

CMPSC/Math 451, Numerical Computation

Wen Shen

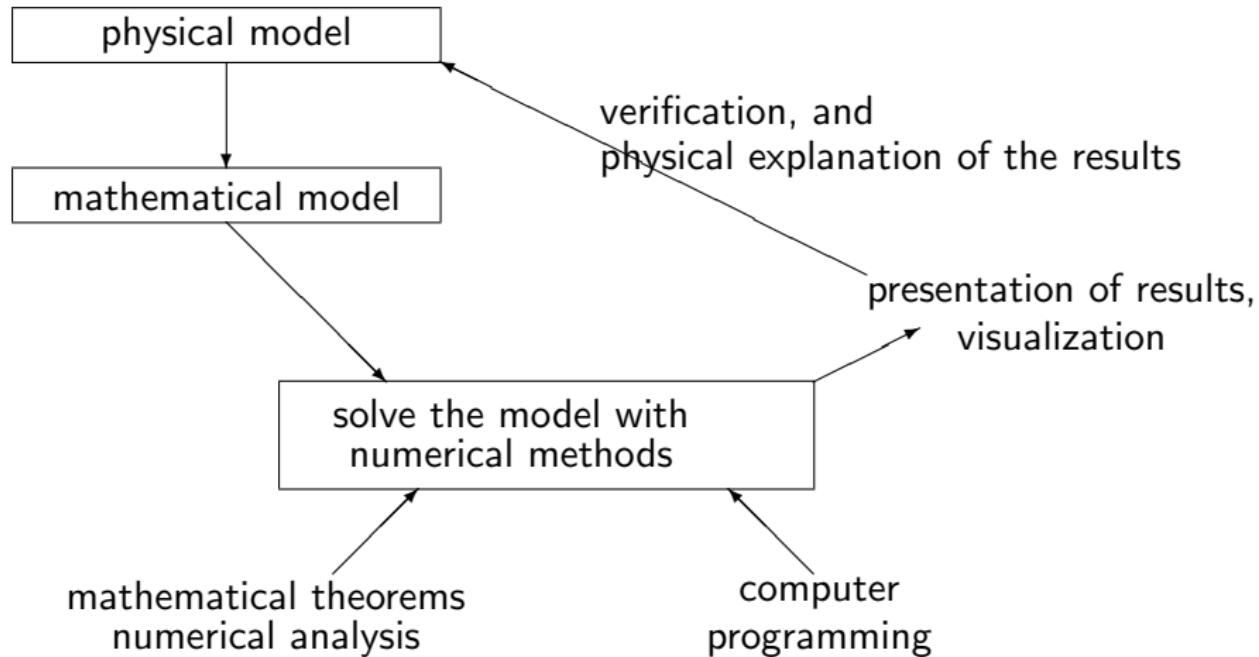
Department of Mathematics, Penn State University

What are numerical methods?

They are **algorithms** that compute **approximations** to functions, their derivatives, their integrations, and solutions to various equations etc.

Such algorithms could be implemented (programmed) on a computer.

Below is an overview on how various aspects are related.



Keep in mind that numerical methods are not about numbers. It is about mathematical ideas and insights.

We will study some basic classical types of problems:

- development of algorithms;
- implementation;
- a little bit of analysis, including error-estimates, convergence, stability etc.

We will use Matlab throughout the course for programming purpose.

Representation of numbers in different bases

Historically, there have been several bases for representing numbers:

- 10: decimal, daily use;
- 2: binary, computer use;
- 8: octal;
- 16: hexadecimal, ancient China;
- 20: vigesimal, used in ancient France (numbers 70-79 are counted as 60+10 to 60+19 in French, and 80 is 4×20);
- 60: sexagesimal, used by the Babylonians.
- etc...

In principle, one can use any number β as the base. Writing such a number with decimal point, we have

$$\begin{aligned} & \text{integer part} \quad \text{fractional part} \\ & \left(\overbrace{a_n a_{n-1} \cdots a_1 a_0} \cdot \overbrace{b_1 b_2 b_3 \cdots} \right)_{\beta} \\ = & \quad a_n \beta^n + a_{n-1} \beta^{n-1} + \cdots + a_1 \beta + a_0 \quad (\text{integer part}) \\ & \quad + b_1 \beta^{-1} + b_2 \beta^{-2} + b_3 \beta^{-3} + \cdots \quad (\text{fractional part}) \end{aligned}$$

Thinking of $\beta = 10$ above, we will understand the decimal representation.

The above formula allows us to convert a number in any base β into decimal base.

One can convert the numbers between different bases. We now go through some examples.

Example 1. octal \rightarrow decimal

$$(45.12)_8 = 4 \times 8^1 + 5 \times 8^0 + 1 \times 8^{-1} + 2 \times 8^{-2} = (37.15625)_{10}$$

Example 2. octal → binary

Observe

$$(1)_8 = (1)_2$$

$$(2)_8 = (10)_2$$

$$(3)_8 = (11)_2$$

$$(4)_8 = (100)_2$$

$$(5)_8 = (101)_2$$

$$(6)_8 = (110)_2$$

$$(7)_8 = (111)_2$$

$$(10)_8 = (1000)_2$$

Then,

$$(5034)_8 = (\underbrace{101}_5 \underbrace{000}_0 \underbrace{011}_3 \underbrace{100}_4)_2$$

Example 3. binary \rightarrow octal

$$(110\ 010\ 111\ 001)_2 = (\underbrace{6}_{110}\ \underbrace{2}_{010}\ \underbrace{7}_{111}\ \underbrace{1}_{001})_8$$

This algorithm is possible because 8 is a power of 2, namely, $8 = 2^3$.

Example 4. decimal \rightarrow binary. Write $(12.45)_{10}$ in binary base.

Answer. This example is of particular interests. Since the computer uses binary base, how would the number $(12.45)_{10}$ look like in binary base? The conversion takes two steps.

First, we treat the integer part and convert $(12)_{10}$ into binary.

Procedure: keep divided by 2, and store the remainders of each step, until one can not divide anymore.

$$\begin{array}{r|rr} & & (\text{remainder}) \\ \hline 2 & \underline{12} & 0 \\ 2 & \underline{6} & 0 \\ 2 & \underline{3} & 1 \\ 2 & \underline{1} & 1 \\ \hline & 0 & \end{array} \Rightarrow (12)_{10} = (1100)_2$$

We now convert $(0.45)_{10}$ into binary.

Procedure: multiply the fractional part by 2, and store the integer part for the result.

0.45		x	2
0.9		x	2
1.8		x	2
1.6		x	2
1.2		x	2
0.4		x	2
0.8		x	2
1.6		x	2
...			

$$\Rightarrow (0.45)_{10} = (0.01110011001100\cdots)_2.$$

Note that the fractional part does not end!

Putting them together, we get

$$(12.45)_{10} = (1100.01110011001100 \dots)_2$$

It is surprising to observe that, a simple finite length decimal number such as 12.45 could have infinite length of fractional numbers in binary form!

How does a computer store such a number? – next video!

Floating point representation

Recall normalized scientific notation for a real number x :

Decimal: $x = \pm r \times 10^n$, $1 \leq r < 10$. (Example: $2345 = 2.345 \times 10^3$)

Binary: $x = \pm r \times 2^n$, $1 \leq r < 2$

Octal: $x = \pm r \times 8^n$, $1 \leq r < 8$ (Just for an example.)

For any base β : $x = \pm r \times \beta^n$, $1 \leq r < \beta$

Information to be stored:

- (1) the sign,
- (2) the exponent n ,
- (3) the value of r .

The computer uses the binary version of the number system. They represent numbers with finite length. These are called *machine numbers*.

r : normalized mantissa. For binary numbers, we have

$$r = 1.(\text{fractional part})$$

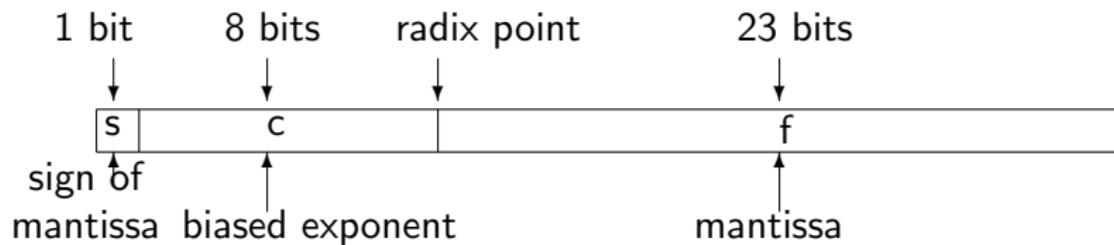
Therefore, in the computer we will only store the fractional part of the number.

n : exponent. If $n > 0$, then $x > 1$. If $n < 0$, then $x < 1$.

s : the sign of the number. If $s = 0$, it is positive; if $s = 1$, it is negative.

Each bit can store the value of either 0 or 1.

Single-precision IEEE standard floating-point, in a 32-bit computer



The exponent: $2^8 = 256$. It can represent numbers from -127 to 128 .

The value of the number:

$$(-1)^s \times 2^{c-127} \times (1.f)_2$$

The smallest representable number in absolute value is

$$x_{\min} = 2^{-127} \approx 5.9 \times 10^{-39}.$$

The largest representable number in absolute value is

$$x_{\max} = 2^{128} \approx 2.4 \times 10^{38}.$$

Computers can only handle numbers with absolute values between x_{\min} and x_{\max} .

We say that x **underflows** if $|x| < x_{\min}$. In this case, we consider $x = 0$.

We say that x **overflows** if $|x| > x_{\max}$. In this case, we consider $x = \infty$.

Error in the floating point representation.

Let $\text{fl}(x)$ denote the floating point representation of the number x . In general it contains error (roundoff or chopping).

$$\text{fl}(x) = x \cdot (1 + \delta)$$

$$\text{relative error: } \delta = \frac{\text{fl}(x) - x}{x}$$

$$\text{absolute error: } = \text{fl}(x) - x = \delta \cdot x$$

Computer errors in representing numbers:

- relative error in rounding off: $\delta \leq 0.5 \times 2^{-23} \approx 0.6 \times 10^{-7}$
- relative error in chopping: $\delta \leq 1 \times 2^{-23} \approx 1.2 \times 10^{-7}$

Error propagation (through arithmetic operation)

Example 1. Consider an addition, say $z = x + y$, done in a computer.
How would the errors be propagated?

Let $x > 0, y > 0$, and let $\text{fl}(x), \text{fl}(y)$ be their floating point representation.

$$\text{fl}(x) = x(1 + \delta_x), \quad \text{fl}(y) = y(1 + \delta_y)$$

where δ_x, δ_y are the relative errors in x, y .

Then

$$\begin{aligned}\text{fl}(z) &= \text{fl}(\text{fl}(x) + \text{fl}(y)) \\ &= (x(1 + \delta_x) + y(1 + \delta_y))(1 + \delta_z) \\ &= (x + y) + x \cdot (\delta_x + \delta_z) + y \cdot (\delta_y + \delta_z) + (x\delta_x\delta_z + y\delta_y\delta_z) \\ &\approx (x + y) + x \cdot (\delta_x + \delta_z) + y \cdot (\delta_y + \delta_z)\end{aligned}$$

Here, δ_z is the round-off error for z .

Then, we have

$$\begin{aligned}\text{absolute error} &= \text{fl}(z) - (x + y) = x \cdot (\delta_x + \delta_z) + y \cdot (\delta_y + \delta_z) \\ &= \underbrace{x \cdot \delta_x}_{\substack{\text{abs. err.} \\ \text{for } x}} + \underbrace{y \cdot \delta_y}_{\substack{\text{abs. err.} \\ \text{for } y}} + \underbrace{(x + y) \cdot \delta_z}_{\text{round off err}} \\ &\quad \underbrace{\hspace{10em}}_{\text{propagated error}}\end{aligned}$$

$$\begin{aligned}\text{relative error} &= \frac{\text{fl}(z) - (x + y)}{x + y} = \underbrace{\frac{x\delta_x + y\delta_y}{x + y}}_{\text{propagated err}} + \underbrace{\delta_z}_{\text{round off err}}\end{aligned}$$

Loss of significance

Loss of significance typically happens when one gets too few significant digits in subtraction of two numbers very close to each other.

Example 1. Find the roots of $x^2 - 40x + 2 = 0$. Use 4 significant digits in the computation.

Answer. The roots for the equation $ax^2 + bx + c = 0$ are

$$r_{1,2} = \frac{1}{2a} \left(-b \pm \sqrt{b^2 - 4ac} \right)$$

In our case, we have

$$x_{1,2} = 20 \pm \sqrt{398} \approx 20.00 \pm 19.95$$

so

$$x_1 \approx 20 + 19.95 = 39.95, \quad (\text{OK})$$

$$x_2 \approx 20 - 19.95 = 0.05, \quad \text{not OK, lost 3 sig. digits}$$

To avoid this: change the algorithm. Observe that $x_1 x_2 = c/a$. Then

$$x_2 = \frac{c}{ax_1} = \frac{2}{1 \cdot 39.95} \approx 0.05006$$

We get back 4 significant digits in the result.

Example 2. Compute the function

$$f(x) = \frac{1}{\sqrt{x^2 + 2x} - x - 1}$$

in a computer. Explain what problem you might run into in certain cases. Find a way to fix the difficulty.

Answer. We see that, for large values of x with $x > 0$, the values $\sqrt{x^2 + 2x}$ and $x + 1$ are very close to each. Therefore, in the subtraction we will lose many significant digits.

To avoid this problem, we manipulate the function $f(x)$ into an equivalent one that does not perform the subtraction. This can be achieved by multiplying both numerator and denominator by the conjugate of the denominator.

$$\begin{aligned} f(x) &= \frac{\sqrt{x^2 + 2x} + x + 1}{(\sqrt{x^2 + 2x} - x - 1)(\sqrt{x^2 + 2x} + x + 1)} \\ &= \frac{\sqrt{x^2 + 2x} + x + 1}{x^2 + 2x - (x + 1)^2} = -(\sqrt{x^2 + 2x} + x + 1). \end{aligned}$$

Review of Taylor Series

Given that $f(x) \in C^\infty$ is a smooth function. Its Taylor expansion about the point $x = c$ is:

$$\begin{aligned}f(x) &= f(c) + f'(c)(x - c) + \frac{1}{2!}f''(c)(x - c)^2 + \frac{1}{3!}f'''(c)(x - c)^3 + \dots \\&= \sum_{k=0}^{\infty} \frac{1}{k!}f^{(k)}(c)(x - c)^k.\end{aligned}$$

This is called *Taylor series of f at the point c* .

If $c = 0$, this is the *MacLaurin series*:

$$f(x) = f(0) + f'(0)x + \frac{1}{2!}f''(0)x^2 + \frac{1}{3!}f'''(0)x^3 + \dots = \sum_{k=0}^{\infty} \frac{1}{k!}f^{(k)}(0)x^k.$$

Familiar examples of MacLaurin Series:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \quad |x| < \infty$$

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots, \quad |x| < \infty$$

$$\cos x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots, \quad |x| < \infty$$

$$\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k = 1 + x + x^2 + x^3 + x^4 + \cdots, \quad |x| < 1$$

Since the computer performs only algebraic operation, these series are actually how a computer calculates many “fancier” functions!

For example, the exponential function is calculated as

$$e^x \approx \sum_{k=0}^N \frac{x^k}{k!}$$

for some large integer N such that the error is sufficiently small.

Note that this is rather “expensive” in computing time! This is why in many algorithm we take care in doing fewer function evaluations!

Example 1. Compute e to 6 digit accuracy.

Answer. We have

$$e = e^1 = 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \dots$$

And

$$\frac{1}{2!} = 0.5$$

$$\frac{1}{3!} = 0.166667$$

$$\frac{1}{4!} = 0.041667$$

...

$$\frac{1}{9!} = 0.0000027 \quad (\text{can stop here})$$

so

$$e \approx 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \dots \frac{1}{9!} = 2.71828$$

Error and convergence:

Assume $f^{(k)}(x)$ ($0 \leq k \leq n$) are continuous functions.

partial sum: $f_n(x) = \sum_{k=0}^n \frac{1}{k!} f^{(k)}(c)(x - c)^k$

Taylor Theorem:

$$E_{n+1} = f(x) - f_n(x) = \sum_{k=n+1}^{\infty} \frac{1}{k!} f^{(k)}(c)(x - c)^k = \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x - c)^{n+1}$$

where ξ is some value between x and c .

This says, for the infinite sum for the error, if it converges, then the sum is “dominated” by the first term in the series.

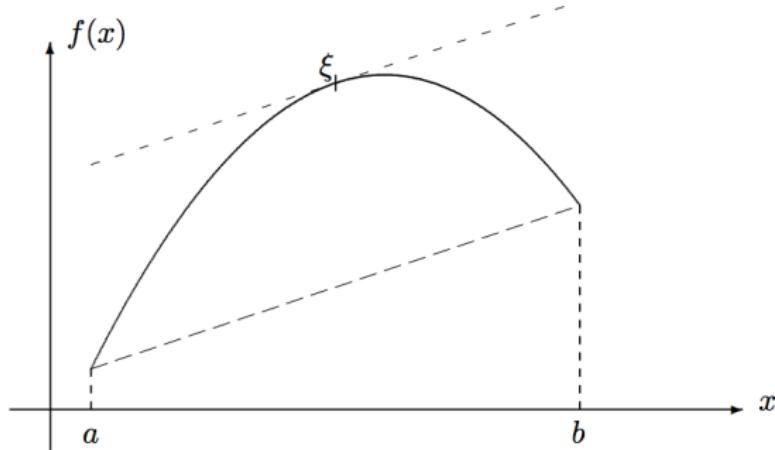
Observation: A Taylor series converges rapidly if x is near c , and slowly (or not at all) if x is far away from c .

Geometric interpretation for the error estimate with $n = 0$.

$$f(b) - f(a) = (b - a)f'(\xi), \quad \text{for some } \xi \text{ in } (a, b)$$

This implies

$$f'(\xi) = \frac{f(b) - f(a)}{b - a}, \quad \text{The Mean-Value Theorem}$$



Finite Difference Approximation to derivatives

Given $h > 0$ sufficiently small, we have 3 ways of approximating $f'(x)$:

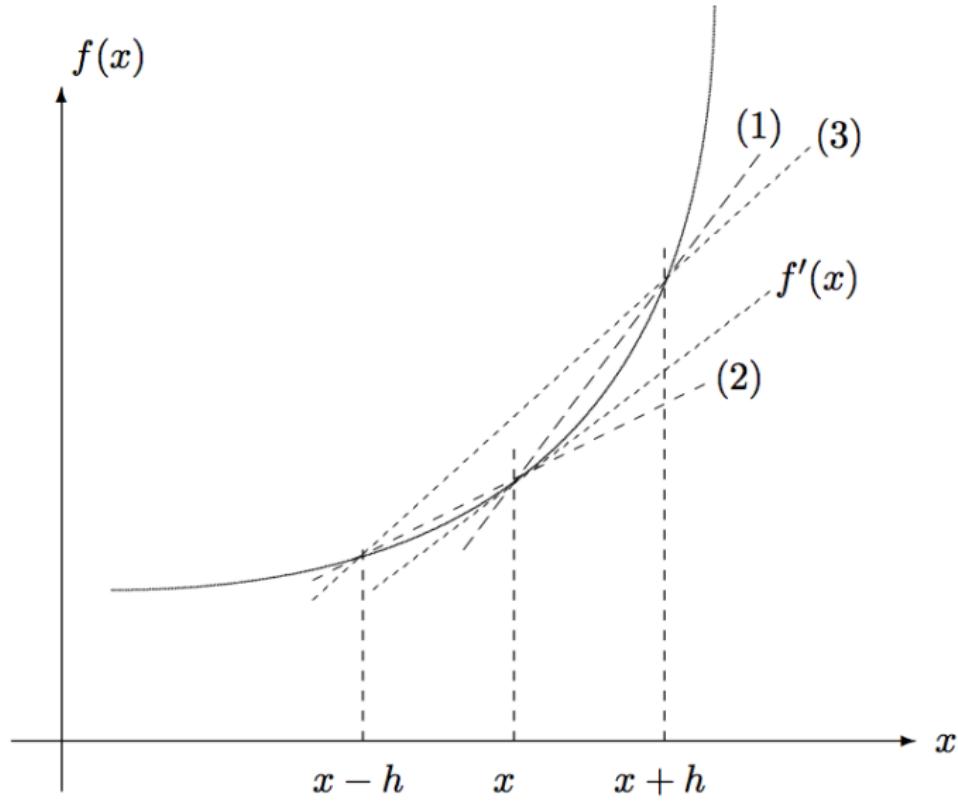
$$(1) \quad f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{Forward Euler}$$

$$(2) \quad f'(x) \approx \frac{f(x) - f(x-h)}{h} \quad \text{Backward Euler}$$

$$(3) \quad f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad \text{Central Finite Difference}$$

For the second derivative f'' , we also have a central finite difference approximation:

$$f''(x) \approx \frac{1}{h^2} (f(x+h) - 2f(x) + f(x-h))$$



Local Truncation Error

The Taylor expansion for $f(x + h)$ about x :

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

where

$$E_{n+1} = \sum_{k=n+1}^{\infty} \frac{1}{k!} f^{(k)}(x) h^k = \frac{1}{(n+1)!} f^{(n+1)}(\xi) h^{n+1} = \mathcal{O}(h^{n+1})$$

Here the notation $\mathcal{O}(h^4)$ indicate a quantity bounded by Ch^4 where C is a bounded constant.

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \frac{1}{6}h^3f'''(x) + \mathcal{O}(h^4),$$

$$f(x-h) = f(x) - hf'(x) + \frac{1}{2}h^2f''(x) - \frac{1}{6}h^3f'''(x) + \mathcal{O}(h^4).$$

Forward Euler:

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{1}{2}hf''(x) + \mathcal{O}(h^2) = f'(x) + \mathcal{O}(h^1), \quad (1^{st} \text{order}),$$

Backward Euler:

$$\frac{f(x) - f(x-h)}{h} = f'(x) - \frac{1}{2}hf''(x) + \mathcal{O}(h^2) = f'(x) + \mathcal{O}(h^1), \quad (1^{st} \text{order}),$$

Central finite difference:

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) - \frac{1}{6}h^2 f'''(x) + \mathcal{O}(h^2) = f'(x) + \mathcal{O}(h^2), \quad (2^{nd} \text{ order}),$$

Central finite difference for the second derivative

$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = f''(x) + \frac{1}{12}h^2 f^{(4)}(x) + \mathcal{O}(h^4) = f''(x) + \mathcal{O}(h^2),$$

second order method.

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Introduction to Matlab

MATLAB is an advanced program package tool for *numerical computation* and *visualization*.

Advantages:

- easy to use and program.
- can be run interactively or from a file.
- 2- and 3-dimensional plot.
- many built-in numerical functions.
- can be developed with “toolboxes” of different kinds.
- possibility to link in FORTRAN or C programs.
- Object-oriented programming.

Disadvantages:

- Relatively slow (compare to FORTRAN or C).

MATLAB: Matrix Laboratory

- Work directly with matrices and vectors
- Specially good in solving systems of linear equations, computing eigenvalues and eigenvectors, factorizing matrices, etc.
- Basic datatypes: Matrices of double precision number.

```
>> a = 2  
a =  
     2  
>> A = [1, 3.5, 4.6; 2, 6.4, -1.28]
```

```
A =  
1.0000    3.5000    4.6000  
2.0000    6.4000   -1.2800
```

```
>> x = [2.6; pi; 1/3]
```

```
x =  
2.6000  
3.1416  
0.3333
```

```
>> whos
```

Name	Size	Bytes	Class
A	2x3	48	double array
a	1x1	8	double array
x	3x1	24	double array

Grand total is 10 elements using 80 bytes

How to solve $Ax = b$. Example

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

```
>> A = [1,2,3;4,5,6;7,8,0]
```

```
A =
```

```
1      2      3  
4      5      6  
7      8      0
```

```
>> b = [1;2;3]
```

```
b =
```

```
1  
2  
3
```

```
>> x = A\b
```

```
x =
```

```
-0.3333  
0.6667  
0
```

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Chapter 2: Polynomial Interpolation

In this Chapter we study how to interpolate a data set with a polynomial.

Problem description:

Given $(n + 1)$ points, say (x_i, y_i) , where $i = 0, 1, 2, \dots, n$, with distinct x_i , not necessarily sorted, we want to find a polynomial of degree n ,

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

such that it interpolates these points, i.e.,

$$P_n(x_i) = y_i, \quad i = 0, 1, 2, \dots, n$$

The goal is to determine the coefficients $a_n, a_{n-1}, \dots, a_1, a_0$.

NB! The total number of data points is 1 larger than the degree of the polynomial.

Why should we do this? Here are some reasons:

- Find the values between the points for discrete data set;
- To approximate a (probably complicated) function by a polynomial;
- Then, it is easier to do computations such as derivative, integration etc.

Example 1. Interpolate the given data set with a polynomial of degree 2:

x_i	0	1	$2/3$
y_i	1	0	0.5

Answer. Let

$$P_2(x) = a_2x^2 + a_1x + a_0$$

We need to find the coefficients a_2, a_1, a_0 .

By the interpolating properties, we have 3 equations:

$$x = 0, y = 1 \quad : \quad P_2(0) = a_0 = 1$$

$$x = 1, y = 0 \quad : \quad P_2(1) = a_2 + a_1 + a_0 = 0$$

$$x = 2/3, y = 0.5 \quad : \quad P_2(2/3) = (4/9)a_2 + (2/3)a_1 + a_0 = 0.5$$

Here we have 3 linear equations and 3 unknowns (a_2, a_1, a_0) .

