  it's binary equivalent : $(1000111)_2$ is

  $1\times 2^6 + 0\times 2^5 \ldots$

- Both represntations are finite :

$$\frac{11}{2} = (5.5)_{10} = 5\times 1 + 5\times \frac{1}{10}$$

- The decimal is finite, but the binary is infinitely long
- Both represntations are infinite and repeating

  $\frac{1}{3} = (0.333\ldots)_{10} = (0.010101\ldots)_2$

  If the representation of a rational number is infinite, it must be repeating

  is i t possible that the decimal represntatio nis infinte, but the binary repesentation is finite ? **NO !*

- We can do multiple sorts of conversions **Binary** $\rightarrow$ **decimal** (straight forward !).
- **Decimal** $\rightarrow$ **binary** : more complicated
- convert the integer and fractional parts separately.

We want to find $a_0, a_1, \ldots, a_n$ all 0 or 1 such that

$$(x)_{10} = a_n a_{n-1}, \ldots, a_0)_2$$

- **[Example]** let 71
- Divide 2 by 71 , remainder $a_0 = 1$.
  Continue this process divide 2 by 35 let $a_1 = 1$
- $\ldots$ divide 2 by 2 with $a_4 = 0$, finally
- divide 2 by 1 twith $a_6 = 1$ so then
  $71 = (1000111)_2$
- We can have a faster way
  $71 = 2^6 + 2^2 + 21 + 2^0 = (1000111)_2$

Q what is a similar approach for decimal fractions ?

Let $0 < x < 1$ then

$$\begin{aligned} x &= (0 \times a-1 * a-2 * \cdots * a-n-)_2 \\ &= \frac{a-1}{2} + \frac{a-2}{2^2} + \cdots + \frac{a-n}{2^n} + \ldots \\ 2x &= a-1 + \underbrace{\frac{a-1}{2} + \cdots + \frac{a-n}{2^{n-1}} + \ldots}_{<1} \end{aligned}$$

Here $a-1$ is the **integral part of** $2x$.

[Example] Suppose $x = 0.625$. We want to use a power represnatation

$$\begin{aligned} x = 0.625 \to 2x = 1.250 \quad &\text{with} \quad a-1 = 1 \\ 4x = 0.50 \quad &\text{with} \quad a-2 = 0 \\ 8x = 1.0 \quad &\text{with} \quad a-3 = 1 \end{aligned}$$

Thus, $0.625 = (101)_2$.

Computer represntations of numbers

- Integers
  1. sign and modulus
  2. 2's complement represntation
- Non-integral real numbers
  1. fixed point (*efficient but serious short coming*)

     > we usually don't use this method, we'll see later why

  2. floating point (*we usually use this !*)

Sign-and-modulus Approach

Use 1 bit to represent the **sign** and store the binary representation of the magnitude of the integer. Exmaple for 71

0	$00\ldots0100011$

Q: what is the largest magnitude which fits a 32 bit word ?

0	$11111\ldots1111$

The largest magnitude is $2^{31} - 1$. $\to$ why though ?

above if we add 1 to the end we get

0	$10\underbrace{\ldots\ldots\ldots\ldots\ldots}_{31}00 = 2^{31}$

> At this point we can explain what cause the explosion of the 64 bit rocket

The rocket required the conversion of a 64 bit but a 16 bit was assigned. The largest number which can fit a 16 bit word is

$$2^{15} - 1 = 32757$$

But the number was alrger than 32767, so the conversion failed and thus the rocket exploded.

---

2. $2's$ complement representation (CR)

> this method is more convenient and used by most machines.

- (i) The nonnegative integers 0 to $2^{31} - 1$ are stored as before , as a bitstring
- (ii) A negative integer $(-x)$ where $1 \leq x \leq 2^{31}$ is stored as the **positive integer**

$$2^{32} - x$$

e.g -71 is stored as the bit string $111\ldots10111001$

There is a way of converting x to the 2's **CR** $2^{32} - x$ of $-x$.

$$2^{32} - x = (2^{32} - 1 - x) + 1$$
$$2^{32} - 1 = (111\ldots111)_2$$

Basically change all 0 bits of x to 1s, 1 bits to 0 and 1.

> Suppose I know the complement representation of $111\ldots10111001$, I want to get the $2's$ representation. How do I do it ?
>
> $\rightarrow$ we can substrcct 1 and flip $0 \leftrightarrow 1$, but this has a short coming, because if the last one is 0.
>
> The answer is to change all from $0 \leftrightarrow 1$ and then add 1 !

Indeed, I know $x = 2^{31} - 1 - (2^{32} - x) + 1$. We know the binary representation of $2^{32} - x$. (we change 0 to 1 and 1 to 0) and then add 1 $(+1)$.

Q what is a quick way of deciding if a number is negative or nonnegative usign 2's CR ?

Advantage of 2's complement representation

-> See slides for comprehension.

Computer representation of nonintegral real numbers

- Real numbers are *approximatively* stored using the binary represntation of the number
    1. fixed point method
    2. Floating point method

For the fixed point : We divide into three fields

- the sign of the number
- the number before the point
- the number after the point

In a32 bit word, with fields width s of $1, 15$ and 16 the number $11/2$ would be stored as

0	0000000000001010	1000000000000000

The fixed point system has a severe limitation on the size of the numbers to be stored. Smallest magnitude is 2^{-16} and largest magnitude is $2^{15} - 2^{-16}$.

> A fix point system is inadequate for most scientific computing. But is fast and is used in some real=time applications.

Q how do we get from LHS bit representation to $2^{15} - 2^{-16}$? Same method as before, we add 1 and change all occurences

---

Explanation of the Patriot Missle Failure (1991)

> lesson don't use fixed point system because unevitable errors that are bieng produced by chopping a binary part.

IEEE Floating point standard

Motivations:

- was satandard use by 50s
- during the subsequent decades different manufacturers had different floating points so programs were not scalable across different machines, which cause difficulties
- in the 1980s some guys created a standard for computer manufacturers to obey,

In this course , the "IEEE standard" refers to the binary standard

Lecture 3

Recap

- Binary and decimal representation of numbers
- computer representation of numbers **Integer** and **non-integer** real nubmers

The ieee standard has 3 binary floating point basic formats:

- **binary32** : single format , single precision ;
- **binary64** : double formaant, double precision;
- **binary128**: quadruple format, quadruple precision

> most computers have the first 2 representations.
>
> In this class we mainly focus on **binary32** because then it is easy to understand he next 2

Any non zero x binary numbers can be written as

$$x = (-1)^s \times m \times 2^E. \qquad \text{where } 1 \leq m < 2,$$

Single format

S	$a_1a_2a_3 \dots a_8$	$b_1b_2b_3 \dots b_23$

- **sign field** 1 bit for s
- exponent field 8 bits
- significant field

[Definition] A number is called (computer) floating pint number if it can be stored exactly this way , e.g.,

$$71 = (1.000111)_2 \times 2^6$$

can be represented as

0	$E = 6$	**00011100000000000000**

> if x is not a floating point number ,it must be roundeded vefore it can be stored on the computer

Special numbers

- 0 : zero cannot be normalized
- -0 : -0 and 0 are two different representations for the same number
- ∞ this allows e.g $1.0/0.0 \to \infty$ instead of terminating with an **overflow** message.
- $-\infty$: $-\infty$ and ∞ areporesent two *very different* numbers
- NaN , or "not a nubmer and is the **error patter**".
- Subnormal numbers (we'll see later)

if exponent is $a_1 \dots a_8$ is	then value is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1 \dots b_{23})_2 \times 2^{-126}$
$\vdots$	$\vdots$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1, \dots, b_{23} = 0$; NaN otherwise

> the exponent representation uses **biased representation** this bitstring is the binarty representation of $E + 127$

Note also that exponent range from

$$E_{\min} = -126 \text{ to } E_{\max} = 127$$

- The smallest postitive number is

  $$(1.000 \dots 0)_2 \times 2^{-126}$$

  approx 1.2×10^{-38}
- THe largest normal positive number is

  $$(1.111 \dots 1)_2 \times 2^{127}$$

  approximatively 3.4×10^{38}

> screw fixed point systems this system stores way way more !!!

Last line:

If exponent $a_1 \dots a_8$ is	Then value is
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1, \dots, b_{23} = 0$; NaN otherwise

Largest is 255.

Nan has two types

- quiet NaN (qNaN) if $b_1 = 1$
- signaling Nan (sNaN) if $b_1 = 0$

First line

$(00..00)_2 = (0)_{10}$	$\pm(0.b_1..b_{23})_2 \times 2^{-126}$

How can I know if b_0 is stored as 0 or 1 since we can't store it ? You look at the exponential field if they are all 0 then we know b_0 is 0.

> - if alll the bit in the exponent fields are 0 then b_0 is 0.
> - if all the bits int the exponent field are 1 then we have $\pm\infty$ or NaN

zero is stored as

0	**000000**	**00000000000000000000**

If exponent is **zero** but fraction is nonzero the number represented is subnormal

---

Although 2^{-126} is the smallest normal positive number, we can represent smaller **subnormal** numbers.
e.g. $2^{-127} = (0.1)_2 \times 2^{-126}$:

0	00000000	10000000000000000000000

and $2^{-149} = (0.0000\ldots01)_2 \times 2^{-126}$:

0	00000000	00000000000000000000001

This is the smallest positive number we can store.

> subnormal numbers cannot be normalized as this would give exponents which do not fit.

==[Note]]] Subnormal numbers are *less accurate* (less room for nonzero bits in the fraction)

Examples

- (i) how is 2 represented ??

  1 and 7 zeros, flushing part is all 0
- (ii) what is the enxt smallest IEE single precision number larger than 2 ?

  just add 1 to the flushing part
- What is the ga[between 2 and the first IEE single precision number larger than 2 ?

  $$2^{-23}\times 2 = 2^{-22}$$

  Explication

  $$\begin{aligned} 2 &= (10\ldots0)_2\times 2^1 \\ &+ (00\ldots01)_2\times 2^1 \\ \overbrace{2^1 \qquad (1+2^{-23}\times 2^1)}^{2^{-23}\times 2^1 = 2^{-22}} & \end{aligned}$$

  If $x = (b_0 b_1 \ldots b_2 3)\times 2^E$ then we have to add $00\ldots01_2\times 2^E$ and we get

  $$\overbrace{x \qquad (x+2^{-23}\times 2^E)}^{2^{-23+E}}$$

  what does 2^{-23+E} tell us ? The gap between 2 consectutive numbers becomes larger and larger, the x just extends. T

Precision Machine Epsilon

[Definition] (Precision) The number of bits in the siginficant is called the precision of the floating ppoint system denoted by p

In the single format , $p = 24$

[Definition] (Machine Epsilon) the gap betwween the number 1 and the next larger floating point number is called the machine epsilon of the floating poitn system denoted ϵ.

In the signle format , the number after 1 is

$$b_0 b_1 \ldots b_{23} = 1.00000000000000000001,$$

so $\epsilon = 2^{-23}$.

> We can further defined ouble and quadruple formats, which are very similar

in double format we use 11 bits for the exponent and 52 bits for the flushing part .

If exponent is $a_1..a_{11}$	Then value is
$(000..0000)_2 = (0)_{10}$	$\pm(0.b_1..b_{52})_2 \times 2^{-1022}$
$(000..0001)_2 = (1)_{10}$	$\pm(1.b_1..b_{52})_2 \times 2^{-1022}$
$(000..0010)_2 = (2)_{10}$	$\pm(1.b_1..b_{52})_2 \times 2^{-1021}$
$\downarrow$	$\downarrow$
$(01..111)_2 = (1023)_{10}$	$\pm(1.b_1..b_{52})_2 \times 2^{0}$
$(10..000)_2 = (1024)_{10}$	$\pm(1.b_1..b_{52})_2 \times 2^{1}$
$\downarrow$	$\downarrow$
$(11..101)_2 = (2045)_{10}$	$\pm(1.b_1..b_{52})_2 \times 2^{1022}$
$(11..110)_2 = (2046)_{10}$	$\pm(1.b_1..b_{52})_2 \times 2^{1023}$
$(11..111)_2 = (2047)_{10}$	$\pm\infty$ if $b_1, , b_{52} = 0$; NaN otherwise

(... similar for quadruple ...)

Single	`p = 24`	$\epsilon = 2^{-23} \approx 1.2\times10^{-7}$
Double	`p=53`	$\epsilon = 2^{-52} \approx 2.2\times10^{-16}$
Quadrule	`p=113`	$\epsilon = 2^{-112} \approx 1.9\times10^{-34}$

Rounding

Floating point data include

- ±0 subnormal
- normal FPNs
- $\pm\infty$ and NaNs.

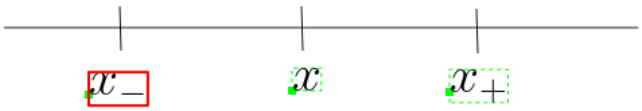
We define

- $N_{\min}$: the minimum positin nrormal FPN
- $N_{\max}$: the maximum positin nrormal FPN

[Question]

Let x be real number and $|x| \leq N_{\max}$. If x is not floating number, what are 2 obvious choices for approximation to x ?

- x_- : the closest FPN **less** than x
- x_+ : the closest FPN **greater** than x



Suppose $x = (b_0b_1 \ldots b_{23}b_{24} \ldots)_2 \times 2^E$ then

$$x_- = (b_0b_1 \ldots b_{23}) \times 2^E$$
$$x_+ = [(b_0b_1 \ldots b_{23})_2 + (0.00 \ldots 1)_2] \times 2^E$$

If positive round to x_-

[Definition] round$(x) = x$

- **Round down:** $\text{round}(x) = x_-$.
- **Round up:** $\text{round}(x) = x_+$.
- **Round towards zero:** $\text{round}(x)$ is either x_- or x_+, whichever is between zero and x.
- **Round to nearest:** $\text{round}(x)$ is either x_- or x_+, whichever is nearer to x. In the case of a tie, the one with its **least significant bit (the last bit) equal to zero** is chosen.

  This rounding mode is almost always used.

> [Rule] suppose same distance for x_- and x_+ w.r.t x. By convention we chose the one which has the last bit 0.

Rounding Errors

[Definition] (absolute rounding error) associated with x is

$$|\text{round}(x) - x|$$

> *we cannot store a real number exactly. We have to do rounding*

For all modes $|\text{round}(x) - x| < |x_+ - x_-|$. (*it is bounded* by the difference)

Suppose $N_{\min} \leq x \leq N_{\max}$,

$$x = (b_0.b_1 b_2 \dots b_{23} b_{24} b_{25} \dots)_2 \times 2^E, \quad b_0 = 1$$

IEEE **single** $x_- = (b_0.b_1 b_2 \dots b_{23})_2 \times 2^E$

IEEE **single** $x_+ = x_- + (0.00\dots01)_2 \times 2^E$

So for **any** mode

$$|\text{round}(x) - x| < 2^{-23} \times 2^E$$

where recall $\epsilon := 2^{-23}$.

In general for any rounding mode

$$|\text{round}(x) - x| < \varepsilon \times 2^E \quad (\star)$$

[Question] does $(\star)$ hold if $0 < x < N_{\min}$. i.e,., $E = -126,\ b_0 = 0$??

> answer is YES

Relative Rounding erro

[Definition] defined by $|\delta|$

$$\delta \equiv \frac{\text{round}(x) - x}{x}$$

Assuming x is in the normal range

$$x = \pm m \times 2^E, \qquad \text{where } m \geq 1$$

so $|x| \geq 2^E$

we get for all rounding modes (given that $|\text{round}(x) - x| < \varepsilon \times 2^E$)

$$|\delta| < \frac{\varepsilon \times 2^E}{2^E} = \varepsilon$$

> We derived indeed the upper bound is the machine precision !!! ε is the bound for any rounding modes.

> For single precision 2^{-23}, for double format 2^{-52} so we say the rounding error in double precision is much smaller than rounding error in single precision.

The relative rounding erro is bounded

$$|\delta| < \varepsilon$$

[Question] Does the bound necessarily hold if $0 < |x| < N_{\min}$?

> answe is NO since
>
> $$\delta = \frac{\text{round}(x)}{x} - 1 \qquad \forall\, x \in \text{ normal range}$$
>
> thus,
>
> $$\boxed{\text{round}(x) = x(1+\delta), \qquad |\delta| < \varepsilon}$$

^ we got this expression through some algebraic manipulation. equal to the true value of x times a factor. where delta is bounded by machine epsilon.

> This we hav eusually 7 accuracy digits when storing a real number on the computer. since
>
> $$\varepsilon = 2^{-23} \equiv 10^{-7}$$

Special case of round to nearest

In IEEE single, for all $|x| = (b_0.b_1b_2\ldots)_2 \times 2^E \le N_{\max}$,

$$|\text{round}(x) - x| \le 2^{-24} \times 2^E,$$

and in general

$$|\text{round}(x) - x| \le \frac{1}{2}\epsilon \times 2^E.$$

For x in the **normal range** (so $b_0 = 1$)

$$\text{round}(x) = x(1+\delta), \quad |\delta| \le \frac{\frac{1}{2}\epsilon \times 2^E}{2^E} = \frac{1}{2}\epsilon.$$

if x_- is close to x then the error is smaller than half the distance between x_- and x_+ (*logic*). We ge the relative error bound for delta in the above picture last line.

Lecture 4

We saw last time

$$\frac{|\text{round}(x)| - x}{|x|} \begin{cases} \varepsilon & \text{all rounding modes} \\ \le \frac{1}{2}\varepsilon & \text{round to nearest} \end{cases}$$

where $N_{\min} \le |x| \le N_{\max}$

[Example] (If x is not in the)

Suppose $x = 2^{-150}$, we know the smallest positive subnormal number is 2^{-149}. x is between 0 and the 2^{-149} on a real line.

We know $\implies \text{round}(x) = 0$ (*rule : we round to that which the last bit in the flushing part is* 0).

So then

$$\left|\frac{\text{round}(x-x)}{x}\right| = \frac{2^{-150}}{2^{-150}} = 1$$

[Example]

Suppose

$$
\begin{aligned}
x &= 0.\underbrace{0010\ldots0}_{23}\underbrace{1}_{\uparrow})\times 2^{-126} \\
x_- &= (0.0010\ldots0)\times 2^{-126} \\
x_+ &= (0.0010\ldots01)\times 2^{-126} \\
\text{round}(x) &= x_- \\
\left|\frac{\text{round}(x)-x}{x}\right| &= \frac{2^{-24}\times 2^{-126}}{(2^{-3}+2^{-24})\times 2^{-126}} \\
&= \frac{2^{-24}}{2^{-3}+2^{-24}} > 2^{-23} \\
&:= \varepsilon
\end{aligned}
$$

> this is why we said subnormal numbers are accurate. That's what it means

---

Today we will cover

- Floating point operations
- Exceptional situations
- Floating point in `C`

Correctly rounded means rounded to fit the destination of the result, using rounding mode in effect

> suppose we are given 2 floating point numbers. Multiply x and y together, then the result is usually not a floating point number necessarily.
>
> > We must round according to the rounding model
> >
> > The computer multiplication is just a rounded value of the exact result

IEE Rule for a Floating Point Operation

Let $\oplus, \ominus, \otimes, \oslash$

The **IEEE rule** for the basic arithmetic operations is then precise

$$
\begin{aligned}
x \oplus y &= \text{round}(x+y), \\
x \ominus y &= \text{round}(x-y), \\
x \otimes y &= \text{round}(x\times y), \\
x \oslash y &= \text{round}(x/y).
\end{aligned}
$$

Therefore when $x+y$ is in the **normal range**

$$
\boxed{x \oplus y} = (x+y)(1+\delta), \qquad |\delta| < \varepsilon
$$

[Note]

$$
|\delta| \le \varepsilon/2
$$

[Example] Suppose a is a *FPN*. Is it true $2 \otimes a = a \oplus a$?

> my answer no because we truncate a decimal point in the multiplication

We apply the rules

$$
\begin{aligned}
2 \oplus a &= \text{round}(2\times a) \\
a \oplus a &= \text{round}(a+a)
\end{aligned}
$$

> So yes it holds mathematically lmao. There's no trick here.

[Example] Suppose a and b are FPNs and $a \le b$, is $a \le (a \oplus b) \oslash 2 \le b$?

$$2a \le a+b \le 2n$$
$$\underbrace{\text{round}(2a)}_{2a} \le \text{round}(a+b) \le \underbrace{\text{round}(2b)}_{2b}$$
$$2a \le a \oplus b \le 2b$$
$$a \le \underbrace{\frac{a \oplus b}{2}}_{\equiv (a \oplus b) \oslash 2} \le b$$
$$\therefore a \le (a \oplus b) \oslash 2 \le b \qquad \checkmark$$

> this inequality is true $\implies$ the claim is true

Format Conversion

The IEE standard requires support for correctly rounded format conversions :

- conversions between floating numbers

  > double format to single format will require rounding ?

- conversion between floating point and integer formats
- Rounding a floating number to an integral value (not an integer format)
- Binary to decimal and decimal to binary conversion

Exceptional Situations

> this part is though the standard is still evolving nowadays

Division by zero

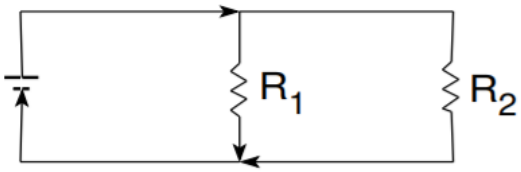
- give the largest FPN as a result.

  rationale: use would notice the large number in the output and conclude something had gone wrong.

- generate a **program interrupt** (*error message - fatal error division by zero*).

  This response is more reasonable but has a problem

[Example] (case where $1/0$ as fatal error fails to work)



the total resistance

$$T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}$$

what if $R_1 = 0$? the current will flow through and avoid the other ; therefore the total resistance in the circuit is zero !not fatal error message lol

in other words,

$$T = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0$$

Other uses of ∞

We used some of the following :

$$
\begin{array}{rcll}
a > 0 & : & a/0 \to \infty, & a * \infty \to \infty, \\
a \text{ finite} & : & a + \infty \to \infty, & a - \infty \to -\infty, \\
 & & a/\infty \to 0, & \infty + \infty \to \infty.
\end{array}
$$

Uses of NaN

- The operations $\infty \times 0, 0/0, \infty/\infty, \infty - \infty$ are indefinite. Computing any of these values results in a **invalid operation** and the IEE standard sets the result to NaN.
- A real operation with a complex result , e.g., the sqrt of a neg number $\to$ produces NaN
- Almost all arithmetic operations with at least one NaN produces a NaN
- sNaN generates interruption while qNaN does not. For instance GCC C compiler generates qNaN unless explicitly specified to behave the other way around.

> caveat NaN^0 , and other stuff in differnt programming languages

[Exmple] Suppose

$$
\begin{pmatrix} 1 & \mathrm{NaN} \\ 0 & 1 \end{pmatrix}
$$

what are the eigenvalue sof this values ? For this specific one form is it 1 or NaN lol ?

> In MATLAB we get a error message without a result.

Overflow and Underflow

Overflow is said to occur when

$$
N_{\max} < \left| \text{ true result } \right| < \infty
$$

where $N_{\max}$ is the **largest** normal FPN

> IN IEEE arithmetic, the standard response depends on the **rounding mode**. If round up ∞ if round down $N_{\max}$.

rounding mode	result
round up	∞
round down	$N_{\max}$
round towards zero	$N_{\max}$
round to nearest	∞

Rounding to nearest is the default rounding mode and any other choice may lead to very misleading final computational results.

Underflow is said to occur when

$$
0 < \left| \text{ true result } \right| < N_{\min}
$$

where $N_{\min}$ is the smallest normal floating point number.

- historically the response was : **replace the result by zero**
- In **IEEE standard the result may be subnormal number instead of zero**.

$$
0 \underbrace{---}_{a} S_{\min} - - x - - S_{\max} - - \underbrace{---}_{b} - N_{\min}
$$

Floating Point in `C`

```c
main ()     /* echo.c: echo input */
{
    float x;
    scanf("%f, &x");
    printf("x = %f", x);
}
```

The following result was for a Sun 4.

With input 0.66666666666666666666 :

```
#cc -o echoinput echo.c
#echoinput
0.66666666666666666666  (typed in)
x=0.666667              (printed out)
```

The two standard format codes used for specifying floating point numbers in these control strings are

- `%f` for **fixed decimal format**
- `%e` for **exponential decimal format**

Output format	Output
%f	0.666667
%e	6.666667e-01
%8.3f	0.667
%8.3e	6.667e-01
%20.15f	0.666666686534882
%20.15e	6.666666865348816e-01

in `%20.15f` and belov why didn't we get the input number back ($0.6666\ldots$), instead we get these weird decimals after $.666$

> no matter how may 5 we put in the input lmao it's in single format , it can only store like 6-7 accurate digits or something
>
> > so we cannot use the `%20.15f` and `%20.15e` formats for more accurate results, we only get garbage
> >
> > instead we must change the program, use something like `long float` ?

- the `scanf` routine calls a decimal to binary conversion routine to convert the input decimal format to internal binary floating point representation;
- the `printf` routing calls a binary decimal conversion routine to convert the binary floating point representation output decimal format
- both conversion routines use the rounding mode that is in teffect to correctly round the result.

A note on the `%f` format code

> this is not a good idea in numerical computing .
>
> Using the %f format code is **NOT** a good idea, unless
>
> it is known the numbers are neither too small nor too large.
>
> e.g., if the input is 1.0e-10, the output using %f is 0.000000.
>
> In numerical computing, usually the %e format code is used.
>
> does $1.0e-10$ represent ten to minus 7 9 1231 ? we don't know lol.

> So we use `%e` format in assignments in tthis course

Double or Long float

```
main ()
{
    double x;
    scanf("%f, &x");
    printf("%e",x);
}
```

same input in this program gives $-6.392091e-236$ which is wrong since we use `%f` format.

- when `scanf` reads a double precision variable we must use the format `%lf` for *long float* so that it stores the result in double format
- `printf` expects double precision, and single format variables are automatically converted to double before being passed to it.

Story : Effect of output format on a parliment election

Parliamentary election in Schleswig-Holstein, Germany, April 5, 1992.

- In the elections, a party with less than 5.0% of the vote cannot be seated.
- It was announced the Greens had a cliff-hanging 5.0% the evening of the election.
- It was discovered after midnight that they really had only 4.97%. The printout had one digit after the decimal point, and the actual percentage was rounded to 5.0%.

> this exmple indicates that we have to be careful about the output format

[Example]

```
main()
{
    float x; int n;
    n = 0; x =1;    /* x =2^0 */
    while ( x!= 0) {
        n++;
        x = x/2;        /* x = 2^(-n) */
        printf("\n n=%d x =%e" , n,x);
    }
}
```

Q: when does this program terminate ?

Initializes x to 1 and repeatedly divides by 2 until it rounds to 0

$$---0--^{2^{-150}}--\underbrace{2^{-149}}_{first\ sub?}---$$

answer : when $n = 150$ it will stop.

Thus x becomes $1/2$, $1/4$, $1/8$, ..., through **subnormal** 2^{-127}, 2^{-128}, ... to **the smallest subnormal** 2^{-149}.

The last value is 0, since 2^{-150} is **not** representable.

Output (various machines with various compliers):

```
n= 1   x=5.000000e-01
n= 2   x=2.500000e-01
. . .
n= 149   x=1.401298e-45
n= 150   x=0.000000e+00
```

Another test if x is zero

```c
main()
{
    float x, y; int n;
    n = 0 ; y =2 ; x =1 ;
    while(y!=1){
        n++;
        x-x/2;
        y=1+x;
        printf("\n n =%d x=%e y=%e", n,x,y);
    }
}
```

At what n does the program terminate ?

$$1 - - - (1 + 2^{-24}) - - - -(1 + 2^{-23})$$

> answer when $n = 24$.

Another test if x is zero

Now instead of using the variable y, change the **test**
`while (y != 1)` to `while (1 + x != 1)` :

```
n=0;   x=1;      /* x = 2^0 */
while(1 + x != 1){... /*same as before*/}
```

It stops at

- $n = 24$ on a PC using Visual Studio or online GBD, and Mac
  It uses registers with **the single precision** $p = 24$
- $n = 53$ on a PC using Visual C++
  It uses registers with **the double precision** $p = 24$
- $n = 64$ on a PC using gcc
  It uses registers with **the extended precision** $p = 64$.

> we get differnt values for n by running the same program on different machines (*different compilers*).

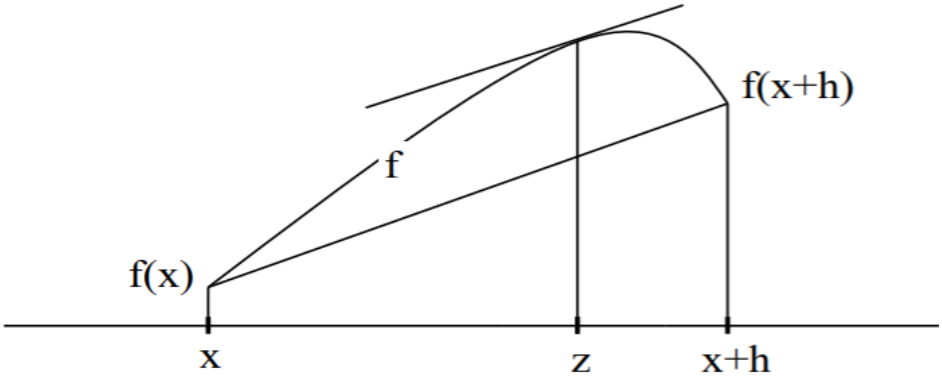
Lecture 5

[Theorem]

Let $f(x)$ be differentiable. For some $z \in [x, x+h]$,

$$\boxed{f'(z) = \frac{f(x+h) - f(x)}{h}}$$

This is *intuitively* clear from:



> the right hand side is just the slope which connects the two points $f(x)$ and $f(x+h)$.
>
> > we can find a point z such that the slope of the tangent line is equal to the slop of the line formed by $f(x)$ and $f(x+h)$, i.e., they are parallel, this is precisely the equality in the theorem formula of the Mean Value Theorem.

We can rewrite the equality of the MV formula as

$$f(x+h) = f(x) + hf'(z)$$

we can generalize if f is twice differentiable

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(z)$$

thus, the **Taylor Theorem**

$$f(x+h) = \sum_{k=0}^{n} \frac{f^{(k)}(x)}{k!}h^k + E_{n+1}$$

this produces the error (remainder)

$$E_{n+1} = \frac{f^{(n+1)}(z)}{(n+1)!}h^{n+1}, \qquad z \in [x, x+h]$$

> we can use the right hand side term as an approximation for E_{n+1}. I.e., the value of the function t a nearby point ?
>
> > we will also use this function later to approximate differential function point ?

We will use the taylor theorem a lot so it's important

If we take $n = \infty$ on the right hand side and assume the error term goes to zero then we have

the **Taylor series**

$$\boxed{f(x+h) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x)}{k!}h^k, \quad |h| < R}$$

[Example] For $f(x) = \sin x$

$$\begin{aligned}\sin(x+h) &= \sin(x) + \sin'(x)h + \frac{\sin''(x)}{2!}h^2 \\ &+ \frac{\sin'''(x)}{3!}h^3 + \frac{\sin''''(x)}{4!}h^4 + \cdots, \quad |h| < \infty\end{aligned}$$

Letting $x = 0$, we get

$$\sin(h) = h - \frac{1}{3!}h^3 + \frac{1}{5!}h^5 - \cdots,$$

Taylor theorem to do approximations

if h is small, $f'x$

> reason to do this approximation : in some application , the function x is very complicated, not necessarily easy to compute it's derivative.
>
> > in some application , we don't have an expression for the expression, so we can only approximate given the values points of the function.

$$f'(x) \approx \underbrace{\frac{f(x+h) - f(x)}{h}}_{\text{forward difference}}$$

How good is the approximation ?

using the Taylor theorem, we get

$$\frac{f(x+h) - f(x)}{h} - f'(x) = \underbrace{\frac{h}{2}f''(z)}_{\text{discretization error}}$$

$$\frac{h}{2}f''(z) \xrightarrow{h \to 0} 0$$

> where we keep only the second degree, why ? Because we want to approximate , so we don't need the higher order terms

The discretization error is $\mathcal{O}(h)$, in this case since the discretization error is a function of order 1 in h.

[Example] We want to approximate the derivative of $f(x) = \sin(x)$ at $x = 1$. Compute the exact discretization errors for $h = 10^{-1} \ldots . 10^{-20}$.

We know the first derivative is $f'(x) = \cos(x)$.

```c
int main()
{
    int n; double x, h ,approx,exact,error;
    x = 1.0 ; h = 1.0; n = 0;
    printf("\n h exact approx error");
    while(n <20) {
        n++;
        h = h /10 ;
        approx - (sin(x+h) - sin(x)) / h;
        exact = cos(x);
        error = approx - exact;
        printf("...\n" , h , approx, exact, error);
    }
}
```

> here h gets lower and lower. Running the program yields

h	approx	exact	error
1.0e-03	5.398815e-01	5.403023e-01	-4.208255e-04
1.0e-04	5.402602e-01	5.403023e-01	-4.207445e-05
1.0e-05	5.402981e-01	5.403023e-01	-4.207362e-06
1.0e-06	5.403019e-01	5.403023e-01	-4.207468e-07
1.0e-07	5.403023e-01	5.403023e-01	-4.182769e-08
1.0e-08	5.403023e-01	5.403023e-01	-1.407212e-08
1.0e-09	5.403024e-01	5.403023e-01	5.254127e-08
1.0e-10	5.403022e-01	5.403023e-01	-5.848104e-08
1.0e-11	5.403011e-01	5.403023e-01	-1.168704e-06
1.0e-12	5.403455e-01	5.403023e-01	4.324022e-05
1.0e-13	5.395684e-01	5.403023e-01	-7.339159e-04
1.0e-14	5.329071e-01	5.403023e-01	-7.395254e-03
1.0e-15	5.551115e-01	5.403023e-01	1.480921e-02
1.0e-16	0.000000e+00	5.403023e-01	-5.403023e-01

> mistake, second column is approximation ???
>
> > [Remark] look at the error. Doe sit behave as it expected ? Yes it goes smaller and smaller. However, when h becomes too small $\sim 10^{-16}$, the approximate value becomes 0.0000, but when h is not too small, we can see $\sim 10^{-9} \rightarrow \sim 10^{-15}$ the error becomes larger and larger
>
> The best result is at $h \sim 10^{-8}$ where the error is the smallest at $\sim 10^{-9}$
>
> > when $h \sim 10^{-16}$ our approx.. has value 0 why ?

Explanation of Accuracy loss ($\sim 10^{-16}$ form above)

> We use double precision so we have 16 accurate digits

if $x = 1$ and $h \leq \epsilon/2 \approx 1.1 \times 10^{-16}$, then $x + h$ has the same numerical value as x so $f(x + h)$ and $f(x)$ cancel to git 0 and the quantity approximation has **no digits of precision**

$$---1\underbrace{----}_{1+h} \oplus ---- (1 + \epsilon)$$

$$\text{where } \epsilon = 2^{-52} \qquad \sin \underbrace{(1 + h)}_{=1?} = \sin (1)$$

But how to understand the line $h \sim 10^{-9} \rightarrow h \sim 10^{-15}$ where the error gets larger again, ?

> the values partially cancel

- When h is a **little** bigger than $\epsilon/2$, the values **partially cancel**. For example, suppose that the first 10 digits of $\sin(x + h)$ and $\sin(x)$ are the same. Then, even though $\sin(x + h)$ and $\sin(x)$ are **accurate to 16 digits**, the **difference** has only **6 accurate digits**.

this is called cancellation error caused by floating point operation

> in pure mathematics we consider only the theoretical error, but in computational mathematics we consider also the rounding error

- in summary using h too big means big discretization error while using h soo small means a big cancellation error

> for the function $f(x) = \sin(x)$, at $x = 1$ the best choice of h is about
>
> $$\boxed{10^{-8} \quad \text{or} \quad \sim \sqrt{\varepsilon}}$$

The cancellation phenomenon occurs when we do a subtraction of two nearly equal numbers and it is one of the main causes for deterioration in accuracy

---

Theoretical analysis of numerical cancellation

instead of correct values x and y (*from experiments for instance*, or previous computations), the computer works with two perturbed floating point numbers. In practice we compute not $x - y$ but : ()only for the normal range

$$\hat{x} = x(1 + \delta_1), \qquad \hat{y} = y(1 + \delta_2)$$

where the errors δ_i may be due to previous computations, experiments, etc

Suppose we want to compute $x - y$, we can only compute $\hat{x} - \hat{y}$, so the computed value is

$$(\hat{x} - \hat{y})(1 + \delta_3) \qquad , |\delta_3| < \varepsilon$$

Is $(\hat{x} - \hat{y})(1 + \delta_3)$ a good approximation for $x - y$? .

The relative error is :

$$\begin{aligned} &\left| \frac{(\hat{x} - \hat{y})(1 + \delta_3) - (x - y)}{x - y} \right| \\ &= \left| \frac{x(1 + \delta_1)(1 + \delta_3) - y(1 + \delta_2)(1 + \delta_3) - (x - y)}{x - y} \right| \\ &= \left| \delta_3 + \frac{x}{x - y}\delta_1 + \frac{x}{x-y}\delta_1\delta_3 + \frac{y}{x-y}\delta_2 + \frac{y}{x-y}\delta_2\delta_3 \right|. \end{aligned}$$

> look at the 4 terms in the last line
>
> $$\frac{x}{x - y} \qquad \text{and} \quad \frac{y}{x - y}$$
>
> , so even if the deltas are small these terms may be very large !!
>
> this is why we get cancellation erros

This suggests

$$|x - y| \ll |x|, \ |y|,$$

> when we do numerical computation we have to be careful. i.e., avoid substraction

---

[Example] How do avoid substraction of 2 FPNs

> use this trick in assignment 2 !!!

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

Given $a = 1, b = -1000, c = 1$

In sing;e precision we get

$$x_1 = 10000.0 \quad \text{very good} \quad , x_2 = 0, \ \text{completely wrong}$$

where the true solutions are

$$x_1 = 999.9998999999998, \quad x_2 = 0001000000010000002$$

[Reason] (why did we get a bad x_2)

$$\sqrt{b^2 - 4ac} \approx -b$$

there is a numerical cancellation error in computing $-b - \sqrt{b^2 - 4ac}$

> here a and c are very small but b is large in magnitude, for x_1 we get $-b + \ldots$ so no problem but we do a substraction for x_2 : $-b - \ldots$.
>
> We do substraction of 2 floating numbers $-b$ and $\sqrt{b^2 - 4ac} \sim -b$ (* since $a = c = 1 \downarrow$*)

How to avoid this problem ? Rationalization

$$\begin{aligned} x_2 &= \frac{(-b-\sqrt{b^2-4ac})(-b+\sqrt{b^2-4ac})}{2a(-b+\sqrt{b^2-4ac})} \\ &= \frac{2c}{-b+\sqrt{b^2-4ac}} \\ &= \frac{c}{ax_1} \end{aligned}$$

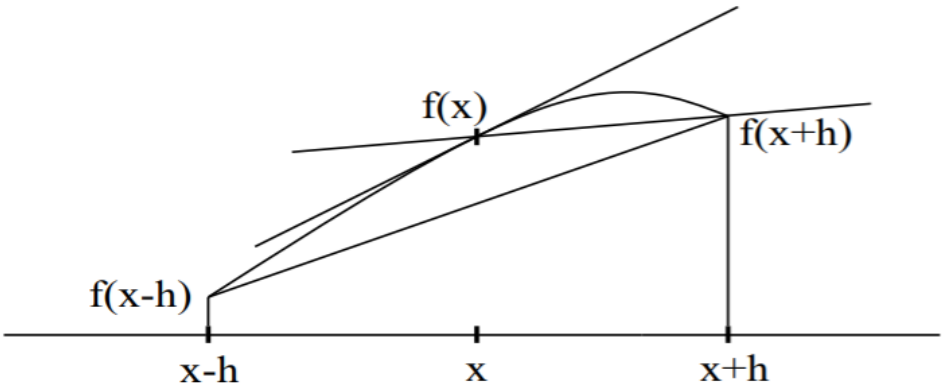
> we effectively truncate the 1000002 trailing part in $x_2 = 0.0001000000010000002$

[Question] Suppose we want to do $x = a + (b - c)$

> if b and c are close, we get cancellation error but if $a >> b \wedge c$ then the cancellation does not matter.
>
> Also if $b = c$ exactly, then no cancellation error obviously.

More accurate numerical differentiation

> can we do better in theory ?
>
> Yes



we said before the approximation

$$f'(x) \sim \frac{f(x+h) - f(x)}{h}$$

The slope of $f(x) \to f(x+h)$ connects the pooints $x \to x+h$ and the slope of the tangent line from $f(x) \to \infty$ is the first derivative. If the angle formed by the two then the two lines are not parallel and the approximation is not good.

the slope of the line from $f(x-h) \to f(x+h)$ is

$$\frac{f(x+h) - f(x-h)}{2h}$$

↑ this is a better approximation geometrically speaking given that the angle formed is smaller (*TODO review*)

So we introduce

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \qquad \text{Central difference formula}$$

Then, from **Taylor theorem,**

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(z_1),$$
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(z_2),$$

with z_1 between x and $x+h$ and z_2 between x and $x-h$.

Subtracting the 2nd from the 1st:

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{12}(f'''(z_1) + f'''(z_2))$$

Discretization error instead of $O(h)$. $\frac{h^2}{12}(f'''(z_1) + f'''(z_2))$ is $O(h^2)$

> note $z_1 \neq z_2$ necessarely in the third order term
>
> Note also that the discretization error is $\mathcal{O}(n^2)$

> [Note] we pick until the third order such that the second order terms cancel (*they are the same*), so we need the third derivative so when we do substraction we cancel them , so then we have the term
>
> $$\frac{h^2}{12}(f'''(z_1) + f'''(z_2))$$

In this case what's a good h ? we do theoretical analysis to justify it ?

Lecture 6

Introduction to MATLAB

> MATLAB stands for A Matrix Laboratory

- Interpreted language not compiled so slower
- Built in performance acceleration techniques exist though
- Can generate C code inside to be portable
- case sensitive
- When we create a MATLAB function the name of the file should match the name of the first function in the file.

Solving a linear system of equations

$$Ax = b$$

$A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, we assume A is non-singular.

We want to develop algorithms and theory for solving $Ax = b$

> Reference : Book C & K Chapter 2, Section 8.1

[Definition] *Norm of vectors and matrices*

> Norm is a measure of the size of a vector or a matrix.

[Definition] Vector Norm

Given a vector $v = [v_1\ v_2\ \dots v_n]^T \in \mathbb{R}^n$, we define

$$\| v \|_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} \text{ Euclidian length}$$
$$\| v \|_1 = \sum_{i=1}^{n} |v_i|$$
$$\| v \|_\infty = \max_i |v_i|$$

Suppose we have a vector $v = [4\ 3]^T$ on a x, y plane, then

- $\| v \|_2 = \sqrt{4^2 + 3^2} = 5$, this is the length of the vector on the plane

- $\| v \|_1 = 4 + 3 = 7$, this is the length of the x component plus the y component

  > Manhattan distance - because there the roads are in a grid, so the distance to walk from point A to point B is the $1-$norm

- $\| v \|_\infty = 4$, this is just the length of the x component in this particular case

[Definition] (*Matrix norm*)

"We measure the size of the matrix,"

Given $A \in \mathbb{R}^{m \times n}$,

$$\| A \|_p = \max_{i \neq 0} \frac{\| Ax \|_p}{\| x \|_p} \quad , p = 1, 2 \infty$$

> For different x we get different ratios so we get the maximum x hat is not 0, A we apply operator A to x , we change it's length, so this ratio shows how the matrix A changes the length of a vector.
>
> > basically how large of a change we can reach.

We can show

$$\| A \|_2 = (\text{largest eigenvalue of } A^T A)^{1/2},$$

where A is symmetric positive definite matrix

We can also show ,

$$\| A \|_1 = \max_j \sum_{i=1}^{m} |a_{ij}| \qquad \text{, maximal column sum}$$

$$\| A \|_\infty = \max_i \sum_{j=1}^{n} |a_{ij}| \qquad \text{, maximal row sum}$$

For 1-norm we consider the column sum, and the row sum for ∞-norm

[Example]

$$A = \begin{pmatrix} -2 & 3 & 4 \\ 5 & -1 & -7 \end{pmatrix} \qquad \begin{cases} \| A \|_2 & = \text{use MATLAB} \\ \| A \|_1 & = 4 + 7 = 11 \\ \| A \|_\infty & = 5 + 1 + 7 = 13 \end{cases}$$

> Trick to know which is column and which is row, the subscript 1 is a column l while the subscript ∞ is a row $--$.

[Definition] (*Forbenius norm*)

$$\| A \|_F = \sqrt{\sum_{i,j} \left| a_{ij}^2 \right|}$$

> note for the other norms we need the definition of a vector norm but nor for the Forbenius ?

For the example above,

$$\| A \|_F = \sqrt{2^2 + 3^2 + 4^2 + 5^2 + 1^2 + 7^2}$$

We Introduce

Gaussian elimination method for solving $Ax = b$.

[Example]

$$A = \begin{pmatrix} 3 & -1 & 2 \\ 9 & -1 & 13 \\ 6 & -12 & -26 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 28 \\ -50 \end{pmatrix}$$

We use Forward Elimination

> By making 9 and 6 0 we have eliminated the unknown x_1 , further we make -12 zero and we have eliminated x_2. Then we have an expression for x_3 and backtrack to find x_1 and x_2

$$\begin{matrix}\text{row } 2 - \frac{9}{3} \times \text{row } 1 \\ \text{row } 3 - \frac{6}{3} \times \text{row } 1\end{matrix} \longrightarrow \begin{pmatrix}3 & -1 & 2 \\ 0 & 2 & 7 \\ 0 & -10 & -26\end{pmatrix}\begin{pmatrix}x_1 \\ x_2 \\ x_2\end{pmatrix}\begin{pmatrix}5 \\ 13 \\ -60\end{pmatrix}$$

$$\text{row } 3 - \frac{-1}{2} \times \text{row } 2 \longrightarrow \underbrace{\begin{pmatrix}3 & -1 & 2 \\ 0 & 2 & 7 \\ 0 & 0 & 5\end{pmatrix}}_{\text{upper triangular}}\begin{pmatrix}x_1 \\ x_2 \\ x_2\end{pmatrix}\begin{pmatrix}5 \\ 13 \\ 5\end{pmatrix}$$

Notice now that the matrix is upper triangular

> [Remark] 3 and 2 on the main diagonal are called the pivot elements, they both appear on the factors of multipliers which are $9/3,\ 6/3,\ -10/2$.

Backward Substitution

From the last row of the reduced matrix, we get

$$5x_3 = 5 \implies x_3 = 5/5 = 1.$$

And then using the second row we get

$$2x_2 + 7x_3 = 13 \implies x_2 = (13 - 7x_3)/2 = (13 - 7 \times 1) = 3.$$

Finally, from the first equation we obtain

$$3x_1 - x_2 + 2x_3 = 5 \implies x_1 = \frac{5 + x_2 - 2x_3}{3} = \frac{5 + 2 - 2 \times 1}{3} = 2.$$

Let $A \in \mathbb{R}^{n \times n}$.