The equations:

$$\begin{aligned} a_0 &= 1 \\ a_2 + a_1 + a_0 &= 0 \\ \frac{4}{9}a_2 + \frac{2}{3}a_1 + a_0 &= 0.5 \end{aligned}$$

In matrix-vector form

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ \frac{4}{9} & \frac{2}{3} & 1 \end{pmatrix} \begin{pmatrix} a_2 \\ a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0.5 \end{pmatrix}$$

Easy to solve in Matlab, or do it by hand:

$$a_2 = -3/4, \quad a_1 = -1/4, \quad a_0 = 1.$$

Then

$$P_2(x) = -\frac{3}{4}x^2 - \frac{1}{4}x + 1.$$

The general case. For the general case with $(n + 1)$ points, we have

$$P_n(x_i) = y_i, \quad i = 0, 1, 2, \dots, n$$

We will have $(n + 1)$ equations and $(n + 1)$ unknowns:

$$P_n(x_0) = y_0 : x_0^n a_n + x_0^{n-1} a_{n-1} + \cdots + x_0 a_1 + a_0 = y_0$$

$$P_n(x_1) = y_1 : x_1^n a_n + x_1^{n-1} a_{n-1} + \cdots + x_1 a_1 + a_0 = y_1$$

⋮

$$P_n(x_n) = y_n : x_n^n a_n + x_n^{n-1} a_{n-1} + \cdots + x_n a_1 + a_0 = y_n$$

Putting this in matrix-vector form

$$\begin{pmatrix} x_0^n & x_0^{n-1} & \cdots & x_0 & 1 \\ x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^n & x_n^{n-1} & \cdots & x_n & 1 \end{pmatrix} \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

i.e.

$$\mathbf{X} \vec{a} = \vec{y}$$

$$\mathbf{X} \vec{a} = \vec{y}$$

- \mathbf{X} : $(n + 1) \times (n + 1)$ matrix, given
(It's called the van der Monde matrix)
- \vec{a} : unknown vector, with length $(n + 1)$
- \vec{y} : given vector, with length $(n + 1)$

Theorem: If x_i 's are distinct, then \mathbf{X} is invertible, therefore \vec{a} has a unique solution.

In Matlab, the command `vander(x)`, where x is a vector that contains the interpolation points $x=[x_1, x_2, \dots, x_n]$, will generate this matrix.

Bad news: \mathbf{X} has very large condition number for large n , therefore not effective to solve if n is large.

Other more efficient and elegant methods include

- Lagrange polynomials
- Newton's divided differences

Lagrange interpolation polynomials

Given points: x_0, x_1, \dots, x_n

Define the **cardinal functions** $l_0, l_1, \dots, l_n \in \mathcal{P}^n$, satisfying the properties

$$l_i(x_j) = \delta_{ij} = \begin{cases} 1 & , \quad i = j \\ 0 & , \quad i \neq j \end{cases} \quad i = 0, 1, \dots, n$$

Here δ_{ij} is called the Kronecker's delta.

Locally supported in discrete sense.

The cardinal functions $l_i(x)$ can be written as

$$\begin{aligned} l_i(x) &= \prod_{j=0, j \neq i}^n \left(\frac{x - x_j}{x_i - x_j} \right) \\ &= \frac{x - x_0}{x_i - x_0} \cdot \frac{x - x_1}{x_i - x_1} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \cdot \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_n}{x_i - x_n} \end{aligned}$$

Verify:

$$l_i(x_i) = 1$$

and for $i \neq k$

$$l_i(x_k) = 0$$

$$l_i(x_k) = \delta_{ik}.$$

Lagrange form of the interpolation polynomial can be simply expressed as

$$P_n(x) = \sum_{i=0}^n l_i(x) \cdot y_i.$$

It is easy to check the interpolating property:

$$P_n(x_j) = \sum_{i=0}^n l_i(x_j) \cdot y_i = y_j, \quad \text{for every } j.$$

Example 2. Write the Lagrange polynomial for the data (same as in Example 1)

x_i	0	2/3	1
y_i	1	0.5	0

Answer. The data set corresponds to

$$x_0 = 0, \quad x_1 = 2/3, \quad x_2 = 1, \quad y_0 = 1, \quad y_1 = 0.5, \quad y_2 = 0.$$

We first compute the cardinal functions

$$l_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{(x - 2/3)(x - 1)}{(0 - 2/3)(0 - 1)} = \frac{3}{2} \left(x - \frac{2}{3} \right) (x - 1)$$

$$l_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} = \frac{(x - 0)(x - 1)}{(2/3 - 0)(2/3 - 1)} = -\frac{9}{2}x(x - 1)$$

$$l_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{(x - 0)(x - 2/3)}{(1 - 0)(1 - 2/3)} = 3x \left(x - \frac{2}{3} \right)$$

so

$$\begin{aligned} P_2(x) &= l_0(x)y_0 + l_1(x)y_1 + l_2(x)y_2 = \frac{3}{2} \left(x - \frac{2}{3} \right) (x - 1) - \frac{9}{2}x(x - 1)(0.5) + 0 \\ &= \dots = -\frac{3}{4}x^2 - \frac{1}{4}x + 1, \quad \text{same as in Example 1} \end{aligned}$$

Pros and cons of Lagrange polynomial:

- (+) Elegant formula,
- (-) Slow to compute, each $l_i(x)$ is different,
- (-) Not flexible: if one changes a points x_j , or add on an additional point x_{n+1} , one must re-compute all l_i 's.

Newton's divided differences

Given a data set

x_i	x_0	x_1	\cdots	x_n
y_i	y_0	y_1	\cdots	y_n

We will describe an algorithm in a recursive form.

Main idea:

Given $P_k(x)$ that interpolates $k + 1$ data points $\{x_i, y_i\}$, $i = 0, 1, 2, \dots, k$, compute $P_{k+1}(x)$ that interpolates one extra point, $\{x_{k+1}, y_{k+1}\}$, by using P_k and adding an extra term.

- For $n = 0$, we set $P_0(x) = y_0$. Then $P_0(x_0) = y_0$.
- For $n = 1$, we set

$$P_1(x) = P_0(x) + a_1(x - x_0) \quad (1)$$

where a_1 is to be determined.

Then, $P_1(x_0) = P_0(x_0) + 0 = y_0$, for any a_1 .

Find a_1 by the interpolation property $y_1 = P_1(x_1)$, we have

$$y_1 = P_0(x_1) + a_1(x_1 - x_0) = y_0 + a_1(x_1 - x_0).$$

This gives us

$$a_1 = \frac{y_1 - y_0}{x_1 - x_0}.$$

- For $n = 2$, we set

$$P_2(x) = P_1(x) + a_2(x - x_0)(x - x_1).$$

Then, $P_2(x_0) = P_1(x_0) = y_0$, $P_2(x_1) = P_1(x_1) = y_1$.

Determine a_2 by the interpolating property $y_2 = P_2(x_2)$.

$$y_2 = P_1(x_2) + a_2(x_2 - x_0)(x_2 - x_1),$$

Then

$$a_2 = \frac{y_2 - P_1(x_2)}{(x_2 - x_0)(x_2 - x_1)}.$$

We would like to express a_2 in a different way. Recall

$$P_1(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0).$$

Then

$$\begin{aligned} P_1(x_2) &= y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0) \\ &= y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_1) + \frac{y_1 - y_0}{x_1 - x_0}(x_1 - x_0) \\ &= y_1 + \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_1). \end{aligned}$$

Then, a_2 can be rewritten as

$$a_2 = \frac{y_2 - P_1(x_2)}{(x_2 - x_0)(x_2 - x_1)} = \frac{y_2 - y_1 - \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0}.$$

The general case for a_n :

Assume that $P_{n-1}(x)$ interpolates (x_i, y_i) for $i = 0, 1, \dots, n-1$.
Let

$$P_n(x) = P_{n-1}(x) + a_n(x - x_0)(x - x_1) \cdots (x - x_{n-1})$$

Then for $i = 0, 1, \dots, n-1$, we have

$$P_n(x_i) = P_{n-1}(x_i) = y_i.$$

Find a_n by the property $P_n(x_n) = y_n$,

$$y_n = P_{n-1}(x_n) + a_n(x_n - x_0)(x_n - x_1) \cdots (x_n - x_{n-1})$$

then

$$a_n = \frac{y_n - P_{n-1}(x_n)}{(x_n - x_0)(x_n - x_1) \cdots (x_n - x_{n-1})}$$

Newton's form for the interpolation polynomial:

$$\begin{aligned}P_n(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots \\&\quad + a_n(x - x_0)(x - x_1) \cdots (x - x_{n-1}),\end{aligned}$$

Newton's divided differences, recursive computation

The recursion is initiated with

$$f[x_i] = y_i, \quad i = 0, 1, 2, \dots$$

Then

$$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}, \quad f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1}, \quad \dots$$

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}, \quad f[x_1, x_2, x_3] = \frac{f[x_3, x_2] - f[x_2, x_1]}{x_3 - x_1}, \quad \dots$$

For the general step, we have

$$f[x_0, x_1, \dots, x_k] = \frac{f[x_1, x_2, \dots, x_k] - f[x_0, x_1, \dots, x_{k-1}]}{x_k - x_0}$$

The constants a_k 's in the Newton's form are computed as

$$a_0 = f[x_0], \quad a_1 = f[x_0, x_1], \quad \dots \quad a_k = f[x_0, x_1, \dots, x_k]$$

Computation of the divided differences

We compute the $f[\dots]$'s through the following table:

x_0	$f[x_0] = y_0$				
x_1	$f[x_1] = y_1$	$f[x_0, x_1]$ $= \frac{f[x_1] - f[x_0]}{x_1 - x_0}$			
x_2	$f[x_2] = y_2$	$f[x_1, x_2]$ $= \frac{f[x_2] - f[x_1]}{x_2 - x_1}$	$f[x_0, x_1, x_2]$		
:	:	:	:	.	.
x_n	$f[x_n] = y_n$	$f[x_{n-1}, x_n]$ $= \frac{f[x_n] - f[x_{n-1}]}{x_n - x_{n-1}}$	$f[x_{n-2}, x_{n-1}, x_n]$...	$f[x_0, x_1, \dots, x_n]$

The diagonal elements give us the a_i 's.

Example 3. Write Newton's form of interpolation polynomial for the data

x_i	0	1	$2/3$	$1/3$
y_i	1	0	$1/2$	0.866

Answer. Set up the triangular table for computation

0	1			
1	0	-1		
$2/3$	0.5	-1.5	-0.75	
$1/3$	0.8660	-1.0981	-0.6029	0.4413

So we have

$$a_0 = 1, \quad a_1 = -1, \quad a_2 = -0.75, \quad a_3 = 0.4413.$$

Then

$$P_3(x) = 1 + (-1)x + (-0.75)x(x-1) + 0.4413x(x-1)(x-\frac{2}{3}).$$

Flexibility of Newton's form: easy to add additional points to interpolate.

Nested form of Newton's polynomial:

$$\begin{aligned}P_n(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots \\&\quad + a_n(x - x_0)(x - x_1) \cdots (x - x_{n-1}) \\&= a_0 + (x - x_0)(a_1 + (x - x_1)(a_2 + (x - x_2)(a_3 + \cdots + a_n(x - x_{n-1}))))\end{aligned}$$

Effective coding:

Given the data x_i and a_i for $i = 0, 1, \dots, n$

the following pseudo-code evaluates the Newton's polynomial $p = P_n(x)$

- $p = a_n$
- for $k = n - 1, n - 2, \dots, 0$
 - $p = p(x - x_k) + a_k$
- end

This requires only $3n$ flops.

Existence and Uniqueness theorem for polynomial interpolation

Theorem. (Fundamental Theorem of Algebra)

Every polynomial of degree n that is not identically zero, has maximum n roots (including multiplicities). These roots may be real or complex. In particular, this implies that if a polynomial of degree n has more than n roots, then it must be identically zero.

Theorem. (Existence and Uniqueness of Polynomial Interpolation)

Given $(x_i, y_i)_{i=0}^n$, with x_i 's distinct. There exists one and only polynomial $P_n(x)$ of degree $\leq n$ such that $P_n(x_i) = y_i$ for $i = 0, 1, \dots, n$.

Proof. The existence: by construction.

Uniqueness: Assume we have two polynomials $p(x), q(x) \in \mathcal{P}_n$, such that

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

Now, let $g(x) = p(x) - q(x)$, a polynomial of degree $\leq n$.

$$g(x_i) = p(x_i) - q(x_i) = y_i - y_i = 0, \quad i = 0, 1, \dots, n$$

So $g(x)$ has $n + 1$ zeros. By the Fundamental Theorem of Algebra, we must have $g(x) \equiv 0$, therefore $p(x) \equiv q(x)$.

Errors in Polynomial Interpolation

Given a function $f(x)$ on $x \in [a, b]$, and a set of distinct points $x_i \in [a, b]$, $i = 0, 1, \dots, n$. Let $P_n(x) \in \mathcal{P}_n$ s.t.,

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

error function : $e(x) = f(x) - P_n(x), \quad x \in [a, b].$

Theorem. *There exists some value $\xi \in [a, b]$, such that*

$$e(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^n (x - x_i), \quad \text{for all } x \in [a, b]. \quad (1)$$

Proof. If $f \in \mathcal{P}_n$, then by Uniqueness Theorem of polynomial interpolation we must have $f(x) = P_n(x)$. Then $e(x) \equiv 0$ and the proof is trivial.

Now assume $f \notin \mathcal{P}_n$. If $x = x_i$ for some i , we have $e(x_i) = f(x_i) - P_n(x_i) = 0$, and the result holds.

Now consider $x \neq x_i$ for any i .

$$W(x) = \prod_{i=0}^n (x - x_i) \in \mathcal{P}_{n+1},$$

it holds

$$W(x_i) = 0, \quad W(x) = x^{n+1} + \dots, \quad W^{(n+1)} = (n+1)!.$$

Fix an y such that $a \leq y \leq b$ and $y \neq x_i$ for any i . We define a constant

$$c = \frac{f(y) - P_n(y)}{W(y)},$$

and another function

$$\varphi(x) = f(x) - P_n(x) - cW(x).$$

We find all the zeros for $\varphi(x)$. We see that x_i 's are zeros since

$$\varphi(x_i) = f(x_i) - P_n(x_i) - cW(x_i) = 0, \quad i = 0, 1, \dots, n$$

and also y is a zero because

$$\varphi(y) = f(y) - P_n(y) - cW(y) = 0$$

So, φ has at least $(n + 2)$ zeros.

Here goes our deduction:

$\varphi(x)$ has at least $n+2$ zeros on $[a, b]$.

$\varphi'(x)$ has at least $n+1$ zeros on $[a, b]$.

$\varphi''(x)$ has at least n zeros on $[a, b]$.

\vdots

$\varphi^{(n+1)}(x)$ has at least 1 zero on $[a, b]$.

Call it ξ s.t. $\varphi^{(n+1)}(\xi) = 0$.

So we have

$$\varphi^{(n+1)}(\xi) = f^{(n+1)}(\xi) - 0 - cW^{(n+1)}(\xi) = 0.$$

Recall $W^{(n+1)} = (n+1)!$, we have, for every y ,

$$f^{(n+1)}(\xi) = cW^{(n+1)}(\xi) = \frac{f(y) - P_n(y)}{W(y)}(n+1)!.$$

Writing y into x , we get

$$e(x) = f(x) - P_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) W(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^n (x - x_i),$$

for some $\xi \in [a, b]$.

Recall the error formula: $e(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^n (x - x_i)$

Example 1. If $n = 1$, $x_0 = a$, $x_1 = b$, $b > a$, find an upper bound for error.

Answer. Let

$$M = \max_{a \leq x \leq b} |f''(x)| = \|f''\|_\infty$$

and observe

$$\max_{a \leq x \leq b} |(x-a)(x-b)| = \dots = \frac{(b-a)^2}{4}.$$

For $x \in [a, b]$, we have

$$|e(x)| = \frac{1}{2} |f''(\xi)| \cdot |(x-a)(x-b)| \leq \frac{1}{2} \|f''\|_\infty \frac{(b-a)^2}{4} = \frac{1}{8} \|f''\|_\infty (b-a)^2.$$

NB! Error depends on the distribution of nodes x_i .

Uniform grid

Equally distribute the nodes (x_i): on $[a, b]$, with $n + 1$ nodes.

$$x_i = a + ih, \quad h = \frac{b - a}{n}, \quad i = 0, 1, \dots, n.$$

One can show that for $x \in [a, b]$, it holds

$$\prod_{i=0}^n |x - x_i| \leq \frac{1}{4} h^{n+1} \cdot n!$$

Proof. If $x = x_i$ for some i , then $x - x_i = 0$ and the product is 0, so it trivially holds.

Now assume $x_i < x < x_{i+1}$ for some i . We have

$$\max_{x_i < x < x_{i+1}} |(x - x_i)(x - x_{i+1})| = \frac{1}{4}(x_{i+1} - x_i)^2 = \frac{h^2}{4}.$$

Now consider the other terms in the product, say $x - x_j$, for either $j > i + 1$ or $j < i$. Then $|x - x_j| \leq h(j - i)$ for $j > i + 1$ and $|x - x_j| \leq h(i + 1 - j)$ for $j < i$. In all cases, the product of these terms are bounded by $h^{n-1} n!$, proving the result.

We have the error estimate

$$|e(x)| \leq \frac{1}{4(n+1)} \left| f^{(n+1)}(x) \right| h^{n+1} \leq \frac{M_{n+1}}{4(n+1)} h^{n+1}$$

where

$$M_{n+1} = \max_{x \in [a,b]} \left| f^{(n+1)}(x) \right| = \| f^{(n+1)} \|_{\infty}$$

Example 2. Consider interpolating $f(x) = \sin(\pi x)$ with polynomial on the interval $[-1, 1]$ with uniform nodes. Give an upper bound for error.

Answer. Since

$$f'(x) = \pi \cos \pi x, \quad f''(x) = -\pi^2 \sin \pi x, \quad f'''(x) = -\pi^3 \cos \pi x$$

we have

$$\left| f^{(n+1)}(x) \right| \leq \pi^{n+1}, \quad M_{n+1} = \pi^{n+1}$$

so the upper bound for error is

$$|e(x)| \leq \frac{M_{n+1}}{4(n+1)} h^{n+1} \leq \frac{\pi^{n+1}}{4(n+1)} \left(\frac{2}{n} \right)^{n+1}.$$

	n	error bound	measured error
Simulation data:	4	4.8×10^{-1}	1.8×10^{-1}
	8	3.2×10^{-3}	1.2×10^{-3}
	16	1.8×10^{-9}	6.6×10^{-10}

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Chebychev nodes: equally distributing the error

Type I: including the end points.

For interval $[-1, 1]$: $\bar{x}_i = \cos\left(\frac{i}{n}\pi\right)$, $i = 0, 1, \dots, n$

For interval $[a, b]$: $\bar{x}_i = \frac{1}{2}(a + b) + \frac{1}{2}(b - a)\cos\left(\frac{i}{n}\pi\right)$, $i = 0, 1, \dots, n$

One can show that

$$\max_{a \leq x \leq b} \left\{ \prod_{k=0}^n |x - \bar{x}_k| \right\} = 2^{-n} \leq \max_{a \leq x \leq b} \left\{ \prod_{k=0}^n |x - x_k| \right\}$$

where x_k is any other choice of nodes.

Error bound: $|e(x)| \leq \frac{1}{(n+1)!} |f^{(n+1)}(x)| 2^{-n}$.

Example Consider the same example with uniform nodes, $f(x) = \sin \pi x$. With Chebyshev nodes, we have

$$|e(x)| \leq \frac{1}{(n+1)!} \pi^{n+1} 2^{-n}.$$

The corresponding table for errors:

n	error bound	measured error
4	1.6×10^{-1}	1.15×10^{-1}
8	3.2×10^{-4}	2.6×10^{-4}
16	1.2×10^{-11}	1.1×10^{-11}

The errors are much smaller!

Type II: Chebyshev nodes can be chosen strictly inside the interval $[a, b]$:

$$\bar{x}_i = \frac{1}{2}(a + b) + \frac{1}{2}(b - a) \cos\left(\frac{2i + 1}{2n + 2}\pi\right), \quad i = 0, 1, \dots, n$$

Discussion:

For large n , polynomials are heavy to deal with.

In general, interpolation polynomials do not converge to the function as $n \rightarrow \infty$.

For small intervals, the error with polynomial interpolation is small.

Conclusion: Better to use piecewise polynomial interpolation. – next chapter.

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Polynomial interpolation: Van der Monde matrix

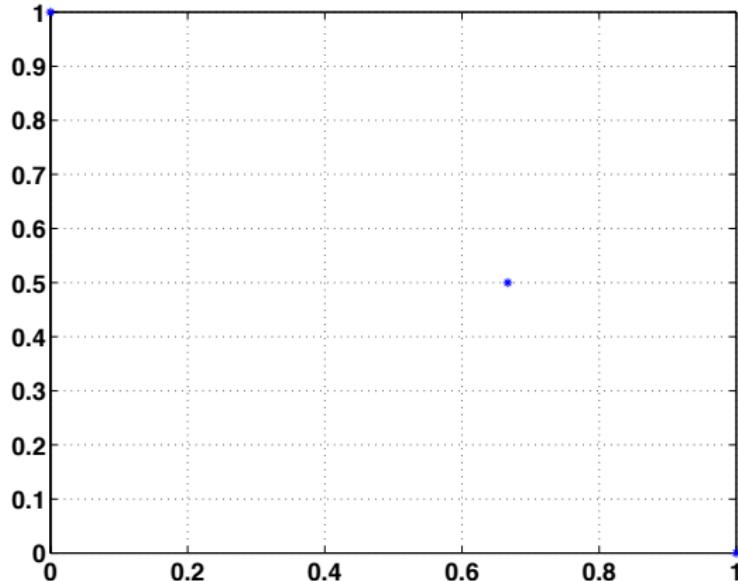
```
>> X = [1,0,0;1,1,1;1,2/3,4/9] % van der Monde matrix
X =
    1.0000      0      0
    1.0000    1.0000    1.0000
    1.0000    0.6667    0.4444

>> y = [1;0;1/2]
y =
    1.0000
    0
    0.5000

>> a = X\y
a =
    1.0000
   -0.2500
   -0.7500
```

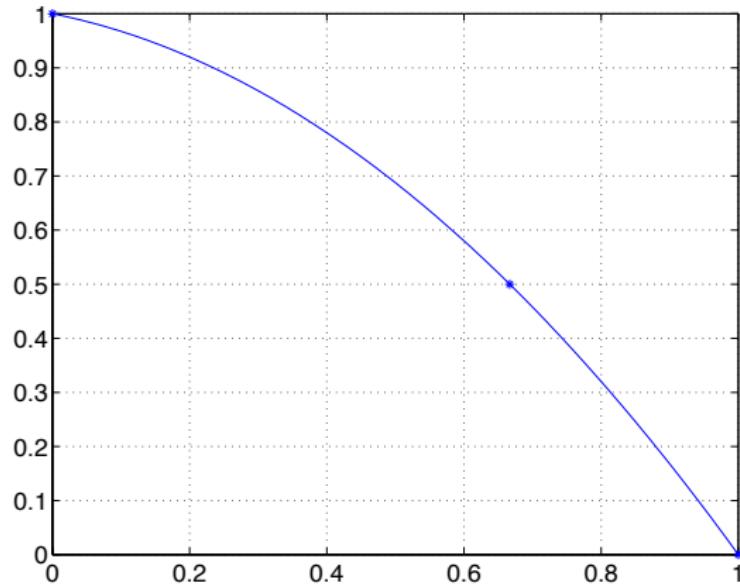
Plot the interpolating points:

```
>> x = [0;1;2/3]; y = [1;0;1/2];  
>> plot(x,y,'*')  
>> grid
```



Plot the interpolating polynomial:

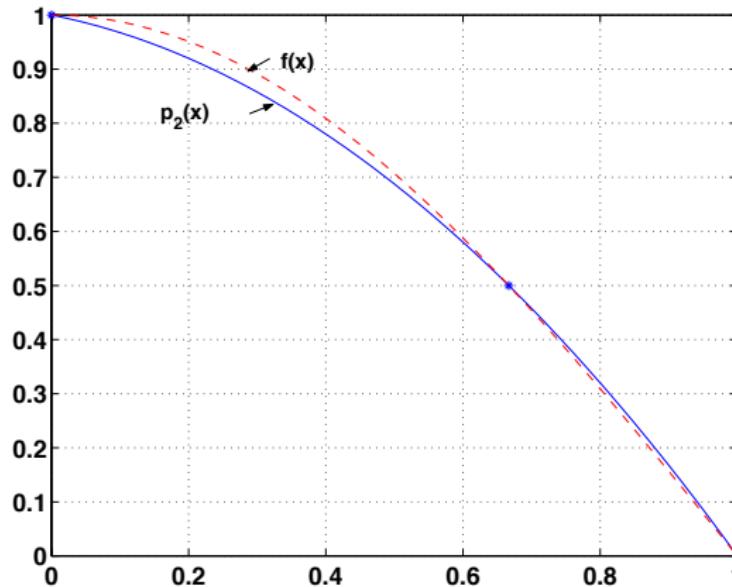
```
>> hold on  
>> t = [0:0.01:1];  
>> p2 = a(1)+a(2)*t+a(3)*t.^2;  
>> plot(t,p2)
```



With $f(x) = \cos(\frac{\pi}{2}x)$, we get:

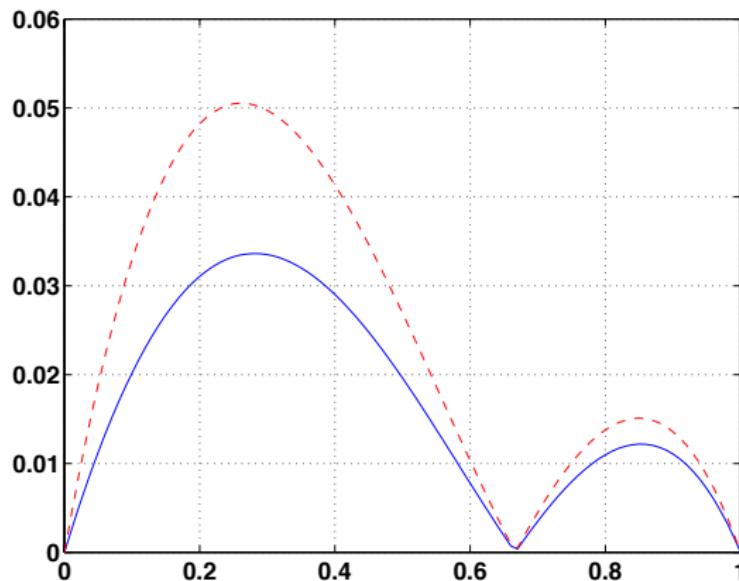
x_i	0	1	$2/3$
$f(x_i)$	1	0	$1/2$

```
>> plot(t,cos(pi/2*t), '--r')
```