- **Forward elimination** $\rightarrow$ requires $n - 1$ steps, since we use 2 steps for a 3×3 matrix

  At the beginning of step k, we have

$$\begin{pmatrix}a_{11} & \dots & a_{1,k-1} & \dots & a_{1n} \\ & \ddots & \vdots & \vdots & \vdots \\ & a_{k-1,k-1} & a_{k-1,k} & \dots & a_{k-1,n} \\ & & a_{k,k} & \dots & \vdots \\ & & a_{i,k} & \dots & a_{kj} \\ & & \vdots & \dots & \vdots \\ & & a_{n,k} & \dots & a_{n,j}\end{pmatrix}\begin{pmatrix}b_1 \\ \vdots \\ b_{k-1]} \\ b_k \\ \vdots \\ b_i \\ b_n\end{pmatrix}$$

$$\text{row } i - \frac{a_{ik}}{a_{kk}} \times \text{row } k, \qquad i = k+1, k+2, \dots, n$$

$$m_{i,k} \leftarrow \frac{a_{ik}}{a_{kk}}$$

$$a_{i,j} \leftarrow a_{ij} - m_{ik} \times a_{kj}, \qquad j = k, k+1, \dots, n$$

Lecture 7

Last lecture we started solving $Ax = b$.

We talked about Gaussian Elimination

- Forward elimination $\rightarrow$ eventually A becomes an upper triangular matrix
- Back substitution

> We want to design a general algorithm.

Forward Elimination

> we need $m-1$ steps. Each step we zero the elements before the pivots ?
>
> At step k, we get

$$\begin{pmatrix} a_{11} & \dots & a_{1,k-1} & \dots & a_{1n} & & \\ & \ddots & \vdots & \vdots & \vdots & & \\ & a_{k-1,k-1} & a_{k-1,k} & \dots & a_{k-1,n} & & \\ & & a_{k,k} & \dots & a_{k,j} & \dots & a_{kn} \\ & & a_{i,k} & \dots & a_{ij} & \dots & a_{in} \\ & & \vdots & \dots & \vdots & & \vdots \\ & & a_{n,k} & \dots & a_{n,j} & \dots & a_{nn} \end{pmatrix}$$

$$m_{i,k} \leftarrow \frac{a_{ik}}{a_{kk}}$$

$$[a_{i,k} \dots a_{i,j} \dots a_{i,n}] - m_{i,k}[a_{kk} \dots a_{k,j} \dots a_{k,k}]$$

$$\text{i.e.,} \quad a_{i,j} \leftarrow a_{i,j} - m_{i,k} a_{k,j}, \qquad i = k+1, \dots, n, \ \text{and} \ j = k+1, \dots, n$$

[Q] why don't we use $[\dots a_{ik} \dots a_{kk}]^T$? Because this will become 0 !

We also need to update

$$b_i \leftarrow b_i - m_{ik} b_k$$

Now we look at backward substitution

After the first phase, we obtain

$$\begin{pmatrix} a_{11} & \dots & a_{ik} & \dots & a_{in} \\ & \ddots & a_{kk} & \dots & a_{kn} \\ & & & \ddots & \vdots \\ & & & & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_k \\ \vdots \\ b_n \end{pmatrix}$$

So we apply

$$a_{nn} x_n = b_n \implies x_n = b_n / a_{nn}$$

$$\vdots$$

$$a_{kk} x_k + a_{k,k+1} x_{k+1} + \dots + a_{k,n} x_n = b_k$$

$$\implies x_k = \left(b_k - \sum_{j=1}^{n} a_{kj} x_j \right) / a_{kk} \qquad , k = n-1, n-2, \dots 1$$

Gaussian Elimination with no pivoting

```
/* forward substitution */
for k=1:n-1
    for i = k+1: n
        m_ik <- a_ik / a_kk
        for j = k+1:n
            a_ik <- a_ij - m_ik * a_kj
        end
        b_i  <- b_i - m_ik * b_k
    end
end

/* backward substition */
x_n <- b_n / a_nn
for k=n-1:-1:1 /* MATLAB notation */
    x_k <-( b_k - sum from j = k+1 to n a_kj x_j / a_kk )
end
```

Computational cost analysis

[Definition] Flop

1 flop is one $+, -, \times, \div$

Let us do a cost analysis for the forward substitution algorithm.

```
for k=1:n-1                                  /* TOTAL : sum_{k=1}^{n-1}(1 + 2(n-k) + 2)(n-k)*/
    for i = k+1: n                           /* TOTAL : (1 + 2(n-k) + 2)(n-k) in for*/
        m_ik <- a_ik / a_kk                   /* 1 operations */
        for j = k+1:n                        /* TOTAL :  2(n-k) total in for */
            a_ik <- a_ij - m_ik * a_kj       /* 2 operations  */
        end
        b_i  <- b_i - m_ik * b_k             /* 2 operations */
    end
end

/* Note in the most outward loop we can't just multiply by n-1 since the inward loops
   actually depend on n and k !
*/
```

Now we do the cost analysis for backward substitution

```
x_n <- b_n / a_nn
for k=n-1:-1:1   /*Now since the variables change we have to do a summation. So we get
    sum_{k=1}^{n-1} (n-k + n-k -1 + 1 +1) */
    x_k <-( b_k - sum from j = k+1 to n a_kj x_j / a_kk ) /* the sum has n-k multiplications and
n-k-1 additions. we also have 1 substraction outside and 1 division. So then in total n-k + nOk
-1 +1 +1. */
end
```

The total cost is

$$\sum_{k=1}^{n-1}(1+2(n-k)+2)(n-k)+\sum_{k=1}^{n-1}(n-k+n-k-1+1+1)$$

We use the fact that

$$1+2+\cdots+n=\frac{1}{2}n(1+n)$$
$$1^2+2^2+\cdots+n^2=\frac{1}{6}n(n+1)(2n+1)$$

So then applying these relationships we conclude that

$$\text{Total} \approx \frac{2}{3}n^3 \text{ flops}, \qquad \text{ignore the lower order terms}$$

> we care about the coefficient because often the algorithm is often n^3, the difference is usually the coefficient, so if we want to improve the algorithm runtime cost we need to care about the coefficient

[Remark]

The cost of backward sub is actually exactly n^2, so here all the cost is in the forward substitution.

> The forward sub is dominant in the flop cost.

MATLAB Implementation

```
function x = genp(A,b)
% genp.m Gaussian elimination with no pivoting
% input:  A is an n x n nonsingular matrix
%         b is an n x 1 vector
% output: x is the solution of Ax=b.
%
n = length(b);
for k = 1:n-1
   for i = k+1:n
     mult = A(i,k)/A(k,k);
     A(i,k+1:n) = A(i,k+1:n)-mult*A(k,k+1:n);
     b(i) = b(i) - mult*b(k);
   end
end
x = zeros(n,1);
for k = n:-1:1
  x(k) = (b(k) - A(k,k+1:n)*x(k+1:n))/A(k,k);
end
```

We give the matrix A a vector b and try to find the solution x.

> - we first get the dimension `n = length(b);`
> - then we do the forward elimination process

> in MATLAB we want to avoid for loops when possible
>
> > Here we can improve the code in the inner-most loop

replace all j with $k+1:n$ and remove the for loop

so from

```
for j = k+1:n
    A(i,j) = A(i,j) - mult*A(k,j)
end
->
A(i,k+1:n) = A(i,k+1:n) - mult*A(k,k+1:n)

/********/
/* we can do even better , same thing for middle for loop !*/
for i = k+1:n
    mult = A(i, k)/ A(k,k)
    A(i,k+1:n) = A(i,k+1:n) - mult*A(k,k+1:n)
    b(i) = b(i) - mult*b(k)
end
->
mult = A(k+1:n, k)/ A(k,k)
A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - mult*A(k,k+1:n)
b(k+1:n) = b(k+1:n) - mult*b(k)

/** But this is hard to read, so define an i to cleanup*/
i = k+1:n
mult = A(i,k)/ A(k,k)
A(i,i) = A(i,i) - mult*A(k,i)
b(i) = b(i) - mult*b(k)

/** so in final we have */
n = length(b);
for k = 1:n-1
    i = k+1:n
    mult = A(i,k)/ A(k,k)
    A(i,i) = A(i,i) - mult*A(k,i)
    b(i) = b(i) - mult*b(k)
end
```

Now for the backward substitution

> - first initialize `x= zeros(n,1)` ; this sets x to $\vec{0}$. It's a column vector with n elements
> - When $k = n$ MATLAB will discard the matrix $A(k, k+1:n) * x(k+1:m)$ as empty. Used to be a problem back in time.

> Note
>
> - If $A(k,k) = 0$ then the algorithm breaks down we get a column vector of NaNs.
> - similarly, if we set the first entry of A to a very small number then $A(k,k)$ is very small, the algorithm doesn't break down but we get skewed poor solution

---

LU Factorization

GENP gives $A = LU$

- where U is the upper triangular matrix obtained by forward elimination
- and L is the lower triangular matrix

$$L = \begin{pmatrix} 1 & & & 0 \\ m_{21} & 1 & & \\ & & \ddots & \\ m_{31} & m_{32} & & \\ m_{41} & m_{42} & m_{43} & 1 \end{pmatrix} \quad m_{ik} \text{ multipliers conputed in the forward elimination.}$$

then we can use the LU decomposition to solve the linear system $Ax = b \implies LUx = b \implies Ly = b$, we do LU again on y lol

Lecture 8

Recap

We saw GENP to solve $Ax = b$.

1. Forward elimination

$$Ax = b \implies Ux = y$$

2. Backward substitution : solve

$$Ux = y$$

```
% Forward elimination process
for k=1:n-1
    i = k+1:n
    A(i,k) = A(i,k)/A(k,k)       - m_{ik}
    A(i,i) = A(i,i) - A(i,k)A(k,i)  %here A(k,i) is a row vector
    b(i) = b(i) - A(i,k)b(k)
end
```

where

- $A(i,i)$ is a matrix
- $A(i,i)$ is a matrix
- $A(i,k)$ is a column vector
- $A(k,i)$ is a row vector

```
% Backward substituion process
x = zeros(n,1)      % important step , we need to initialize this
for k=n:-1:1
    x(k) = (b(k) - A(k,k+1:n) x(k+1:n)) / A(k,k)
end
```

Here the cost of GENP is

$$\frac{2}{3}n^3 \quad \text{flops.}$$

LU Factorization

GENP finds the matrices $M_1, M_2, \dots M_{k-1}$ such that

$$\frac{M_{n-1} \dots M_2 M_1 A = U}{M_{n-1} \dots M_2 M_1 b = u} \implies A = \underbrace{M_1^{-1} M_2^{-1} \dots M_{n-1}^{-1}}_{L} U = LU$$

$$Ax = b \implies \underbrace{M_{n-1} \dots M_1}_{Ux=y} Ax = \underbrace{M_{n-1} \dots M_1} b$$

[Recall Example]

$$\begin{pmatrix} 3 & -1 & 2 \\ 9 & -1 & 13 \\ 16 & -12 & -16 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 28 \\ -50 \end{pmatrix}$$

We multiply A by the following matrix

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{10}{2} & 1 \end{pmatrix}}_{M_2} \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ -\frac{9}{3} & 1 & 0 \\ -\frac{6}{3} & 0 & 1 \end{pmatrix}}_{M_1} \underbrace{\begin{pmatrix} 3 & -1 & 2 \\ 9 & -1 & 13 \\ 16 & -12 & -16 \end{pmatrix}}_{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 7 \\ 0 & \frac{10}{2} & 1 \end{pmatrix} \begin{pmatrix} 3 & -1 & 2 \\ 0 & 2 & 7 \\ 0 & -10 & -30 \end{pmatrix}$$

$$= \underbrace{\begin{pmatrix} 3 & -1 & 2 \\ 0 & 2 & 7 \\ 0 & 0 & 5 \end{pmatrix}}_{U}$$

Second row times first column we get the first row in the result

> we want to make -10 zero so we need to find M_1 for that matter

From the above we get

$$M_2 M_1 \begin{pmatrix} 5 \\ 28 \\ -50 \end{pmatrix} = \underbrace{\begin{pmatrix} 5 \\ 13 \\ 5 \end{pmatrix}}_{y}$$

In general ,

$$M_k = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & -m_{k+1,k} & 1 & \\ & & \vdots & \ddots & \\ & & -m_{i,k} & & 1 \end{pmatrix}$$

where in the matrix above the row and line with the most entries corresponds to k index.

Easy to verify :

$$M_k^{-1} = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & m_{k+1,k} & 1 & \\ & & \vdots & \ddots & \\ & & m_{i,k} & & 1 \end{pmatrix}$$

> NOTE the invers matrix is the same but with a flipped sign
>
> > So their product is indeed the Identity matrix.

$$A = M_1^{-1} M_2^{-1} \dots M_{n-1}^{-1} U \qquad\qquad U$$

MATLAB Code

```
function [L,U] = lunp(A)
% lunp.m LU factorization (with no pivoting)
% input: A is an n x n nonsingular matrix
% output: Unit lower triangular L and upper triangular U
                    %%%%%% such that A = LU %%%%%%
%
n = size(A,1);
for k = 1:n-1
    i = k+1:n;
    A(i,k) = A(i,k)/A(k,k);          %
    A(i,i) = A(i,i) - A(i,k)*A(k,i);
end
L = tril(A,-1) + eye(n);     %built in function to get the strictly lower part of A
U = triu(A);                 %built in function to get the strictly upper part of A

% if we remove the -1 in tril(A,-1) then we don't get the diagonal elements
% eye is the identity matrix to get the diagonal entries
```

$$\begin{pmatrix} -2^{-n} & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$\text{when } n = 53, \qquad \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

So we get

$$n_4 = \frac{1}{-2^{-n}} \begin{pmatrix} -2^{-n} & 1 \\ 0 & 1 - \frac{1}{-2^{-n}} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 - \frac{1}{-2^{-n}} \end{pmatrix}$$

$$\text{Also,} \qquad \begin{cases} x_2 &= \dfrac{2 - \dfrac{1}{-2^{-n}}}{1 - \dfrac{1}{-2^{-n}}} \\ &= \dfrac{2^{1-n} + 1}{2^n + 1} \approx 1 \\ x_1 &= \dfrac{1 - x_2}{-2^{-n}} \\ &= \dfrac{1 - \dfrac{2^{1-n} + 1}{2^{-n} + 1}}{-2^{-n}} \\ &\approx 1 \end{cases}$$

Then ,

$$\begin{aligned} x_2^{\circ} &= (2 \ominus ((1 \oslash (-2^{-n})) \otimes 1)) \oslash (1 \ominus ((1 \oslash (-2^{-n})) \oslash 1)) \\ &= \underbrace{(2 \ominus -2^n)}_{\equiv \text{round}(2+2^{-n})} \oslash \underbrace{(1 \ominus (-2^n))}_{\equiv \text{round}(1+2^n)} \end{aligned}$$

Moreover ,

$$2 + 2^{-n} = 2 + 2^{53} = 2^{53}(1 + 2^{-52})$$
$$\text{round}(2 + 2^{53}) = 2^{53}(1 + 2^{-52})$$

Recall machine epsilon ε is 2^{-52} so next FPN of 1 is $1 + 2^{-52}$. So indeed the expression above includes a FPN

$$1 + 2^n = 1 + 2^{53} = 2^{53}(1 + 2^{-53})$$

> [Note] the number $1+2^{-53}$ is in the middle of 1 and the next FPN $1+2^{-52}$ so it is not actually a FPN !

So then this amounts to , given that we round to 1 (*rules for rounding when there is a tie , last bit* 0)

$$\text{round}(1+2^{53}) = 2^{53}$$

Now we look at x_1°

$$\begin{aligned} x_1^\circ &= (1 \ominus (1 \otimes x_2^\circ)) \oslash (-2^{-53})) \\ &= \frac{(1 \ominus x_2^\circ)}{2^{-53}} \end{aligned}$$

Whence

$$1 \ominus x_2^\circ = \text{round}(1 - x_2^\circ) = \text{round}(1-(1+2^{-52})) = 2^{-52}$$

We plug this back , and obtain

$$\boxed{x_1^\circ = \frac{-2^{-52}}{-2^{-53}} = 2}$$

If $n \geq 54$, we want to understand why the computed solution for this case is 0 or 1 ??

$$x_2^\circ = (2 \ominus (2^{-n})) \oslash (1 \ominus)(-2^n)$$

here

$$\begin{aligned} 2 \ominus (-2^n) = \text{round}(2+2^n) = \text{round}((2^n(1+2^{1-n}))) \\ \text{for } n \geq 54, \qquad \text{round}(1+2^{1-n}) = 1 \end{aligned}$$

So we conclude that the above expression is actually

$$2 \ominus (-2^n) = 2^n$$

So,

$$\begin{aligned} 1 \ominus (-2^n) = \text{round}(2^n(1+2^{-n})) \\ \text{for } n \geq 54 \qquad , \text{round}(1+2^{-n}) = 1 \end{aligned}$$

It follows that

$$1 \ominus (-2^n) = 2^n$$

Finally,

$$x_1^\circ = (1 \ominus \underbrace{x_2^\circ}_{=1}) \oslash (-2^{-n}) = 0$$

This is why we get the result by MATLAB. When $n \geq 54$ the computer outputs 0.

> MATLAB outputs at ≤ 52 , $x(1) = x(2) = 1.000\ldots0$, but at $n = 53$ $x(1) = 2.000\ldots0$, $x(2) = 1.00\ldots0$, but of course for $n \geq 54$ we get $x(1) = -0.00\ldots0$ and $x(2) = 1.00\ldots0$.

Lecture 3

Last time we saw in GENP that

> At step k of the forward elimination
>
> - if $a_{kk} = 0$ then the algorithm breaks down.
> - if $a_{kk} \neq 0$ but $|a_{kk}|$ is small then the solution will have poor accuracy.

To overcome this trouble we just multiply the GENP.

Indeed, at the k th step of the forward elimination we chose /find

$$|a_{qk}| = \max\{|a_{ik}|, \quad i = k, k+1, \ldots, n\}.$$

Then we do the interchange (swap of the 2 rows) :

$$a_{kj} \longleftrightarrow a_{qj} \quad , j = k, k+1, \dots n$$
$$b_k \longleftrightarrow b_q$$

This strategy is called Partial Pivoting because we chose the largest magnitude for a_{qk} along a_{ik}.

Complete computing (strategy)

> Usually we don't use this strategy though, too costy ?

```
%GEPP Algorithm
function x = gepp(A,b)
% genp.m GE with partial pivoting
% input: A is an n x n nonsingular matrix
% b is an n x 1 vector
% output: x is the solution of Ax=b.
%
n = length(b);
for k = 1:n-1
    [maxval, maxindex] = max(abs(A(k:n,k)));
    q = maxindex+k-1;
    if maxval == 0, error('A is singular'),
    A([k,q],k:n) = A([q,k],k:n);        %column from k to column n, for the preceeding columns we have zeros
    b([k,q]) = b([q,k]);        % here we interchange vectors b
     end
    i = k+1:n;
    mult = A(i,k)/A(k,k);
    A(i,i) = A(i,i) - mult*A(k,i);
    b(i) = b(i) - mult*b(k);
end
x = zeros(n,1);
for k = n:-1:1
    x(k) = (b(k) - A(k,k+1:n)*x(k+1:n))/A(k,k);
end
```

Last time we saw that we had bad results ??? So now if we use GEPP , we have a much more accurate solution.

> IN practice use GEPP to solve a linear system of equations

In GENP we know the L matrix is formed by the multipliers. In $A = LU$, the ik entry of L is chosen by $m_{ik}

LU factorization with partial pivoting

$$PA = LU$$

where

- $P = P_{n-1} \dots P_2 P_1$ with P_k being the permutation matrix used in the k th step of the forward elimination process (P_k is the matrix obtained by swapping rows k and q of the identity matrix I_n)

> In ass 3 we need to apply GEPP to get a solution for Linear System, by hand row swapping

The MATLAB program `lupp.m`, with modification `gepp.m` computes the LU factorization . Once this factorization is available , to solve $Ax = b$, we can solve two triangular system

$$Ly = Pb, \quad Ux = y$$

to obtain the solution x.

> We pay the price to get more accurate solutions but the cost is to check the row comparisons at each step
>
> At step k we do $a_{kk}, a_{k+1,k}, \dots$ so we need todo $n-1$ comparisons, so half of n^2 comparisons cost

Theoretical results for GEPP

Here we consider all norms $\|\cdot\|$ first second , ∞ norms etc.

- Suppose element sof A and b are FPNs, we can show that if we use GEPP, then we can find $E \in \mathbb{R}^{n \times n}$ such that the computed solution x_c satisfies

$$(A + E)x_c = bm$$

and usually (in 540 we'll se the other cares),

$$\|E\| \approx \epsilon \|A\|$$

so $A + E$ is close to A In other words the exact solution is a nearby problem we ay GEPP is usually numerically stable.

> Why do we look at nearby problem ? When we store A and b we have rounding errors !! and this error is about $\approx \epsilon \|A\|$.

[Question] This means the computed seolution fits a linear system quite well . But does this mean x_c has high accuracy ?? NO

- **Residual vector:** $r = b - Ax_c$. If x_c is the exact solution the $r = 0$. But usually $r \neq 0$. If the previous 2 equations hold we deduce
- $$\begin{gathered} \| r \| \lessapprox \epsilon \|A\| \cdot \|x_c\| \\ \frac{\|x_c - x\|}{\| x \|} \lessapprox \epsilon \|A\| \cdot \left\|A^{-1}\right\| \end{gathered}$$

[Definition] We define the condition number of $Ax = b$ linear system.

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

[Remark] we can show that $\kappa(A) \geq 1$

> The condition number depends on the matrix. If condition number is large then computed solution is very likely to have very poor accuracy.

> although GEPP is numerically stable, the computed solution may have poor accuracy since accuracy will also depend on the condition number of the problem, which has nothing to do with the algorithm.
>
> > when we use algorithm to solve a problem ,the accuracy of the computed solution depends on 2 things :
> >
> > 1. numerically stability
> > 2. conditioning of the problem

Remarks

- the size of the residual is usually very small compared to the product of the size of A and the size of x_c
- let $\epsilon \approx 10^{-t}$ and $\kappa(A) \approx 10^{p}$ then usually x_c has approximately $t - p$ accurate decimal digits. if $\kappa(A)$ is large we saty the problem is ill-condition
- If we use GENP then (2) doesn't hold

> Note MATLAB's default norm is $\| \cdot \|_2$

> Summary of key points
>
> 1. GEPP is good (numerically stable usually) , so relative residual is small (bounded by machine epsilon) so computed solution fits the linear system very well. The accuracy of result depends on the condition number. If problem is bad (ill condition) then x_c will have very poor accuracy (most likely)
> 2. when we solve LS the accuracy of the computed solutions depends on 2
>   1. numerical stab of the algorithm
>   2. condition number

[Q] suppsoe we have 2 algorithms to solve a problem , how do you know which algorithm is better than the other ?

check residual. If res is small then this algorithm is better than the other one. Just like in Academia lol.

[Q] get poor solution using an algorithm, should I say my algorithm is bad ?

No because the accuracy depends on the condition number as well. If relative residual is large then algorithm is bad.

Diagonally Dominant Matrices

[Definition] $A = (a_{ij})_{n \times n}$ is strictly diagonally dominant by column if

$$|a_{jj}| > \sum_{i=1, i \neq j}^{n} |a_{ij}| \quad , j = 1 : n \quad , \text{e.g} \quad \begin{pmatrix} -6 & 3 & 4 \\ 1 & 10 & -2 \\ -1 & 4 & 7 \end{pmatrix}$$

A is strictly diagonally dominat by row if

$$|a_{ii}| > \sum_{j=1, i\neq j}^{n} |a_{ij}| \quad , i = 1:n \quad \text{, e.g} \quad \begin{pmatrix} -6 & 3 & 2 \\ 1 & 10 & -6 \\ -1 & 4 & 7 \end{pmatrix}$$

$6 > 3 + 2$ also $10 > 1 + 6$ and $7 > 1 + 4$.