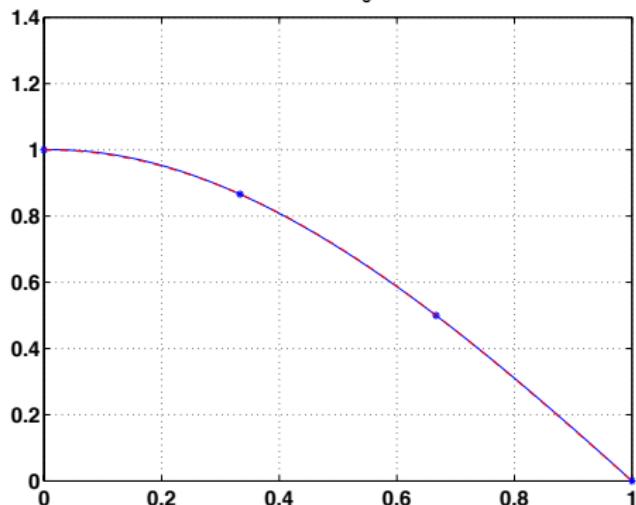


Now let us plot the error $e(x) = f(x) - p_2(x)$ (—) and upper error bound (---)

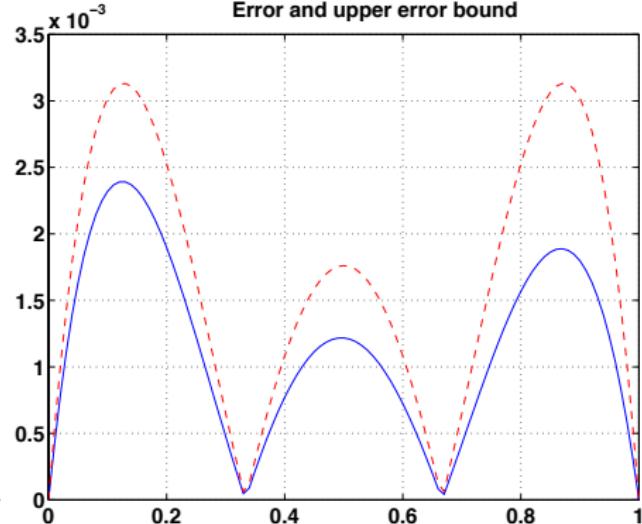
```
>> hold off  
>> errorbound = abs(pi^3/48*t.*(t-1).*(t-2/3));  
>> error = abs(cos(pi/2*t)-p2);  
>> plot(t,error,t,errorbound,'--r')
```



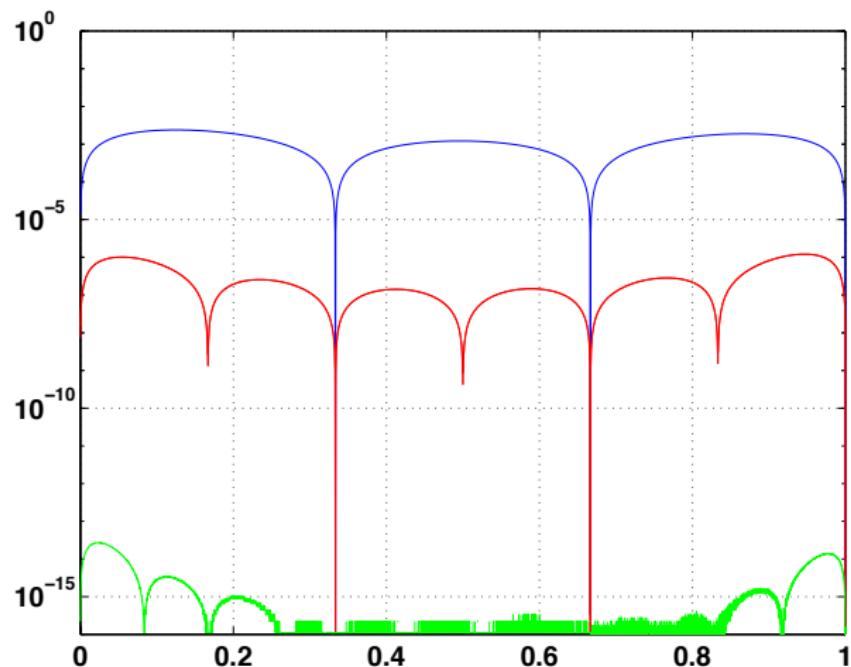
$f(x)$ og $p_3(x)$



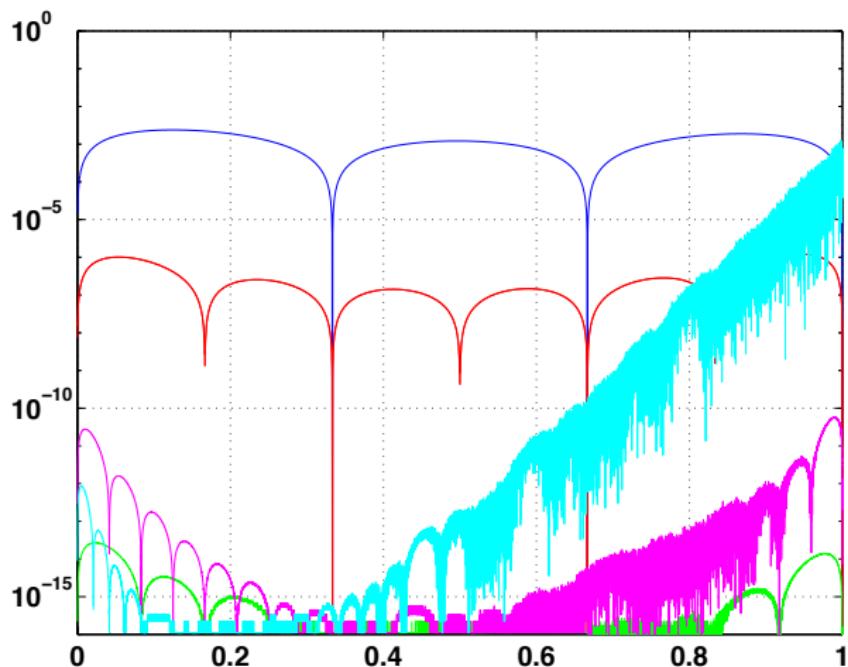
Error and upper error bound



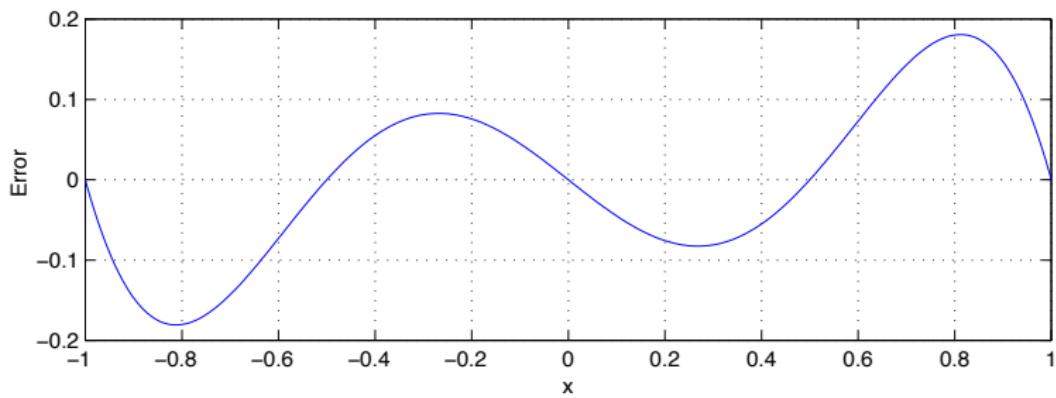
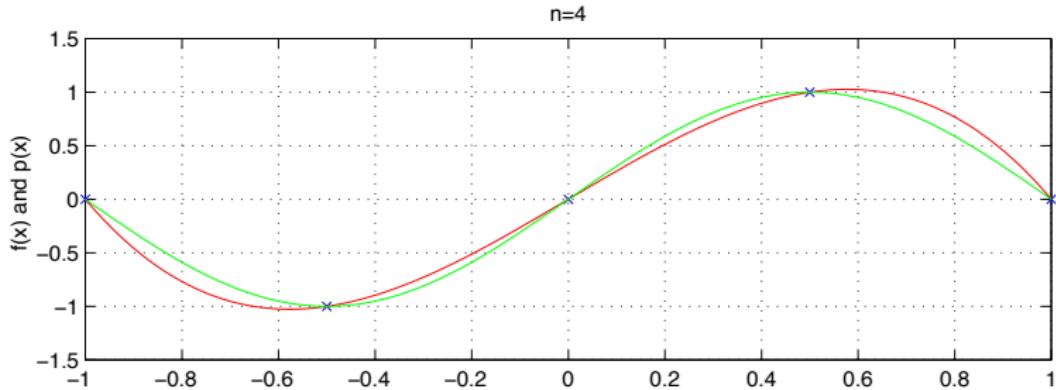
Error when interpolating number increases:



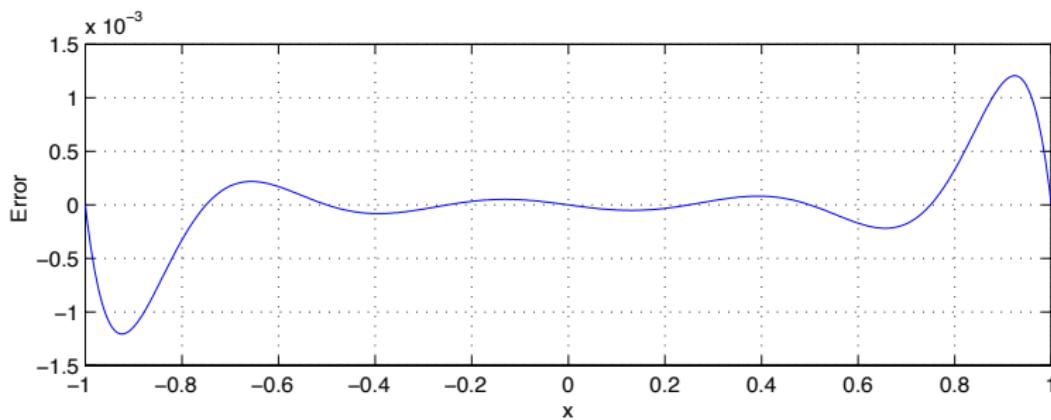
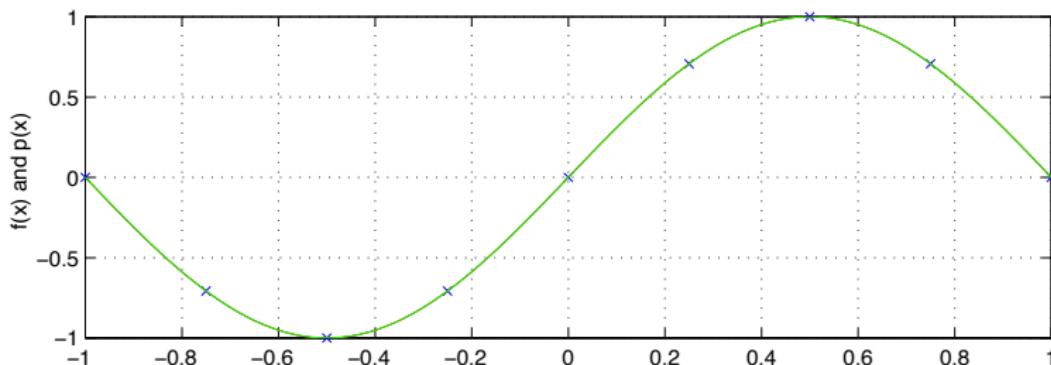
Error when interpolating number increases even more:



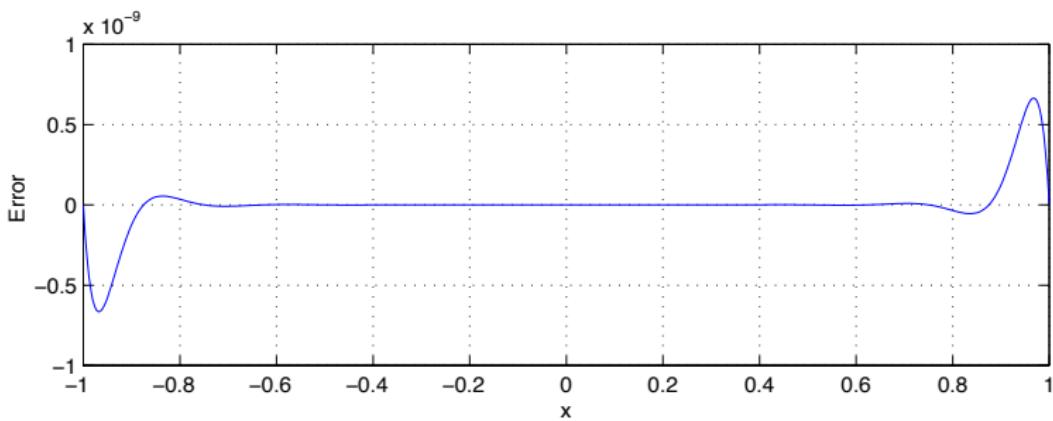
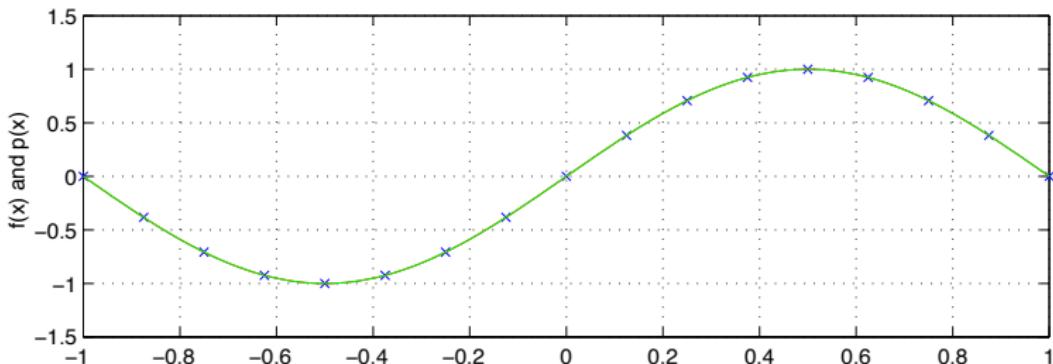
Error with uniform nodes: $n=4$



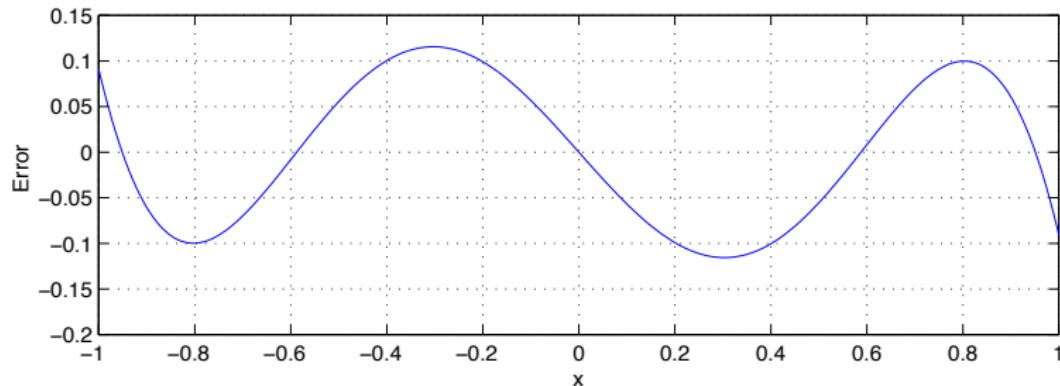
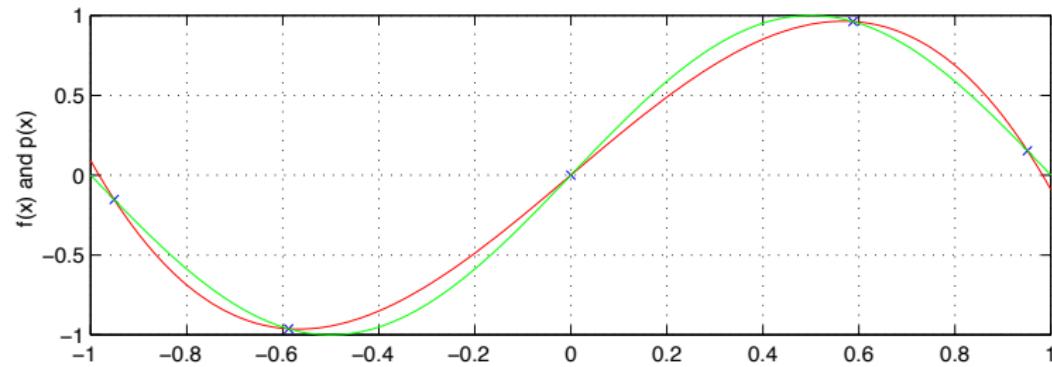
Error with uniform nodes: $n=8$



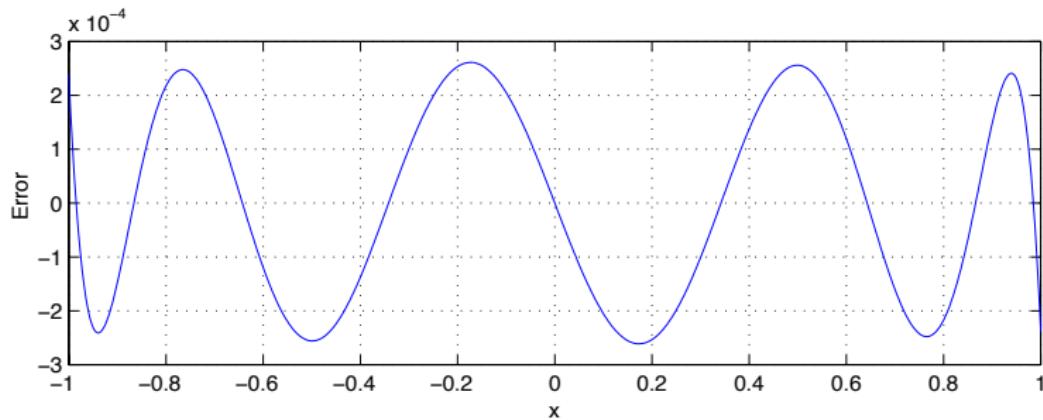
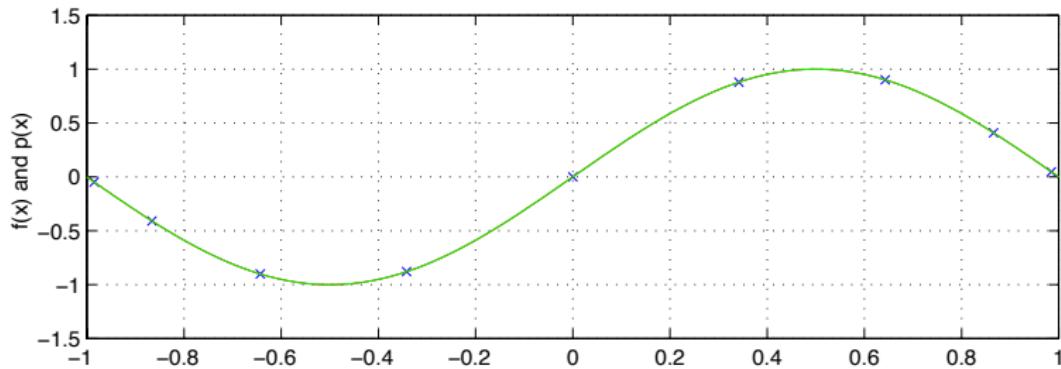
Error with uniform nodes: $n=16$



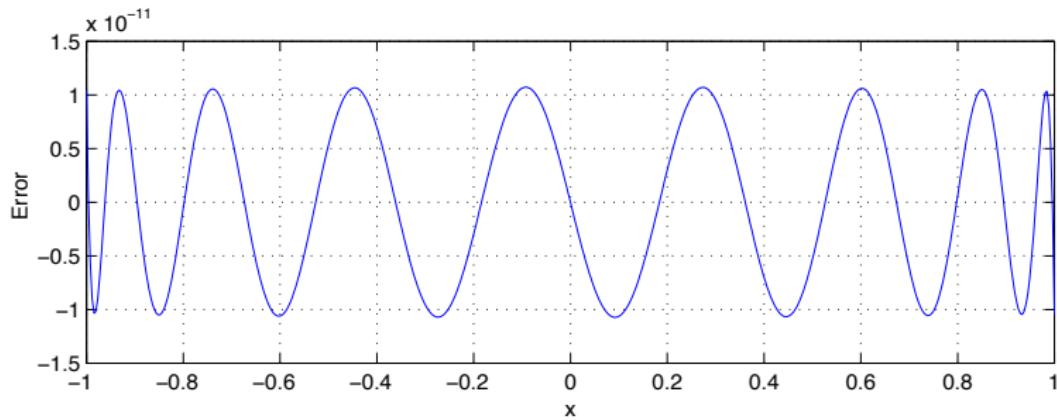
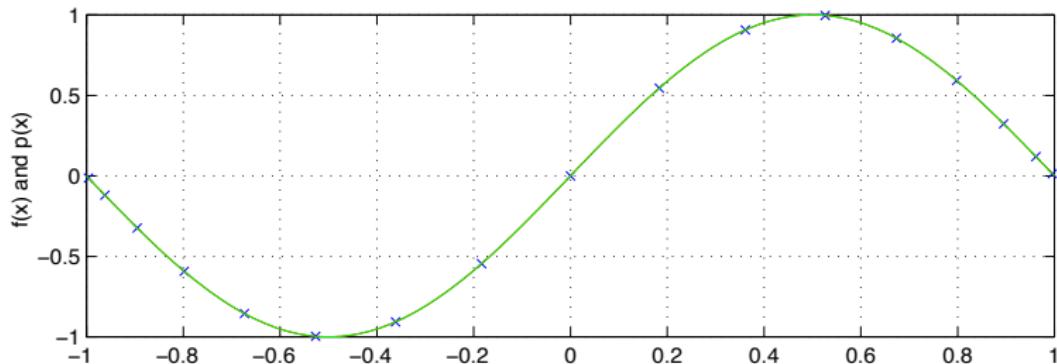
Chebyshev-nodes, $n = 4$



Error with Chebyshev nodes: $n=8$



Error with Chebyshev nodes: $n=16$



CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Introduction

Disadvantages of polynomial interpolation $P_n(x)$

- n -time differentiable. We do not need such high smoothness;
- big error in certain intervals (esp. near the ends);
- no convergence result;
- Heavy to compute for large n

Suggestion: use piecewise polynomial interpolation.

Usage:

- visualization of discrete data
- graphic design

Requirement:

- interpolation
- certain degree of smoothness

Problem setting

Given a set of data

x	t_0	t_1	\cdots	t_n
y	y_0	y_1	\cdots	y_n

Find a function $S(x)$ which interpolates the points $(t_i, y_i)_{i=0}^n$.

The set $t_0 < t_1 < \cdots < t_n$ are called knots. Note that they need to be ordered.

$S(x)$ consists of piecewise polynomials

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

Definition for a spline of degree k

$\mathcal{S}(x)$ is called a *spline of degree k* , if

- $\mathcal{S}_i(x)$ is a polynomial of degree k ;
- $\mathcal{S}(x)$ is $(k - 1)$ times continuous differentiable, i.e., for $i = 1, 2, \dots, k - 1$ we have

$$\begin{aligned}\mathcal{S}_{i-1}(t_i) &= \mathcal{S}_i(t_i), \\ \mathcal{S}'_{i-1}(t_i) &= \mathcal{S}'_i(t_i), \\ &\vdots \\ \mathcal{S}_{i-1}^{(k-1)}(t_i) &= \mathcal{S}_i^{(k-1)}(t_i),\end{aligned}$$

Commonly used splines:

- $k = 1$: linear spline (simplest)
- $k = 2$: quadratic spline (less popular)
- $k = 3$: cubic spline (most used)

Example 1. Determine whether this function is a first-degree spline function:

$$S(x) = \begin{cases} x & x \in [-1, 0] \\ 1 - x & x \in (0, 1) \\ 2x - 2 & x \in [1, 2] \end{cases}$$

Answer. Check all the properties of a linear spline.

- Linear polynomial for each piece: OK.
- $S(x)$ is continuous at inner knots:

At $x = 0$, $S(x)$ is discontinuous, because from the left we get 0 and from the right we get 1.

Therefore this is NOT a linear spline.

Example 2. Determine whether the following function is a quadratic spline:

$$S(x) = \begin{cases} x^2 & x \in [-10, 0] \\ -x^2 & x \in (0, 1) \\ 1 - 2x & x \geq 1 \end{cases}$$

Answer. Let's label each piece:

$$Q_0(x) = x^2, \quad Q_1(x) = -x^2, \quad Q_2(x) = 1 - 2x.$$

We now check all the conditions, i.e, the continuity of Q and Q' at inner knots 0, 1:

$$Q_0(0) = 0, \quad Q_1(0) = 0, \quad \text{OK}$$

$$Q_1(1) = -1, \quad Q_2(1) = -1, \quad \text{OK}$$

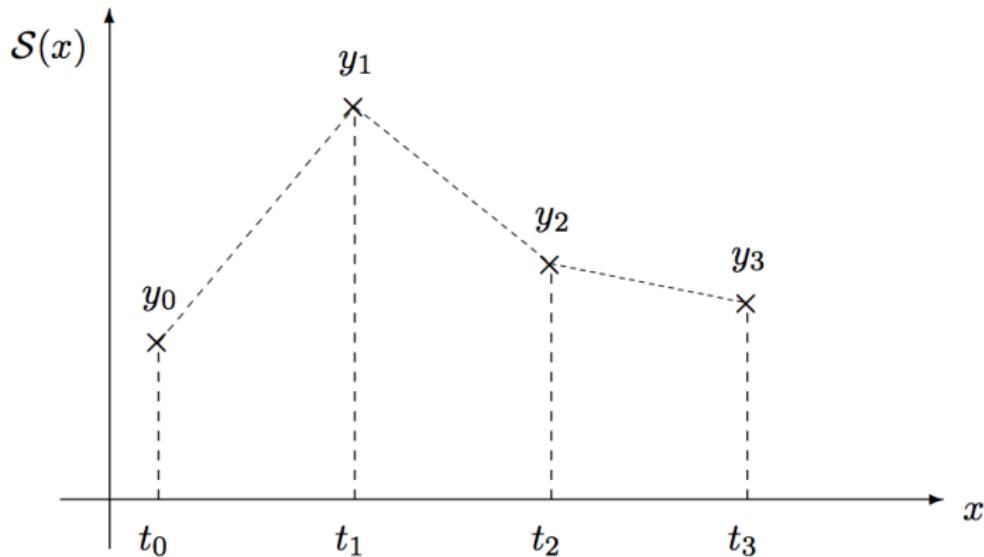
$$Q'_0(0) = 0, \quad Q'_1(0) = 0, \quad \text{OK}$$

$$Q'_1(1) = -2, \quad Q'_2(1) = -2, \quad \text{OK}$$

It passes all the test, so it is a quadratic spline.