We can show

- if a tridiagonal A is strictly bounded by column then partial pivoting has the same effect as no pivoting ,i.e GEPP = GPP
- if a triadiagonal A is strictly bounded by row then GENP will not break down

[Q] why tridiagonal matrices ? most usual in this course ?

[Theorem] If a tridiagonal A is strictly diagonally dominant by row, then GENP will not break down.

$$Pf.\ A = \begin{pmatrix} d_1 & c_1 & & & \\ a_1 & d_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-2} & d_{n-1} & c_{n-1} \\ & & & a_{n-1} & d_n \end{pmatrix} \qquad \begin{aligned} |d_1| &> |c_1| \\ |d_i| &> |a_{i-1}| + |c_1| \\ |d_n| &> |a_{n-1}| \end{aligned}$$

> can d_1 be zero here ? NO because $|d_1| > |c_1|$ so d_1 is not zero

> after we make $a_1 = 0$ we don't need to to compute $m_{3,1}, m_{2,1}$ etc we just start next step

(see slides 3 missing lines here)

After step 1 the matrix becomes

$$\begin{pmatrix} d_1 & c_1 & & & \\ 0 & d_1' & c_2' & & \\ & a_2 & d_3 & c_3 & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-2} & d_{n-1} & c_{n-1} \\ & & & a_{n-1} & d_n \end{pmatrix} \qquad \begin{aligned} |d_1| &> |c_1| \\ |d_i| &> |a_{i-1}| + |c_1| \end{aligned}$$

$d_2 > a_1 + c_2$ (since strictly dominant row) so $d_2' > c_2'$. In first step we only changed d_2 on the main diagonal .

Lecture 10

Recap

Last time we talked about solving tridiagonal linear system

$$A = \begin{pmatrix} d_1 & c_1 & & & \\ a_1 & d_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-2} & d_{n-1} & c_{n-1} \\ & & & a_{n-1} & d_n \end{pmatrix}$$

```
// forward elimination process
for k = 1 : n-1
    mult <- a_k / d_l
    d_k+1 <- d_k+1 - mult * c_k
    b_k+1 <- b_k+1 - mult * b_k
end
// we get an *upper bidiagonal* a special matrix who has the main diagonal and a sub diagonal
for k = n : -1:1
    x_l <- (b_k - c_k * mult x_k+1) / d_k
end
```

> - the cost here is $8n$ flops and .Given the algorithm defined above, because for each k we have we have one division, one subtraction, one multiplication , one subtraction and one multiplication (*forward elimination*) and we need 3 flops in the backward substitution process
> - the storage here is a, b, c, d and x (can use b to store x)
>
> > we don't store the whole matrix since to many zeros, the matrix is tridiagonal. We store the diagonal, sub diagonal and super diagonal.

If we apply GEPP how do we solve this problem ?

MATLAB commands and functions

- the solution of $Ax = b$: we do $A\ b$ to solve this linear system , MATLAB here uses GEPP for this command
- The LU factorization there exists an inbuild function with partial pivoting : $\text{lu}(A)$

  > in computation there doesn't exist a no pivoting function apparently. This one is much better
- THe determinant of A : $\det(A)$.

We have $\det(A)$ with $PA = LU$. We use

$$\begin{aligned}\det(P)\det(A) &= \underbrace{\det(L)\det(U)}_{\text{lower } \triangle\ I\ \text{ upper } \triangle} \\ &= 1 \times u_{11}u_{22}\dots u_{nn}\end{aligned}$$

Given

$$P = P_{n-1}\dots P_1$$

If we do row interchange at the first slope then (interchange row 1 and row q of the identity matrix)

$$\det(P_1) = -1$$

So we have that

$$P = P_{n-1}\dots P_1 \implies \det(P) = \Pi_{k=1}^{n-1}\det(P_k) = (-1)$$

So then we can get $\det(A)$:

$$\det(A) = (-1)^{\#\text{ of permutations}}u_{11}u_{22}\dots u_{nn}$$

> In Ass.3 we are asked to compute $\alpha = c^T A^{-1} d$
>
> > Do we need to compute the inverse ? No since
> >
> > $Ax = b \implies x = A^{-1}b$ no need to compute the inverse. To compute the inverse we need to solve a sequence of systems lmao. We solve just $Ax = b$ using GEPP or linear factorization with pivoting !
> >
> > When we see the inverse of a matrix in some formula we should immediately think of pivoting !

If we really needed to solve, we solve $A\bar{x} = I$, in ass.3 $A\bar{x} = B$ and apply LEPP ??

Material for Mid term (14 October)

1. Computer numbers and arithmetic

1. conversion between decimal and binary representation
2. Computer representation of integers
   1. sign $ modulus (*simple*)
   2. 2's complement representation *(harder)*. know its advantageous. If I give number we should be able to give what's the 2s complement. Or vice verse this is 2s complement can you tell me what the number is?
3. Computer representation of real numbers
   1. Fixed point system , know the limitations of this system. serious limitation of size. In practice we dpnt use this system except in signal procession for some real time applications because it's fast. No need for big small number so fixed point can work, but usually we don't use this since we need large numbers or good accuracy.
   2. For the reason above we use Floating point system.

1. IEEE FPS. We introduced the tables for single , double, quadruble format.

$\pm$	$a_1 \dots a_8$	$b_{1\dots b_{23}}$

$\pm$	$a_1 \dots a_{11}$	$b_{1\dots b_{32}}$

We can store ± 0, subnormal numbers, normal numbers and $\pm\infty$, NaN. Many questions withthe table, be familiar witth he table. What is the largest subnormal number, smallest subnormalnumber ? macine epsilon ε , here

$$\epsilon = \begin{cases} 2^{-23} & \text{single} \\ 2^{-52} & \text{double} \end{cases}$$

it's the gap . What is the largest FPN smaller than x ?

$$---- s_1 ----- x ------ s_2 --$$

need to find s_i FPNs given x .

2. Rounding

Given real number x we may not be able to store exactly so we need to round .

1. 4 rounding modes. In midterm given real number and asked what is the rounding value if we use one of this rounding modes ?
2. We have the bound

$$\frac{|\mathrm{round}(x) - x|}{|x|} \quad \begin{cases} < \epsilon & \text{any number} \\ \leq \frac{\epsilon}{2} & \text{round to enarest} \end{cases}$$

Here we have a condition : x is in the normal range , i.e.,

$$N_{\min} \leq |x| \leq N_{\max}$$

where the Ns are the smallest and biggest FPNs. We assume x is in the range. If the above condition doesn't hold then the relitve bounding above doesn't hold

We need to prove these things !!!

3. Arithmetic (IEEE rule)

Suppose we have x and y FPNs, then

$$x \circ y = \mathrm{round}(x \circ y) \quad \circ : \oplus, \ominus, \oslash, \otimes$$

> using this equality we can prove anything ? Very important equality

The computed result should be the rounding value of the exact result.

4. Overflow and underflow

Know the definitions of course. We always try to avoid overflow. For example if asked to calculate $\sqrt{x^2 + y^2}$ then this is a vector $[x\ y]^T$ how do we compute this length ? We can code of course but is this reliable ? suppose x is very large then x^2 could be larger than the largest normal number so we will have overflow. Although the true value of this length is smaller than N_max. So we may have overflow indeed. So how do we compute this in a reliable way ? Exercise. "do we use double format lol ?"

5. Exceptional situations

$1/0?$, $0/0$? , $\infty/0$

2. Derivative approximation

We saw 2 methods to approximate.

$$\begin{aligned} f'(x) &\approx \frac{f(x+h) - f(x)}{h} \\ &\approx \frac{f(x+h) - 2f(x) + f(x-h)}{2h} \end{aligned}$$

we need to be able to derive the errors. Give some bounds or expressions to estimate the error, the difference. $\mathcal{O}(h)$ and down is $\mathcal{O}(h^2)$. We cannot get $f(x+h)$ nor $f(x)$ exactly, we have errors in them, note the subtraction ! we have numerical cancelation error, catastrophic cancellation . When h is very small then $f(x+h) \approx f(x)$ so big trouble.

Numerical cancellation important concept. Possible in midterm we are given a formula is this reliable to use this formula , just like the question in Ass.2 ; and also how to reformulate, to avoid the substraction of to FPNs.

3. Solving $Ax = b$

1. GENP

   One of the most basic algorithms. $A = LU$ by forward elimination process

2. GEPP

   we use partial pivoting strategy. Why do we need to use aprtial pivoting strategy? . $PA = LU$ by forward elimination process (Ass.3 we are given A, we are asked to get P and LU) it is possible in the midterm we will have a very similar question, but matrix will be 3×3 of course.

3. Some theoretical results for GEPP

   The following holds only for GEPP, for GENP it doesn't work see nots why

   $$|| r || \approx< \epsilon \|A\| \|x_c\| \quad r = b - Ax_c$$

   we have computer solution x_c , this r is jsut to show how the x_c FITS the linear system. if r small then the x_c fits the system very well.

   $$\frac{\|x_c - x\|}{|| x ||} \approx< \epsilon \|A\| \left\|A^{-1}\right\|$$

   although the x_c fits the system very well, it is possible its accuracy is not good , the above is the relative error and the RHS is the upper bound, we have machine epsilon very small but $\|A\| \left\|A^{-1}\right\|$ can be very large (called condition number). This quantity has nothing to do with the algorithm, only dependent on A. if it's vrey large we say it's ill-condition and the bound will be very large , for instance 10^{-4}, then the computed solution has about 4 accurate digits. if con number close to 1 then about 16 accurate digits since double format.

   1. numerical stability of the algorithm (GEPP is numerically stable)
   2. condition number of the problem (although GEPP is good the computed solution might have poor accuracy since the problem is bad)

      Conceptual questions in midterm about this

4. Tridiagonal $Ax = b$

   Apply the GENP approach to solve this tridiagonal system. we don't apply directly, we can solve $Ax = b$ very efficiently. IF asked to use GEPP can we describe the algorithm and storage and computation cost?

   Case : A is strictly diagonally dominant by row(SDDC. then GENP will not break down since pivot element sin gaussian elimination cannot be zero, in this case the algorithm is much table) or strictly diagonally bounded by column (SDDC. this case GENP is just GEPP; no difference. We proved that, we need to be able to prove it, check first step then show remaining matrix is still SDCC (induction?))

Solving Nonlinear equations

Solving the nonlinear equation $f(x) = 0$

[Example] $x \cosh(50/x) - x - 10 = 0$. We want to get a root.

This problem is very different from the linear equation.

Difficulty : in general there are no closed forms formulas for roots of $f(x) = 0$.

for $ax = b \implies x = b/a$, here it doesn't work. even if $f(x)$ is polynomial with degree 4 of higher there is no formula in theory for the roots. Someone even proved there are no roots recently.

> So we have to be satisfied with approximate roots.

We design Iterative Methods

We construct $x_1, x_2, \ldots, x_n, \ldots$ and hope it will converge to a root of $f(x) = 0$.

3 major issues though.

- where to start the iteration ?
- does the iteration converge and how fast ?

Here we have to do theoretical analysis. Does it converge slow , fast ?

- when to terminate the iteration ?

3 methods

- Bisection method
- Newtons method

Bisection method

[Fact] Suppose $f(x)$ is continuous over $[a,b]$ and $f(a) \times f(b) < 0$, then we know there must be a root over this given interval. i.e., $\exists\, r \mid f(r) = 0$

We check that value of the sign at the mid point. If we're lucky we get a zero lol but this usually doesnt happen.

- if $f(c) = 0$ then c is a root
- if $f(a)f(c) < 0$ then a root exists in $[a,c]$
- if $f(c)f(b) < 0$ then a root exists in $[c,b]$

we continue this slicing in half until the interval is small enough, usually when it's smaller than a given

$$\text{tolerance}: \quad \delta$$

[Algorithm] given the main thing andtolerence δ

```
c <- (a+b) / 2 , error_bound <- }b-a|/2
wile error_bound > delta
    if f(c) = 0 then c is a root, stop
    else
        if f(a)f(c) < 0 then b <- c alese a <- c end
    end
    c <- (a+b)/2 , erro_bound <- error_bound/ 2
end
root <- c
```

Lecture 11

Recall

- we were developing an algorithm where we were picking the middle point to have smaller and smaller intervals. The last iteration returns a good root given a δ.

```
c <- (a+b) / 2 , error_bound <- }b-a|/2
wile error_bound > delta
    if f(c) = 0 then c is a root, stop                  // if lucky
    else
        if f(a)f(c) < 0 then b <- c alese a <- c end    // check the sign
    end
    c <- (a+b)/2 , erro_bound <- error_bound/ 2
end
root <- c
```

MATLAB implementation of this algorithm

```
function r = bisection(f,a,b,delta,display)

fa = f(a); fb = f(b);
if sign(fa)*sign(fb) > 0 % check if same sign
    disp('function has the same sign at a and b')
    return
end

% start the first iteration and compute the middle point c, error bound and other important
information
n = 1; c = (a+b)/2; fc = f(c); e_bound = abs(b-a)/2;
if display,
```

```
    disp(' ');
    disp(' n c f(c)')
    disp('-------------------------------------------       -------')
    disp(sprintf('%4d %23.15e %23.15e', n, c, fc))
end
while e_bound > delta
    n = n+1;
    if fc == 0, r = c; return, end % found the root (lucky case)
    if sign(fa)*sign(fc) < 0 % There is a root in [a,c].
        b = c; fb = fc; % update b
    else % There is a root in [c,b].
        a = c; fa = fc; % update a
    end
    c = (a+b)/2; fc = f(c); e_bound = e_bound/2;
    if display, disp(sprintf('%4d %23.15e %23.15e', n, c,   fc)), end
end
r = c;
end
```

We want to solve

$$f(x) = x^2 - 2 \qquad , [1,2]$$

In MATLAB we run

```
>> f = @(x) x^2 -2
>> f(1)
ans = -1
>> f(2)
ans = 2

>> r =bisection(f,1,2,1.e-12,1)
n           c               f(c)
-----------------------------------------
40  1.41421356238243e+00    -199....e-12
```

after 40 iterations we get a iteration smaller than the error bound.

> if we change the MATLAB implementation to include the error at each step we add

```
disp(sprintf(.....,n,c,e,f(c)))
```

where $e = \left|c - \sqrt{2}\right|$.

When we display the function again , we see the error progressivle decreases but with some noise. After each step we have one more accurate digits. $-2 \implies$ 2 accurate digits.

Convergence and efficiency

Suppose the initial interval is $[a,b] \equiv [a_1, b_1]$ and r is a root.

At step $1 (n = 1)$,

$$c_1 = \frac{1}{2}(a_1 + b_1), \quad |r - c_1| \leq \frac{1}{2}|b_1 - a)1|$$

after n steps we get interval $[a_n, b_n], c_n = 1/2(a_n + b_n)$

we get

$$|r - c_n| \leq \frac{1}{2^2}|b_{n-1} - a_{n-1}| = \cdots = \frac{1}{2^n}|b - a|,$$

$$\lim_{n \to \infty} c_n = r$$

> convergence analysis is simple and easy to understand

[Q] How many steps are require dto ensure $|r - c_n| \leq \delta$ for a general continuous function ?

To ensure this bound wre require

$$\frac{1}{2^n}|b-a| \leq \delta$$

From this we obtain

$$n \geq \log_2(|b-a|/\delta)$$

[Definition] *Linear Convergence* : A sequence $\{x_n\}$ is said to have linear convergence to x if there exists $c \in (0,1)$ such that

$$\lim_{n\to\infty} \frac{|x_{n+1}| - x}{|x_n - n|} = c$$

> remark the numerator is the difference between x_{n+1} and x. SO if x_a is approximation then the numerator is the error at step $n+1$ and the denominator is error at step n .
>
> > when n is big enough, this ratio will be close to c. Since $c < 1$ so it becomes smaller.

For the bisection method, $|r-c| \leq \frac{1}{2^n}|b-a|$ where c_n is the approximation ? Here the upper bound is a sequence and has linear convergence to 0 , but what is the convergence rate ?

Actually we have this squence

$$\left\{\frac{1}{2^n}(n-a)\right\} \to 0 \qquad \left|\frac{x_{n+1}-0}{x_n - 0}\right| = \frac{1}{2}$$

if we use this definition then does the upper bound have linear convergence ?

> no $\{c_n\}$ does not have linear convergence to r. Recall the output of the MATLAB code, the c keeps jumping left and right. The error was oscillating between large and small but it was converging nevertheless.

Usually we see that the bisection method has linear convergence

[Note] in the bisection method we use only the sign information only.

So we introduce a faster method

Newton's Method

> use the value and the derivative of the function. Unlike the bisection method where we use just the sign

A basic strategy behind many numerical algorithms

We replace the complicated problem (nonlinear) by a sequence of simpler problems. Replace nonlinear function by sequences of linear

If the sequence is infinite, we hope that solutions to the simpler problems will converge to a solution of the original problem.

We want to construct a sequence of linear equations and hope the solution of these linear equations to converge to a root of $f(x) = 0$.

We achieve this by using Taylor Series

$$f(x) = \underbrace{f(x_0) + f'(x_0)(x-x_0)}_{\text{linear func of } x} + \underbrace{\frac{1}{2}f''(x_0)(x-x_0)^2}_{\text{error term ?}}$$

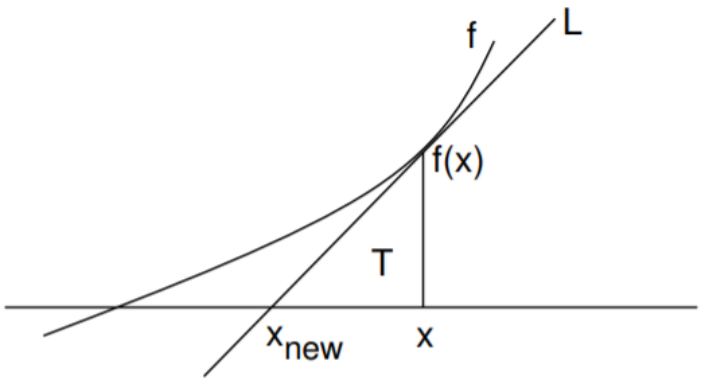
We use $l(x) \equiv f(x_0) + f'(x_0)(x-x_0)$ as an approximation to $f(x)$.

> Set $l(x) = 0$, we get
>
> $$\implies \text{root } x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$
>
> we use x_1 as an approximation to a root of $f(x) = 0$.
>
> [Note] If the approximation is not good, we continue this process by setting
>
> $$x_2 : \qquad x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

In general we have Newton's iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad , f'(x_n) \neq 0$$

Geometrical interpretation



L is just the first derivative of function at x_0

$$\text{given } y = f(x)$$
$$y - f(x_0) = f'(x_0)(x - x_0)$$

Set $y = 0$ and we get the intersection point of L and x-axis

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)},$$

which is exactly the Newton's iteration when $n = 0$.

> maybe we'll have to derive this in exam ? He says it's easy and can derive the formula this way.

when do we stop the iteration?

when $f(x) = 0$ or $f(x_n)$ is small enough (given a tolerance of course).

Stopping criteria

1. $|x_{n+1} - x_n| \leq x + l$
2. $|f(x_{n+1})| \leq f + l$
3. Maximum number of iterations

MATLAB Implementation for $f(x) = x^2 - 2 = 0$

```
function r = newton(f,fd,x,xtol,ftol,nmax,display)
% Newton's method for solving f(x)=0.
% r = newton(f,fd,x,xtol,ftol,n_max,display)
% input: f is the handle of the function f(x).
% fd is the handle of the derivative f'(x).
% x is the initial point
% xtol and ftol are termination tolerances
% nmax is the maximum number of iterations
% display = 1 if step-by-step display is desired,
% = 0 otherwise
% output: root is the computed root of f(x)=0
%
n = 0;
fx = f(x); % f is either a string name or the handle of f(x)
if display,
    disp(' n x f(x)')
    disp('-----------------------------------------------       -------')
    disp(sprintf('%4d %23.15e %23.15e', n, x, fx))
end
if abs(fx) <= ftol, r = x; return, end  % this usually deosnt happen

for n = 1:nmax
```

```
    fdx = fd(x);
    d = fx/fdx;
    x = x - d; % x_{n+1} = x_{n} - d
    fx = f(x);
    if display, disp(sprintf('%4d %23.15e %23.15e', n, x, fx)), end
    if abs(d) <= xtol | abs(fx) <= ftol % if d is small enough less than x tolerance or
abs(x_{n+1}) less than f tolerance
        r = x;
        return  % stop
    end
end
r = x;
end
```

Lecture 13

- Final exam will have similar difficulty and style will be similar.
- The average of this course is usually B or B+

Suppose $x \leq y$ does it imply $\text{round}(\text{x}) \leq \text{round}(\text{y})$? Yes

1. Case 1 : No FPN between x and y , then x_- and y_- are both to the left of x. and similarly for x_+ and y_+ to the right of y. So this holds for this case.

   Is it possible that $\text{round}(x) = x_+$ and $\text{round}(\text{y}) = x_-$? No. So we can conclude that it holds for round to nearest as well.

2. Case 2 : There are FPNs between x and y. then it must be true that $x_+ \leq y_-$. So then $\text{round}(x) \leq \text{round}(y)$ for all rounding modes.

Polynomial Interpolation

Given a set of data points for population in the U.S population from 1900 to 2020 we want to estimate the population in 1985 (no data point at that year). **Which function to use ? That is the question**

Problem ; Given $n+1$ points (x_0, y_0) and $(x_1, y_1), \ldots,$ or table

x	x_0	x_1	$\ldots x_n$
y	y_0	y_1	$\ldots y_n$

we want to find a polynomial $p(x)$ with least degree such that

$$p(x_1) = y_1, \quad i = 0, 1, \ldots n$$

we have

- x_1 : nodes
- $p(x)$: interpolating polynomial

[Theorem] Suppose all the node x_1 are distinct. There exists a unique polynomial $p(x)$ with degree $\leq n$. Such that

$$p(x_i) = y_i, \qquad i = 0, 1, \; dotsn$$

[Proof] Let $p(x) = c_0 + c_1 x + \ldots c_n x^n$. Set $p(x_n) = y_i \quad , i = 0, 1, \ldots n.$

Then,

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_0 & x_1^2 & \ldots & x_1^n \\ 1 & x_0 & x_2^2 & \ldots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_0 & x_n^2 & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \iff Ac = y$$

do we have a unique solution to this system ? Need to check if A is nonsingular of course. Here A is a square matrix, so it should be easy

[Remark] This matrix is called *the Vandermonde Matrix* .

$$\det(A) = \Pi_{0 \le i < j \le n}(x_i - x_j) \neq 0$$

so A is non singular so $Ac = y$ has a unique solution c.

[Remark] We can solve this matrix in $\mathcal{O}(n(\log n)^2$.

Lecture 14

Recall the Vandermode approach

A square matrix with special form

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_0 & x_1^2 & \dots & x_1^n \\ 1 & x_0 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_0 & x_n^2 & \dots & x_n^n \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \iff Ac = y$$

Here A is nonsingular.

[Algorithm] For finding $c_0, c_1, \dots, c_n$

- step 1 : From Ay
- step 2 Solve $Ac = y$
- Form A : $A(:, j) = A(:, j-1)$.

[Example]

x	-1	0	1
y	-15	-5	-3

Let $p_2(x) = c_0 + c_1 x + c_2 x^2$

$$\begin{pmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} -15 \\ -5 \\ -3 \end{pmatrix}$$

> we use the vandermode approach to find polynomial fittings

More efficient flop way for this

- 8 flops : $p_3(x) = c_0 + c_1 * x + c_2 * x * x + c_3 * x^2 * x$
- 6 flops : $p_3(x) = c_0 + x(c_1 + c_2 x + c_3 x^2)$

In general ,

$$p_n(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + \dots + x(c_{n-1} + x(c_n)))))$$

[Algorithm] for evaluating $p_n(x)$ for some x in Vandermode approach

```
p <- c_n
for i = n-1 : -1 : 0        // the full for 2n flops
    p <- c_i + x * p        //2 flops
end
//Nested multiplication
```

Second approach to get the interpolating polynomial

Lagrange Approach

$$p_n(x) = \sum_{i=0}^{n} l_i(x) y_i$$

$$l_i(x) = \frac{(x-x_0)(x-x_1)\dots(x-x_{i-1})(x-x_{i+1})\dots(x-x_0)}{(x_i-x_0)(x_i-x_1)\dots(x_1-xi-1)(x_i-x_{i+1})\dots(x_1-x_0)}$$

this is a polynomial of degree n

$$l_i(x_j) = \delta_{ij} = \begin{cases} 0 & j \neq i \\ 1 & j = 1 \end{cases} \qquad \text{cardinal functions}$$

$$p_n(x_j) = \sum_{i=0}^{n} l_i(x_j) y_j = l_j(x_j) y_j = y_i$$

the above line is true for all j and is the polynomial that we want.

[Example]

x	-1	0	1
y	-15	-5	-3

we have

$$\begin{aligned}
p_2(x) &= l_o(x)y_0 + l_1(x)y_1 + l_2(x)y_2 \\
l_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{(x-0)(x-1)}{(-1-0)(-1-1)} = \frac{1}{2}x(x-1) \\
l_1(x) &= \frac{(x-x_2)(x-x_0)}{(x_1-x_2)(x_1-x_0)} = \frac{(x-1)(x+1)}{(0-1)(0+1)} = 1-x^2 \\
l_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{(x+1)(x-0)}{(1+1)(1-0)} = \frac{1}{2}x(x+1) \\
p_2(x) &= \frac{1}{2}x(x-1)(-15) + (1-x^2)(-5) + \frac{1}{2}x(x+1)(-3) \\
&= -4x^2 + 6x - 5
\end{aligned}$$

which is exactly the same that we got in the last example

> The advantage of this approach is that we can write the in polynomial by hand, don't need MATLAB

Computation

$$\begin{aligned}
p_n(x) &= \sum_{i=0}^{n} l_i(x) y_1 \\
&= \sum_{i=0}^{n} \frac{\Pi_{j=0\ ,j\neq i}^{n}(x-x_j)}{\Pi_{j=0,\ j\neq i}^{n}(x_i-x_j)} y_i \\
&= \sum_{i=0}^{n} \Pi_{j=0}^{n}(x-x_j) \frac{y_i}{\Pi_{j=0,\ j\neq i}(x_i-x_j)} \frac{1}{x-x_j} \\
&= \Pi_{j=0}^{n}(x-x_j) \sum_{i=0}^{n} \frac{c_i}{x-x_j}
\end{aligned}$$

The term, independent of x

$$\frac{y_i}{\Pi_{j=0,\ j\neq i}(x_i-x_j)} := c_i$$

The rest afterwards are dependent on x .

Computing c_i for $i = 0, 1, \ldots, n$

$$c_i = \frac{y_i}{\Pi_{j=0,\, j\neq i}(x_i - x_j)}$$

the cost is n of '-' , $n-1$ of '*' and 1 of '/' , also

- $2n$ flops for computing each c_i
- Total cost for computing all c_i is $2n * (n+1) \approx 2n^2$ flops

---

Given c_i for $i = 0, 1, \ldots n$

Consider computing $p_n(x)$

$$p_n(x)\underbrace{\Pi_{j=0}^{n}(x - x_j)}_{n+1\ \mathrm{n+1+n}}\underbrace{\sum_{i=0}^{n}\frac{c_i}{x - x_j}}_{2(n+1)}$$

$$\therefore \text{ cost } = (n+1+n) + (2(n+1)+n) + 1 \approx 5n \text{ flops.}$$

lecture 15

We have found poly $p_n(x)$ to interpolate $(x_0, y_0), (x_1, y_1), \ldots, (x_k, y_k)$. Now (x_{k+1}, y_{k+1}) a new point is available we want to find a new more inter-poly $p_{k+1}(x)$ to interpolate all $k+2$ points.

With Lagrange we have to recalculate the coefficients and for the vandermode we have to rewrite A, so it's a waste of resources.

> here with this approach we can accommodate such situation.

$$p_{k+1} = p_k(x) + a_{k+1}(x - x_0)(x - x_1)\ldots(x - x_k)$$

here

$$\text{for } i = 0 : k \quad , p_{k+1}(x_i) = p_k(x_i) = y_i$$

to find p_{k+1} we need to solve the function

$$p_{k+1}(x_{k+1}) = y_{k+1}$$

we substitute this in the above and solve for the unknown a_{k+1}. We obtain

$$a_{k+1} = \frac{y_{k+1} - p_k(x_{k+1})}{(x_{k+1} - x_0)(x_{k+1} - x_1)\ldots(x_{k+1} - x_k)}$$

so we get p_{k+1} from p_k , we get the general expression for $p_n(x)$:

[Definition] (*Newton form*) of interpolating polynomial

$$p_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + \underbrace{a_n(x - x_0)\ldots(x - x_{n-1})}_{\text{degree } \leq n}$$

> this looks very similar to the Vandermode form but we just changed the format of the polynomial. Now we have $x - x_k$ instead of x in the Vandermode.
>
> > Q does this interpolating polynomial use the point x_n ?
> >
> > Although we don't have it in this expression we will use this point.
> >
> > Q what is a_0 ? It's y_0 because we know that
> >
> > $$p_0(x) = a_0 \longrightarrow p_0(x_0) = a_0 = y_0$$
>
> the degree of a_0 is 0

[Example]

x	-1	0	1
y	-15	-5	-3

$$
\begin{aligned}
p_0(x) &= a_0 \\
&\implies p_0(x_0) = y_0 = a_0 = -15 \\
p_1(x) &= a_0 + a_1(x - x_0) \\
&= -15 + a_1(x + 1) \\
&\implies p_1(x_1) = y_1 \implies -15 + a_1(0 + 1) = -5 \\
&\implies a_1 = 10 \\
p_2(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) \\
&= -15 + 10(x + 1) + a_2(x + 1)x \\
&\implies p_2(x_2) = y_2 \\
&\implies -15 + 10(x + 1) + a_2(1 + 1) = -3 \\
&\implies a_2 = -4
\end{aligned}
$$

We conclude that

$$
p_2(x) = -15 + 10(x + 1) - 4(x + 1)x = \boxed{-4x^2 + 6x - 5}
$$

Evaluation of $p_n(x)$ for some x.

Suppose $a_0, \dots a_n$ are known (given.)

> observe that for almost all terms we have the common term $x - x_0$ so we can use some nested multiplication stuff ?

$$
\begin{aligned}
p_3(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)\dots \\
&= a_0 + (x - x_0)\big[a_1 + a_2(x - x_1) + a_2(x - x_1)(x - x_2)\big] \\
&= a_0 + (x - x_0)\big[a_1 + (x - x_1)\big[\underbrace{a_2 + (x - x_2)a_3}_{u_2}\big]\big] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \underbrace{\qquad\qquad\qquad\qquad}_{u_1} \\
&\qquad \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{u_0}
\end{aligned}
$$

where we defined

$$
\begin{aligned}
u_3 &:= a_3 \\
u_2 &:= a_2 + (x - x_2)u_3 \\
u_1 &:= a_1 + (x - x_1)u_2 \\
u_0 &:= a_0 + (x - x_0)u_1
\end{aligned}
$$

so we can generate an algorithm for computing $p_n(x)$.

```
/* Algorithm for evaluating p_n(x) for some x */
p <-- a_n
for i=n-1:-1:0
    p <-- a_i + (x-x_i) * p  /* recall all the a_i are given */
end
p = p_n(x)
```

> **flop cost for the above algorithm**
>
> - inside the loop : 3 flops
> - we do this n times so the total cost is $3n$ flops.
>
> **Cost for computing a_{k+1}**
>
> - $k + 1$ flops for the subtraction (denominator)
> - k flops for multiplications (denominator)
> - $3k + 1$ (numerator) +1 (for division) (numerator)
> - $\therefore$ total cost for computing coefficients $a_1, a_2, \dots, a_n$ is
>
> $$
> \sum_{k=0}^{n-1} 5k + 3 \approx \frac{5}{2}n^2 \text{ flops}
> $$

A more efficient algorithm for computing $a_0, a_1, \dots a_n$

$$p_n(x_i) = y_i, \quad i = 0, 1, \dots n$$

we can consider

$$p_3(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3 \dots$$
$$p_3(x_i) = y_i, \quad i = 0, 1, \dots n$$

we have $n + 1$ variables and with $n + 1$ unknowns so we can construct a linear system (*general form* [in our example we are limited to 4 rows])

$$\begin{pmatrix} 1 & & & & \\ 1 & x_1 - x_0 & & & \\ 1 & x_2 - x_0 & \prod_{j=0}^{1}(x_2 - x_j) & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & x_n - x_0 & \prod_{j=0}^{1}(x_n - x_j) & \dots & \prod_{j=0}^{n-1}(x_n - x_j) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

We can find the coefficients

- step 0 : $a_0 = y_0$
- step 1 : put zeros under the first pivot. i.e., substract equation 1 from equation 2 and substract equation 1 from equation 3 .Substract equation 1 from equation 4, obtain

  $$\begin{pmatrix} 1 & & & \\ 0 & x_1 - x_0 & & \\ 0 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & \\ 0 & x_3 - x_0 & (x_3 - x_0)(x_3 - x_1) & (x_3 - x_0)(x_3 - x_1)(x_3 - x_2) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 - y_0 \\ y_2 - y_0 \\ y_3 - y_0 \end{pmatrix}$$

  Then divide equation 2 by $x_1 - x_0$. Also divide equation 3 by $x_2 - x_0$. Divide equation 4 by $x_3 - x_0$, we obtain the new matrix

  $$\begin{pmatrix} 1 & & & \\ 0 & 1 & & \\ 0 & 1 & (x_2 - x_1) & \\ 0 & 1 & (x_3 - x_1) & (x_3 - x_1)(x_3 - x_2) \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ \frac{y_1 - y_0}{x_1 - x_0} \\ \frac{y_2 - y_0}{x_2 - x_0} \\ \frac{y_3 - y_0}{x_3 - x_0} \end{pmatrix}$$

Lecture 17

[Definition] (*Spline*) : A function f is called a spline of degree k if

1. the domain of S in $[a, b]$
2. $S, S', \dots, S^{k-1}$ continuous on $[a, b]$
3. There is a partition of the interval $a = t_0 < t_1 < \dots < t_n = b$ such that S is a polynomial of degree at most k on each $[t_i, t_{i+1}]$, here $t_i =$ knots

Special cases

$k = 1$ then we have a Linear spline. We have a table with $n + 1$ points

x	t_0	t_1	$\dots$
y	y_0	y_1	$\dots$

where $t_0 < t_1 < \dots < t_1$ i.e., the knots are ordedred.

We can write

$$S(x) = \begin{cases} S_0(x) & ,x \in [t_0, t_1] \\ S_1(x) & ,x \in [t_1, t_2] \\ \vdots & \\ S_{n-1}(x) & ,x \in [t_{n-1}, t_n] \end{cases}$$

[Example]

$$--- \underbrace{t_i}_{} ----- \underbrace{t_{i+1}}_{} --- \\ \underbrace{\qquad\qquad\qquad\qquad\qquad}_{S_{i(x)}}$$

with point s(t_i, y_i) and (t_{i+1}, y_{i+1}) on top in that order, then we can write

$$S_i(x = y_1 + m_i(x - y_i)) \qquad ,m_i = \frac{y_{i+1} - y_i}{t_{i+1} - t_i}$$

> For a linear spline, generally it's first derivative S' is not conitnous on the interval. So it's graph lacks of smoothness.

- Evaluation of Linear Spline $S(x)$:

  we look at x and check it's interval.

```
for i = 0:n-1
    if x <= t_{i+1},
        exit loop --> x in [t_i , t_{i+1}]
    end
end
s<-- y_i + m_i(x-t_i)
```

  We extend the definition :

  - if $x < t_0$ then $S(x) = y_0 + m_o(x - t_0)$.
  - if $x > t_n$ then $S(x) = y_n + m_n(x - t_n)$

For $k = 2$ We have a quadratic spline

Think of a irregular curve. It may collapse at some point ? S'' is the curvature of the curve.

> For a quadratic spline , generally S'' is not continuous. So the curve of it's graph changes abruptly at each knots

---

In application we use the cubic spline ($k = 3$)

1. even if $S'''(x)$ is not continuous you cannot detect it by eyes
2. experiments show using spline of degree $k > 3$ it suddenly gives us advantage, soldom yields any advantage

---

Cubic Spline $k = 3$.

$$S(x) = \begin{cases} S_1(x) & ,x \in [t_0, t_1] \\ S_2(x) & ,x \in [t_1, t_2] \\ \vdots & \\ S_{n-1}(x) & ,x \in [t_{n-1}, t_n] \end{cases}$$

$S_i(x)$ is a polynomial degree at most 3

We have a smooth curve with no abrupt change ?

> there are 4 coefficients for every subinterval, so we need to find $4n$ unknowns

Conditions for cubic spline :

1. $S(t_i) = y_i$, $i = 0, 1, \ldots, n$,.i.e, the spline has to pass through all the $n + 1$ points.

   This condition gives us $n + 1$ equations.
2. continouity: $S_{i-1}(t_i) = S_i(t_i)$, then the spline on this interval is continuous
3. $S'_{i-1}(t_i) = S'_i t_i$ for $i = 1, \ldots, n - 1$ (first deriv. is cont. on the interval)
4. $S''_{i-1}(t_i) = S''_i t_i$ for $i = 1, \ldots, n - 1$ (second deriv. is cont. on the interval)

the last 3 conditions give $3(n - 1)$ conditions.

In total we have

$$n+1+3(n-1)=4n-2 \qquad \text{equations}$$

to set up.

Q But we have $4n$ unknows to solve, so can we know the cubic spline in that case ?

> Yes but the solution is not unique ! If we want unique solution we need 2 more equations.

$\star$ We have to impose 2 extra conditions to find a unique solution.

[Definition] (*Natural cubic spline*)

By imposing the conditions

$$S''(t_0)=S''(t_n)=0$$

we are now using the natural cubic spline.

[Example] Derive the natural cubic spline to interpolate

x	**-1**	**0**	**1**
y	1	2	-1

3 points $\implies$ 2 pieces of the polynomial to interpret this table (partitions).

$$\text{Let } S=\begin{cases} S_0(x)=a_0+a_1x+a_2x^2+a_3x^3 & ,x\in[-1,0] \\ S_1(x)=b_0+b_1x+b_2x^2+b_3x^3 & ,x\in[0,1] \end{cases}$$

- $$\begin{aligned} 1.\quad S_0(-1)=1 &\implies & a_0-a_1+a_2-a_3&=1 \\ 2.\quad S_0(0)=2 &\implies & a_0&=2 \\ 3.\quad S_1(0)=2 &\implies & b_0&=2 \\ 4.\quad S_1(1)=-1 &\implies & b_0+b_1+b_3&=-1 \end{aligned}$$

- $$\begin{aligned} S_0'(x)&=a_1+2a_2x+3s_3x^2 \\ S_0''(x)&=2a_2+6a_3x \\ S_1'(x)&=b_1+2b_2x+3b_3x^2 \\ S_1''(x)&=2b_2+6b_3x \end{aligned}$$

- $$\begin{aligned} 5.\quad (S_0'(0)=a_1, \quad S_1'(0)=b_1) &\implies a_1=b_1 \\ 6.\quad (S_0''(0)=2a_2, \quad S_1''(0)=2b_2) &\implies a_2=b_2 \\ 7.\quad S_0'(-1)=2a_2-6a_3=0 &\implies a_3=\frac{1}{3}a_2 \\ 8.\quad S_1''(1)=2b_2+2b_3=0 &\implies b_3=-\frac{1}{3}b_2 \end{aligned}$$

By substitution we get

$$\left.\begin{aligned} 2-a_1+a_2-\tfrac{1}{3}a_2&=1 \\ 2+a_1+a_2-\tfrac{1}{3}a_2&=-1 \end{aligned}\right\} \implies \begin{aligned} a_2&=-3 \\ a_1&=-1 \\ a_3&=-1 \end{aligned}$$

then $b_1=-1$,$b_2=-3$, $b_3=1$ so we get

$$S(x)=\begin{cases} 2-x-3x^2-x^3 \\ 2-x-3x^2+x^3 \end{cases}$$

Design of algorithm for NCS

Let $Z_i=S''(t_i)$ for $i=0:n$ i.e., the second derivative at knot t_i.

On interval $[t_i,t_{i-1}]$, $S''(x)$ is a linear polynomial and $S_i''(t_i)=z_i$ and $S_2''(t_{i+1})=z_{i+1}$. By that we mean that for 2 points t_i and t_{i+1} we have the equivalent codomain points (t_i,z_i) and (t_{i+1},z_{i+1}) with a straight line joining these two codomain points, by the function $S_i''(x)$.

We can write out a second derivative

$$S_i''(x)=\frac{x-t_{i+1}}{t_i-t_{i+1}}z_i+\frac{x-t_i}{t_{i+1}-t_i}z_{i+1} \qquad \longrightarrow \text{Lagrange form !} \tag{1}$$

↑ note the Lagrange form !

Of course we can also write in the form :

$$S_i''(x) = z_1 + \frac{z_{i+1} - z_i}{t_{i+1} - t + i}(x - t_i),$$

but we use the first form for convenience.

We integrate (1) twice to get the expression for S_i . Denote $h_i := t_{i+1} - t_i$ for notation convenience.

$$S_i(x) = (t_{i+1} - x)^3 \frac{z_i}{6h_i} + (x - t_i)^3 \frac{z_{i+1}}{6h_i} + c_i x + d_i,$$

where c_i and d_i are integration constants. We would like to dewtermine z_i z_{i+1} c_i and d_i. We look for these variables to find an expression for the ith piece, and thence we can find an expression for all pieces function.

$S_i(t_i) = y_i$ for all i. This gives us $S_{i+1}(t_{i+1}) = y_{i+1}$ (since it holds for all i) ,since it's continuous that vies us $S_i(t_{i+1}) = y_{i+1}$. So we get

conditions of function of all knots

- $S_i(t_i) = y_i$
- $S_i(t_{i+1}) = y_{i+1}$

the above gives us (by substitution)

$$\begin{cases} h_i^2 \frac{z_i}{6h_i} + c_i t_i + d_i = y_i \\ h_i^3 \frac{z_{i+1}}{6h_i} + c_i t_{i+1} + d_i = y_{i+1} \end{cases}$$

solving these two equations gives us

$$\begin{cases} c_i = \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{6}(z_{i+1} - z_i) \\ d_i = \frac{y_i t_{i+1} - y_{i+1} t_i}{h_i} + \frac{h_i}{6}(t_i z_{i+1} - t_{i+1} z_i) \end{cases}$$

Now we find all the z_i. We use conitnouity of S' .

$$\begin{aligned} S_i'(x) &= -(t_{i+1} - x)^2 \frac{z_i}{2h_i} + (x - t_i^2) \frac{z_{i+1}}{2h_i} + c_i \\ S_i'(t_i) = S_{i+1}'(t_i) &\implies h_{i-1} z_{i-1} + 2(h_{i-1} + h_i) + h_i z_{i+1} = b_6(b_i - b_{i-1}) \\ &\text{where } b_i = \frac{y_{i+1} - y_i}{h_i} \end{aligned}$$

Lecture 17

Recap :

$Z_i = S''(t_i)$, with S_i'' on $[t_i, t_{i+1}]$ is continuous. $S_i''(t_i) = z_i$, $S_i''(t_{i+1}) = z_{i+1}$

$S_i''(x)$ on $[t_i, t_{i+1}]$ is linear polynomial

$$S_i''(x) = \frac{x - t_{i+1}}{t_i - t_{i+1}} z_i + \frac{x - t_i}{t_{i+1} - t_i} z_{i+1},$$

Let $h_i = t_{i+1} - t_i then

$$S_i(x) = (t_{i+1} - x)^3 \frac{z_i}{6h_i} + (x - t_i)^3 \frac{z_{i+1}}{6h_i} + z_i x + d_i \qquad (\star)$$

By $S_i(t_i) = y_i$, and $S_i(t_{i+1}) = y_{i+1}$

$$\text{get } \begin{cases} c_i &= \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{6}(z_{i+1} - z_i) \\ d_i &= \frac{y_i t_{i+1} - y_{i+1} t_i}{h_i} + \frac{h_i}{6}(t_i z_{i+1} - t_{i+1} z + i) \end{cases}$$

We are left with the continouity of the first derivative

$$S_{i-1}'(t_i)' = S_i'(t_i).$$

From $(\star)$,

$$S_i'(x) = (t_{i+1}-x)^2\frac{z_i}{2h_i} + (x-t_i)^2\frac{z_{i+1}}{2h_i} + \frac{y_{i+1}yy_i}{h_i} - \frac{h_i}{6}(z_{i+1}-z_i)$$
$$\text{using } S_{i-1}'(t_i)' = S_i'(t_i) \text{ we get}$$
$$h_{i-1}z_{i-1} + 2(h_{i-1}+h_i)z_i + h_iz_{i+1} = 6(b_i - b_{i-1}) \qquad \text{for } i = 1,2,\ldots,n-1$$

Let $b_i = (y_{i+1}-y_i)/h_i$ then

$$h_0z_0 + 2(h_0+h_i)z_i + h_iz_1 = 6(b_1-b_0)$$

we use the conditions

$$\{z_0 = 0 \qquad \text{and } z_n = 0$$

We get the matrix form

$$\begin{pmatrix} 2(h_0+h_1) & h_1 & & & \\ h_1 & 2(h_1+h_2) & h_2 & & \\ & \ddots & & & \\ & h_{n-3} & 2(h_{n-3}+h_{n-2}) & h_{n-2} \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_{n-2} \\ z_{n-1} \end{pmatrix} = \begin{pmatrix} 6(b_1-b_2) \\ 6(b_2-b_1) \\ \vdots \\ 6(b_{n-2}-b_{n-3}) \\ 6(b_{n-1}-b_{n-2}) \end{pmatrix}$$

> Notice that this tridiagonal linear system is SDDC (since all $h_i > 0$), because $2(h_0+h_1) > h_1$ (recall that $h_i = t_{i+1}-t_i > 0$ since ordered)

Since SDDC then we won't have row permutation ,thence we need not use GEPP we can use GENP to solve this, it is enough.

Evaluating $S(x)$

We have the expression for the ith piece.

$$S_i(x) = (t_{i+1}-x)^3\frac{z_i}{6h_i} + (x-t_i)^3\frac{z_{i+1}}{6h_i} + \left(\frac{y_{i+1}-y_i}{h_i} - \frac{h_i}{6}(z_{i+1}-z_i)\right)x + \frac{y_it_{i+1}-y_{i+1}t_i}{h_i} + \frac{h_i}{6}(t_iz_{i+1}) - t_{i+1} - z_i)$$

We impose

$$S_i(x) = A_i + B_i(x-t_i) + C_i(x-t_i)^2 + D_i(x-t_i)^2$$

↑ we can used nested evaluation if we can make our expression int his form.