Linear Spline

$n = 1$: piecewise linear interpolation, i.e., straight line between 2 neighboring points.



Requirements:

$$\begin{aligned}\mathcal{S}_0(t_0) &= y_0 \\ \mathcal{S}_{i-1}(t_i) = \mathcal{S}_i(t_i) &= y_i, \quad i = 1, 2, \dots, n-1 \\ \mathcal{S}_{n-1}(t_n) &= y_n.\end{aligned}$$

Easy to find: write the equation for a line through two points: (t_i, y_i) and (t_{i+1}, y_{i+1}) ,

$$\mathcal{S}_i(x) = y_i + \frac{y_{i+1} - y_i}{t_{i+1} - t_i}(x - t_i), \quad i = 0, 1, \dots, n-1.$$

Accuracy Theorem for linear spline:

Assume $t_0 < t_1 < t_2 < \cdots < t_n$, and let $h_i = t_{i+1} - t_i$, $h = \max_i h_i$.

$f(x)$: given function, $\mathcal{S}(x)$: a linear spline

$$\mathcal{S}(t_i) = f(t_i), \quad i = 0, 1, \dots, n$$

We have the following, for $x \in [t_0, t_n]$,

(1) If f'' exists and is continuous, then

$$|f(x) - \mathcal{S}(x)| \leq \max_i \left\{ \frac{1}{8} h_i^2 \max_{t_i \leq x \leq t_{i+1}} |f''(x)| \right\} \leq \frac{1}{8} h^2 \max_x |f''(x)|.$$

(2) If f' exists and is continuous, then

$$|f(x) - \mathcal{S}(x)| \leq \max_i \left\{ \frac{1}{2} h_i \max_{t_i \leq x \leq t_{i+1}} |f'(x)| \right\} \leq \frac{1}{2} h \max_x |f'(x)|.$$

To minimize error, it is obvious that one should add more knots where the function has large first or second derivative.

Quadratic spline: Self study.

Natural cubic spline

Given $t_0 < t_1 < \cdots < t_n$, we define the *cubic spline*, with

$$S(x) = S_i(x) \quad \text{for} \quad t_i \leq x \leq t_{i+1}$$

Write

$$S_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i, \quad i = 0, 1, \dots, n-1$$

Total number of unknowns = $4 \cdot n$.

Requirements: S, S', S'' are all continuous.

Equations we have

equation		number	total = 4n.
(1) $\mathcal{S}_i(t_i) = y_i,$	$i = 0, 1, \dots, n - 1$	n	
(2) $\mathcal{S}_i(t_{i+1}) = y_{i+1},$	$i = 0, 1, \dots, n - 1$	n	
(3) $\mathcal{S}'_i(t_{i+1}) = \mathcal{S}'_{i+1}(t_{i+1}),$	$i = 0, 1, \dots, n - 2$	$n - 1$	
(4) $\mathcal{S}''_i(t_{i+1}) = \mathcal{S}''_{i+1}(t_{i+1}),$	$i = 0, 1, \dots, n - 2$	$n - 1$	
(5) $\mathcal{S}''_0(t_0) = 0, \mathcal{S}''_{n-1}(t_n) = 0,$		2	

How to compute $S_i(x)$? We know:

S_i : polynomial of degree 3

S'_i : polynomial of degree 2

S''_i : polynomial of degree 1

Procedure:

- Start with $S''_i(x)$, they are all linear, one can use Lagrange form,
- Integrate $S''_i(x)$ twice to get $S_i(x)$, you will get 2 integration constant
- Determine these constants. Various tricks on the way...

Natural Cubic Splines; Derivation of Algorithm

Define: $z_i = S''(t_i)$, $i = 1, 2, \dots, n - 1$, $z_0 = z_n = 0$

NB! These z_i 's are our unknowns.

Let $h_i = t_{i+1} - t_i$. Lagrange form for S_i'' :

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

Then

$$S_i'(x) = \frac{z_{i+1}}{2h_i}(x - t_i)^2 - \frac{z_i}{2h_i}(x - t_{i+1})^2 + C_i - D_i$$

$$S_i(x) = \frac{z_{i+1}}{6h_i}(x - t_i)^3 - \frac{z_i}{6h_i}(x - t_{i+1})^3 + C_i(x - t_i) - D_i(x - t_{i+1}).$$

You can check by yourself that these S_i, S_i' are correct.

Interpolating properties: (1). $\mathcal{S}_i(t_i) = y_i$ gives

$$y_i = -\frac{z_i}{6h_i}(-h_i)^3 - D_i(-h_i) = \frac{1}{6}z_i h_i^2 + D_i h_i \quad \Rightarrow \quad D_i = \frac{y_i}{h_i} - \frac{h_i}{6} z_i$$

(2). $\mathcal{S}_i(t_{i+1}) = y_{i+1}$ gives

$$y_{i+1} = \frac{z_{i+1}}{6h_i} h_i^3 + C_i h_i, \quad \Rightarrow \quad C_i = \frac{y_{i+1}}{h_i} - \frac{h_i}{6} z_{i+1}.$$

We see that, once z_i 's are known, then (C_i, D_i) 's are known, and so $\mathcal{S}_i, \mathcal{S}'_i$ are known.

$$\begin{aligned}\mathcal{S}_i(x) &= \frac{z_{i+1}}{6h_i}(x - t_i)^3 - \frac{z_i}{6h_i}(x - t_{i+1})^3 + \left(\frac{y_{i+1}}{h_i} - \frac{h_i}{6} z_{i+1} \right) (x - t_i) \\ &\quad - \left(\frac{y_i}{h_i} - \frac{h_i}{6} z_i \right) (x - t_{i+1}). \\ \mathcal{S}'_i(x) &= \frac{z_{i+1}}{2h_i}(x - t_i)^2 - \frac{z_i}{2h_i}(x - t_{i+1})^2 + \frac{y_{i+1} - y_i}{h_i} - \frac{z_{i+1} - z_i}{6} h_i.\end{aligned}$$

Continuity of $S'(x)$ requires

$$S'_{i-1}(t_i) = S'_i(t_i), \quad i = 1, 2, \dots, n-1$$

$$\begin{aligned} S'_i(t_i) &= -\frac{z_i}{2h_i}(-h_i)^2 + \underbrace{\frac{y_{i+1}-y_i}{h_i}}_{b_i} - \frac{z_{i+1}-z_i}{6}h_i \\ &= -\frac{1}{6}h_iz_{i+1} - \frac{1}{3}h_iz_i + b_i \\ S'_{i-1}(t_i) &= \frac{1}{6}z_{i-1}h_{i-1} + \frac{1}{3}z_ih_{i-1} + b_{i-1} \end{aligned}$$

Set them equal to each other, we get

$$\begin{cases} h_{i-1}z_{i-1} + 2(h_{i-1} + h_i)z_i + h_iz_{i+1} = 6(b_i - b_{i-1}), & i = 1, 2, \dots, n-1 \\ z_0 = z_n = 0. \end{cases}$$

$$\begin{cases} h_{i-1}z_{i-1} + 2(h_{i-1} + h_i)z_i + h_iz_{i+1} = 6(b_i - b_{i-1}), & i = 1, 2, \dots, n-1 \\ z_0 = z_n = 0. \end{cases}$$

In matrix-vector form: $\mathbf{H} \cdot \vec{z} = \vec{b}$

$$\mathbf{H} = \begin{pmatrix} 2(h_0 + h_1) & h_1 & & & & \\ h_1 & 2(h_1 + h_2) & h_2 & & & \\ & h_2 & 2(h_2 + h_3) & h_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & h_{n-3} & 2(h_{n-3} + h_{n-2}) & h_{n-2} \\ & & & & h_{n-2} & 2(h_{n-2} + h_{n-1}) \end{pmatrix}$$

\mathbf{H} : tri-diagonal, symmetric, and diagonal dominant

$$2|h_{i-1} + h_i| > |h_i| + |h_{i-1}|$$

which implies unique solution.

$$\begin{cases} h_{i-1}z_{i-1} + 2(h_{i-1} + h_i)z_i + h_iz_{i+1} = 6(b_i - b_{i-1}), & i = 1, 2, \dots, n-1 \\ z_0 = z_n = 0. \end{cases}$$

$$\vec{z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_{n-2} \\ z_{n-1} \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 6(b_1 - b_0) \\ 6(b_2 - b_1) \\ 6(b_3 - b_2) \\ \vdots \\ 6(b_{n-2} - b_{n-3}) \\ 6(b_{n-1} - b_{n-2}) \end{pmatrix}.$$

Summarizing the algorithm:

- Set up the matrix-vector equation and solve for z_i .
- Compute $S_i(x)$ using these z_i 's.

See Matlab codes.

Theorem on smoothness of cubic splines.

Theorem. If S is the natural cubic spline function that interpolates a twice-continuously differentiable function f at knots

$$a = t_0 < t_1 < \cdots < t_n = b$$

then

$$\int_a^b [S''(x)]^2 dx \leq \int_a^b [f''(x)]^2 dx.$$

Note that $\int(f'')^2$ is related to the curvature of f .

Cubic spline gives the least curvature, \Rightarrow most smooth, so best choice.

Proof. Let

$$g(x) = f(x) - \mathcal{S}(x)$$

Then

$$g(t_i) = 0, \quad i = 0, 1, \dots, n$$

and

$$f'' = \mathcal{S}'' + g'', \quad (f'')^2 = (\mathcal{S}'')^2 + (g'')^2 + 2\mathcal{S}''g''$$

$$\Rightarrow \int_a^b (f'')^2 dx = \int_a^b (\mathcal{S}'')^2 dx + \int_a^b (g'')^2 dx + \int_a^b 2\mathcal{S}''g'' dx$$

We claim that

$$\int_a^b \mathcal{S}''g'' dx = 0$$

then this would imply

$$\int_a^b (f'')^2 dx \geq \int_a^b (\mathcal{S}'')^2 dx$$

and we are done.

Proof of the claim:

$$\int_a^b \mathcal{S}'' g'' dx = 0$$

Using integration-by-parts,

$$\int_a^b \mathcal{S}'' g'' dx = \mathcal{S}'' g' \Big|_a^b - \int_a^b \mathcal{S}''' g' dx$$

Since $\mathcal{S}''(a) = \mathcal{S}''(b) = 0$, the first term is 0.

For the second term, since \mathcal{S}''' is piecewise constant. Call

$$c_i = \mathcal{S}'''(x), \quad \text{for } x \in [t_i, t_{i+1}].$$

Then

$$\int_a^b \mathcal{S}''' g' dx = \sum_{i=0}^{n-1} c_i \int_{t_i}^{t_{i+1}} g'(x) dx = \sum_{i=0}^{n-1} c_i [g(t_{i+1}) - g(t_i)] = 0,$$

(b/c $g(t_i) = 0$).

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Natural cubic spline

Computing z_i :

```
function z = cspline(t,y)
n = length(t);
z = zeros(n,1); h = zeros(n-1,1); b = zeros(n-1,1);
u = zeros(n,1); v = zeros(n,1);

h = t(2:n)-t(1:n-1); b = (y(2:n)-y(1:n-1))./h;
u(2) = 2*(h(1)+h(2)); v(2) = 6*(b(2)-b(1));

for i=3:n-1
    u(i) = 2*(h(i)+h(i-1))-h(i-1)^2/u(i-1);
    v(i) = 6*(b(i)-b(i-1))-h(i-1)*v(i-1)/u(i-1);
end

for i=n-1:-1:2
    z(i) = (v(i)-h(i)*z(i+1))/u(i);
end
```

Computing $S(x)$ for a given x :

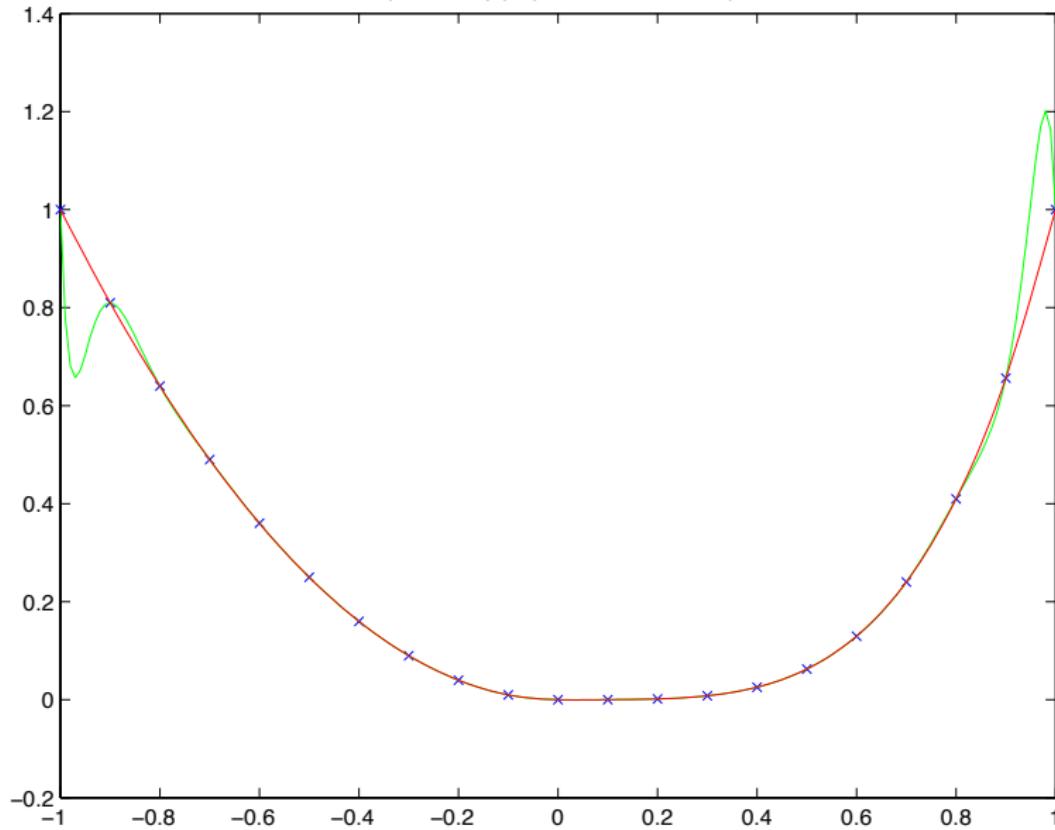
```
function S = cspline_eval(t,y,z,x)
m = length(x);
n = length(t);
for i=n-1:-1:1
    if (x-t(i)) >= 0
        break
    end
end
h = t(i+1)-t(i);
S = z(i+1)/(6*h)*(x-t(i))^3 ...
    -z(i)/(6*h)*(x-t(i+1))^3 ...
    +(y(i+1)/h-z(i+1)*h/6)*(x-t(i)) ...
    -(y(i)/h-z(i)*h/6)*(x-t(i+1));
```

Use your functions:

```
>> t = [0.9,1.3,1.9,2.1]
t =
    0.9000    1.3000    1.9000    2.1000
>> y = [1.3,1.5,1.85,2.1]
y =
    1.3000    1.5000    1.8500    2.1000
>> z = cspline(t,y)
z =
    0
   -0.5634
    2.7113
    0
>> cspline_eval(t,y,z,1.5)
ans =
    1.5810
```

Comparing polynomial interpolation with cubic spline:

interpolation by polynomial and cubic spline



knots (x), polynomial interpolation (-), cubic spline (-)

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Trapezoid Rule : $\int_a^b f(x) dx \approx h \left[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n) \right]$

Example 1: Let $f(x) = \sqrt{x^2 + 1}$, compute $I = \int_{-1}^1 f(x) dx$ by Trapezoid rule with $n = 10$.

Answer. We can set up the data

i	x_i	f_i
0	-1	1.4142136
1	-0.8	1.2806248
2	-0.6	1.1661904
3	-0.4	1.077033
4	-0.2	1.0198039
5	0	1.0
6	0.2	1.0198039
7	0.4	1.077033
8	0.6	1.1661904
9	0.8	1.2806248
10	1	1.4142136

Here $h = 2/10 = 0.2$.

By the formula, we get

$$T = h \left[(f_0 + f_{10})/2 + \sum_{i=1}^9 f_i \right] = 2.3003035.$$

Sample codes for Trapezoid Rule in Matlab.

Let $f(x) = x^2 + \sin(x)$. It can be defined in file “func.m” as:

```
function v=func(x)
    v=x.^2 + sin(x);
end
```

In the following script, the integral value is stored in the variable ‘T’.

```
h=(b-a)/n; x=[a:h:b];
T = (func(a)+func(b))/2;
for i=2:1:n, T = T + func(x(i)); end
T = T*h;
```

Or, one may use directly the Matlab vector function ‘sum’, and the code could be very short:

```
h=(b-a)/n;
x=[a+h:h:b-h]; % inner points
T = ((func(a)+func(b))/2 + sum(func(x)))*h;
```

Numerical integration: Introduction

Problem Description:

Given a function $f(x)$, defined on an interval $[a, b]$, we want to find an approximation to the integral

$$I(f) = \int_a^b f(x) dx .$$

Main ideas:

- Cut up $[a, b]$ into smaller sub-intervals;
- In each sub-interval, find a polynomial $p_i(x) \approx f(x)$;
- Integrate $p_i(x)$ on each sub-interval, and sum them up.

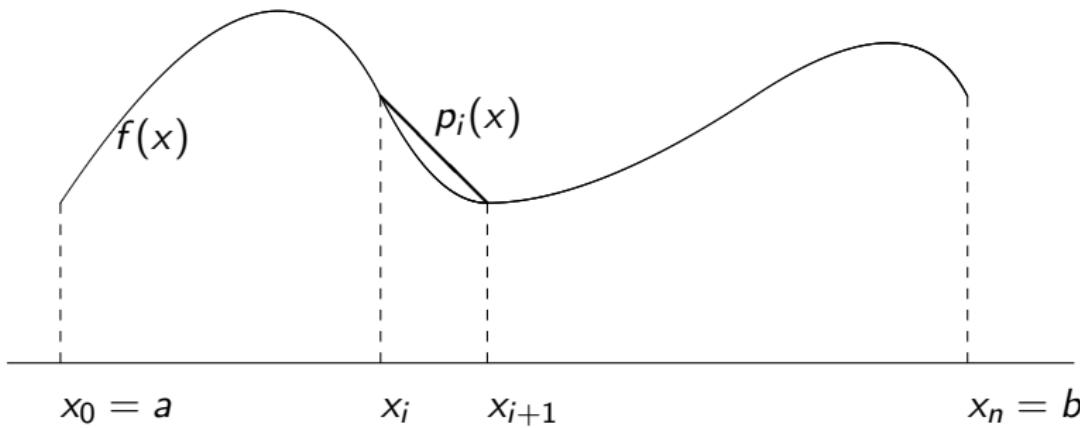
Trapezoid rule

The grid: We cut up $[a, b]$ into n sub-intervals:

$$x_0 = a, \quad x_i < x_{i+1}, \quad x_n = b$$

On $[x_i, x_{i+1}]$, approximate $f(x)$ by a linear polynomial p_i :

$$p_i(x_i) = f(x_i), \quad p_i(x_{i+1}) = f(x_{i+1}).$$



On each sub-interval, the integral of p_i equals to the area of a trapezium:

$$\int_{x_i}^{x_{i+1}} p_i(x) dx = \frac{1}{2}(f(x_i) + f(x_{i+1}))(x_{i+1} - x_i).$$

Now, we use

$$\int_{x_i}^{x_{i+1}} f(x) dx \approx \int_{x_i}^{x_{i+1}} p_i(x) dx = \frac{1}{2} (f(x_{i+1}) + f(x_i)) (x_{i+1} - x_i),$$

and we sum up all the sub-intervals

$$\begin{aligned}\int_a^b f(x) dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x) dx \approx \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} p_i(x) dx \\ &= \sum_{i=0}^{n-1} \frac{1}{2} (f(x_{i+1}) + f(x_i)) (x_{i+1} - x_i)\end{aligned}$$

Uniform Grid

$$h = \frac{b - a}{n}, \quad x_{i+1} - x_i = h.$$

$$\begin{aligned}\int_a^b f(x) dx &\approx \sum_{i=0}^{n-1} \frac{h}{2} (f(x_i) + f(x_{i+1})) \\&= \frac{h}{2} [(f(x_0) + f(x_1)) + (f(x_1) + f(x_2)) + \cdots + (f(x_{n-1}) + f(x_n))] \\&= \frac{h}{2} \left[f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n) \right] \\&= \underbrace{h \left[\frac{1}{2}(f(x_0) + f(x_n)) + \sum_{i=1}^{n-1} f(x_i) \right]}_{T(f; h)}\end{aligned}$$

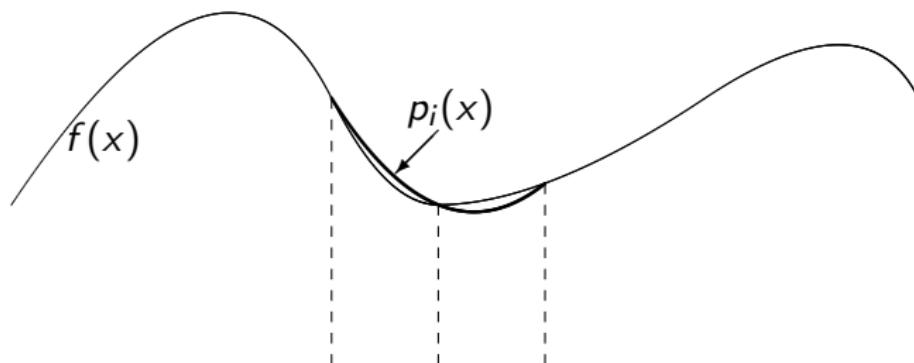
Simpson's rule

We now explore possibility of using higher order polynomials.

We cut up $[a, b]$ into $2n$ equal sub-intervals

$$x_0 = a, \quad x_{2n} = b, \quad h = \frac{b - a}{2n}, \quad x_{i+1} - x_i = h$$

On $[x_{2i}, x_{2i+2}]$, interpolates $f(x)$ at the points $x_{2i}, x_{2i+1}, x_{2i+2}$ with a quadratic polynomial $p_i(x)$.



$$x_{2i} \quad x_{2i+1} \quad x_{2i+2}$$

Note that in each sub-interval there is a point in the interior.

Lagrange form for $p_i(x)$:

$$\begin{aligned} p_i(x) &= f(x_{2i}) \frac{(x - x_{2i+1})(x - x_{2i+2})}{(x_{2i} - x_{2i+1})(x_{2i} - x_{2i+2})} + f(x_{2i+1}) \frac{(x - x_{2i})(x - x_{2i+2})}{(x_{2i+1} - x_{2i})(x_{2i+1} - x_{2i+2})} \\ &\quad + f(x_{2i+2}) \frac{(x - x_{2i})(x - x_{2i+1})}{(x_{2i+2} - x_{2i})(x_{2i+2} - x_{2i+1})} \end{aligned}$$

With uniform nodes, this becomes

$$\begin{aligned} p_i(x) &= \frac{1}{2h^2} f(x_{2i})(x - x_{2i+1})(x - x_{2i+2}) - \frac{1}{h^2} f(x_{2i+1})(x - x_{2i})(x - x_{2i+2}) \\ &\quad + \frac{1}{2h^2} f(x_{2i+2})(x - x_{2i})(x - x_{2i+1}) \end{aligned}$$

We work out the integrals (try to fill in the details yourself!)

$$I_1 = \int_{x_{2i}}^{x_{2i+2}} (x - x_{2i+1})(x - x_{2i+2}) dx = \frac{2}{3}h^3,$$

$$I_2 = \int_{x_{2i}}^{x_{2i+2}} -(x - x_{2i})(x - x_{2i+2}) dx = \frac{4}{3}h^3,$$

$$I_3 = \int_{x_{2i}}^{x_{2i+2}} (x - x_{2i})(x - x_{2i+1}) dx = \frac{2}{3}h^3,$$

Then

$$\begin{aligned}\int_{x_{2i}}^{x_{2i+2}} p_i(x) dx &= \frac{1}{2h^2} f(x_{2i}) \cdot I_1 + \frac{1}{h^2} f(x_{2i+1}) \cdot I_2 + \frac{1}{2h^2} f(x_{2i+2}) \cdot I_3 \\&= \frac{1}{2h^2} f(x_{2i}) \frac{2}{3}h^3 + \frac{1}{h^2} f(x_{2i+1}) \frac{4}{3}h^3 + \frac{1}{2h^2} f(x_{2i+2}) \frac{2}{3}h^3 \\&= \frac{h}{3} [f(x_{2i}) + 4f(x_{2i+1}) + f(x_{2i+2})].\end{aligned}$$

We now sum them up

$$\begin{aligned}\int_a^b f(x) dx &\approx S(f; h) = \sum_{i=0}^{n-1} \int_{x_{2i}}^{x_{2i+2}} p_i(x) dx \\ &= \frac{h}{3} \sum_{i=0}^{n-1} [f(x_{2i}) + 4f(x_{2i+1}) + f(x_{2i+2})].\end{aligned}$$

$$\begin{array}{ccccc} & \boxed{1} & & 4 & 1 \\ 1 & & 4 & & \\ & \boxed{1} & & & \end{array}$$

$$x_{2i-2} \quad x_{2i-1} \quad x_{2i} \quad x_{2i+1} \quad x_{2i+2}$$

Simpson's Rule:

$$S(f; h) = \frac{h}{3} \left[f(x_0) + 4 \sum_{i=1}^n f(x_{2i-1}) + 2 \sum_{i=1}^{n-1} f(x_{2i}) + f(x_{2n}) \right]$$

Error estimates for Trapezoid rule.

We define the error:

$$E_T(f; h) \doteq I(f) - T(f; h) = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} [f(x) - p_i(x)] dx = \sum_{i=0}^{n-1} E_{T,i}(f; h),$$

where $E_{T,i}(f; h)$ is the error on each sub-interval

$$E_{T,i}(f; h) = \int_{x_i}^{x_{i+1}} [f(x) - p_i(x)] dx, \quad (i = 0, 1, \dots, n-1)$$

Error bound with polynomial interpolation:

$$f(x) - p_i(x) = \frac{1}{2} f''(\xi_i)(x - x_i)(x - x_{i+1}), \quad (x_i < \xi_i < x_{i+1})$$

Error estimate on each sub-interval:

$$E_{T,i}(f; h) = \frac{1}{2} f''(\xi_i) \int_{x_i}^{x_{i+1}} (x - x_i)(x - x_{i+1}) dx = -\frac{1}{12} h^3 f''(\xi_i).$$

(You may work out the details of the integral!)

The total error is:

$$\begin{aligned} E_T(f; h) &= \sum_{i=0}^{n-1} E_{T,i}(f; h) = \sum_{i=0}^{n-1} -\frac{1}{12} h^3 f''(\xi_i) \\ &= -\frac{1}{12} h^3 \underbrace{\left[\sum_{i=0}^{n-1} f''(\xi_i) \right]}_{= f''(\xi)} \cdot \frac{1}{n} \cdot \underbrace{\frac{b-a}{h}}_{= n} \end{aligned}$$

which gives

$$E_T(f; h) = -\frac{b-a}{12} h^2 f''(\xi), \quad \xi \in (a, b).$$

Error bound

$$|E_T(f; h)| \leq \frac{b-a}{12} h^2 \max_{x \in (a,b)} |f''(x)|.$$

Example 2. Consider function $f(x) = e^x$, and the integral $I(f) = \int_0^2 e^x dx$. What is the minimum number of points to be used in the trapezoid rule to ensure an error $\leq 0.5 \times 10^{-4}$?

Answer. We have

$$f'(x) = e^x, \quad f''(x) = e^x, \quad a = 0, \quad b = 2, \quad \max_{x \in (a,b)} |f''(x)| = e^2.$$

By error bound, it is sufficient to require

$$\begin{aligned} |E_T(f; h)| &\leq \frac{1}{6} h^2 e^2 \leq 0.5 \times 10^{-4} \\ \Rightarrow h^2 &\leq 0.5 \times 10^{-4} \times 6 \times e^{-2} \approx 4.06 \times 10^{-5} \\ \Rightarrow \frac{2}{n} &= h \leq \sqrt{4.06 \times 10^{-5}} = 0.0064 \\ \Rightarrow n &\geq \frac{2}{0.0064} \approx 313.8 \end{aligned}$$

We need at least 314 points.

Romberg Algorithm

Given $T(f; h)$, $T(f; h/2)$, compute

$$U(h) = T(f; h/2) + \frac{T(f; h/2) - T(f; h)}{2^2 - 1}$$

then

$$I(f) = U(h) + \tilde{a}_4 h^4 + \tilde{a}_6 h^6 + \dots$$

We can iterate this idea. Assume we have computed $U(h)$, $U(h/2)$,

$$(3) \quad I(f) = U(h) + \tilde{a}_4 h^4 + \tilde{a}_6 h^6 + \dots$$

$$(4) \quad I(f) = U(h/2) + \tilde{a}_4 (h/2)^4 + \tilde{a}_6 (h/2)^6 + \dots$$

Cancel the leading error term: $(4) \times 2^4 - (3)$

$$(2^4 - 1)I(f) = 2^4 U(h/2) - U(h) + \tilde{a}'_6 h^6 + \dots$$

Let

$$V(h) = \frac{2^4 U(h/2) - U(h)}{2^4 - 1} = U(h/2) + \frac{U(h/2) - U(h)}{2^4 - 1}.$$

Then

$$I(f) = V(h) + \tilde{a}'_6 h^6 + \dots \quad \text{6th order approximation}$$

So $V(h)$ is even better than $U(h)$.

One can keep doing this several layers, until desired accuracy is reached.

This gives the **Romberg Algorithm**

Romberg Algorithm

Set $H = b - a$, define:

$$R(0, 0) = T(f; H) = \frac{H}{2}(f(a) + f(b))$$

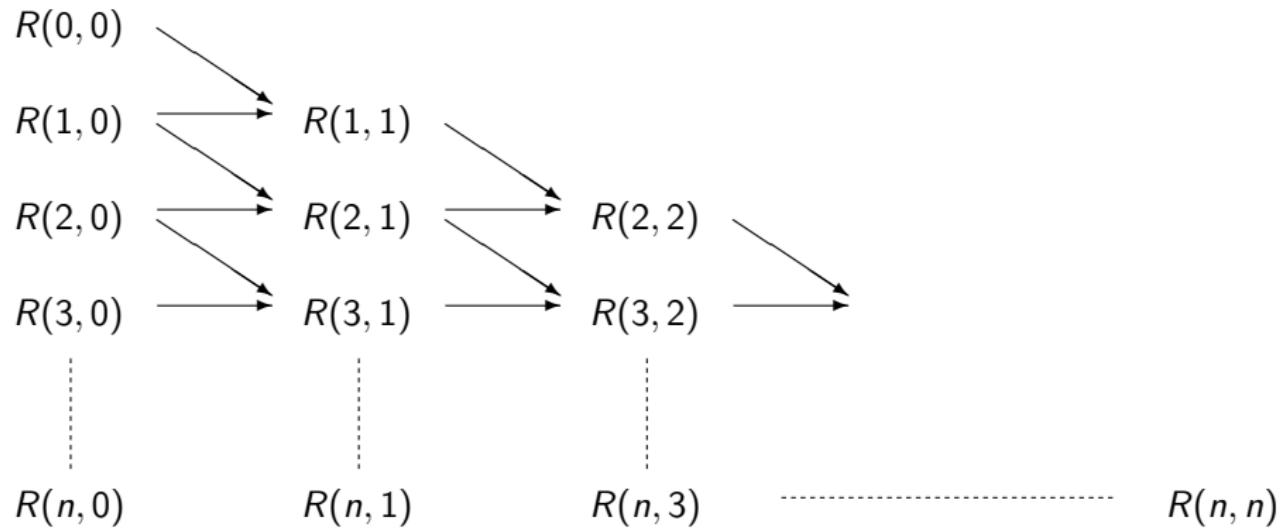
$$R(1, 0) = T(f; H/2)$$

$$R(2, 0) = T(f; H/(2^2))$$

⋮

$$R(n, 0) = T(f; H/(2^n))$$

Romberg triangle



The entry $R(n, m)$ is computed as

$$R(n, m) = R(n, m - 1) + \frac{R(n, m - 1) - R(n - 1, m - 1)}{2^{2m} - 1}$$

Accuracy:

$$I(f) = R(n, m) + \mathcal{O}(h^{2(m+1)}), \quad h = \frac{H}{2^m}.$$

Algorithm can be done either column-by-column or row-by-row.

Romberg algorithm pseudo-code, using column-by-column

$R = \text{romberg}(f, a, b, n)$

$R = n \times n$ matrix

$h = b - a; R(1, 1) = [f(a) + f(b)] * h/2;$

for $i = 1$ to $n - 1$ do % 1st column recursive trapezoid

$R(i + 1, 1) = R(i, 1)/2; h = h/2;$

for $k = 1$ to 2^{i-1} do

$R(i + 1, 1) = R(i + 1, 1) + h * f(a + (2k - 1)h)$

end

end

for $j = 2$ to n do % 2 to n column

for $i = j$ to n do

$R(i, j) = R(i, j - 1) + \frac{1}{4^{j-1}-1} [R(i, j - 1) - R(i - 1, j - 1)]$

end

end

Richardson Extrapolation

Given $T(f; h), T(f; h/2), T(f; h/4), \dots$, a sequence of approximations by Trapezoid rule with different values of h .

Idea: One could combine these numbers in particular ways to get much higher order approximations.

The particular form of the eventual algorithm depends on the detailed error formula.

One can show: If $f^{(n)}$ exists and is bounded, the error for trapezoid rule satisfies the Euler MacLaurin's formula

$$E(f; h) = I(f) - T(f; h) = a_2 h^2 + a_4 h^4 + a_6 h^6 + \cdots + a_n h^n$$

Here a_n depends on the derivatives $f^{(n)}$.

Proof can be achieved by Taylor series.

Error formula: $E(f; h) = I(f) - T(f; h) = a_2 h^2 + a_4 h^4 + a_6 h^6 + \cdots + a_n h^n$

When we half the grid size h , the error formula becomes

$$E(f; \frac{h}{2}) = I(f) - T(f; \frac{h}{2}) = a_2(\frac{h}{2})^2 + a_4(\frac{h}{2})^4 + a_6(\frac{h}{2})^6 + \cdots + a_n(\frac{h}{2})^n$$

$$(1) \quad I(f) = T(f; h) + a_2 h^2 + a_4 h^4 + a_6 h^6 + \cdots$$

$$(2) \quad I(f) = T(f; \frac{h}{2}) + a_2(\frac{h}{2})^2 + a_4(\frac{h}{2})^4 + a_6(\frac{h}{2})^6 + \cdots + a_n(\frac{h}{2})^n$$

The goal: cancel the leading error term to get a higher order approximation.
Multiplying (2) by 2^2 and subtract (1), we get

$$\begin{aligned} (2^2 - 1) \cdot I(f) &= 2^2 \cdot T(f; h/2) - T(f; h) + a'_4 h^4 + a'_6 h^6 + \cdots \\ \Rightarrow I(f) &= \underbrace{\frac{4}{3} T(f; h/2) - \frac{1}{3} T(f; h)}_{U(h)} + \tilde{a}_4 h^4 + \tilde{a}_6 h^6 + \cdots \end{aligned}$$

A 4th order approximation:

$$U(h) = \frac{4}{3} T(f; h/2) - \frac{1}{3} T(f; h) = \frac{2^2 T(f; h/2) - T(f; h)}{2^2 - 1}$$

This idea is called the *Richardson extrapolation*.

We do not have to stop here. One can go into many levels of this manipulation! \Rightarrow Romberg Algorithm

Simpson's Rule: $S(f; h) = \frac{h}{3} \left[f(x_0) + 4 \sum_{i=1}^n f(x_{2i-1}) + 2 \sum_{i=1}^{n-1} f(x_{2i}) + f(x_{2n}) \right]$

Example 3: Let $f(x) = \sqrt{x^2 + 1}$, and compute $I = \int_{-1}^1 f(x) dx$ by Simpson's rule with $n = 5$, i.e. with $2n + 1 = 11$ points.

Answer. We can set up the data (same as in Example 1):

i	x_i	f_i
0	-1	1.4142136
1	-0.8	1.2806248
2	-0.6	1.1661904
3	-0.4	1.077033
4	-0.2	1.0198039
5	0	1.0
6	0.2	1.0198039
7	0.4	1.077033
8	0.6	1.1661904
9	0.8	1.2806248
10	1	1.4142136

Here $h = 2/10 = 0.2$.

$$\begin{aligned} S(f; 0.2) &= \frac{h}{3} [f_0 + 4(f_1 + f_3 + f_5 + f_7 + f_9) \\ &\quad + 2(f_2 + f_4 + f_6 + f_8) + f_{10}] \\ &= 2.2955778. \end{aligned}$$

Question to ponder:

This value is somewhat smaller than the number we get with trapezoid rule, and it is actually more accurate. Could you intuitively explain that for this particular example?

Sample codes: Let a, b, n be given, and let the function 'func' be defined.

To find the integral with Simpson's rule, one could possibly follow the following algorithm:

```
h=(b-a)/2/n;
xodd=[a+h:2*h:b-h]; % x_i with odd indices
xeven=[a+2*h:2*h:b-2*h]; % x_i with even indices
S=(h/3)*(func(a)+4*sum(func(xodd))+2*sum(func(xeven))+func(b));
```

Error Estimate for Simpson's Rule.

The basic error on each sub-interval is

$$E_{S,i}(f; h) = -\frac{1}{90} h^5 f^{(4)}(\xi_i), \quad \xi_i \in (x_{2i}, x_{2i+2}). \quad (1)$$

(See lecture notes or textbook for the proof.)

Then, the total error is

$$E_S(f; h) = I(f) - S(f; h) = -\frac{1}{90} h^5 \sum_{i=0}^{n-1} f^{(4)}(\xi_i) \frac{1}{n} \cdot \frac{b-a}{2h} = -\frac{b-a}{180} h^4 f^{(4)}(\xi),$$

This gives us the error bound

$$|E_S(f; h)| \leq \frac{b-a}{180} h^4 \max_{x \in (a,b)} |f^{(4)}(x)|.$$

Example 4. With $f(x) = e^x$ defined on $[0, 2]$, use Simpson's rule to compute $\int_0^2 f(x) dx$. In order to achieve an error $\leq 0.5 \times 10^{-4}$, how many points must we take?

Answer. We have

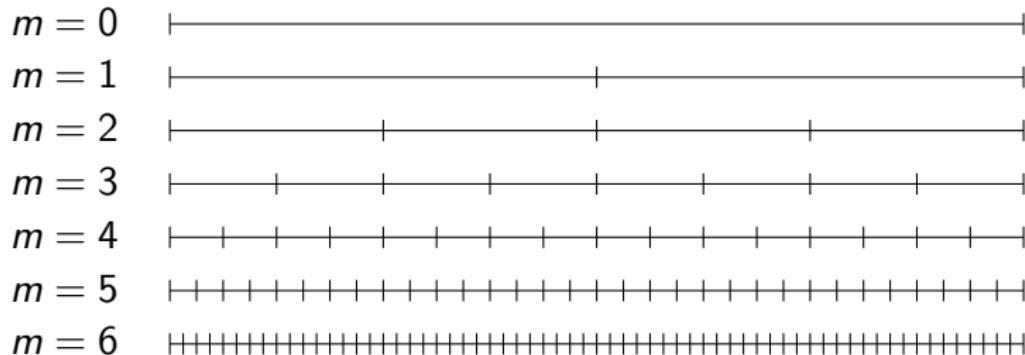
$$\begin{aligned}|E_S(f; h)| &\leq \frac{2}{180} h^4 e^2 \leq 0.5 \times 10^{-4} \\ \Rightarrow h^4 &\leq 45/e^2 \times 10^{-4} \times = 6.09 \times 10^{-4} \\ \Rightarrow h &\leq 0.1571 \\ \Rightarrow n &= \frac{b-a}{2h} = 6.36 \approx 7\end{aligned}$$

We need at least $2n + 1 = 15$ points.

Recall: With trapezoid rule, we need at least 314 points. The Simpson's rule uses much fewer points.

Recursive trapezoid rule; composite schemes

Divide $[a, b]$ into 2^m equal sub-intervals.



$$h_m = \frac{b - a}{2^m}, \quad h_{m+1} = \frac{1}{2} h_m$$

Trapezoid rule:

$$T(f; h_m) = h_m \cdot \left[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{i=1}^{2^m-1} f(a + ih_m) \right]$$

$$T(f; h_{m+1}) = h_{m+1} \cdot \left[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{i=1}^{2^{m+1}-1} f(a + ih_{m+1}) \right]$$

We can re-arrange the terms in $T(f; h_{m+1})$:

$$\begin{aligned} T(f; h_{m+1}) &= \frac{h_m}{2} \left[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{i=1}^{2^m-1} f(a + ih_m) \right. \\ &\quad \left. + \sum_{i=0}^{2^m-1} f(a + (2i+1)h_{m+1}) \right] \\ &= \frac{1}{2} T(f; h_m) + h_{m+1} \sum_{i=0}^{2^m-1} f(a + (2i+1)h_{m+1}) \end{aligned}$$

Advantages:

1. One can keep the computation for a level m . If this turns out to be not accurate enough, then add one more level to get better approximation. \Rightarrow flexibility.
2. This formula allows us to compute a sequence of approximations to a definite integral using the trapezoid rule without re-evaluating the integrand at points where it has already been evaluated. \Rightarrow efficiency.

Question: A similar recursive algorithm could be defined using Simpson's rule. Would you like to try it?

Useful for the next topic: Romberg Algorithm.

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Matlab: Effective programming

Three different ways of computing a vector product of two vectors.

$$z_i = x_i \cdot y_i, \quad i = 1, 2, \dots, n$$

Method 1:

Do not allocate memory space for z in advance.

Compute the elements in the new vector by a for-loop.

```
x = rand(n,1);      % random vector
y = rand(n,1);      % of length n
clear z;
t = cputime;
for i=1:n
    z(i) = x(i)*y(i);
end
cputime-t
```

Method 2:

Allocate space for z in advance, but still use a for loop.

```
z = zeros(n,1);
for i=1:n
    z(i) = x(i)*y(i);
end
```

Method 3:

Use vector operation in Matlab directly.

```
z = x.*y;
```

Result (The CPU-time measured in seconds):

n	Method 1	Method 2	Method 3
5000	2.86	0.24	0.00
10000	14.22	0.49	0.00
20000	59.65	0.97	0.01
100000	--	4.87	0.03
1000000	--	48.84	0.30

Moral: use pre-defined Matlab functions!

Romberg integration

$$I(f) = \int_0^{\pi/2} \cos(2x)e^{-x} dx, \quad (= 0.2415759)$$

Result:

0.6221			
0.3111	0.2074		
0.2575	0.2397	0.2419	
0.2455	0.2415	0.2416	0.2416

Error:

3.80e-01			
6.94e-02	3.41e-02		
1.59e-02	1.87e-03	2.81e-04	
3.90e-03	1.11e-04	5.85e-06	1.47e-06

Expand the integration interval to 2π . Exact value= 0.1996265.

Result:

3.1475

1.7095 1.2302

0.5141 0.1156 0.0413

0.2570 0.1714 0.1751 0.1772

Error:

2.94e+00

1.50e+00 1.03e+00

3.14e-01 8.39e-02 1.58e-01

5.74e-02 2.82e-02 2.45e-02 2.24e-02

Matlab's adaptive Simpsons methode: quad

Syntax:

```
q = quad('f',a,b,toleranse,trace)
```

If the option $trace \neq 0$, Matlab will show the development of the integration.

```
quad('f3',0,2*pi,1.e-6,1)
```

Matlab's recursive numerical integration: quadl and quad8

Syntax and usage would be the same as for quad.
The functions use a high order recursive algorithm.

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Numerical solution of nonlinear equations; Introduction

Problem: Given $f(x)$: continuous, real-valued, possibly non-linear.
Find a root r of $f(x)$ such that $f(r) = 0$.

Example 1. Quadratic polynomials: $f(x) = x^2 + 5x + 6$.

$$f(x) = (x + 2)(x + 3) = 0, \quad \Rightarrow \quad r_1 = -2, \quad r_2 = -3.$$

Observation: Roots are not unique.

Example 2. $f(x) = x^2 + 4x + 10 = (x + 2)^2 + 6$. No roots.

Observation: There are no real r that would satisfy $f(r) = 0$.

Example 3. $f(x) = x^2 + \cos x + e^x + \sqrt{x + 1}$.

Observation: Roots can be difficult/impossible to find analytically.

Overview of the chapter:

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

Bisection method

Basic idea:

Given $f(x)$, a continuous function.

If we find some a and b , such that $f(a)$ and $f(b)$ are of opposite sign, then, there exists a point c , between a and b , such that $f(c) = 0$.

This fact follows from the continuity of f .

We can iterate on this idea!

Procedure:

- Initialization: Find a, b such that $f(a) \cdot f(b) < 0$.
This means there is a root $r \in (a, b)$ s.t. $f(r) = 0$.
- Let $c = \frac{a+b}{2}$, mid-point.
- If $f(c) = 0$, done (lucky!)
else:
 - if $f(c) \cdot f(a) < 0$, pick the interval $[a, c]$
 - if $f(c) \cdot f(b) < 0$, pick the interval $[c, b]$,
- Iterate the procedure until stop criteria satisfied.

Stop Criteria:

- 1) interval small enough, i.e., $(b - a) \leq \epsilon$,
- 2) $|f(c)|$ very small, i.e, $|f(c)| \leq \epsilon$
- 3) max number of iteration reached. (to avoid dead loop, in case the method does not converge.)
- 4) any combination of the previous ones.

Convergence analysis:

Consider $[a_0, b_0]$, $c_0 = \frac{a_0+b_0}{2}$, let $r \in (a_0, b_0)$ be a root.

The error: $e_0 = |r - c_0| \leq \frac{b_0 - a_0}{2}$

Denote the further intervals as $[a_n, b_n]$ for iteration number n .

$$e_n = |r - c_n| \leq \frac{b_n - a_n}{2} \leq \frac{b_0 - a_0}{2^{n+1}} = \frac{e_0}{2^n}.$$

If the error tolerance is ε , we require $e_n \leq \varepsilon$, then

$$\frac{b_0 - a_0}{2^{n+1}} \leq \varepsilon \quad \Rightarrow \quad n \geq \frac{\ln(b-a) - \ln(2\varepsilon)}{\ln 2}, \quad (\# \text{ of steps})$$

Remark: very slow convergence.

Fixed point iterations

We rewrite the equation $f(x) = 0$ into the form $x = g(x)$.

Remark: This can always be achieved, for example: $x = f(x) + x$.
However, the choice of g makes a difference in convergence.

Main idea:

Make a guess of the solution, say \bar{x} . If the function $g(x)$ is “nice”, then hopefully, $g(\bar{x})$ should be closer to the answer than \bar{x} . If that is the case, then we can iterate.

Iteration algorithm:

- Choose a start point x_0 ,
- Do the iteration $x_{k+1} = g(x_k)$, $k = 0, 1, 2, \dots$ until meeting stop criteria.

Stop Criteria: Let ε be the tolerance

- $|x_k - x_{k-1}| \leq \varepsilon$,
- max # of iteration reached,
- any combination.

Example 1. Find an approximate solution to $f(x) = x - \cos x = 0$, with 4 digits accuracy.

Choose $g(x) = \cos x$, we have $x = \cos x$.

Choose $x_0 = 1$, and do the iteration $x_{k+1} = \cos(x_k)$:

$$x_1 = \cos x_0 = 0.5403$$

$$x_2 = \cos x_1 = 0.8576$$

$$x_3 = \cos x_2 = 0.6543$$

⋮

$$x_{23} = \cos x_{22} = 0.7390$$

$$x_{24} = \cos x_{23} = 0.7391$$

$$x_{25} = \cos x_{24} = 0.7391 \quad \text{stop here}$$

Our approximation to the root is 0.7391.

Example 2. Consider $f(x) = e^{-2x}(x - 1) = 0$. (root: $r = 1$).

Rewrite as

$$x = g(x) = e^{-2x}(x - 1) + x$$

Choose an initial guess $x_0 = 0.99$, very close to the real root. Iterations:

$$x_1 = g(x_0) = 0.9886$$

$$x_2 = g(x_1) = 0.9870$$

$$x_3 = g(x_2) = 0.9852$$

⋮

$$x_{27} = 0.1655$$

$$x_{28} = -0.4338$$

$$x_{29} = -3.8477 \quad \text{Diverges. It does not work.}$$

What went wrong?

Fixed point iteration; Convergence analysis.

Our iteration is $x_{k+1} = g(x_k)$. Let r be the exact root, s.t., $r = g(r)$. Define the error: $e_k = |x_k - r|$.

$$\begin{aligned} e_{k+1} &= |x_{k+1} - r| = |g(x_k) - r| = |g(x_k) - g(r)| \\ &= |g'(\xi)| |(x_k - r)| \quad (\xi \in (x_k, r), \text{ since } g \text{ is continuous}) \\ &= |g'(\xi)| e_k \end{aligned}$$

$$\Rightarrow e_{k+1} = |g'(\xi)| e_k.$$

Observation:

- If $|g'(\xi)| < 1$, then $e_{k+1} < e_k$, error decreases, the iteration converges. (linear convergence)
- If $|g'(\xi)| > 1$, then $e_{k+1} > e_k$, error increases, the iteration diverges.

Convergence condition:

There exists an interval around r , say $[r - a, r + a]$ for some $a > 0$, such that $|g'(x)| < 1$ for almost all $x \in [r - a, r + a]$, and the initial guess x_0 lies in this interval.

In Example 1, $g(x) = \cos x$, $g'(x) = \sin x$, $r = 0.7391$,

$$|g'(r)| = |\sin(0.7391)| < 1. \quad \text{OK, convergence.}$$

In Example 2, we have

$$\begin{aligned} g(x) &= e^{-2x}(x - 1) + x, \\ g'(x) &= -2e^{-2x}(x - 1) + x^{-2x} + 1 \end{aligned}$$

With $r = 1$, we have

$$|g'(r)| = e^{-2} + 1 > 1, \quad \text{Divergence.}$$

Pseudo code

```
r=fixedpoint('g', x,tol,nmax)
r=g(x); % first iteration
nit=1;
while (abs(r-g(r))>tol and nit < nmax) do
    r=g(r);
    nit=nit+1;
end
end
```

A practical error estimate

Assume $|g'(x)| \leq m < 1$ in $[r - a, r + a]$. We have $e_{k+1} \leq me_k$.
This gives

$$e_1 \leq me_0, \quad e_2 \leq me_1 \leq m^2 e_0, \quad \dots \quad e_k \leq m^k e_0.$$

This result is useless unless we find a way to estimating e_0 .

$$e_0 = |r - x_0| = |r - x_1 + x_1 - x_0| \leq e_1 + |x_1 - x_0| \leq me_0 + |x_1 - x_0|$$

then

$$e_0 \leq \frac{1}{1-m} |x_1 - x_0|, \quad (\text{can be computed})$$

Put together

$$e_k \leq \frac{m^k}{1-m} |x_1 - x_0|.$$

$$e_k \leq \frac{m^k}{1-m} |x_1 - x_0|.$$

If the error tolerance is ε , then

$$\frac{m^k}{1-m} |x_1 - x_0| \leq \varepsilon,$$

$$m^k \leq \frac{\varepsilon(1-m)}{|x_1 - x_0|}$$

$$k \geq \frac{\ln(\varepsilon(1-m)) - \ln|x_1 - x_0|}{\ln m}$$

which give the minimum number of iterations needed to achieve an error $\leq \varepsilon$.

Example 3. We want to solve $\cos x - x = 0$ with the fixed point iteration

$$x = g(x) = \cos x, \quad x_0 = 1,$$

with error tolerance $\varepsilon = 10^{-5}$. Find the min# iterations.

We know $r \in [0, 1]$.

We see that the iteration happens between $x = 0$ and $x = 1$.

For $x \in [0, 1]$, we have

$$|g'(x)| = |\sin x| \leq \sin 1 = 0.8415, \quad m \doteq 0.8415$$

And $x_1 = \cos x_0 = 0.5403$. Using the formula

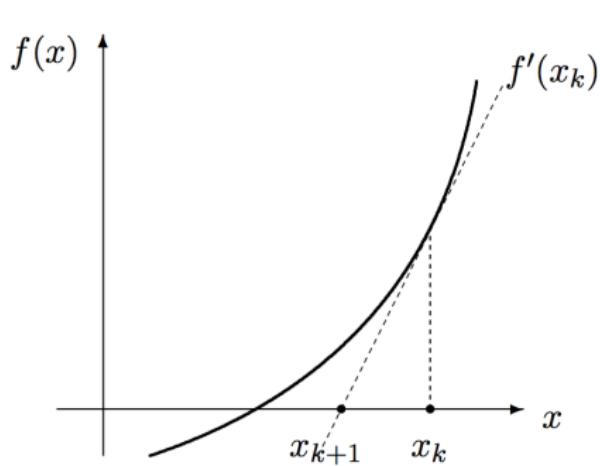
$$k \geq \frac{\ln(\varepsilon(1 - m)) - \ln|x_1 - x_0|}{\ln m} \approx 73 \quad \text{#iterations needed}$$

In actual simulation $k = 25$ is enough.