So then

$$\begin{aligned} S_i(x) &= A_i + B_i(x-t_i) + C_i(x-t_i)^2 + D_i(x-t_i)^2 \\ &= A_i + (x-t_i)\left(B_i + C_i(x-t_i) + D_i(x-t_i)^2\right) \end{aligned}$$

and we continue this way with the nested evaluation algorithm. Because if $x = t_i$ int the above the terms cancel out, then

$$A_i = S_i(t_i) = y_i \tag{1}$$

if we take the first derivative then A_i cancels out ; we get

$$B_i = S_i'(t_i) = -\frac{h_i}{6}z_{i+1} - \frac{h_i}{3}z_i + \frac{y_{i+1}-y_i}{h_i} \tag{2}$$

Similarly, we take the second derivative and evaluate it at point t_i ; we get

$$C_i = \frac{1}{2}S_2''(t_i) = \frac{1}{2}z_i \tag{3}$$

Lastly, we take the third derivative and evaluate at t_i as well and we get

$$D_i = \frac{1}{6}S_i'''(t_i) = \frac{1}{6h_i}(z_{i+1}-z_i) \tag{4}$$

> So if we know the z_is , we can get the ith piece in $S_i(x)$ over which we use nested calculation to find x at that piece !

Least Squares Approximation

Motivation : we want to avoid high degree polynomial for interpolation since it's not good idea. We make the polynomial get close to the data points not go exactly through the data points.

Given $m+1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, we find a function to approximate them.

For example for 5 points we need a polynomial of degree exactly 4. But if we pass a straight line, it's just degree 1.

> Both polynomial interpolation and the spline require $p(x_i) = y_i \ \ \forall\, i$, but here, for many applications , x_i, y_i they are from sensors for instance time and temperature, so they have errors, so it is not necessary for our polynomial to pass through all of them necessarily.

Q How do we measure distance between and point and our line $y = C_0 + C_1 x$?.

Well for a given point (x_k, y_k) the distance is $(C_0 + C_1 x_k - y_k)^2$.

To find C_1 and C_2 we solve

$$\min \sum_{k=0}^{m} |C_0 + C_1 x_k - y_k|,$$

this is called L_1-norm approximation , which is not differentiable with respect to our coefficients. So in practice, better is

$$\min \sum_{k=0}^{m} (C_0 + C_1 x_k - y_k)^2 := \phi(c_o, c_1),$$

this is called the least-square problem.

[Definition]

1. the vertical line from data point to line is called least square
2. the horizontal line from data point to line is called data least square
3. The resulting vector (distance to the line) is called the total least square

To solve the least square problem above, we do

$$\frac{\partial \phi}{\partial C_0} = \sum_{k=0}^{m} 2(C_0 + C_1 x_k - y_k) \times 1 = 0$$
$$\frac{\partial \phi}{\partial C_1} = \sum_{k=0}^{m} 2(C_0 + C_1 x_k - y_k) \times x_k = 0$$

in both cases we set the PDVs to be zero (2 gets cancelled out) and get a linear system

$$\implies \begin{pmatrix} m+1 & \sum_{k=0}^{m} x_k \\ \sum_{k=0}^{m} x_k & \sum_{k=0}^{m} x_k^2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} \sum_{k=0}^{m} y_k \\ \sum_{k=0}^{m} y_k x_k \end{pmatrix}$$

[Example]

We want to use a straight line to approximate the following data :

x	**1**	**2**	**2.5**
y	2.1	4.1	5.2

We write the linear system immediately

$$\begin{pmatrix} 4 & 8.5 \\ 8.5 & 20.25 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \end{pmatrix} = \begin{pmatrix} 17.1 \\ 37.65 \end{pmatrix} \implies C_0 = 3 \ \ C_1 = 0.6 \ \ ,\therefore y = 3 + 0.6x$$

[Note] the first element is 4 NOT 5 because we have $m+1$ data points and the sum is from $k = 0 \to m$.

General Linear Family of Function

Assume our data can be fit by such a function

$$y = \sum_{j=0}^{m} C_i g_j(x)$$

where we took a linear combination of $n+1$ basis functions g, which are known. IF we take our base functions to be

$$\begin{cases} g_0(x) = 1 \\ g_1(x) = x \end{cases}$$

then we get a straight line. But we are interested in something more general.

To determine $C_0, C_1, \ldots, C_n$ ($n+1$ coefficients and $n+1$ base functions).

Let (x_m, y_m) with $m \geq n$,

$$\sum_{k=0}^{m} \left(\sum_{j=0}^{n} C_j g_j(x_k) - y_k \right)^2 = \phi(C_0, C_1, \ldots, C_n),$$

we want to minimize this phi in terms of C_i, i.e.,

$$\min_{C_0, C_1, \ldots, C_n} \phi(C_0, C_1, \ldots, C_n)$$

Lecture 18

Given $m+1$ points,

1. Data fitting by a straight line

   $$y = C_0 + C_1 x$$

   we have

   $$\min_{C_0, C_1} \phi(C_0, C_1)$$
   $$\phi(C_0, C_1) = \sum_{k=0}^{m} (C_0 + C_1 x_k - y_k)^2$$

   then

   $$\begin{cases} \frac{\partial \phi}{\partial C_0} \\ \frac{\partial \phi}{\partial C_1} \end{cases} \implies \begin{pmatrix} m+1 & \sum_{k=0}^{m} x_k \\ \sum_{k=0}^{m} x_k & \sum_{k=0}^{m} x_k^2 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \end{pmatrix} = \begin{pmatrix} \sum_{k=0}^{m} y_k \\ \sum_{k=0}^{m} x_k y_k \end{pmatrix}$$

2. By a general linear family of functions

   $$y = \sum_{j=0}^{m} C_j g_j(x)$$

   We have $n+1$ known basis functions $g_0, g_1, \ldots, g_n$, e.g., $g_1(x) = x$. Then,

   $$\min_{C_0, C_1} \phi(C_0, C_1, \ldots, C_n)$$
   $$\phi(C_0, \ldots, C_n) = \sum_{k=0}^{m} \left(\sum_{j=0}^{n} C_j g_j(x_k) - y_k \right)^2$$

We have $n+1$ base functions and we have $n+1$ data points ? We also use the condition

$$m > n.$$

To solve the problem compute :

[Definition] (*Normal equations*) :

$$\frac{\partial \phi}{\partial C_i} = \sum_{k=0}^{m} 2 \left(\sum_{j=0}^{n} C_j g_j(x_k) - y_k \right) g_i(x_k), \quad i = 0, \ldots, n \qquad (\star)$$

Now set $\partial\phi/\partial C_i = 0$ then the resulting coefficients $C_0, C_1, \ldots, C_n$ can produce the resulting functions in the form

$$y = \sum_{j=0}^{n} C_j g_j(x)$$

is as close to the data point as possible.

Now set in $(\star)$

$$\frac{\partial \phi}{\partial C_i} = 0$$

we get

$$\sum_{j=0}^{n} \left(\sum_{k=0}^{m} g_j(x_k) g_i(x_k) \right) C_j = \sum_{k=0}^{m} y_k g_i(x_k), \quad i = 0, \ldots, n$$

Then we construct the matrix

$$\begin{pmatrix} \sum_{k=0}^{m} g_0(x_k) g_0(x_k) & \cdots & \sum_{k=0}^{m} g_n(x_k) g_0(x_k) \\ \vdots & \ddots & \vdots \\ \cdot & & \cdot \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_n \end{pmatrix} = \begin{pmatrix} \sum_{k=0}^{m} y_k g_0(x_k) \\ \vdots \\ \vdots \end{pmatrix} \qquad (\star\ 2)$$

This is a $n+1$ by $n+1$ for which we can just use GENP.

Denote

$$A = \begin{pmatrix} g_0(x_o) & g_1(x_0) & \cdots & g_n(x_0) \\ g_0(x_1) & g_1(x_1) & \vdots & g_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ g_0(x_n) & g_1(x_n) & \cdots & g_n(x_n) \end{pmatrix} \in \mathbb{R}^{(n+1)\times(n+1)}$$

Q What is the relationship between the last 2 matrices ?

$\rightarrow A^T A$ gives the previous matrix !

So $(\star\ 2)$ becomes , by setting

$$A^T A c = A^T y$$

> the optimal C that we want to solve the minimization problem is the c that satisfies this equation

Letting

$$y = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix} \qquad \text{and } c = \begin{pmatrix} C_0 \\ \vdots \\ C_n \end{pmatrix}$$

And also , the objective functions

$$\phi(C_0, \ldots, C_n) = \sum_{k=0}^{m} \left(\sum_{j=0}^{n} C_j g_j(x_k) - y_k \right)^2$$

becomes

$$\phi(C_0, \ldots, C_n) = \sum_{k=0}^{m} \left(\sum_{j=0}^{n} C_j g_j(x_k) - y_k \right)^2 = \|Ac - y\|_2^2$$

If c satisfies $A^T A c = A^T y$ then we can rewrite this as

$$A^T (Ac - y) = 0 \qquad (\star\ 3)$$

The geometrical interpretation of this can be seen by looking at matrix $A = [a_0\ a_1\ \ldots\ a_n]$ where each a_i is a column of A ,then $\star\ 3$ becomes

$$a_j^T(Ac - y) = 0, \quad j = 0, \ldots, n$$

each column of A is orthogonal to $Ac - y$.

$$Ac = C_0a_0 + C_1a_1 + \ldots C_na_n$$

this is a linear combination of columns of A.

> we want to find a specific c in the plane (space) Ac (since Ac is a linear combination of all C_i !!).
>
> Then for a given y not on the plane (it can't be) then we need to find a point C (linear combination of $C_0, C_1, \ldots, C_n$) on the plane such that $Ac - y$(the distance between Ac and y) is minimized. Such c is normal to the plane and perpendicular to the plane. Then $Ac - y$ will be minimized for that c.

When

$$y - Ac \perp Ac_{\text{plane}}$$

then we find such optimal c .

Then because $y - Ac \perp$ to any vector on this plane. And $a_j = Ae_j$, for $e_j = [0\ 1\ 0 \ldots\ 0]^T$ basis matrix column, it means that it can be written as A times a special c , then we can get

$$a_j^T \underbrace{(y - Ac)}_{\text{normal ?}} = 0, \qquad j = 0, 1, \ldots, n$$

This is how we can understand these objective functions geometrically.

Because A has full column rank $m > n$, then A^TA in $A^TAc = A^Ty$ nonsingular. Because it is nonsingular we can write a solution as

$$c = (A^TA)^{-1}A^Ty.$$

> Also, A^TA is symmetric, and actually, A^TA is symmetric positive definite so we have a more efficient algorithm than GENP to solve $A^TAc = A^Ty$!

[Note] The MATLAB function to solve for the c is $c = A \setminus y$.

Relationship with poly-interpolation

if our base functions $g_j(x) = x^j$ for $j = 0, \ldots, n$ (we look at this specific set of the basis functions) , and $m = n$ then

$$A = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & & \ddots & \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix}$$

this is precisely the Vandermode Matrix ! So in this case we can have a polynomial to interpolate our data points.

Then the problem is just polynomial interpolation because we want to minimize $Ac = y$.

Numerical Interpolation

Motivation:

For example we want to calculate

$$\int_0^1 x^d \, \mathrm{d}x = \frac{1}{3}x^3$$

by sometimes we can't find the exact solution. So if we can find the anti-derivative F of f , i.e., the derivative of the function $F'(x) = f(x)$ then the integral

$$\int_a^b f(x)\mathrm{d}x = F(b) - F(a).$$

However, in many applications, such F may not exist.

For example

$$f(x) = e^{-x^2},$$

we can't solve this, we need to develop an algorithm to solve this iteratively. We need to compute

$$\int_a^b f(x)\mathrm{d}x$$

numerically.

To design such algorithm we use an important fact that for a given function the integral is the positive are minus the negative area under the curve.

> [Fact] : The integral is the area between the graph and the x-axis (may be negative).

[Example] (*Rectangle Rule*)

(Like in Riemann sums approximation) Given a curve f over an interval a to b, then we can draw a rectangle from $a \to f(a) \to f(b) \to b \to a$. We use this as an approximation. Then the exact integral is

$$I = \int_a^b f(x)\mathrm{d}x$$

and our approximation is

$$I_R = (b-a)f(a)$$

But we can do better, we can divide into more subintervals. Divide into partitions $[a, x_1, x_2, \ldots, x_{n-1}, b]$ where $a = x_0$ and $b = x_n$ then the are of the rectangle over $[x_i, x_{i+1}]$ as $hf(x_i)$ where h is just the width of every rectangle

$$h = \frac{b-1}{n} \qquad \text{(width of the rectangle panels)}$$

We look at expression for x_i, that is $x_i = a + ih$. Then the total area across all n rectangles can be rewritten as

$$I_R = \sum_{i=0}^{n-1} hf(x_i)$$

> [Note] From calculus we know that if we take the limit we get the exact integral, but here in numerical computation we can't let $h = 0$! So we use this approximation method.

So I_R is an approximation to the real area I, and is called the Composite Rectangle Rule .

[Example] We want to approximate

$$I = \int_0^1 e^x \mathrm{d}x = e^x\Big|_0^1 = e - 1.$$

```
function I_R = rect(a,b,n) /*n is the number of subintervals */
 h = (b-a)/n ;
 x = linspace(a,b-h,n)
 fx = exp(x);
 I_R = sum(fx)*h;
```

Then we display the differences

```
 I = exp(1) - 1;
 disp ('     h    I_R   error')
 disp (' ')
  n=1;
  for i = 1:10
    n = 2*n;
    I_R = rect(0,1,n);
    h = 1/n;
    disp([hmI_R, I-I_R]);
  end
```

In the result we not that $h \sim \text{error}$ in magnitude. We have some reason that error introduced by the rectangle approximation will be

$$\mathcal{O}(h)$$

Lecture 19

> Recap: We saw the rectangle rule for integration interval. We didivded the interval $a \to b$ into n panels. And we said that
>
> $$h = \frac{b-a}{n} \qquad \text{, with } \; x_i = a + ih$$

We also said that the error is

$$I - I_R = \text{error}$$

[Proof]

We use the Mean-Value Theorem.

- Sum: Let $q(x)$ be continuous on $[a,b]$ if $p(x_i) \geq 0$ for $i = 1, 2, \ldots, n$, then

  $$\sum_{i=1}^{n} p(x_i)q(x_i) = q(z)\sum_{i=1}^{n} p(x_i) \qquad \text{for some } z \in [a,b].$$

  In other words, we find a z to pull the $q(\cdot)$ outside the sum .
- Integral: Let $q(x)$ and $p(x)$ be continuous on the interval $[a,b]$, if $p(x) \geq 0$, then

  $$\int_a^b p(x)q(x)\mathrm{d}x = q(z)\int_a^b p(x)\mathrm{d}x \qquad \text{for some } z \in [a,b].$$

Then it follows that

$$I - I_R = \frac{1}{2}(b-a)hf'(z) = \mathcal{O}(h) \qquad \text{, } z \in [a,b] \tag{1}$$

> Here we assume that the first derivative f' of the function is continuous and is bounded.

By the Taylor Theorem, we can expand our function through

$$f(x) = f(a) + f(x-a)f'(z_x) \qquad \text{, } z_x \in [a,b]$$

here z_x since z depends on x. It is not necessarely a constant.

1. Base case: $(n = 1)$:

   $$\begin{aligned} I - I_R &= \int_a^b f(x)\mathrm{d}x - \underbrace{fa(b-a)}_{\text{Rect rule}} \\ &= \int_a^b f(x)\mathrm{d}x - \int_a^b f(a)\mathrm{d}x \\ &= \int_a^b [f(x) - f(a)]\mathrm{d}x \\ &= \int_a^b (x-a)f'(z_x)\mathrm{d}x && \text{(By Taylor)} \\ &= f'(z)\int_a^b (x-a)\mathrm{d}x && \text{(By MVT for integral)} \\ &= \frac{1}{2}(b-a)^2 - f'(z) && (h = b-a) \\ &= \frac{1}{2}(b-a)hf'(z) \end{aligned}$$

   So for $n = 1$ we proved that the error for the rectangle error can be expressed as in the last line above.
2. Inductive step: $(n > 1)$: We use $h = b - a/n$.

$$
\begin{aligned}
I - I_R &= \int_a^b f(x)\mathrm{d}x - h\sum_{i=0}^{n-1} f(x_i) \\
&= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x - h\sum_{i=0}^{n-1} f(x_i) \\
&= \sum_{i=0}^{n-1} \Big[\int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x - hf(x_i) \Big] \\
&= \sum_{i=0}^{n-1} \frac{1}{2}(\underbrace{x_{i+1} - x_i}_{h})^2 f'(z_i) \qquad z_i \in [x_i, x_{i+1}] \qquad \text{formula from Basic Rect Rule} \\
&= \frac{1}{2}h^2 \sum_{i=0}^{n-1} f'(z_i)
\end{aligned}
$$

Here let $1 := p(x)$ and $f'(z_i) := q(x)$ in the Summation MVT, then we can simplify the last expression

$$
\begin{aligned}
I - I_r &= \frac{1}{2}h^2 f'(z) \underbrace{\sum_{i=0}^{n-1} 1}_{=n} \\
&= \frac{1}{2}nh^2 f'(z) \\
&= \frac{1}{2}(b-a)hf'(z) \qquad (\text{since } h = b - a/n)
\end{aligned}
$$

We have proved both cases, so indeed we conclude that

$$
I - I_R := \text{error is } \mathcal{O}(h).
$$

> [Note] The error formula is not only useful in theory but also useful in practice

[Example] Suppose we use I_R to compute $\int_0^1 e^{-x^2}\mathrm{d}x$. Then how many points will there be required to ensure the absolute error is bounded above by 10^{-4}.

> if we take more points we reduce the error which is $\mathcal{O}(h)$

$$
\begin{aligned}
I - I_R &= \frac{1}{2}(b-a)hf'(z) \\
&= \frac{1}{2}\frac{1}{n}f'(z)
\end{aligned}
\qquad
\begin{cases}
b - a = 1 - 0 = 1 \\
h = \dfrac{b-a}{n} = \dfrac{1}{n} \\
z \in [0,1]
\end{cases}
$$

We calculate the maximum value of $f'(z)$ and make sure that this maximum with the constants in front is less than 10^{-4}.

- we do not know the value of z and $f'(z)$ but we can know $\max f'(z)$?

Now, get

$$
\max_{z\in[0,1]} \left|f'(z)\right|
$$

Set $f'' = 0$ to get the stationary point for $f'(x)$ \

$$
\begin{aligned}
f'(x) &= -2xe^{-x^2} \\
f''(x) &= -2e^{-x^2} - 2xe^{-x^2}(-2x) \\
&= 0 \implies x^\star = \sqrt{1/2} \\
\max_{z\in[0,1]} &= \max\{\left|f'(0), \left|f'(1), ' \, abs f'(x^\star)'\right|\right|\} \\
&= \max\{0, 2e^{-1}, \sqrt{2}e^{-1/2}\} \\
&= \sqrt{2}e^{-1/2}
\end{aligned}
$$

Now, to ensure that $|I - I_R| \leq 10^{-4}$, we require

$$
\begin{aligned}
\frac{1}{2}\frac{1}{n}\sqrt{2}e^{-1/2} &\leq 10^{-4} \\
n \geq \frac{1}{2}\sqrt{2}e^{-1/2}10^4 &= 4288.819 \\
\therefore n &= 4289
\end{aligned}
$$

- WIth the mid-point rule, on an interval $[a, b]$ and mid-point at $a + b - a/2$, the error is

$$(b-a)f(a+\frac{b-a}{2})$$

- with the trapezoid rule, on an interval $[a, b]$,the error is

$$\frac{1}{2}(b-a)[f(a)+f(b)]$$

Midpoint Rule

> We divide $[a, b]$ into multiple intervals and take the value at mid point for every subinterval, and use that value as height of the interval.

The width of each subinterval is h so we can write the area as :

$$I_M = h\sum_{i=0}^{n-1} f\left(a+\left(i+\frac{1}{2}\right)h\right)$$

The midpoint of each subinterval $a + (i + 1/2)h$.

Trapezoid Rule

> We divide $[a, b]$ into n subinterval and connect left and right subsequent points

The area is then

$$I_T = \frac{1}{2}h[f(a)+f(a+h)] + \frac{1}{2}h[f(a+h)+f(a+2h)] + \cdots + \frac{1}{2}h[f(b-h)+f(b)]$$

All terms except first and lasts appear twice so we can simplify this expression to

$$I_T = h\frac{1}{2}[f(a)+f(b)] + \sum_{i=1}^{n-1} f(a+ih)$$

> We can show that
>
> $$|I - I_M| = \frac{1}{24}(b-a)h^2 f''(z) \qquad z \in [a, b]$$
> $$|I - I_T| = -\frac{1}{12}(b-a)h^2 f''(z) \qquad z \in [a, b]$$
>
> the midpoint rule is slightly more accurate than the trapezoid rule. (**Proof left as an exercise**)
>
> [Note] Both are h^2

1. Midpoint and trapezoid rule are more accurate than rectangle rule.
2. Error in midpoint rule is slightly more accurate than the trapezoid rule.

[Remark] If our f is a linear polynomial function then the errors will be 0 because second derivative of a linear function is 0. i.e., $I = I_M = I_T$. It makes sense geometrically since it's a straight line so for the mid point the two excess cancel one another and for the trapezoid the new line goes over the original line.

[Remark] If $S(x)$ is a linear spline that interpolate $(x_i, f(x_i))$ ofr $i = 0, 1, \ldots, b$ then with $x = a + ih$,

$$I_T = \int_a^b S(x)\,\mathrm{d}x$$

Recursive Trapezoi Rule

Suppose $[a, b]$ is divided into 2^n equal subintervals. Then

$$I_T(2^n) = \frac{1}{2}h[f(a)+f(b)] + h\sum_{i=1}^{2n-1} f(a+ih), \quad h = \frac{b-1}{2^n}$$

The trapezoid rule for 2^{n-1} equal subintervals is

$$I_T(2^{n-1}) = \frac{1}{2}\tilde{h}[f(a)+f(b)] + \tilde{h}\sum_{i=1}^{2^{n-1}-1} f(a+i\tilde{h}), \quad \tilde{h} = \frac{b-1}{2^{n-1}} = 2h$$

Q after computing $I_T(2^{n-1})$ can we compute $I_t(2^n)$ by using $I_T(2^{n-1})$ and avoid reevaluating f at the old points?

Lecture 20

Recursive Trapezoid Rule

Suppose $[a,b]$ is divided into 2^n equal subintervals. Then

$$I_T(2^n) = \frac{1}{2}h[f(a)+f(b)] + h\sum_{i=1}^{2n-1} f(a+ih), \quad h = \frac{b-a}{2^n}$$

The trapezoid rule for 2^{n-1} equal subintervals is

$$I_T(2^{n-1}) = \frac{1}{2}\tilde{h}[f(a)+f(b)] + \tilde{h}\sum_{i=1}^{2^{n-1}-1} f(a+i\tilde{h}), \quad \tilde{h} = \frac{b-a}{2^{n-1}} = 2h$$

Q after computing $I_T(2^{n-1})$ can we compute $I_t(2^n)$ by using $I_T(2^{n-1})$ and avoid reevaluating f at the old points?

> To compute the first approximation based on the second approximation $I_T(2^{n-1})$ we use reccursion.