Newton's method

Goal: Given $f(x)$, find a root r s.t. $f(r) = 0$.

Main idea: Given x_k , the next approximation x_{k+1} is determined by approximating $f(x)$ as a linear function at x_k .



We have

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

which gives

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

Newton's method is a fixed point iteration.

Newton's method is a fixed point iteration

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

If r is a fixed point, assume $f'(r) \neq 0$, then

$$r = g(r), \quad r = r - \frac{f(r)}{f'(r)}, \quad \frac{f(r)}{f'(r)} = 0, \quad f(r) = 0.$$

then r is a root for f .

Newton's method is the “best” fixed point iteration.

Observation: A fixed point iteration $x = g(x)$ is optimal if $g'(r) = 0$ where $r = g(r)$.

For Newton, we have

$$g'(x) = 1 - \frac{f'(x)f'(x) - f''(x)f(x)}{(f'(x))^2} = \frac{f''(x)f(x)}{(f'(x))^2}$$

Then

$$g'(r) = \frac{f''(r)f(r)}{(f'(r))^2} = 0$$

which is the “best” possible scenario!

Newton's method; Convergence analysis.

Let r be the root so $f(r) = 0$ and $r = g(r)$. Recall $g'(r) = 0$.

Define Error: $e_{k+1} = |x_{k+1} - r| = |g(x_k) - g(r)|$

Taylor expansion for $g(x_k)$ at r :

$$\begin{aligned}g(x_k) &= g(r) + (x_k - r)g'(r) + \frac{1}{2}(x_k - r)^2g''(\xi), \quad \xi \in (x_k, r) \\&= g(r) + \frac{1}{2}(x_k - r)^2g''(\xi)\end{aligned}$$

$$e_{k+1} = \frac{1}{2}(x_k - r)^2 |g''(\xi)| = \frac{1}{2}e_k^2 |g''(\xi)|$$

Write now $M = \frac{1}{2} \max_x |g''(\xi)|$, we have

$$e_{k+1} \leq M(e_k)^2$$

This is called *quadratic convergence*.

Theorem: If $e_{k+1} \leq M e_k^2$, then $\lim_{k \rightarrow +\infty} e_k = 0$ if e_0 is sufficiently small.
(This means, M can be big, but it would not effect the convergence!)

Proof for the convergence: We have

$$e_1 \leq (M e_0) e_0$$

If e_0 is small enough, such that $(M e_0) < 1$, then $e_1 < e_0$.

This means $(M e_1) < M e_0 < 1$, and so

$$e_2 \leq (M e_1) e_1 < e_1, \quad \Rightarrow \quad M e_2 < M e_1 < 1$$

By an induction argument, we conclude that $e_{k+1} < e_k$ for all k , i.e., error is strictly decreasing after each iteration. \Rightarrow convergence.

Newton's iterations; Examples

Example. Find a numerical method to compute \sqrt{a} using only $+, -, *, /$ arithmetic operations. Test it for $a = 3$.

Answer. It's easy to see that \sqrt{a} is a root for $f(x) = x^2 - a$.
Newton's method gives

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = x_k - \frac{x_k^2 - a}{2x_k} = \frac{x_k}{2} + \frac{a}{2x_k}$$

Test it on $a = 3$: Choose $x_0 = 1.7$.

	error
$x_0 = 1.7$	7.2×10^{-2}
$x_1 = 1.7324$	3.0×10^{-4}
$x_2 = 1.7321$	2.6×10^{-8}
$x_3 = 1.7321$	4.4×10^{-16}

Note the extremely fast convergence.

Sample Code:

```
r=newton('f','df',x0,nmax,tol)
x=x0;
n=0;
dx=f(x)/df(x);
while ((dx>tol) and (f(x)>tol)) or (n<nmax) do
    n=n+1;
    x=x-dx;
    dx=f(x)/df(x);
end
r=x-dx;
```

Secant method

If $f(x)$ is complicated, $f'(x)$ might not be available.

Solution for this situation: using approximation for f' , i.e.,

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

This is *secant method*:

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

Initial guess needed: x_0, x_1 .

Advantages include

- No computation of f' ;
- One $f(x)$ computation each step;
- Also rapid convergence.

A bit on convergence: One can show that

$$e_{k+1} \leq Ce_k^\alpha, \quad \alpha = \frac{1}{2}(1 + \sqrt{5}) \approx 1.62$$

This is called *super linear convergence*. ($1 < \alpha < 2$)

It converges for all function f if x_0 and x_1 are close to the root r .

Example Use secant method for computing \sqrt{a} .

Answer. The iteration now becomes

$$x_{k+1} = x_k - \frac{(x_k^2 - a)(x_k - x_{k-1})}{(x_k^2 - a) - (x_{k-1}^2 - a)} = x_k - \frac{x_k^2 - a}{x_k + x_{k-1}}$$

Test with $a = 3$, with initial data $x_0 = 1.65$, $x_1 = 1.7$.

	error
$x_1 = 1.7$	7.2×10^{-2}
$x_2 = 1.7328$	7.9×10^{-4}
$x_3 = 1.7320$	7.3×10^{-6}
$x_4 = 1.7321$	1.7×10^{-9}
$x_5 = 1.7321$	3.6×10^{-15}

It is a little but slower than Newton's method, but not much.

The most practical algorithm: hybrid methods

- Sample the function to find a and b such that $f(a) \cdot f(b) < 0$.
- Use bisection's method, maybe 5-6 iterations, to get some good initial guess x_0 ;
- Use either Newton or secant method, with this x_0 , for 3-4 iterations.

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Direct methods for systems of linear equations

The problem: n equations, n unknowns,

$$\left\{ \begin{array}{lcl} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = & b_2 \\ & \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = & b_n \end{array} \right. \quad \begin{array}{l} (1) \\ (2) \\ \vdots \\ (n) \end{array}$$

In matrix-vector form:

$$A\vec{x} = \vec{b},$$

where $A \in \mathbb{R}^{n \times n}$, $\vec{x} \in \mathbb{R}^n$, $\vec{b} \in \mathbb{R}^n$

$$A = \{a_{ij}\} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

Naive Gaussian elimination

Step 1: Forward elimination.

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \right. \quad \begin{array}{l} (1) \\ (2) \\ \vdots \\ (n) \end{array}$$

Make an upper triangular system!

Algorithm:

```
for  $k = 1, 2, 3, \dots, n - 1$ 
    for  $j = k + 1, k + 2, \dots, n$ 
         $(j) \leftarrow (j) - (k) \times \frac{a_{jk}}{a_{kk}},$ 
    end
end
```

Work count: #flops = $\frac{1}{3}(n^3 - n) = \mathcal{O}(n^3)$

Step 2: Backward substitution

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{nn}x_n = b_n \end{array} \right. \quad \begin{array}{l} (1) \\ (2) \\ \vdots \\ (n) \end{array}$$

Algorithm:

$$x_n = \frac{b_n}{a_{nn}}$$

for $i = n - 1, n - 2, \dots, 1$

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^n a_{ij}x_j \right)$$

end

Work count: #flops = $\frac{1}{2}(n^2 - n) = \mathcal{O}(n^2)$

Total work count: $= \mathcal{O}(n^3)$. Extremely slow.

Tridiagonal system

$$A = \begin{pmatrix} d_1 & c_1 & 0 & \cdots & 0 & 0 & 0 \\ a_1 & d_2 & c_2 & \cdots & 0 & 0 & 0 \\ 0 & a_2 & d_3 & \ddots & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & d_{n-2} & c_{n-2} & 0 \\ 0 & 0 & 0 & \cdots & a_{n-2} & d_{n-1} & c_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & a_{n-1} & d_n \end{pmatrix}$$

Gaussian Elimination can be very efficiently:

Step 1: Forward Elimination:

for $i = 2, 3, \dots, n$

$$d_i \leftarrow d_i - \frac{a_{i-1}}{d_{i-1}} c_{i-1}$$

$$b_i \leftarrow b_i - \frac{a_{i-1}}{d_{i-1}} b_{i-1}$$

end

Now the A matrix looks like

$$A = \begin{pmatrix} d_1 & c_1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & d_2 & c_2 & \cdots & 0 & 0 & 0 \\ 0 & 0 & d_3 & \ddots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & 0 & 0 & \cdots & d_{n-2} & c_{n-2} & 0 \\ 0 & 0 & 0 & \cdots & 0 & d_{n-1} & c_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & 0 & d_n \end{pmatrix}$$

Step 2: Backward substitution:

$$x_n \leftarrow b_n / d_n$$

for $i = n - 1, n - 2, \dots, 1$

$$x_i \leftarrow \frac{1}{d_i}(b_i - c_i x_{i+1})$$

end

Amount of work: $\mathcal{O}(n)$. Very efficient!

Review of linear algebra

Consider a square matrix $A = \{a_{ij}\}$. A is called *strictly diagonal dominant* if

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|, \quad i = 1, 2, \dots, n$$

Properties:

- A is regular, invertible, A^{-1} exists, and $Ax = b$ has a unique solution.
- $Ax = b$ can be solved by Gaussian Elimination without pivoting.

One such example: the system from natural cubic spline.

Vector and matrix norms

A norm: measures the “size” of the vector and matrix.

General norm properties: Denote $\|x\|$ the norm of x . Then

- ① $\|x\| \geq 0$, equal if and only if $x = 0$;
- ② $\|ax\| = |a| \cdot \|x\|$, a : is a constant;
- ③ $\|x + y\| \leq \|x\| + \|y\|$, triangle inequality.

Examples of vector norms: $x \in \mathbb{R}^n$

① $\|x\|_1 = \sum_{i=1}^n |x_i|,$ l_1 -norm

② $\|x\|_2 = \left(\sum_{i=1}^n x_i^2 \right)^{1/2},$ l_2 -norm

③ $\|x\|_\infty = \max_{1 \leq i \leq n} |x_i|,$ l_∞ -norm

Matrix norms

Matrix norm is defined in term of the corresponding vector norm:

$$\|A\| = \max_{\vec{x} \neq 0} \frac{\|Ax\|}{\|x\|}$$

Properties:

$$\|A\| \geq \frac{\|Ax\|}{\|x\|} \quad \Rightarrow \quad \|Ax\| \leq \|A\| \cdot \|x\|$$

$$\|I\| = 1, \quad \|AB\| \leq \|A\| \cdot \|B\|.$$

Examples of matrix norms:

$$l_1 - \text{norm} : \|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^n |a_{ij}|$$

$$l_2 - \text{norm} : \|A\|_2 = \max_i |\lambda_i|, \quad \lambda_i : \text{eigenvalues of } A$$

$$l_\infty - \text{norm} : \|A\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$$

Eigenvalues λ_i for A

$$Av = \lambda v, \quad \lambda : \text{ eigenvalue}, \quad v : \text{ eigenvector}$$

$$(A - \lambda I)v = 0, \quad \Rightarrow \quad \det(A - \lambda I) = 0 : \text{ polynomial of degree } n$$

Property:

$$\lambda_i(A^{-1}) = \frac{1}{\lambda_i(A)}$$

This implies

$$\|A^{-1}\|_2 = \max_i |\lambda_i(A^{-1})| = \max_i \frac{1}{|\lambda_i(A)|} = \frac{1}{\min_i |\lambda_i(A)|}$$

Condition number of a matrix A

Want to solve: $Ax = b$

Put some perturbation: $A\bar{x} = b + p$

Relative errors:

$$e_b = \frac{\|p\|}{\|b\|}, \quad e_x = \frac{\|\bar{x} - x\|}{\|x\|}$$

relation between them?

We have

$$A(\bar{x} - x) = p, \quad \Rightarrow \quad \bar{x} - x = A^{-1}p$$

so

$$e_x = \frac{\|\bar{x} - x\|}{\|x\|} = \frac{\|A^{-1}p\|}{\|x\|} \leq \frac{\|A^{-1}\| \cdot \|p\|}{\|x\|}.$$

$$Ax = b \quad \Rightarrow \quad \|Ax\| = \|b\| \quad \Rightarrow \quad \|A\| \|x\| \geq \|b\| \quad \Rightarrow \quad \frac{1}{\|x\|} \leq \frac{\|A\|}{\|b\|}$$

we get

$$e_x \leq \frac{\|A^{-1}\| \cdot \|p\|}{\|x\|} \leq \|A^{-1}\| \cdot \|p\| \cdot \frac{\|A\|}{\|b\|} = \|A\| \cdot \|A^{-1}\| e_b = \kappa(A) \cdot e_b,$$

$$\kappa(A) = \|A\| \cdot \|A^{-1}\| : \quad \text{the condition number of } A$$

$$\text{Using } l_2\text{-norm: } \kappa(A) = \|A\|_2 \cdot \|A^{-1}\|_2 = \frac{\max_i |\lambda_i(A)|}{\min_i |\lambda_i(A)|}$$

Error in b propagates with a factor of $\kappa(A)$ into the solution.

If $\kappa(A)$ is very large, $Ax = b$ is very sensitive to perturbation, therefore difficult to solve. We call this *ill-conditioned system*.

Some Matlab commands:

```
norm(x);          % vector norm  
eig(A);          % eigenvalue/eigen vector of a matrix  
cond(A);          % condition number of A
```

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Fixed point iterative solvers for linear systems

Problem: Find approximate solution to $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ has properties:

- Large, n is very big, for example $n = \mathcal{O}(10^6)$.
- A is sparse, with a large percent of 0 entries.
- A is structured. (meaning: the product Ax can be computed efficiently)

Idea: Avoiding computing A^{-1} . Perform the “cheap” operation Ax .

Jacobi iterations

Want to solve:

$$\left\{ \begin{array}{lcl} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = & b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = & b_n \end{array} \right.$$

Rewrite it:

$$\left\{ \begin{array}{lcl} x_1 & = & \frac{1}{a_{11}}(b_1 - a_{12}x_2 - \cdots - a_{1n}x_n) \\ x_2 & = & \frac{1}{a_{22}}(b_2 - a_{21}x_1 - \cdots - a_{2n}x_n) \\ \vdots \\ x_n & = & \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots 0) \end{array} \right.$$

In a compact form: $x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^n a_{ij}x_j \right), \quad i = 1, 2, \dots, n$

This gives the Jacobi iterations:

- Choose a start point, $x^0 = (x_1^0, x_2^0, \dots, x_n^0)^t$.
- for $k = 0, 1, 2, \dots$ until stop criteria

for $i = 1, 2, \dots, n$

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^n a_{ij} x_j^k \right)$$

end

end

Choices of starting vector x^0 : Anything goes.

- A vector with all entries 1: $x_i^0 = 1$ for all i , or $x_i = b_i/a_{ii}$.
- The load vector: $x^0 = b$;
- Best choice: $x_i = b_i/a_{ii}$, for $i = 1, 2, \dots, n$.

Stop Criteria could be any combinations of the following

- $\|x^k - x^{k-1}\| \leq \varepsilon$ for certain vector norms.
- Residual $r^k = Ax^k - b$ is small, i.e., $\|r^k\| \leq \varepsilon$.
- Max number of iteration reached.

About the algorithm:

- Must make 2 vectors for the computation, x^k and x^{k+1} .
- Non-sequential. Great for parallel computing.

Jacobi iterations; example

Example 1. Solve the following system with Jacobi iterations.

$$\begin{cases} 2x_1 - x_2 = 0 \\ -x_1 + 2x_2 - x_3 = 1 \\ \quad -x_2 + 2x_3 = 2 \end{cases}$$

Given the exact solution $x = (1, 2, 2)^t$.

Answer. Choose x^0 by $x_i^0 = b_i/a_{ii}$:

$$x^0 = (0, 1/2, 1)^t$$

The iteration is

$$\begin{cases} x_1^{k+1} = \frac{1}{2}x_2^k \\ x_2^{k+1} = \frac{1}{2}(1 + x_1^k + x_3^k) \\ x_3^{k+1} = \frac{1}{2}(2 + x_2^k) \end{cases}$$

We run a couple of iterations, and get

$$x^1 = (0.25, 1, 1.25)^t$$

$$x^2 = (0.5, 1.25, 1.5)^t$$

$$x^3 = (0.625, 1.5, 1.625)^t$$

Observations:

- Looks like it is converging. Need to run more steps to be sure.
- Rather slow convergence rate.

Gauss-Seidal iterations

Recall Jacobi iteration

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^n a_{ij} x_j^k \right) = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^k - \sum_{j=i+1}^n a_{ij} x_j^k \right)$$

If the computation is done in a sequential way, for $i = 1, 2, \dots$, then in the first summation term, all x_j^k are already computed for step $k + 1$. We will replace all these x_j^k with x_j^{k+1} .

Gauss-Seidal iterations

Use the latest computed values of x_i .

for $k = 0, 1, 2, \dots$, until stop criteria

 for $i = 1, 2, \dots, n$

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^n a_{ij}x_j^k \right)$$

 end

end

About the algorithm:

- Need only one vector for both x^k and x^{k+1} , saves memory space.
- Not good for parallel computing.

Example 2. Try it on the same Example 1, with $x^0 = (0, 0.5, 1)^t$.
The iteration now is:

$$\begin{cases} x_1^{k+1} = \frac{1}{2}x_2^k \\ x_2^{k+1} = \frac{1}{2}(1 + x_1^{k+1} + x_3^k) \\ x_3^{k+1} = \frac{1}{2}(2 + x_2^{k+1}) \end{cases}$$

We run a couple of iterations:

$$\begin{aligned} x^1 &= (0.25, 1.125, 1.5625)^t \\ x^2 &= (0.5625, 1.5625, 1.7813)^t \end{aligned}$$

Observation: Converges a bit faster than Jacobi iterations.

SOR (Successive Over Relaxation)

SOR is a more general iterative method.

A version based on Gauss-Seidal.

$$x_i^{k+1} = (1 - w)x_i^k + w \cdot \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^n a_{ij}x_j^k \right)$$

Note the second term is the Gauss-Seidal iteration multiplied with w .

w : relaxation parameter.

Usual value: $0 < w < 2$ (for convergence reason)

- $w = 1$: Gauss-Seidal
- $0 < w < 1$: under relaxation
- $1 < w < 2$: over relaxation

Example 3. Try this on the same example with $w = 1.2$. General iteration is now:

$$\begin{cases} x_1^{k+1} &= -0.2x_1^k + 0.6x_2^k \\ x_2^{k+1} &= -0.2x_2^k + 0.6 * (1 + x_1^{k+1} + x_3^k) \\ x_3^{k+1} &= -0.2x_3^k + 0.6 * (2 + x_2^{k+1}) \end{cases}$$

With $x^0 = (0, 0.5, 1)^t$, we get

$$x^1 = (0.3, 1.28, 1.708)^t$$

$$x_2 = (0.708, 1.8290, 1.9442)^t$$

Observation: faster convergence than both Jacobi and G-S.

Writing all methods in standard form

Want to solve $Ax = b$. We change it into a fixed-point problem

$x = Mx + y$ for some matrix M and vector y , with fixed point iteration
 $x^{k+1} = Mx^k + y$.

Splitting of the matrix A :

$$A = L + D + U$$

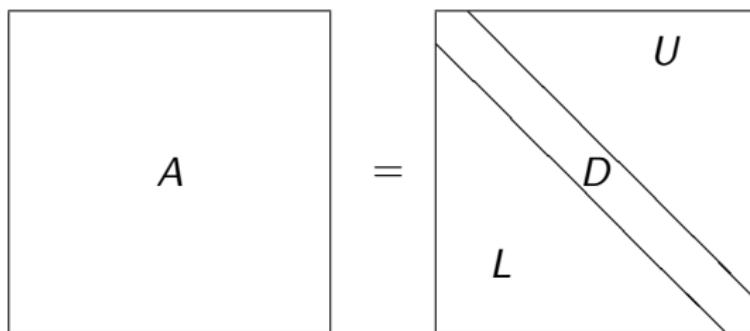


Figure: Splitting of A .

Now we have

$$Ax = (L + D + U)x = Lx + Dx + Ux = b$$

Jacobi iterations:

$$Dx^{k+1} = b - Lx^k - Ux^k$$

so

$$x^{k+1} = D^{-1}b - D^{-1}(L + U)x^k = y_J + M_Jx^k$$

where

$$y_J = D^{-1}b, \quad M_J = -D^{-1}(L + U).$$

Gauss-Seidal:

$$Dx^{k+1} + Lx^{k+1} = b - Ux^k$$

so

$$x^{k+1} = (D + L)^{-1}b - (D + L)^{-1}Ux^k = y_{GS} + M_{GS}x^k$$

where

$$y_{GS} = (D + L)^{-1}b, \quad M_{GS} = -(D + L)^{-1}U.$$

SOR:

$$x^{k+1} = (1 - w)x^k + wD^{-1}(b - Lx^{k+1} - Ux^k)$$

$$\Rightarrow Dx^{k+1} = (1 - w)Dx^k + wb - wLx^{k+1} - wUx^k$$

$$\Rightarrow (D + wL)x^{k+1} = wb + [(1 - w)D - wU]x^k$$

so

$$x^{k+1} = (D + wL)^{-1}b + (D + wL)^{-1}[(1 - w)D - wU]x^k = y_{SOR} + M_{SOR}x^k$$

where

$$y_{SOR} = (D + wL)^{-1}b, \quad M_{SOR} = (D + wL)^{-1}[(1 - w)D - wU].$$

Analysis for errors and convergence

Iteration $x^{k+1} = y + Mx^k$ for solving $Ax = b$

Assume s is the solution: $As = b$, $s = y + Ms$.

Define the error vector: $e^k = x^k - s$

$$e^{k+1} = x^{k+1} - s = y + Mx^k - (y + Ms) = M(x^k - s) = Me^k.$$

This gives the propagation of error:

$$e^{k+1} = M e^k.$$

Take the norm on both sides:

$$\|e^{k+1}\| = \|Me^k\| \leq \|M\| \cdot \|e^k\|$$

This implies:

$$\|e^k\| \leq \|M\|^k \|e^0\|, \quad e^0 = x^0 - s.$$

Theorem *If $\|M\| < 1$ for some norm $\|\cdot\|$, then the iterations converge in that norm.*

NB! Convergence only depends on the iteration matrix M .

Check our methods: $A = D + L + U$.

- Jacobi: $M_J = -D^{-1}(L + U)$. Determined by A ;
- G-S: $M_{GS} = -(D + L)^{-1}U$. Determined by A ;
- SOR: $M_{SOR} = (D + wL)^{-1}[(1 - w)D - wU]$.
One can adjust w to get a smallest possible $\|M\|$.
More flexible.

Check the same example we have been using. We have

$$A = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix},$$

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

The iteration matrix for each method:

$$M_J = \begin{pmatrix} 0 & 0.5 & 0 \\ 0.5 & 0 & 0.5 \\ 0 & 0.5 & 0 \end{pmatrix}, \quad M_{GS} = \begin{pmatrix} 0 & 0.5 & 0 \\ 0 & 0.25 & 0.5 \\ 0 & 0.125 & 0.25 \end{pmatrix},$$

$$M_{SOR} = \begin{pmatrix} -0.2 & 0.6 & 0 \\ -0.12 & 0.16 & 0.6 \\ -0.072 & 0.096 & 0.16 \end{pmatrix}$$

We list their various norms:

M	l_1 norm	l_2 norm	l_∞ norm
Jacobi	1	0.707	1
G-S	0.875	0.5	0.75
SOR	0.856	0.2	0.88

The l_2 norm is the most significant one. We see now why SOR converges fastest.

Convergence Theorem. If A is diagonal dominant, i.e.,

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|, \quad \text{for every } i = 1, 2, \dots, n.$$

Then, all three iteration methods converge for all initial choice of x^0 .

NB! If A is not diagonal dominant, it might still converge, but there is no guarantee.

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Iterative methods for systems of linear equations in Matlab

Example: We wish to solve a system: $Ax = b$ where A is a 6×6 matrix

$A =$

$$\begin{matrix} 4 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & 0 & -1 & 0 & 0 \\ -1 & 0 & 4 & -1 & -1 & 0 \\ 0 & -1 & -1 & 4 & 0 & -1 \\ 0 & 0 & -1 & 0 & 4 & -1 \\ 0 & 0 & 0 & -1 & -1 & 4 \end{matrix}$$

and the rhs vector is: $b = [1; 5; 0; 3; 1; 5]$.

The exact solution becomes: $x = [1; 2; 1; 2; 1; 2]$.

We solve the system with iterative methods, with the initial value:

$$x^{(0)} = [0.25; 1.25; 0; 0.75; 0.25; 1.25].$$

Jacobi iterations: tolerance $\varepsilon = 10^{-5}$, with $\|\cdot\|_\infty$

k	x1	x2	x3	x4	x5	x6
1	0.2500	1.2500	0	0.7500	0.2500	1.2500
2	0.5625	1.5000	0.3125	1.3750	0.5625	1.5000
3	0.7031	1.7344	0.6250	1.5781	0.7031	1.7344
4	0.8398	1.8203	0.7461	1.7734	0.8398	1.8203
5	0.8916	1.9033	0.8633	1.8467	0.8916	1.9033
6	0.9417	1.9346	0.9075	1.9175	0.9417	1.9346
7	0.9605	1.9648	0.9502	1.9442	0.9605	1.9648
8	0.9787	1.9762	0.9663	1.9699	0.9787	1.9762
9	0.9856	1.9872	0.9819	1.9797	0.9856	1.9872
10	0.9923	1.9913	0.9877	1.9890	0.9923	1.9913
...						
18	0.9999	1.9998	0.9998	1.9998	0.9999	1.9998
19	0.9999	1.9999	0.9999	1.9999	0.9999	1.9999
20	1.0000	1.9999	0.9999	1.9999	1.0000	1.9999

One needs more iterations to get the convergence.

Gauss-Seidal iterations:

k	x1	x2	x3	x4	x5	x6
1	0.2500	1.2500	0	0.7500	0.2500	1.2500
2	0.5625	1.5781	0.3906	1.5547	0.6602	1.8037
3	0.7422	1.8242	0.7393	1.8418	0.8857	1.9319
4	0.8909	1.9332	0.9046	1.9424	0.9591	1.9754
5	0.9594	1.9755	0.9652	1.9790	0.9852	1.9910
6	0.9852	1.9911	0.9873	1.9924	0.9946	1.9967
7	0.9946	1.9967	0.9954	1.9972	0.9980	1.9988
8	0.9980	1.9988	0.9983	1.9990	0.9993	1.9996
9	0.9993	1.9996	0.9994	1.9996	0.9997	1.9998
10	0.9997	1.9998	0.9998	1.9999	0.9999	1.9999
11	0.9999	1.9999	0.9999	2.0000	1.0000	2.0000
12	1.0000	2.0000	1.0000	2.0000	1.0000	2.0000
=====						
13	1.0000	2.0000	1.0000	2.0000	1.0000	2.0000

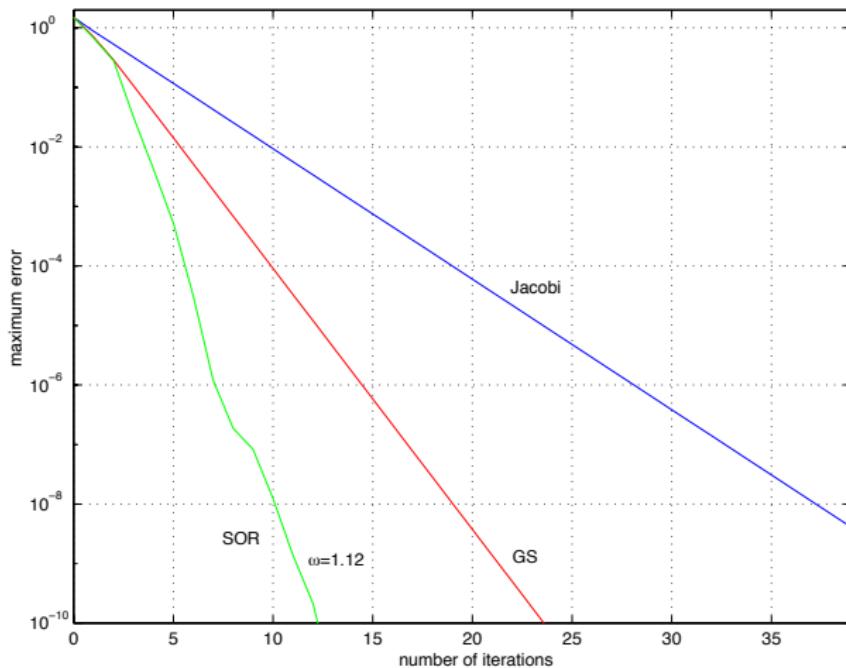
We see that after 12 iterations the method converges.

SOR iterations with $\omega = 1.12$:

k	x1	x2	x3	x4	x5	x6
1	0.2500	1.2500	0	0.7500	0.2500	1.2500
2	0.6000	1.6280	0.4480	1.6813	0.7254	1.9239
3	0.7893	1.8964	0.8411	1.9434	0.9671	1.9841
4	0.9518	1.9831	0.9805	1.9922	0.9940	1.9980
5	0.9956	1.9986	0.9972	1.9992	0.9994	1.9998
6	0.9994	1.9998	0.9998	1.9999	1.0000	2.0000
7	0.9999	2.0000	1.0000	2.0000	1.0000	2.0000
8	1.0000	2.0000	1.0000	2.0000	1.0000	2.0000
=====						
9	1.0000	2.0000	1.0000	2.0000	1.0000	2.0000

We see that after 8 iterations the method converges.

Error against number of iterations



CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

The Method of Least Squares

Problem description

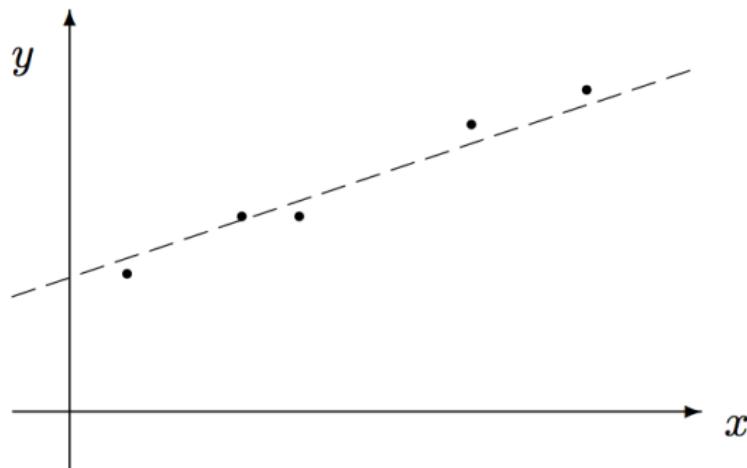
Given data set

x	x_0	x_1	x_2	\cdots	x_m
y	y_0	y_1	y_2	\cdots	y_M

Want to fit in a function $y = f(x)$
such that the error $e_k = f(x_k) - y_k$ is minimized.

Linear regression and basic derivation

We want to fit in $y(x) = ax + b$.



Problem:

Find a, b , such that when we use $y = ax + b$, the “error” becomes smallest possible.

How to measure error?

- ① $\max_k |y(x_k) - y_k| — l_\infty \text{ norm}$
- ② $\sum_{k=0}^m |y(x_k) - y_k| — l_1 \text{ norm}$
- ③ $\sum_{k=0}^m [y(x_k) - y_k]^2 — l_2 \text{ norm, used in Least Square Method. (LSM)}$

A minimization problem:

Find a and b such that the error function $\psi(a, b)$ defined as

$$\psi(a, b) = \sum_{k=0}^m (ax_k + b - y_k)^2$$

is minimized.

How to find a and b ?

At the minimum of a function, we have

$$\frac{\partial \psi}{\partial a} = \frac{\partial \psi}{\partial b} = 0$$

error function: $\psi(a, b) = \sum_{k=0}^m (ax_k + b - y_k)^2$

$$\frac{\partial \psi}{\partial a} = 0 : \sum_{k=0}^m 2(ax_k + b - y_k)x_k = 0, \quad (I)$$

$$\frac{\partial \psi}{\partial b} = 0 : \sum_{k=0}^m 2(ax_k + b - y_k) = 0, \quad (II)$$

Solve (I), (II) for (a, b) . Rewrite it as a system

$$\begin{cases} \left(\sum_{k=0}^m x_k^2 \right) \cdot a + \left(\sum_{k=0}^m x_k \right) \cdot b &= \sum_{k=0}^m x_k \cdot y_k \\ \left(\sum_{k=0}^m x_k \right) \cdot a + (m+1) \cdot b &= \sum_{k=0}^m y_k \end{cases}$$

These are called the normal equations.

Example 1. Data set for S (liquid surface tension) and T (temperature)

T_k	0	10	20	30	40	80	90	95
S_k	68.0	67.1	66.4	65.6	64.6	61.8	61.0	60.0

From physics we know that they have a linear relation $S = aT + b$. Use MLS to find the best fitting a, b .

Answer. We have $m = 7$, and the sums:

$$\sum_{k=0}^7 T_k^2 = 26525, \quad \sum_{k=0}^7 T_k = 365, \quad \sum_{k=0}^7 T_k S_k = 22685, \quad \sum_{k=0}^7 S_k = 514.5.$$

The normal equations:
$$\begin{cases} 26525 a + 365 b &= 22685 \\ 365 a + 8 b &= 514.5 \end{cases}$$

Solve it: $a = -0.079930$, $b = 67.9593$.

Linear LSM with 3 functions

Given data set (x_k, y_k) for $k = 0, 1, \dots, m$. Find a function

$$y(x) = a \cdot f(x) + b \cdot g(x) + c \cdot h(x)$$

which best fit the data.

This means we need to find (a, b, c) .

Here $f(x), g(x), h(x)$ are given functions, for example

$$f(x) = e^x, \quad g(x) = \ln(x), \quad h(x) = \cos x,$$

but not restricted to these.

Define the error function:

$$\psi(a, b, c) = \sum_{k=0}^m (y(x_k) - y_k)^2 = \sum_{k=0}^m (a \cdot f(x_k) + b \cdot g(x_k) + c \cdot h(x_k) - y_k)^2$$

At minimum, we have

$$\frac{\partial \psi}{\partial a} = 0 \quad : \quad \sum_{k=0}^m 2 \left[a \cdot f(x_k) + b \cdot g(x_k) + c \cdot h(x_k) - y_k \right] \cdot f(x_k) = 0$$

$$\frac{\partial \psi}{\partial b} = 0 \quad : \quad \sum_{k=0}^m 2 \left[a \cdot f(x_k) + b \cdot g(x_k) + c \cdot h(x_k) - y_k \right] \cdot g(x_k) = 0$$

$$\frac{\partial \psi}{\partial c} = 0 \quad : \quad \sum_{k=0}^m 2 \left[a \cdot f(x_k) + b \cdot g(x_k) + c \cdot h(x_k) - y_k \right] \cdot h(x_k) = 0$$

The normal equations are

$$\left(\sum_{k=0}^m f(x_k)^2 \right) a + \left(\sum_{k=0}^m f(x_k)g(x_k) \right) b + \left(\sum_{k=0}^m f(x_k)h(x_k) \right) c = \sum_{k=0}^m f(x_k)y_k$$

$$\left(\sum_{k=0}^m f(x_k)g(x_k) \right) a + \left(\sum_{k=0}^m g(x_k)^2 \right) b + \left(\sum_{k=0}^m h(x_k)g(x_k) \right) c = \sum_{k=0}^m g(x_k)y_k$$

$$\left(\sum_{k=0}^m f(x_k)h(x_k) \right) a + \left(\sum_{k=0}^m g(x_k)h(x_k) \right) b + \left(\sum_{k=0}^m h(x_k)^2 \right) c = \sum_{k=0}^m h(x_k)y_k$$

We note that the system of normal equations is always symmetric. We only need to compute half of the entries.

General linear LSM

Given data set (x_k, y_k) , $k = 0, 1, \dots, m$.

Let $g_0, g_1, g_2, \dots, g_n$ be $n + 1$ given functions, linearly independent.

We search for a function in the form

$$y(x) = \sum_{i=0}^n c_i g_i(x)$$

that best fit the data.

Here g_i 's are called *basis functions*.

Define error function

$$\psi(c_0, c_1, \dots, c_n) = \sum_{k=0}^m \left[y(x_k) - y_k \right]^2 = \sum_{k=0}^m \left[\left(\sum_{i=0}^n c_i g_i(x_k) \right) - y_k \right]^2$$

At minimum, we have

$$\frac{\partial \psi}{\partial c_j} = 0, \quad j = 0, 1, \dots, n.$$

This gives:

$$\sum_{k=0}^m 2 \left[\left(\sum_{i=0}^n c_i g_i(x_k) \right) - y_k \right] g_j(x_k) = 0$$

Re-arranging the ordering of summation signs:

$$\sum_{i=0}^n \left(\sum_{k=0}^m g_i(x_k) g_j(x_k) \right) c_i = \sum_{k=0}^m g_j(x_k) y_k, \quad j = 0, 1, \dots, n.$$

$$\sum_{i=0}^n \left(\sum_{k=0}^m g_i(x_k) g_j(x_k) \right) c_i = \sum_{k=0}^m g_j(x_k) y_k, \quad j = 0, 1, \dots, n.$$

This gives the system of normal equations:

$$A\vec{c} = \vec{b}$$

where $\vec{c} = (c_0, c_1, \dots, c_n)^t$ and

$$A = \{a_{ij}\}, \quad a_{ij} = \sum_{k=0}^m g_i(x_k) g_j(x_k)$$
$$\vec{b} = \{b_j\}, \quad b_j = \sum_{k=0}^m g_j(x_k) y_k.$$

We note that this A is symmetric, $a_{ij} = a_{ji}$.

Non-linear LSM: quasi-linear

Consider fitting the data set (x_k, y_k) with the function $y(x) = a \cdot b^x$. This means, we need to find (a, b) such that this $y(x)$ best fit the data.

$$\text{variable change: } \ln y = \ln a + x \cdot \ln b.$$

Let

$$S = \ln y, \quad \bar{a} = \ln a, \quad \bar{b} = \ln b, \quad \rightarrow \quad S(x) = \bar{a} + \bar{b}x.$$

Generate data (x_k, S_k) where $S_k = \ln y_k$ for all k .

Use linear LSM and find (\bar{a}, \bar{b}) such that $S(x)$ best fits (x_k, S_k) .

Then, transform back to the original variable

$$a = \exp\{\bar{a}\}, \quad b = \exp\{\bar{b}\}.$$

Non-linear LSM

For the data (x_k, y_k) , fit in the function $y(x) = ax \cdot \sin(bx)$.

No variable change that can change this problem into a linear one.

The function arises in the solution of 2nd order ODE with resonance.

Define error

$$\psi(a, b) = \sum_{k=0}^m [y(x_k) - y_k]^2 = \sum_{k=0}^m [ax_k \cdot \sin(bx_k) - y_k]^2.$$

At minimum:

$$\frac{\partial \psi}{\partial a} = 0 : \quad \sum_{k=0}^m 2 \left[ax_k \cdot \sin(bx_k) - y_k \right] \cdot [x_k \cdot \sin(bx_k)] = 0$$

$$\frac{\partial \psi}{\partial b} = 0 : \quad \sum_{k=0}^m 2 \left[ax_k \cdot \sin(bx_k) - y_k \right] \cdot [ax_k \cdot \cos(bx_k)x_k] = 0$$

→ 2×2 system of non-linear equations to solve for (a, b) !

May use Newton's method to find a root. May not have unique solution.

Least square for continuous functions

Problem. Given: function $f(x)$ on $x \in [a, b]$,
and a set of basis functions g_i ($i = 1, 2, \dots, n$) on $x \in [a, b]$.

Find a function:
$$g(x) = \sum_{i=1}^n a_i g_i(x)$$

to minimize the error:
$$E(f, g) = \|f - g\|_2^2 = \int_a^b (f(x) - g(x))^2 dx$$

This means:
$$E(a_1, a_2, \dots, a_n) = \int_a^b \left(f(x) - \sum_{i=1}^n a_i g_i(x) \right)^2 dx$$

At the minimum, we must have

$$\frac{\partial E}{\partial a_i} = 0, \quad i = 1, 2, \dots, n$$

$$E(a_1, a_2, \dots, a_n) = \int_a^b \left(f(x) - \sum_{i=1}^n a_i g_i(x) \right)^2 dx.$$

$$\begin{aligned}\frac{\partial E}{\partial a_i} &= -2 \int_a^b g_i(x) \left(f(x) - \sum_{j=1}^n a_j g_j(x) \right) dx = 0 \\ &\int_a^b g_i(x) f(x) dx - \int_a^b g_i(x) \sum_{j=1}^n a_j g_j(x) dx = 0\end{aligned}$$

we have

$$\sum_{j=1}^n a_j \int_a^b g_i(x) g_j(x) dx = \int_a^b g_i(x) f(x) dx, \quad i = 1, 2, \dots, n$$

$$\sum_{j=1}^n a_j \int_a^b g_i(x)g_j(x) dx = \int_a^b g_i(x)f(x) dx, \quad i = 1, 2, \dots, n$$

Writing this into matrix-vector form:

$$C\vec{a} = \vec{b}$$

where

$$C = \{c_{ij}\}, \quad c_{ij} = \int_a^b g_i(x)g_j(x) dx, \quad b_i = \int_a^b g_i(x)f(x) dx.$$

Note that the matrix C is symmetric, since $c_{ij} = c_{ji}$.

If the basis functions $\{g_i\}$ are linearly independent, then the matrix C is non-singular, and the system $C\vec{a} = \vec{b}$ has a unique solution.

Orthogonal basis.

If the basis function $g_i(x)$ are orthogonal to each other, i.e., if $i \neq j$, then

$$\int_a^b g_i(x)g_j(x) dx = 0, \quad \rightarrow \quad c_{ij} = 0$$

then the C matrix is diagonal, and the system is trivial to solve.

$$c_{ii} = \int_a^b (g_i(x))^2 dx, \quad a_i = \frac{b_i}{a_{ii}} = \frac{\int_a^b g_i(x)f(x) dx}{\int_a^b (g_i(x))^2 dx}.$$

Examples of families of orthogonal functions

Legendre polynomials

For interval $[-1, 1]$, they are

$$P_0(x) = 1,$$

$$P_1(x) = x,$$

$$P_2(x) = (3x^2 - 1)/2,$$

$$P_3(x) = (5x^3 - 3x)/2,$$

$$P_4(x) = (35x^4 - 30x^2 + 3)/8,$$

...

They are solutions to Legendre equation.

Example 1. Verify that $P_0 = 1$, $P_1 = x$, $P_2 = (3x^2 - 1)/2$ are orthogonal to each other.

Answer. Note: P_0, P_2 are even functions, and P_1 is odd.

Then, $P_0(x)P_1(x)$ and $P_1(x)P_2(x)$ are odd.

$$\int_{-1}^1 P_0(x)P_1(x) dx = 0, \quad \int_{-1}^1 P_1(x)P_2(x) dx = 0.$$

We only need to check

$$\int_{-1}^1 P_0(x)P_2(x) dx = \frac{1}{2} \int_{-1}^1 (3x^2 - 1) dx = \frac{1}{2} (x^3 - x) \Big|_{-1}^1 = 0,$$

proving that P_0, P_1, P_2 are orthogonal to each other.

Example 2. Find a function $g(x) = a_0 P_0(x) + a_1 P_1(x) + a_2 P_2(x)$ on $-1 \leq x \leq 1$, that “best” approximates the function

$$f(x) = \begin{cases} -1, & -1 \leq x \leq 0 \\ 1, & 0 < x \leq 1 \end{cases} .$$

Answer. With orthogonal basis, the coefficients a_i are simply computed as

$$a_i = \frac{\int_{-1}^1 f(x)P_i(x) dx}{\int_{-1}^1 P_i^2(x) dx}.$$

Note that $f(x)$ is odd, therefore $f(x)P_0(x)$ and $f(x)P_2(x)$ are odd, and $a_0 = 0$, $a_2 = 0$. We only need to compute a_1 :

$$a_1 = \frac{\int_{-1}^1 f(x)P_1(x) dx}{\int_{-1}^1 P_1^2(x) dx} = \frac{2 \int_0^1 x dx}{\int_{-1}^1 x^2 dx} = \frac{2(0.5)}{2/3} = \frac{3}{2}.$$

Therefore, the “best” approximation is $g(x) = \frac{3}{2}P_1(x) = \frac{3}{2}x$.

From Differential Equation Course, we learned that the set of trig functions, defined on the interval $x \in [-\pi, \pi]$,

$$1, \sin nx, \cos nx, \quad n = 1, 2, \dots$$

are orthogonal to each other. This means,

$$\int_{-\pi}^{\pi} 1 \cdot \sin nx \, dx = 0, \quad \text{for every } n$$

$$\int_{-\pi}^{\pi} 1 \cdot \cos nx \, dx = 0, \quad \text{for every } n$$

$$\int_{-\pi}^{\pi} \sin nx \cdot \sin mx \, dx = 0 \quad \text{for every } m \neq n$$

$$\int_{-\pi}^{\pi} \cos nx \cdot \cos mx \, dx = 0 \quad \text{for every } m \neq n$$

$$\int_{-\pi}^{\pi} \sin nx \cdot \cos mx \, dx = 0 \quad \text{for every } m, n$$

Example 4. Given M . Approximate a function $f(x)$ defined on $[-\pi, \pi]$ by

$$g(x) = c_0 + \sum_{n=1}^M \left[a_n \sin nx + b_n \cos nx \right],$$

in the “best” possible way.

Answer. Using the orthogonal property, the coefficients are computed as

$$\begin{aligned} a_n &= \frac{\int_{-\pi}^{\pi} f(x) \sin nx \, dx}{\int_{-\pi}^{\pi} \sin^2 nx \, dx}, & b_n &= \frac{\int_{-\pi}^{\pi} f(x) \cos nx \, dx}{\int_{-\pi}^{\pi} \cos^2 nx \, dx} \\ c_0 &= \frac{1}{2} \int_{-\pi}^{\pi} f(x) \, dx. \end{aligned}$$

As $M \rightarrow \infty$, we obtain the Fourier series for $f(x)$!

CMPSC/Math 451, Numerical Computation

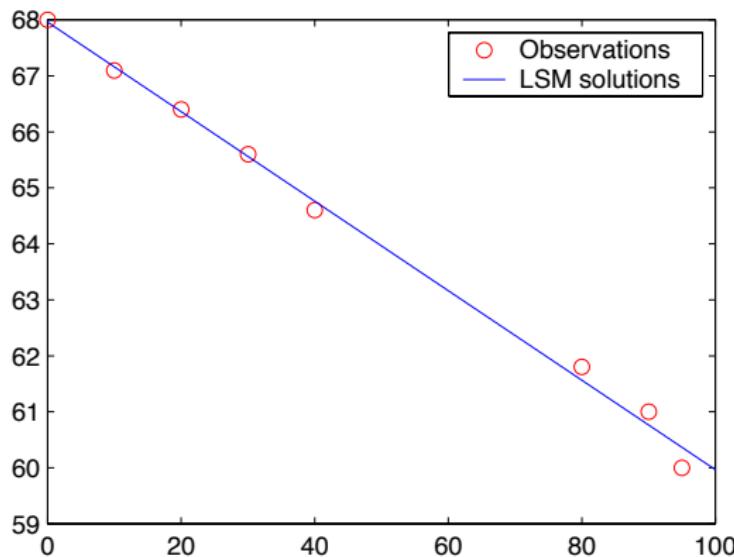
Wen Shen

Department of Mathematics, Penn State University

Matlab Simulations for Least Squares Method

Example 1: Linear regression $S = aT + b$ for the data:

T_k	0	10	20	30	40	80	90	95
S_k	68.0	67.1	66.4	65.6	64.6	61.8	61.0	60.0



Example 2:

The water tide in the North Sea can be characterized by the following formulae for the height of water $H(t)$, a periodic function of time t with the period equals to 12 hours:

$$H(t) = a_0 + a_1 \sin \frac{2\pi t}{12} + a_2 \cos \frac{2\pi t}{12}$$

We have the following observation data of the height of the water:

t_k	0.0	2.0	4.0	6.0	8.0	10.0	(hours)
H_k	1.0	1.6	1.4	0.6	0.2	0.8	(meters)

What is a_0 , a_1 and a_2 ?

Answer. The normal equations:

$$\sum_k a_0 + a_1 \sin \frac{\pi t_k}{6} + a_2 \cos \frac{\pi t_k}{6} - H_k = 0$$

$$\sum_k \left[a_0 + a_1 \sin \frac{\pi t_k}{6} + a_2 \cos \frac{2\pi t_k}{12} - H_k \right] \sin \frac{\pi t_k}{6} = 0$$

$$\sum_k \left[a_0 + a_1 \sin \frac{\pi t_k}{6} + a_2 \cos \frac{2\pi t_k}{12} - H_k \right] \cos \frac{\pi t_k}{6} = 0$$

i.e.,

$$a_0(n+1) + a_1 \sum_k \sin \frac{\pi t_k}{6} + a_2 \sum_k \cos \frac{\pi t_k}{6} = \sum_k H_k$$

$$a_0 \sum_k \sin \frac{\pi t_k}{6} + a_1 \sum_k \sin^2 \frac{\pi t_k}{6} + a_2 \sum_k \cos \frac{\pi t_k}{6} \sin \frac{\pi t_k}{6} = \sum_k H_k \sin \frac{\pi t_k}{6}$$

$$a_0 \sum_k \cos \frac{\pi t_k}{6} + a_1 \sum_k \sin \frac{\pi t_k}{6} \cos \frac{\pi t_k}{6} + a_2 \sum_k \cos^2 \frac{\pi t_k}{6} = \sum_k H_k \cos \frac{\pi t_k}{6}$$

Simple Matlab codes:

```
t=[0 2 4 6 8 10];
H=[1 1.6 1.4 0.6 0.2 0.8];
n=length(t); va=pi/6;

s1=sum(sin(va*t));
s2=sum(cos(va*t));
s3=sum(sin(va*t).^2);
s4=sum(cos(va*t).*sin(va*t));
s5=sum(cos(va*t).^2);

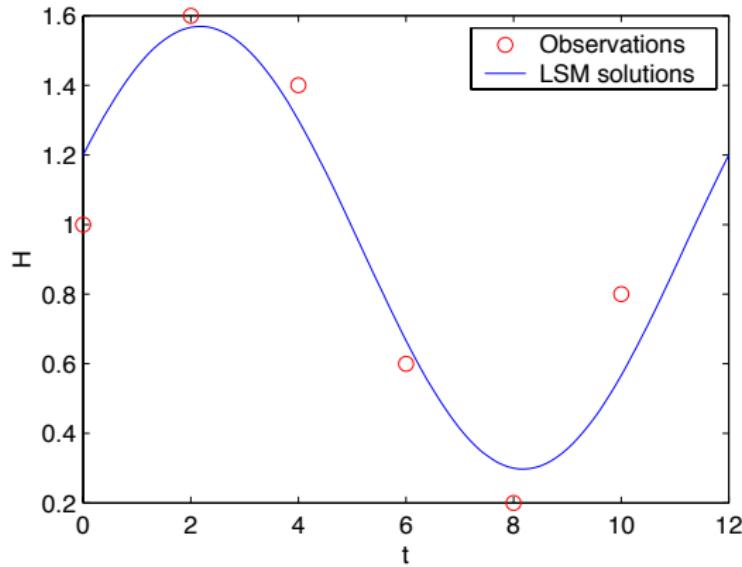
A=[n,s1,s2; s1, s3, s4; s2, s4, s5];
h=[sum(H);sum(H.*sin(va*t));sum(H.*cos(va*t))];
a=A\h;

x=[0:0.05:12];
fx=a(1)+a(2)*sin(va*x)+a(3)*cos(va*x);
plot(x,fx,'b',t,H,'ro')
```

The code gives:

$$a_0 = 0.9333, a_1 = 0.5774, a_2 = 0.2667$$

The plot:



CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Numerical solution of ordinary differential equations (ODE)

Definition of ODE:

an equation which contains one or more ordinary derivatives of an unknown function.

Example 1. Let $x = x(t)$ be the unknown function of t , ODE examples can be

$$x' = x^2, \quad x'' + x \cdot x' + 4 = 0, \quad \text{etc.}$$

We consider the initial-value problem for first-order ODE

$$\begin{cases} x' = f(t, x), & \text{differential equation} \\ x(t_0) = x_0 & \text{given initial condition (IC)} \end{cases}$$

Some examples:

$$x'(t) = 2, \quad x(0) = 0. \quad \text{solution:} \quad x(t) = 2t.$$

$$x'(t) = 2t, \quad x(0) = 0. \quad \text{solution:} \quad x(t) = t^2.$$

$$x'(t) = x + 1, \quad x(0) = 0. \quad \text{solution:} \quad x(t) = e^t - 1.$$

In many situations, exact solutions can be very difficult/impossible to obtain.