[Theorem]

$$I_T(2^n) = \frac{1}{2}I_T(2^{n-1}) + \sum_{i=1}^{2^{n-1}} f[a+(2i-1)h]$$

[Proof]

$$I_T(2^{n-1}) = h[f(a)+f(b)] + 2h\sum_{i=1}^{2^n-1} f(a+2ih)$$

On a line we have the points $a, a+h, a+2h, a+3h, \ldots, b$.

$$\begin{aligned} I_T(2^n) &= \frac{1}{2}h[f(a)+f(b)] + \sum_{i=1}^{2^n-1} f(a+ih) \qquad\qquad h = \frac{a-b}{2^n} \\ &= \frac{1}{2}I_T(2^{n-1}) + \Big[\cancelto{0}{\frac{1}{2}h[f(a)+f(b)]} + \underbrace{h\sum_{i=1}^{2^n-1} f(a+ih)}_{\star}\Big] \\ &\quad - \cancelto{0}{\frac{1}{2}h[f(a)+f(b)]} - h\sum_{i=1}^{2^{n-1}-1} f(a+2ih) \end{aligned}$$

Some terms cancel here,

We want to cancel the last term out so we put it in 2 parts

$$\sum_{i=1}^{2^n-1} f(a+ih) = \sum f(a+(2i-1)h) + \sum f(a+2ih)$$

The largest odd number in $(\star)$ is 2^{n-1} so in our decomposition (since $2i-1 = 2^n-1 \implies i = 2^{n-1}$)for the odd we have the bounds

$$\sum_{i=1}^{2^{n-1}} f(a+(2i-1)h)$$

We look at the last even number in $(\star)$, it is $2^{n-1}-1$ since $2i = 2^n-1-1 \implies i = 2^{n-1}-1$ so our bounds in the even decomposition becomes

$$\sum_{i=1}^{2^{n-1}-1} f(a+2ih)$$

Therefore, we have

$$
\begin{aligned}
I_T(2^n) &= \frac{1}{2}h[f(a)+f(b)] + \sum_{i=1}^{2^n-1} f(a+ih) \qquad h = \frac{a-b}{2^n} \\
&= \frac{1}{2}I_T(2^{n-1}) + \Big[\cancelto{0}{\frac{1}{2}h[f(a)+f(b)]} + \underbrace{h\sum_{i=1}^{2^n-1} f(a+ih)}_{\star} \Big] \\
&- \cancelto{0}{\frac{1}{2}h[f(a)+f(b)]} - h\sum_{i=1}^{2^{n-1}-1} f(a+2ih) \\
&= \frac{1}{2}I_T(2^{n-1}) + h\Big[\sum_{i=1}^{2^{n-1}} f(a+(2i-1)h) + \sum_{i=1}^{2^{n-1}-1} f(a+2ih) - \sum_{i=1}^{2^{n-1}-1} f(a+2ih)\Big] \\
&= \frac{1}{2}I_T(2^{n-1}) + h\sum_{i=1}^{2^{n-1}} f[a+(2i-1)h]
\end{aligned}
$$

> Q Why do we need this recursive formula ? What's the point ? Not all error formulas are easy to get, if we cannot get the error formulas, then we don't know how many points are needed for a certain level of accuracy, we can instead start with $n = 1$to get an approximation then $n = 2$ (divide interval into new parts) and get a new and better approximation. We continue this process until we reach desired accuracy.

Formally,

Motivation We need a recursive formula because sometimes it's difficult to get the error formula. Then we do not know how many points are needed. With the recursive formula, you can start with $n = 1$, then you just half and double n to get a better approximation. When the difference between $I_T(2^{n-1})$ and $I_T(2^n)$ is small you stop and use imation. $I_T(2^{n-1})$ as the final approx.

Simpson's Rule

> For the rectangle and trapezoid we used straight lines, but that is actually not necessary !

Let a function defined on $a, a+h, a+2h, \ldots$. Let $p_2(x)$ be the polynomial passing through the 3 points. Then we use that quadratic polynomial's integral for our rule.

We need to divide our interval a, b into n subintervals where n is even. For each pair, we have 3 points : $(a, f(a)), (a+h, f(a+h)), (a+2h, f(a+2h))$. Then add $+2h$ for next iteration pairs on the elements.

The are of the first 2 panels can be shown to be

$$
\int_a^{a+2h} p_2(x)\mathrm{d}x = \frac{h}{3}[f(a) + 4f(a+h) + f(a+2h)]
$$

Q How do we obtain this ?

Find the quadratic function $p_2(x)$ to interpolate the 3 points then compute the integral. (Lagrange interpolation) ?

We take the summation of all the expressions (each integral of 3 points) , we find that many terms cancel out

$$
\frac{h}{3}[f(a) + 4f(a+h + f(a+2h)]
$$

the green term appear once, the red term appear on every iteration and the orange term appear at most twice in every iteration ?

It can be show that for some $z \in [a, b]$,

$$
I - I_S = -\frac{1}{180}(b-a)h^4 f^(4)(z) = \mathcal{O}(h^4)
$$

The error formula only involves the fourth derivative of the function, so then we have

[Remark] If f is a polynomial of degree ≤ 3, then $I = I_S$. Since the derivative of a first order function is 0 ?

Formally,

- if the function itself is a quadratic poly, then the inter-poly $p_2(x)$ is just equal to $f(x)$. Then $I_S = I$.

- if the function is a cubic polynomial, although $p_2(x)$ is not equal to $f(x)$ but their integration are,

---

Adaptive Methods

Motivation

consider the left part of a a, b partition to be changing dramatically , while the right part is flat, then our panels have all the same weight. We need to adapt in that we have to use custom width of the panels.

$(\star)$ But we don't know in advance which part of the functions change much and which don't change too much

- it is not efficient to use the same panel width h everywhere on $[a, b]$.
- we can't know where the function changes rapidly.
- we want to develop an adaptive method to deal with this situation.

Idea divide $[a, b]$ into 2 equal subinterval ,apply approximation procedure, estimate error using these approximation , and if we are not satisfied with the error on a subinterval we divide that subinterval and continue that process.

> Similar procedure to the recursive formula.

```
function numl = adapt(f,a,b,epsilon, ...)
    /* Compute the integral from a and b in two ways and call the values I_1 and I_2 (assume I)2
is better than I_1), Estimate the error I-I_2 based on |I_2 - I_1|*/
    if |the estimated erro| <= epsilom, then
        nul = I_2 + the estimated error
    else
        c = (a+b)/2
        numl = adapt(f,a,c,epsilon/2, ...) + adapt(f,c,b,epsilon/2 ,...)
    end
```

Note for the subintervals we use tolerance $\epsilon/2$.

This will guarantee $|I - \text{numl}| \approx< \epsilon$

If the $|\text{the estimated error}| \leq \epsilon$ is true then the conclusion is true the interval a, b is divided in the intervals a, c and c, b with

$$I_L = adapt(f, a, c, \epsilon/2, \ldots), \qquad I_R = adapt(f, c, b, \epsilon/2, \ldots)$$

Details on how to setup the adaptive Simpson's rule

Define I_1 and I_2

- Simpson's rule for $n = 2$ gives $I = I_1 + E_1$

$$I_1 = \frac{b-a}{6}[f(a) + 4g\left(\frac{a+b}{2}\right) + f(b)]$$
$$E_1 = -\frac{1}{180}(b-a)\left(\frac{b-a}{2}\right)^4 f^{(4)}(z)$$

- Simpson's rule for $n = 4$ gives $I + I_2 + E_2$

$$I_2 = \frac{b-a}{12}\left[f(a) + 4f\left(a + \frac{b-a}{4}\right) + 2f(a + \frac{b-a}{2}) + 4f\,()\right]\ldots$$
$$E_2 = -\frac{1}{180}(b-a)\left(\frac{b-a}{4}\right)^4 f^{(4)}(\tilde{z})$$

> Forth derivative are on different points. In practice we regard

Estimating the error in I_2

We assume $f^{(4)}(z)$ in E_1 is equal to $f^{(4)}(\tilde{z})$ in E_2. Then we observe that

$$E_1 = 16E_2$$

Now since $I = I_1 + E_1 = I_2 + E_2$, we have

$$I_2 - I_1 = E_1 - E_2 = 16E_2 - E_2 = 15E_2$$

This gives an error estimate in I_2 as

$$E_2 = \frac{1}{15}(I_2 - I_1)$$

> so in every step we got approximation I_1 and I_2 using the Simpson's rule. We assumed the forth derivative is the same and got an expression for the error E_2.

[Example]

level 1	$[a - c - b]$
level 2	$[a - d - c - e - b]$
level 3	$[a - d_1 - d - d_2 - c - e_1 - e - e_2 - b]$
	$\vdots$

Lecture 21

Gaussian Quadrature Rules

> For any $n+1$ nodes $x_0 < x_1 < \cdots < x_n$ on $[a, b]$ we can find polynomial $p(x)$ of degree $\leq n$ to interpolate $(x_0, f(x_0)), \ldots (x_n, f(x_n))$.

We do not want to use regular points. We chose the nodes to get better approximations ?

Write

$$p(x) = \sum_{i=0}^{n} I_i(x) f(x_i), \qquad l_i(x) = \prod_{j=0, j\neq i}^{n} \left(\frac{x - x_j}{x_i - x_j} \right) \qquad (\star)$$

Then $\tilde{I} := \int_a^b p(x)\mathrm{d}x$ can be used as an approximation to $I = \int_a^b f(x)\mathrm{d}x$.

> We know we can find the polynomial $p(x)$ and interpolate this polynomial using the Lagrange form ($\star$) that passes through the $n+1$ points and has degree less or equal to n.

We use $\tilde{I}$ to denote it's integral

$$\tilde{I} = \sum_{i=0}^{n} \int_a^b I_i(x)\mathrm{d}x \times f(x_i) = \sum_{i=0}^{n} A_i f(x_i), \qquad A_i = \int_a^b I_i(x)\mathrm{d}x$$

Then if $f(x)$ is a polynomial with degree less or equal to n, then $p(x) = f(x)$ and we must have $\tilde{I} = I$, no matter how those nodes are chosen.

[Remark] we know that a polynomial of degree less or equal to n that passes through $n+1$ points (no matter which) , implies the polynomial is unique.

[Definition] (*Newton-Cotes formula*) : If we take $x_i = a + ih$ with $h = (b-a)/n$ then $\tilde{I} = \sum_{i=0}^{n} A_i f(x_i)$ is called the newton-cotes formula.

Q Can we chose node such that we can still get the exact integral $\tilde{I} = I$ with a polynomial of higher degree? I.e., a polynomial with degree $\not\leq n$?

A Yes, because we have $n+1$ points so unique polynomial. Now we have $n+1$ parameters to choose and we can expect to increase the highest degree from $n \to 2n+1$.

We can understand this by regarding A_i and x_i as unknowns. So there would be $2n+2$ unknowns. We want $\tilde{I} = I$ to hold for

$$f(x) = x^j, \qquad j = 0, 1, \ldots, k$$

if we can get exact integration for all the exact functions $x^0, x^1, \ldots x^k$, then we can find an exact polynomial with higher degree. So we have $k+1$ equations. we can expect the highest k will satisfy $k + 1 = 2n + 2$.

> Formally, if we can get exact integral for these monomials ($f(x) = x^j,\ j = 0 \to k$) ,where j is from $0 \to k$, then we can get exact integral for any polynomial with degree less than or equal to k .
>
> - Because any poly with degree less or equal to k is a linear combination of these monomials. So if we can get exact integration of these monomials then we can get exact integration of the polynomial with degree $\leq k$.
>
> Formally, because any polynomial with degree $\leq k$ is a linear combination of these monomials. $\implies$ its integral is also a linear combination of integrals of monomials

But how de we find the x_i and A_is ? Well if we have the x_is we can get the A_i.

[Theorem] Let q be a polynomial of degree $n+1$ such that

$$\int_a^b x^k q(x)\mathrm{d}x = 0, \qquad k = 0, 1, \ldots, n$$

Let $x_0, x_1, \ldots, x_n$ be the zeros of $q(x)$. Then for any polynomial o$f(x)$ of degree less than or equal to $2n+1$,

$$\int_a^b f(x)\mathrm{d}x = \sum_{i=0}^{n} A_i f(x_i),$$

$$\text{with} \quad A_i = \int_a^b I_i(x)\mathrm{d}x, \qquad I_i(x) = \prod_{j=0, j\neq i}^{n} \left(\frac{x - x_k}{x_i - x_j}\right)$$

[proof] Let $f(x)$ be any polynomial of degree $\leq wn+1$. Dividing $f(x)$ by $q(x)$ to obtain the quotient $p(x)$ and the remainder $r(x)$ (both polynomials with degree $\leq n$), then

$$f(x) = p(x)q(x) + r(x) \qquad (\text{here } q(x) \text{ has degree exactly } n+1)$$

Since $q(x_i) = 0, f(x_i) = r(x_i)$ for $i = 0 \to n$, then

$$r(x) = \sum_{i=0}^{n} I_i(x) r(x_i) = \sum_{i=0}^{n} I_i(x) f(x_i)$$

Thus, with $p(x) := \sum_{i=0}^{n} c_i x^i$,

$$\begin{aligned}
\int_a^b f(x)\mathrm{d}x &= \int_a^b p(x)q(x)\mathrm{d}x = \int_a^b f(x)\mathrm{d}x \\
&= \int_a^b \left(\sum_{i=0}^{n} c_i x^i\right) q(x)\mathrm{d}x + \int_a^b \sum_{i=0}^{n} I_i(x) f(x_i)\mathrm{d}x \\
&= \sum_{i=0}^{n} \left(\int_a^b I_i(x)\mathrm{d}x\right) f(x_i)
\end{aligned}$$

[Definition] (*Gaussian quadrature rule*) : Any $I_G = \sum_{i=0}^{n} A_i f(x_i)$ with x_i and A_i $(i = 0, 1, \ldots, n)$ defined as in the above theorem is called Gaussian quadrature rule.

Q How to find $q(x)$?

If the interval $[a, b]$ is $[-1, 1]$, the Legendre polynomial $q_{n+1}(x)$ is defined by

$$q_{n+1}(x) = \frac{2n+1}{n+1} q_n(x) - \frac{n}{n+1} q_{n-1}(x), \quad q_o(x) = 1. \quad q_1(x) = x$$

satisfies

$$\int_a^b x^k q_{n+1}(x)\mathrm{d}x = 0$$

Thus, the roots of $q_{n+1}(x)$ = 0 are the nodes of Gaussian quadrature rule for

$$\int_{-1}^{1} f(x)\mathrm{d}x$$

[Example] Derive the Gaussain quadrature of the form

$$I_G(1) = A_0 f(x_0) + A_1 f(x_1) \qquad \text{for} \quad \int_{-1}^{1} f(x)\mathrm{d}x$$

we have 4 unknowns.

- Method 1 : The Legendre polynomial $q_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$ (where we took $n = 1$in $q_{n+1}(x)$ formula defined above)

$$q_2(x) = \frac{2 \times 1 + 1}{1 + 1} \times q_1(x) - \frac{1}{2} q_0(x) \implies \int_a^b x^k q_{n+1}\mathrm{d}x = \frac{3}{2}x^2 - \frac{1}{2}$$

. thus the two nodes are

$$\frac{3}{2}x^2 - \frac{1}{2} = 0 \implies 3x^2 - 1 = 0 \xrightarrow{\text{so then we get}} x_0 = -\frac{1}{\sqrt{3}}, \; x_1 = \frac{1}{\sqrt{3}}$$

Then,

$$A_0 = \int_{-1}^{1} \underbrace{\frac{x - \frac{1}{\sqrt{3}}}{-\frac{2}{\sqrt{3}}}}_{\frac{x - x_1}{x_0 - x_1}} \mathrm{d}x = 1. \quad A_1 = \int_{-1}^{1} \underbrace{\frac{x + \frac{1}{\sqrt{3}}}{\frac{2}{\sqrt{3}}}}_{\frac{x - x_0}{x_1 - x_0}} \mathrm{d}x = 1$$

Finally,

$$A_0 f(x_0) + A_1 f(x_1) = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right)$$

$2n + 2k + 1$ so $n = 3$ is the highest degree for exact solution.

- Method 2 (if you don't remember the Legendre polynomials) Let $q(x) = c_0 + c_1 x + c_2 x^2$. Set
  - $$\int_{-1}^{1} x^k q(x) \mathrm{d}x = 0, \quad k = 0, 1$$

$$k = 0, \quad c_0 x\Big|_{-1}^{1} + \frac{1}{2} c_1 x^2\Big|_{-1}^{1} + \frac{1}{3} c_2 x^3\Big|_{-1}^{1} = 0 \implies 2c_0 + \frac{2}{3} = 0$$
$$k = 1, \quad \frac{1}{2} c_0 x^2\Big|_{-1}^{1} + \frac{1}{3} c_1 x^3\Big|_{-1}^{1} + \frac{1}{4} c_2 x^4\Big|_{-1}^{1} = 0 \implies \frac{2}{3} c_1 = 0$$

  Where we used $x^k = 1$ and $x^k = x$ in the above integral. We solve and get $c_0 = -\frac{1}{3} c_2$ and $c_1 = 0$

  Thus,

$$q(x) = c_0 + c_2 x^2 = c_0 - 3c_0 x^2 = c_0(1 - 3x^2)$$

  > Note that the solution is not unique here which is fine since we only want the zeros of this

The zeros of $q(x)$ are $x_0 = -1/\sqrt{3}$ and $x_1 = 1/\sqrt{3}$. Then we can compute A_0 and A_1 with the same integrals as in Method 1 :

$$A_0 = \int_{-1}^{1} \underbrace{\frac{x - \frac{1}{\sqrt{3}}}{-\frac{2}{\sqrt{3}}}}_{\frac{x - x_1}{x_0 - x_1}} \mathrm{d}x = 1. \quad A_1 = \int_{-1}^{1} \underbrace{\frac{x + \frac{1}{\sqrt{3}}}{\frac{2}{\sqrt{3}}}}_{\frac{x - x_0}{x_1 - x_0}} \mathrm{d}x = 1$$

- Method 3 Take $f(x) = x^j$. Then,

$$I = \int_{-1}^{1} f(x) \mathrm{d}x = \int_{-1}^{1} x^j \, dx = \frac{1 - (-1)^{j+1}}{j + 1}$$

Set

$$A_0 x_0^j + A_1 x_1^j = \frac{1 - (-1)^{j+1}}{j + 1}$$

For $j = 0, 1, 2, 3$ we obtain,

$$\text{for } j = 0, \ \ x^0 = 1, \qquad \int_{-1}^{1} x^0 \mathrm{d}x = 2 = A_0 + A_1 \tag{1}$$
$$\text{for } j = 1 \ \ x^1 = x, \qquad \int_{-1}^{1} x \mathrm{d}x = 0 = A_0 x_0 + A_1 x_1 \tag{2}$$
$$\text{for } j = 2 \ \ x^2, \qquad \int_{-1}^{1} x^2 \mathrm{d}x = \frac{2}{3} = A_0 x_0^2 + A_1 x_1^2 \tag{3}$$
$$\text{for } j = 3 \ \ x^3, \qquad \int_{-1}^{1} x^3 \mathrm{d}x = 0 = A_0 x_0^3 + A_1 x_1^3 \tag{4}$$

This is a system of 4 equations which we can solve by hands.

- Equation 2 and 4 gives us $A_0 x_0 (x_0^2 - x_1^2) = 0$.
- if $A_0 x_0 = 0$, then equation 2 gives $A_1 x_1 = 0$ which contradicts equation 3. Thus we must have $x_0^2 - x_1^2 = 0$
- if $x_0 = x_1$ then equation 2 says $A_0 + A_1 = 0$ which contradicts 1. Thus, we must have $x_0 = x_1$

So

$$\left.\begin{array}{ll}(2) & A_0 - A_1 = 0 \\ (1) & A_0 + A_1 = 2\end{array}\right\} \implies A_0 = 1, \quad A_1 = 1$$

And from equation 3 we get $x_0^2 = 1/3$ since $x_0^2 + x_1^2 = \frac{1}{3}$.

Interval transformation

We defined the rule for the interval $[-1, 1]$, but we want to extend this rule. To convert $[a, b] \to [-1, 1]$, let

$$x = \alpha + \beta t$$

Set $a = \alpha - \beta$ and $b = \alpha + \beta$. Then,

$$\int_a^b f(x)\mathrm{d}x = \beta \int_{-1}^{1} \underbrace{f(\alpha + \beta t)}_{:= g(t)}\mathrm{d}t := \beta \int_{-1}^{1} g(t)\mathrm{d}t$$

Then we obtain

$$I_G[a, b] = \beta \sum_{i=0}^{n} A_i g(t_i) = \beta \sum_{i=0}^{n} A_i f(\alpha + \beta t_i)$$

Q If $f(x)$ is a polynomial with degree $\leq 2n + 1$ is it true that

$$I_G[a, b] = \int_a^b f(x)\mathrm{d}x \ ?$$

Yes , we do linear transformation and we still have the property.

Lecture 22

ODEs

Euler's Method (1768)

The goal is to find approximate values of the solution of the IVP over the interval $[a, b]$.

We use $n + 1$ points $t_0, t_1, \ldots, t_n$ to equally partition $[a, b]$, with

[Definition] (*Step size*): $h = t_{i+1} - t_i = (b - a)/n$

Suppose we have already obtained x_i, an approximation to $x(t_i)$ and would like to get x_{i+1}, an approximation to $x(t_{i+1})$.

Then the Taylor series expansion is

$$x(t_{i+1}) \approx x(t_i) + (t_{i+1} - t_i)x'(t_i) = x(t_i) + hf(t_i, x(t_i))$$

Out ODE here is

$$x'(t) = f(t, x(t))$$

> we do not know $x(t_i)$, but we assume we have t_i as an approximation to $x(t_i)$?

This leads to Euler's method

$$x_{i+1} = x_i + hf(t_i, x_i), \qquad i = 0, 1, \ldots, n - 1$$

where $x_0 = x(a)$ is given.

Q Derive Euler's method by the rectangle rule for integration.

Since $x'(t) = f(t, x(t))$, we have

$$\int_{t_i}^{t_{i+1}} x'(t)\mathrm{d}t = \int_{t_i}^{t_{i+1}} f(x, x(t))\mathrm{d}t$$

This gives

$$x(t_{i+1}) - x(t_i)) = \int_{t_i}^{t_{i+1}} f(t, x(t))\mathrm{d}t = hf(t_i, x(t_i)) + \underbrace{\frac{1}{2}h^2 x''(z_i)}_{\text{error from rect rule}},$$

again leading to Euler's scheme (*ignoring the error term ?*):

$$x_{i+1} - x_i = hf(t_i, x_i)$$

[Remark] In Euler's method, we chose a constant step size h. But it may be more efficient to choose a different step size h_i at each point t_i according to $f(x,t)$

[Example] Use Euler's method to solve

$$\begin{cases} x' &= x \\ x(0) &= 1 \end{cases} \qquad \text{over } [0,4]$$

with $n = 20$. What do you observe ? How do you explain what you observed ? The exact solution is $x(t) = e^x$

```
function [t,x] = euler(f,a,b,x0,n)

h = (b-a)/n;
t = linspace(a,b,n+1);
x(1) = x0;
for i = 1:n
    fval = f(t(i) , x(i));
    x(i+1) = x(i) + h*fval;
end
```

We use this with

```
a=0; b=4; x0=1; n=20;
s= 0:0.01:4;
x = exp(s);
t  = linspace(a,b,n+1);
f= @(t,x)x;
[t,x_e]= euler(f,a,b,x0,n);
figure
plot(s,y 'k-', t , x_e, 'ro');
%legend label panel,etc
```

This produces an exponential function with discrete points computed by EM, and an exact solution curve above the points.

$\forall\ t_i$ our approximated value is smaller than the exact solution, and the difference becomes larger and larger

> approximation becomes worse and worse as t increases.

Q Why do we get a worse approximation as t increases ?

A Because error accumulates based on previous errors.

Back to the example

The taylor theorem gives

$$x(t_{i+1}) - x(t_i)) = \int_{t_i}^{t_{i+1}} f(t, x(t))\mathrm{d}t = hf(t_i, x(t_i)) + \underbrace{\frac{1}{2}h^2 x''(z_i)}_{\text{error from rect rule}}, \quad z \in [t_i, t_{i+1}]$$

Euler's method gives

$$x_{i+1} = x_i + hf(t_i, x_i)$$
$$\implies x(t_{i+1}) - x_{i+1} = x(t_i) - x_i hf[f(t_i, x(t_i)) - f(t_i, x_i)] + \frac{1}{2}h^2 x''(z_{i+1})$$

[Note] If $x(t_i) = x_i$, then the first two terms above vanish

The global error at t_{i+1} : $x(t_{i+1}) - x_{i+1}$ comes from two sources:

1. the local truncation error : $\frac{1}{2}h^2 x''(z_{i+1})$. If $x_i = x(t_i)$, then the local truncation error is jus the global error.

2. the propagation error : $x(t_i) - x_i + h[f(t_i, x(t_i)) - f(t_i, x_i)]$. This is due to the accumulate effects of all local truncation erros at $t_1, t_2, \ldots, t_n$.

[Note] when performing computation there is an additional rounding error.

[remark] THere is a few techniques to determine the step size h according to error analysis (*not seen in this class*)

The trapezoid Euler Method

This is more accurate than Rectangle rule.

$$\int_{t_i}^{t_{i+1}} x'(t)\mathrm{d}t = \int_{t_i}^{t_{i+1}} f(t, x(t))\mathrm{d}t$$

Thus,

$$\begin{aligned} x(t_{i+1}) - x(t_i) &= \int_{t_i}^{t_{i+1}} f(t, x(t))\mathrm{d}t \\ &= \frac{1}{2}h\Big[f(t_i, x(t_i)) + f(t_{i+1}, x(t_{i+1}))\Big] - \frac{1}{12}h^3 x'''(z_i) \end{aligned}$$

Use x_i and x_{i+1} to replace $x(t_i)$ and $x(t_{i+1})$ and ignore the error term :

remark The RHS involves x_{i+1} so we can't just use this rule?

To get around it, replace that x_{i+1} by $x_i + hf()$

[Definitiom] (*Improved Euler's Method*)

$$\begin{cases} \hat{x}_{i+1} = x_i + hf(t_i, x_i) \\ x_{i+1} = x_i + \frac{1}{2}h[f(t_i, x_i) + f(t_{i+1}, \hat{x}_{x+1}], \quad i = 0, 1, \ldots \\ x_0 = x(a) \end{cases}$$

But here we will call it The trapezoind Euler method.

[Definition] (*The midpoint Euler method*)

We want to compare these three methods

[Example] Recall the example from before

> Here the Middle Euler's method and Trapezoid Euler's method are better than Euler's method, since it matches the exact solution.

Q Can we conclude that TEM and MEM is better than EM ?

A no because the formers have much more cost. So we need to be fair here. Indeed for the formers we do 2 methods evaluations at each step while for EM it's only one method evaluation at each step .

To make a fair comparison,

```
% we set
[t,x_e] = euler(f,a,b,x0,n)
[t,x_t] = trap_euler(f,a,b,x0,2n) % 2n now make sure it takes twice longer
```

Even with this modification , EM is still not as accurate, despite having the same cost, so we can conclude that MEM and TEM are actually better than EM !

General Taylor series method

$$x(t_{i+1}) = x(t_i) + hx'(x_i) + \frac{1}{2!}h^2 x''(t_i) + \cdots + \frac{1}{m!}h^m x^(m)(t_i) + \mathcal{O}(h^{m-1})$$

From $x^t = f(t, x)$ we can compute $x'', \ldots, x^(m)$. Define $x_i', x_i'', \ldots, x_i^(m)$ as approximations to $x'(t_i), x''(t_i), \ldots, x^(m)(t_i)$, respectively,

then we have a taylor expansion :

[Example] $x' = 1 + x^2 + t^3$ with $x(1) = -4$, over $[1, 2]$.

$$\begin{aligned} x' &= 1 + x^2 + t^3, \quad \underbrace{x_i' = 1 + x_i^2 + t_i^3}_{\text{apprixmated value}} \\ x'' &= 2xx' + 3t^2, \quad x_i' = 2x_i x_i' + 3t_i^2 \\ x''' &= 2x'^2 + 2xx'' + 6t, \quad x_i''' = 2x_i'^2 + 2x_i x_i'' + 6t_i \\ x^{(}4) &= 4x'x'' + 2x'x'' + 2xx''' + 6, \quad x^{(}4) = 4x_i' x_i'' + 2x_i' x_i'' + 2x_i x_i''' + 6 \end{aligned}$$

The taylor series method of order 4 is

$$\begin{aligned} x_{i+1} &= x_i + hx_i' + \frac{1}{2!}h^2 x_i'' + \frac{1}{3!}h^3 x_i''' + \frac{1}{4!}h^4 x_i^{(}4) \\ &= x_i + h\left(x_i' + h\left(\frac{1}{2!}x_i'' + h\left(\frac{1}{3!}x_i''' + h\frac{1}{4!}x_i^{(}4)\right)\right)\right) \end{aligned}$$

[Remarks]

1. Euler's method is a Taylor series method of order 1
2. if $f(t, x)$ is complicated, then high-order Taylor series method may be very complication

> this method is not often used because it requires a lot of analytical work

---

Runge-Kutta methods of order 2

Goal develop a class of methods, including the trapezoid Euler method and the midpoint Euler method.

The trapezoid Euler scheme : (setting $w_1 = 1/2, w_2 = 1/2, \alpha = 0, \beta - 1$)

$$x_{i+1} = x_i + \frac{1}{2}h\Big[f(t_i, x_i) + f(t_{i+1}, x_9 + hf(t_i, x_i))\Big]$$

The midpoint Euler scheme : (setting $w_1 = 0, w_2 = 1, \alpha = 1/2, \beta = 1/2$)

$$x_{i+1} = x_i + hf\left(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hf(t_i, x_i)\right)$$

Write a general scheme

$$\begin{aligned} x_{i+1} &= x_i + w_1 K_1 + w_2 K_2 \\ K_1 &= hf(t_i, x_i), \dots \end{aligned}$$

Since $x'(t) = f(t, x(t))$,

$$\begin{aligned} x''(t) &= \frac{\partial f(t, x(t))}{\partial t} + \frac{\partial f(t, x(t))}{\partial x}x'(t) \\ &= \frac{\partial f(t, x(t))}{\partial t} + \frac{\partial f(t, x(t))}{\partial x}f(t, x(t)) \end{aligned}$$

Setting

$$x'(t) = f(t, x(t)) \qquad \text{, and } x'(t_i) = f(t_i, x(t)i)$$

Then by Taylor's theorem we have

$$\begin{aligned} x(t_{i+1}) &= x(t_i) + x'(t_i)h + \frac{1}{2}x''(t_i)h^2 + \mathcal{O}(h^3) \\ &= x(t_i) + f(t_i, x(t_i))h \\ &+ \frac{1}{2}\Big[\frac{\partial f(t, x(t))}{\partial t} + \frac{\partial f(t, x(t))}{\partial x}f(t_i, x(t_i))\Big]h^2 + \mathcal{O}(h^3) \end{aligned} \tag{1}$$

In the Range-Kutta method

We let

$$\begin{cases} K_1 = hf(t_i, x_i) \\ K_2 = hf(t_i + \alpha h, x_i + \beta K_1) \end{cases}$$

$$
\begin{aligned}
x_{i+1} &= x_i + w_1 K_1 + w_2 K_2 \\
&= x_i + w_1 h f(t_i, x_i) + w_2 h f(t_i, x_i) \\
&+ \frac{\partial f(t_i, x_i)}{\partial t} w_2 \alpha h^2 + \frac{\partial f(t_i, x_i)}{\partial x} w_2 \beta h^2 f(t_i, x_i) + \mathcal{O}(h^3) \\
&= x_i + (w_1 + w_2) f(t_i, x_i) h \\
&= \left[w_2 \alpha \frac{\partial f(t_i, x_i)}{\partial t} + w_2 \beta \frac{\partial f(t_i, x_i)}{\partial x} f(t_i, x_i) \right] h^2 + \mathcal{O}(h^3)
\end{aligned}
$$

Lecture 23

ODEs

> Recall in MATLAB we took $2n$ in Euler method to make a fair comparison, to have the same cost. Then Euler, trapezoid Euler and mid point Euler all have the same cost.
>
> For Rankouta method we take $n/2$ to have the same cost as well

- ⋆ The above produce RKM and MEME is very close to the exact solution.
- The computed solution TEM is slightly lover
- EM $(2n)$ is even lower and EM normal is lowermost.

We conclude that RKM $(n/2)$ is the most accurate method of the all.

Runge-Kutta Methods of order 2

We said we can extend it to a higher order. We have order 4 . K_1, K_2, K_3, K_4.

[Note] we don't need to know the K_i by hearth ? Not sure.

[Remark] The local truncation error is $\mathcal{O}(h^5)$, while that of order 2 is $\mathcal{O}(h^3)$

> In order 2 at each step we do 2 function evaluations while for order 4 we do 4 evaluations for each step.

[Note] (*MATLAB*) : There exist the inbuild methods `ode23` (2n and 3rd order Runge-Kutta) and `ode45` (4th and 5rd order Runge-Kutta)

Q Why do these methods use a pair of methods ? Because it uses an adaptive Simpson's rule. Use pair of methods to get estimated error for better method then if approximation is good we use larger step size and if bad we use smaller step size for better approximation.

> Formally, we use a pair to estimate the error then use adaptative stepsize.

[Note] Understand very well the adaptive idea.

Final

All material *before* and *after* midterm.

- 6 big questions

⋆ 1 question for polynomial interpolation and spline interpolation , least squares approximation (worth 25 points)

approximate through these points

Review Interpolation

> we are given a table of points and want to find $p(x)$ such that . There exist a unique polynomial with degree less or equal to n to interpolate $n+1$ points.

Recall existence and uniqueness.

- Vandermode approach $p(x) = c_0 + c_1 x + \cdots + c_n x^n$, this is the standard expression. This polynomial has $n+1$ coefficients and we need to find them. We solve $p(x_i) = y_i$, and solve the linear system to get the coefficients c_i. Advantage :
- Lagrange approach , we write $p(x) = \sum_{i=0}^{n} l_i(x) y_i$, we can write the polynomial explicitly.

  Recall know the formula for $l_i(x)$ Lagrange coefficients (*Cardinal functions*)
- Newton approach $p(x) = a_0 + a_1(x - x_0) + \cdots + a_n(x - x_0)(x - x_1) \ldots (x - x_n)$. Then we solve the system formed by $p(x_i) = y_i$.

Polynomial evaluation $\rightarrow$ we are given a special x and we want to compute the $f(x)$ in an efficient way

- nested multiplication technique to reduce cost of evaluation

Recall ($\star$) (*Runge Function*) : using high degree polynomials to do interpolation is dangerous

$$f(x) = \frac{1}{25x^2}, \quad x \in [-1, 1]$$

we found that the interpolating polynomial does not converge to x for all points. At the end of the interval the interpolation becomes worse and worse. We use this *runge* function to illustrate the high degree polynomial danger. Because of the *round* phenomenon.

[Note] no need to know proof for runge phenomenon (or round phenomenon not sure what prof said)

Spline interpolation

high degree interpolation is not a good idea so we use piecewise interpolation (so we keep the degree low).

- need to understand *smoothness* conditions.

> The t_is in the table are called *knots* and they are in order. Assume this order to get piecewise polynomial

- Linear spline : connect points to get a straight line

  Need lots of points otherwise curve doesn't look smooth

- Cubic spline : Piecewise cubic polynomials. Each S_i is a cubic polynomial.

  $$\overbrace{t_{i-1} - - - t_i}^{S_{i-1}()} \overbrace{- - - t_{i+1}}^{S_i(x)} \qquad , i = 1, \ldots, n-1$$

  We have also

  $$y_i = S_{i-1}(t_i) = S_i(t_i) \qquad S_{i-1}(t_i) = S_i'(t_i) \qquad S_{i-1}''(t_i) = S_i''(t_i)$$

  and end point conditions

  $$S_0(t_0) = y_0 \qquad S_{n-1}(t_n) = y_n$$

  So we have $4n - 2$ unknowns.

  This leads us to Natural Cubic spline because it gives us 2 extras conditions so it's unique !

  $$S_0''(t_0) = S_{n-1}''(t_n) = 0$$

  [Note] Don't use tridiagonal system approach, because need to recall difficult equations. Just remember the conditions and use them to setup equations to solve for the natural cubic spline.

- Quadratic spline : Maybe we have to derive in the final. Need to know the equations that the quadratic spline need to satisfy. Need to know why we need to impose these conditions

---

Least Squares Approximation

> Motivation is similar, we want low degree polynomial for interpolation.

We are given $m + 1$ points and want to find $p(x)$ which may be a polynomial or something else
$p(x) = c_0 g_0(x) \mid c_1 g_1(x) \mid \ldots$, such that

$$\sum_{i=0}^{m} (p(x_i) - y_i)^2$$

is minimum. Here the $g(x)_i$ are the basis functions; they are known.

The least squares problem is

$$\min_c \|Ac - y\|_2^2$$

[Note] The solution (*optimal* c) satisfies the normal equations $A^T Ac = A^T y$.

[Remark] ($\star$) We use matrix vector language to describe the problem

$$\text{Given } c = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}, \qquad y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}, \qquad A = \begin{pmatrix} g_0(x_0) & g_1(x_0) & \dots & g_n(x_0) \\ g_0(x_1) & g_1(x_1) & \dots & g_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ g_0(x_m) & g_1(x_m) & \dots & g_n(x_m) \end{pmatrix}$$

[Note] Here we have $m+1$ rows (number of points) and $n+1$ columns (basis functions). It has more rows than columns

We get c by solving $A^TAc = A^Ty$

[Remark] A^TA is nonsingular. Because indeed the columns of matrix A are linearly independent (full column rank) then A^TA is nonsingular so we have a unique solution

[Remark] (*Special case*) : if $m = c$ then A becomes a square matrix , then $g_i(x) = x^i$ and the problem becomes polynomial interpolation problem.

- setup A
- write $A^TA = A^Ty$
- solve for c

Numerical Integration

(⋆) 1 question worth 25 points (several small questions)

> We want to design numerical solution algorithm to approximate value of integral
>
> $$\int_a^b f(x)\mathrm{d}x$$
>
> we can have analytical solutions, but in practice and applications we can't always get the antiderivative.

- Rectangle Rule : Most simple, $I - I_R = \mathcal{O}(h)$
- Midpoint rule : $I - I_M = \mathcal{O}(h^2)$
- Trapezoid rules $I - I_T = \mathcal{O}(h^2)$

(⋆)For these rules need to know derivation and for the rectangle rule need to know how to additionally get it's error.

(⋆) Need to know the $\mathcal{O}(h^i)$ values for each, (*order of error*). Know the motivations.

To get error divide and apply basic rectangle rule, get error for composite rectangle rule ,and show the error behaves like $\mathcal{O}(h)$

Derive formulas for all 3 rules. Look at the geometrical and derive the formulas.

- Simpson's rule $I - I_MS = \mathcal{O}(h^4)$

Need to know how to derive the Simpson's rule.

- Recursive trapezoid rules (basic ideas and it's enough) No need to know details
- (⋆)Adaptive Simpson's Method

why use adaptive and how it works, need to know ideas and why and how it works not all details.

- Gaussian quadrature rule

In the previous rules we take the nodes and partititon in equal subinterval. But here we assume that the knodes are unknowns , We approximate with

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{n} A_i f(x_i)$$

We have $2n+2$ unknowns (A_i and x_i), so we get exact polynomial for degree less or equal to $2n+1$.

[Remark] Take the $f(x_i)$ to be the monomials x_i . Force the equality to get exact integrals for the monomials, get the system of equations and solve the unknowns.

[Note] Need to know why we use the monomials ?

Need to know how to do interval transformation to extend rule from $[-1,1]$ to $[a,b]$.

Numerical Solutions to ODE

(⋆) 1 big question for 10 points.

- We know

$$\begin{cases} x' = f(t, x(t)) \\ x(a) = x_0 \end{cases}$$

  We want to get a numerical solution for $x(t)$. We find a sequence of points (t_i, x_i) where $x_i \approx x(t_i)$ is an approximation.

  We want to get continuous approximation using spline interpolation or polynomial interpolation

- Euler's method
  1. Integration the rectangle rule for integral
  2. Taylor's series expansion
- Midpoint Euler's method
- Trapezoid Euler's method

Need to know the formulas, and how to derive them for Euler's emthod, Midpoint and trapezoid.

[Note] (*Error analysis*) : know error analysis regarding Euler's method .The errors have 2 sources

1. local truncation error
2. propagation error : caused by previous local truncation error (accumulate to become the propagation error).

When we solve the ODE nuemrically when t increases error becomes larger because we accumulate the local truncation error.

- Taylor's series method

[remark Need to calculate high order derivaitve by hand., so in practice it;s not used

- Runge-Kutta method of order 2
- Runge-Kutta method of order 4

in practice we use order 4

[Remark] Only need basic ideas of these methods, nor how to derive. To difficult to memorize.

Computer numbers adn arithmetic (1 big question 15 points [*quite long btw*])

- IEE single format F P system (all questions use single format)

  Hidden bit, exponent bias, machine precision, machine epsilon

- (computer) FPN
- special numbers
- normal numbers (largest and smallest)
- subnormal numbers (largest and smallest)
- round to nearest

  Also need to know : if we're given the rounded nomber then who is the candidate for the original number

- Absolute rounding error and relative rounding erros

  their bounds, specificaly for round to nearest, the absolute rounding error is half the gap

$$- - _{x_-} - - x - - - x_+ - --$$

  Relative round error :

$$\frac{|\text{round}(x) - x|}{|x|} \begin{cases} \leq \epsilon \\ \leq \frac{1}{2}\epsilon \end{cases}$$

  when x is subnormal number then this bound may not hold.

- F P operations

  computed result should be the correct rounded value of the result. Last question of Ass 1 , question about rounding error with operation on FPN, we need to be able to do that question !!!!

calculate the gaps, able to do some derivations about these gaps, rounding errors ,Ass questions about the F P operations.

(⋆) Other contents for this chapter will not be in the final!!

Taylor series, derivative approx, numerical cancellation

> no big question on this top

But need to know the techniques introduced in these topics, and understand them to apply in other topics.

- Newton method for solving nonlinear equations (need taylor expansions)
- error analysis for composite rectangle rule (neeed ?) *numerial theorem* ????????????? wtf is she saying

$$E_{n+1} = \frac{f^{(n+1)(z)}}{(n+1)!} h^{(n+1)}, z \in [x, x+h]$$

Solving a linear system of equations (1 question 10 points)

- Solving tridiagonal system by GENP (Important)

$$\begin{pmatrix} d_1 & c_1 & & & & \\ a_1 & d_2 & c_2 & & & \\ & & \ddots & & & \\ & & a_{n-1} & d_{n-1} & c_{n-1} & \\ & & & & a_{n-1} & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}$$

  We only store (update) d and b during the algorith musing the values of c and a. so we don't store the whole tridiagonal matrix. we only use a 4 array to store a, b, c, d , instead of a 2D array for the whole matrix (so we save space and cost)

  Need to be able to use the algorithm ,do cost analysis

  We need to be able to solve using both GENP and GEPP (in lecture notes it's GENP but we need to be able to do both).

  For SDDR then GEPP will not break down.

  > When we talk about natural cubic spline, we probed that we get a tridiagonal matrix. That matrix is SDDC, in that case we don't ahve row permutation so GEPP is GENP in that case only.

  Neeed to be familiar with SDDC and SDDR.

$$\text{SDDC} \qquad |a_{jj}| > \sum_{i=1, i\neq 1 j}^{n} |a_{ij}|$$

- $Ax = b$, A nonsingular $n \times n$ matrix
- To sovle this problem we introduce
  - GENP (1. Forward elimination , get upper triangular U 2. Back substitution ; $A = LU$)

    Understand why it gives the LU factorization and what is LU factorization. It is the forward elimination process .

    Formally, for GENP , $Ax = b \implies LU$ of A , then we can just solve $Ly = b$ and $Ux = y$, both L, U are triangular matrices so we can do this by back substition

    In some cases it can break down, it has disadvantages.
  - GEPP $PA = LU$: elimination with partial pivoting.

    Difference is that in forwar elimination process we need to do partial pivoting. Need to understand what is partial pivoting. In the kth step of forward elimination , the matrix loks like

$$\begin{pmatrix} a_{11} & \dots & a_{1k} & \dots & a_{in} \\ & \ddots & & & \\ & & a_{kk} & \dots & a_{kn} \\ & & \vdots & & \vdots \\ & & a_{ik} & & \\ & & a_{nk} & & a_{nn} \end{pmatrix}$$

we exchange the largest magnitude element (swap). because we have $n-1$ steps then we use $n-1$ permutations in forwad elimination.

P_k is the identity matrix with swapped kth row.

$$P_k = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 0 & \dots & 1 \\ & & & \ddots & \\ & & 1 & & 0 \\ & & & & 1 \end{pmatrix}$$

where k is the column $0, 1$

$$P = P_{n-1} \dots P_2 P_1$$

(⋆) we store the multipliers in the lower matrix L.

With GEPP , we solve $Ly = Pb$ then $Ux = y$.

We showed GEPP is numerically stabl, so compute solution is nearby ?

- condition number of problem (of the matrix A)
- stability of algorithm

> Cost analysis flops and comparisons for both algorithms

Solving nonlinear equations (1 question 20 points)

- $f(x) = 0$
- Iterative method: to approximat ethe rules ? It consutrcts a sequence $x_1 \dots x_n \dots$ and we hope that it converges to a root of $f(x) = 0$

  Need to know the motivation for iterative method
- Bisection method : $[a, b]$ interval, if $f(a)f(b) < 0$ then if f is continuous we know there's a root in this interval. we define a mid point c and iterative reccursively by dividing the interval into half. When the length is small enough we stop the iteration. After all the steps,

  $$|r - C_n| \leq \frac{1}{2^n}|b - a| \qquad C_n \xrightarrow{n \to \infty} r$$

  so solution is guaranteed. Although our sequence c_n does not have a converge to r , the upper bound $1/2^n\,|b-a|$ has linear convergence to 0.

  ⋆ need to explain why convergence solution is guaranteed.
- Newton Method

  Has quadratic convergence rate. Need to compute fisrst derivative of function. Gives us faster convergence.

  Need to assume first derivative not zer and that it exists because of the iteration

  $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

  Need to know quadratic convergence rate : if $\{x_n\} \to r$, then

  $$\lim_{n \to \infty} \frac{|x_{n+1} - r|}{|x_n - r|^2} = \left| \frac{f''(r)}{2f'(r)} \right| = C$$

  (⋆) ↑ Need to show and prove the quadratic converge rate for Newton's method . Usually this C is not 0. Accuracy goes from $3 \to 6 \to 12$ digits at each iteration .

In Ass. we proved

$$\lim_{n\to\infty} \frac{f(x_{n+1})}{f^2(x_n)} = C'$$

Make sure we understand it.

In the lecture notes (*her notes latex*) we have the remarks

- if $f''(r) = 0$ but $f'(r) \neq 0$ then some function converges faster than quadratic convergence rate. (*see notes*).
- if $f'(r) = 0$ then we can show that NM has linear covnergence rate
- if initial point is not close to the root r then NM mmay not converge.

Need to be familiar with these special cases.

- Secant method : Something computing the derivative is impossible, so we use 2 points. We replace $f'(r)$ in the newton method by the joindd line between the two points

  No need to know how to prove the its superlinear convergence rate (do know that it's slower, and that it has advantage that we don't need to compute first derivative). Alsom it may NOT converge for same reason Newton method may not converge.

  > - for all Bisection, method, newton , secont need to know how to do the iterations
  > - need to know convergence rate for bisection
  > - special cases listed in the lecture notes.

- Comparisons of the Three Methods (advantages and disadvantages) (*see her notes latex*)