Numerical solutions

We seek approximate values of the solution at discrete sampling points.

Uniform grid for time variable. Let h be the time step length

$$t_{k+1} - t_k = h, \quad t_k = t_0 + kh, \quad t_0 < t_1 < \cdots < t_N.$$

Given an ODE, and a final computing time t_N .

We seek values $x_n \approx x(t_n)$, $n = 1, 2, \dots, N$, and $t_0 < t_1 < \cdots < t_N$.

Overview of the Chapter:

- Taylor series method, and error estimates
- Runge-Kutta methods
- Multi-step methods
- System of ODE
- High order equations and systems
- Stiff systems
- Matlab solvers

Taylor series methods for ODEs

Given

$$x'(t) = f(t, x(t)), \quad x(t_0) = x_0.$$

Let $t_1 = t_0 + h$. Let's find the value $x_1 \approx x(t_1) = x(t_0 + h)$.

Taylor expansion of $x(t_0 + h)$ expand at t_0 gives

$$x(t_0 + h) = x(t_0) + hx'(t_0) + \frac{1}{2}h^2x''(t_0) + \dots = \sum_{m=0}^{\infty} \frac{1}{m!} h^m x^{(m)}(t_0)$$

Taylor series method of order m

We take the first $(m + 1)$ terms in Taylor expansion.

$$x(t_0 + h) \approx x(t_0) + hx'(t_0) + \frac{1}{2}h^2x''(t_0) + \cdots + \frac{1}{m!}h^m x^{(m)}(t_0).$$

Error in each step:

$$x(t_0 + h) - x_1 = \sum_{k=m+1}^{\infty} \frac{1}{k!} h^k x^{(k)}(t_0) = \frac{1}{(m+1)!} h^{m+1} x^{(m+1)}(\xi)$$

for some $\xi \in (t_0, t_1)$.

For $m = 1$:

$$x_1 = x_0 + h x'(t_0) = x_0 + h \cdot f(t_0, x_0)$$

This is called **forward Euler step**.

General formula for step number k :

$$x_{k+1} = x_k + h \cdot f(t_k, x_k), \quad k = 0, 1, 2, \dots, N - 1$$

For $m = 2$:

$$x_1 = x_0 + h x'(t_0) + \frac{1}{2} h^2 x''(t_0)$$

Computing x'' :

$$x'' = \frac{d}{dt} x'(t) = \frac{d}{dt} f(t, x(t)) = f_t + f_x \cdot x' = f_t + f_x \cdot f$$

we get

$$x_1 = x_0 + h f(t_0, x_0) + \frac{1}{2} h^2 [f_t(t_0, x_0) + f_x(t_0, x_0) \cdot f(t_0, x_0)]$$

For general step k , $k = 0, 1, 2, \dots, N - 1$, we have

$$x_{k+1} = x_k + h f(t_k, x_k) + \frac{1}{2} h^2 [f_t(t_k, x_k) + f_x(t_k, x_k) \cdot f(t_k, x_k)]$$

Examples of Taylor Series Method

Example 1. Set up Taylor series methods with $m = 1$ and $m = 2$ for

$$x' = -x + e^{-t}, \quad x(0) = 0.$$

The exact solution is $x(t) = te^{-t}$.

Answer. The initial data gives $t_0 = 0, x_0 = 0$.

For $m = 1$, we have

$$x_{k+1} = x_k + h(-x_k + e^{-t_k}) = (1 - h)x_k + he^{-t_k}$$

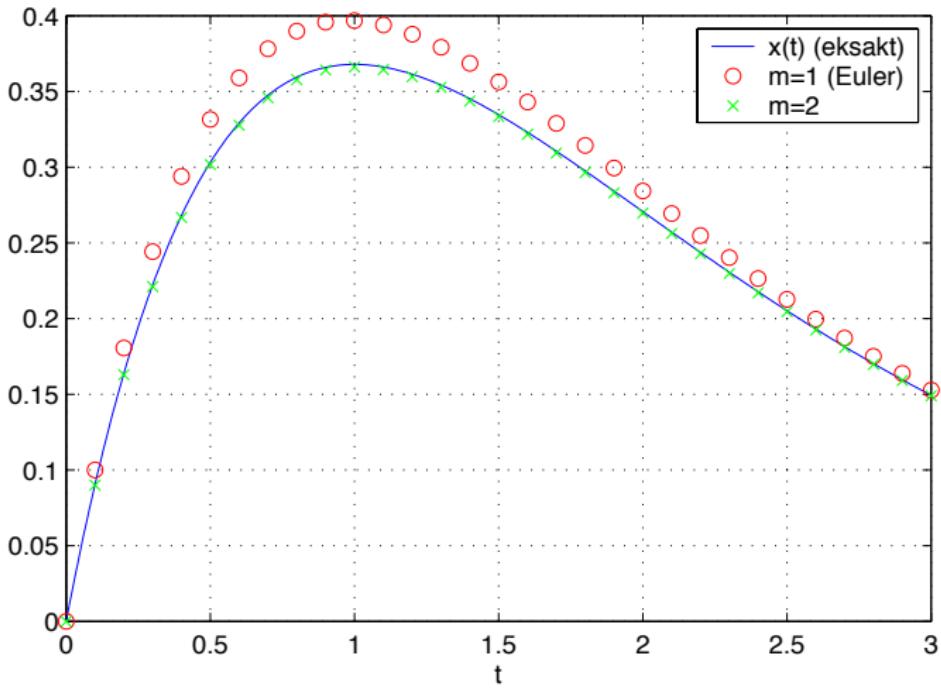
For $m = 2$, we have

$$x'' = (-x + e^{-t})' = -x' - e^{-t} = x - e^{-t} - e^{-t} = x - 2e^{-t}$$

so

$$\begin{aligned} x_{k+1} &= x_k + hx'_k + 0.5h^2x''_k \\ &= x_k + h(-x_k + e^{-t_k}) + 0.5h^2(x_k - 2e^{-t_k}) \\ &= (1 - h + 0.5h^2)x_k + (h - h^2)e^{-t_k} \end{aligned}$$

Simulation result



Example 2. Set up Taylor series methods with $m = 1, 2, 3, 4$ for

$$x' = x, \quad x(0) = 1.$$

The exact solution is $x(t) = e^t$.

Answer. We set $t_0 = 0, x_0 = 1$. Note that

$$x'' = x' = x, \quad x''' = x'' = x, \quad \dots \quad x^{(m)} = x$$

Taylor series method of order m :

$$x_{k+1} = x_k + hx_k + h^2x_k/2 + \dots + h^mx_k/(m!)$$

So

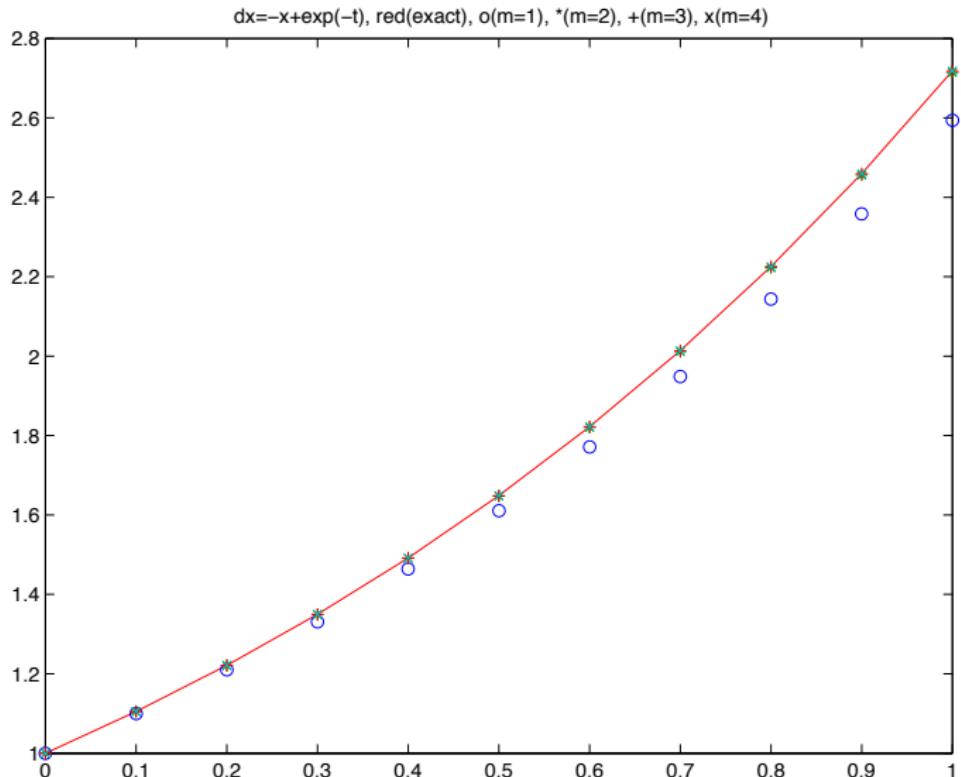
$$m = 1 : \quad x_{k+1} = x_k + hx_k = (1 + h)x_k$$

$$m = 2 : \quad x_{k+1} = x_k + hx_k + h^2x_k/2 = (1 + h + h^2/2)x_k$$

$$m = 3 : \quad x_{k+1} = x_k + hx_k + h^2x_k/2 + h^3x_k/6 = (1 + h + h^2/2 + h^3/6)x_k$$

$$m = 4 : \quad x_{k+1} = \dots = (1 + h + h^2/2 + h^3/6 + h^4/24)x_k$$

Simulation result



Error analysis for Taylor Series Methods

Given ODE

$$x' = f(t, x), \quad x(t_0) = x_0. \quad (*)$$

Local error (error in each time step) for Taylor series method of order m .

Let t_k, x_k be given,

let x_{k+1} be the numerical solution after one iteration,

and let $x(t_k + h)$ be the exact solution for the IVP

$$x' = f(t, x), \quad x(t_k) = x_k.$$

Then, the local truncation error is define as

$$e_k \doteq |x_{k+1} - x(t_k + h)|.$$

Theorem For Taylor series method of order m at step k , the local error is of order $m + 1$, i.e., $e_k \leq Mh^{m+1}$ for some bounded constant M .

Proof. We have

$$e_k = |x_{k+1} - x(t_k + h)| = \frac{h^{m+1}}{(m+1)!} \left| x^{(m+1)}(\xi) \right| = \frac{h^{m+1}}{(m+1)!} \left| \frac{d^m f}{dt^m}(\xi, x(\xi)) \right|,$$

for some $\xi \in (t_k, t_{k+1})$.

We assume

$$\left| \frac{d^m f}{dt^m} \right| \leq M,$$

Now we have

$$e_k \leq \frac{M}{(m+1)!} h^{m+1} = \mathcal{O}(h^{m+1}).$$

Definition. The ODE $x' = f(t, x)$ is called **well-posed** if it is stable w.r.t. perturbations in initial data. This means, let $x(t)$ and $\tilde{x}(t)$ be the solutions with two different initial conditions $x(t_0) = x_0$ and $\tilde{x}(t_0) = \tilde{x}_0$. Fix a final time T . Then, there exists a constant C , independent of t , such that

$$|x(t) - \tilde{x}(t)| \leq C |x_0 - \tilde{x}_0|.$$

Total error is the error at the final computing time T .

Let

$$N = \frac{T}{h}, \quad \text{i.e.,} \quad T = Nh.$$

The total error is defined as

$$E \doteq |x(T) - x_N|.$$

Theorem. Assume that the ODE is well-posed. If the local error of a numerical iteration satisfies

$$e_k \leq Mh^{m+1}$$

then the total error satisfies

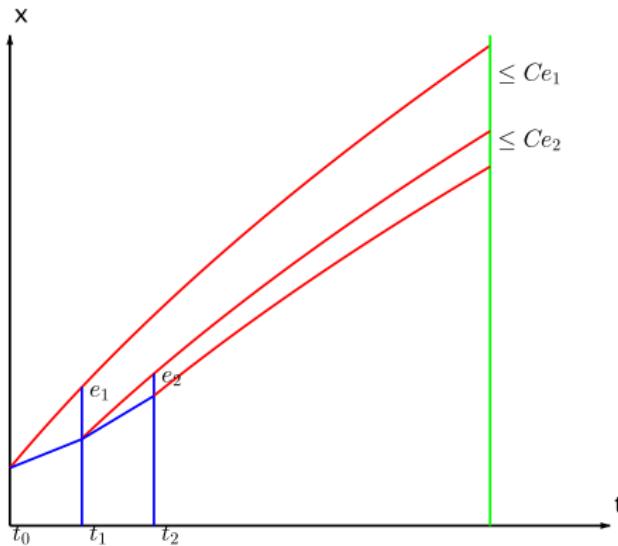
$$E \leq Ch^m$$

for some bounded constant C , where C is uniform in t .

This means that the order of the method is always 1 less than the order of the local truncation error.

Proof. We observe two facts about the errors. First, at every step k , the local error is being carried on through the rest of the simulation. Second, the local errors accumulate through time iteration steps.

By the well-posedness assumption, at each time step k , the local error e_k is amplified at most by a factor of C in the answer at the final time T .



We can add up all the accumulated errors at T caused by all the local errors

$$\begin{aligned} E &= C \sum_{k=1}^N \left| e_L^{(k)} \right| \leq C \sum_{k=1}^N \frac{M}{(m+1)!} h^{m+1} \\ &= CN \frac{M}{(m+1)!} h^{m+1} = C(Nh) \frac{M}{(m+1)!} h^m = \frac{CMT}{(m+1)!} h^m = \mathcal{O}(h^m) \end{aligned}$$

Therefore, the method is of order m .

Runge-Kutta Methods

One difficulty in high order Taylor series methods lies in the fact that it uses the higher order derivatives x'', x''', \dots , which might be very difficult to get.

A better method should only use $f(t, x)$, not its derivatives.

These methods are called Runge-Kutta methods.

1st order method: The same as Euler's method.

2nd order method: Let $h = t_{k+1} - t_k$. Given x_k , the next value x_{k+1} is computed as

$$x_{k+1} = x_k + \frac{1}{2}(K_1 + K_2)$$

where

$$\begin{cases} K_1 &= h \cdot f(t_k, x_k) \\ K_2 &= h \cdot f(t_k + h, x_k + K_1) \end{cases}$$

This is called Heun's method.

Theorem. *Heun's method is of second order.*

Proof. It suffices to show that the local truncation error is of order 3.
Taylor expansion in two variables gives

$$f(t_k + h, x_k + K_1) = f(t_k, x_k) + hf_t(t_k, x_k) + K_1 f_x(t_k, x_k) + \mathcal{O}(h^2, K_1^2).$$

We have $K_1 = hf(t_k, x_k)$, so the last term above is actually $\mathcal{O}(h^2)$. We also have

$$K_2 = h \left[f(t_k, x_k) + hf_t(t_k, x_k) + hf(t_k, x_k)f_x(t_k, x_k) + \mathcal{O}(h^2) \right]$$

Then, our method is:

$$\begin{aligned} x_{k+1} &= x_k + \frac{1}{2} \left[hf + hf + h^2 f_t + h^2 f f_x + \mathcal{O}(h^3) \right] \\ &= x_k + hf + \frac{1}{2} h^2 [f_t + f f_x] + \mathcal{O}(h^3) \end{aligned}$$

$$x_{k+1} = x_k + hf + \frac{1}{2}h^2[f_t + ff_x] + \mathcal{O}(h^3)$$

Compare this with Taylor expansion for $x(t_{k+1}) = x(t_k + h)$

$$\begin{aligned}x(t_k + h) &= x(t_k) + hx'(t_k) + \frac{1}{2}h^2x''(t_k) + \mathcal{O}(h^3) \\&= x(t_k) + hf(t_k, x_k) + \frac{1}{2}h^2[f_t + f_x x'] + \mathcal{O}(h^3) \\&= x(t_k) + hf + \frac{1}{2}h^2[f_t + f_x f] + \mathcal{O}(h^3).\end{aligned}$$

We see the first 3 terms are identical, this gives the local truncation error:

$$e_L = |x_{k+1} - x(t_k + h)| = \mathcal{O}(h^3)$$

Viewing Heun's Method as trapezoid method.

Integrating the ODE $x' = f(t, x)$ over the integral $t \in [t_k, t_k + h]$, we get

$$x(t_k + h) = x(t_k) + \int_{t_k}^{t_k+h} x'(t) dt = x(t_k) + \int_{t_k}^{t_k+h} f(t, x(t)) dt. \quad (1)$$

Once $x(t_k) \approx x_k$ is given, then $x_{k+1} \approx x(t_k + h)$ can be computed by suitably approximating the integral.

For the Heun's method, we see that

$$K_1 \approx hx'(t_k), \quad K_2 \approx hx'(t_k + h).$$

Then, the trapezoid rule

$$\int_{t_k}^{t_k+h} x'(t) dt \approx \frac{h}{2} [x'(t_k) + x'(t_k + h)] = \frac{1}{2}(K_1 + K_2)$$

exactly gives the Heun's iteration.

General Rung-Kutta methods of order m

These methods take the form

$$x_{k+1} = x_k + w_1 K_1 + w_2 K_2 + \cdots + w_m K_m$$

where

$$\left\{ \begin{array}{lcl} K_1 & = & h \cdot f(t_k, x_k) \\ K_2 & = & h \cdot f(t_k + a_2 h, x + b_2 K_1) \\ K_3 & = & h \cdot f(t_k + a_3 h, x + b_3 K_1 + c_3 K_2) \\ & \vdots & \\ K_m & = & h \cdot f(t_k + a_m h, x + \sum_{i=1}^{m-1} \phi_i K_i) \end{array} \right.$$

The parameters w_i, a_i, b_i, ϕ_i are carefully chosen to guarantee the order m .
NB! The choice is NOT unique!

The classical RK4

This elegant 4th order method takes the form

$$x_{k+1} = x_k + \frac{1}{6} [K_1 + 2K_2 + 2K_3 + K_4]$$

where

$$K_1 = h \cdot f(t_k, x_k),$$

$$K_2 = h \cdot f\left(t_k + \frac{1}{2}h, x_k + \frac{1}{2}K_1\right),$$

$$K_3 = h \cdot f\left(t_k + \frac{1}{2}h, x_k + \frac{1}{2}K_2\right),$$

$$K_4 = h \cdot f(t_k + h, x_k + K_3).$$

Viewing RK4 Method as Simpsons rule.

The integral form of the ODE $x' = f(t, x)$ gives

$$x(t_k + h) = x(t_k) + \int_{t_k}^{t_k+h} x'(t) dt = x(t_k) + \int_{t_k}^{t_k+h} f(t, x(t)) dt.$$

For the RK4 method, we see that

$$K_1 \approx hx'(t_k), \quad K_2 \approx hx'(t_k + h/2), \quad K_3 \approx hx'(t_k + h/2), \quad K_4 \approx hx'(t_k + h)$$

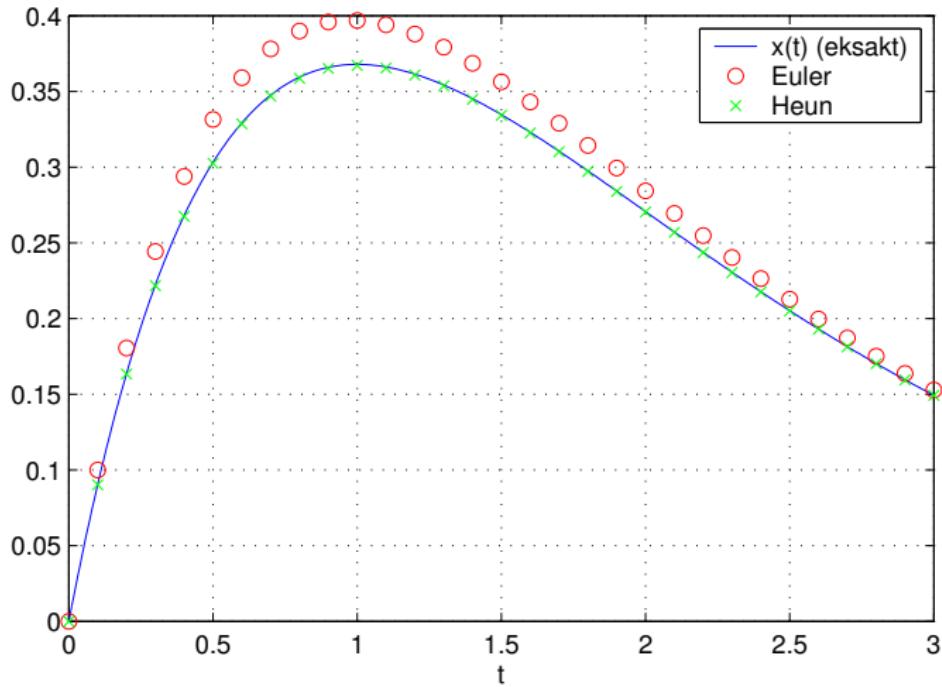
Then, the Simpson's rule

$$\int_{t_k}^{t_k+h} x'(t) dt \approx \frac{h}{6} [x'(t_k) + 4x'(t_k + h/2) + x'(t_k + h)] = \frac{1}{2}(K_1 + 2K_2 + 2K_3 + K_4)$$

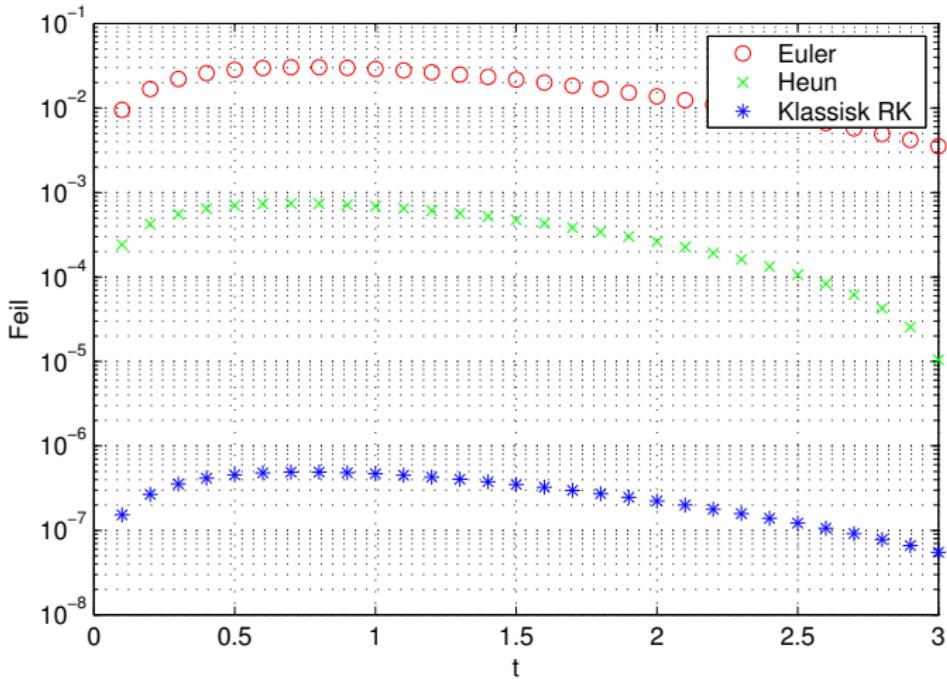
exactly gives the classical RK4 iteration.

Numerical Simulations of Rung-Kutta methods

The ODE: $x' = -x + e^{-t}$, $x(0) = 0$



Plot for the errors with various Runge-Kutta methods.



One sees that the higher order methods perform significantly better than the lower order ones.

An adaptive Runge-Kutta-Fehlberg method

In general, we have

- ① Smaller time step h gives smaller error;
- ② Higher order methods give better approximations;
- ③ With uniform grid, the local error varies at each step, depending on properties of f and its derivatives.

Optimal situation: h varies each step to get uniform error at each step.
This leads to *adaptive methods*.

Key point:

How to get an error estimate at each time step?

Main observation:

If we use two different methods, one is more accurate than the other, then we can assume the better solution is very close to the exact solution.

Thus, the difference between these two solutions becomes a measurement for the local error.

The algorithm reads:

- Compute $x(t + h)$ from $x(t)$ with Method 1, call it $x(t + h)$;
- Compute $x(t + h)$ from $x(t)$ with Method 2, a better method that generates a more accurate approximation. Call this solution $\bar{x}(t + h)$;
- Then, $|x(t + h) - \bar{x}(t + h)|$ gives a measure to error;
- if error $>>$ tol, half the step size;
- if error $<<$ tol, double the step size;
- if error \approx tol, keep the step size;

Now we need to find two methods with different accuracy.
By our observation earlier we may propose the following:

- Method 1: Compute $x(t + h)$ from $x(t)$ with step h , using a specific method, say RK4 method;
- Method 2: Compute $x(t + \frac{1}{2}h)$ from $x(t)$ with step $\frac{1}{2}h$, then compute $x(t + h)$ from $x(t + \frac{1}{2}h)$ with step $\frac{1}{2}h$, with the same methods in Method 1. This approximation is better.

But this is rather wasteful of computing time.

A better approach due to Fehlberg, is built upon some higher order RK methods.
He has a 4th order method:

$$x(t+h) = x(t) + \frac{25}{216}K_1 + \frac{1408}{2565}K_3 + \frac{2197}{4104}K_4 - \frac{1}{5}K_5$$

where

$$K_1 = h \cdot f(t, x),$$

$$K_2 = h \cdot f\left(t + \frac{1}{4}h, x + \frac{1}{4}K_1\right),$$

$$K_3 = h \cdot f\left(t + \frac{3}{8}h, x + \frac{3}{32}K_1 + \frac{9}{32}K_2\right),$$

$$K_4 = h \cdot f\left(t + \frac{12}{13}h, x + \frac{1932}{2197}K_1 - \frac{7200}{2197}K_2 + \frac{7296}{2197}K_3\right),$$

$$K_5 = h \cdot f\left(t + h, x + \frac{439}{216}K_1 - 8K_2 + \frac{3680}{513}K_3 - \frac{845}{4104}K_4\right).$$

Adding an additional term:

$$K_6 = h \cdot f \left(t + \frac{1}{2}h, x - \frac{8}{27}K_1 + 2K_2 - \frac{3544}{2565}K_3 + \frac{1859}{4104}K_4 - \frac{11}{40}K_5 \right)$$

one obtains a 5th order method:

$$\bar{x}(t+h) = x(t) + \frac{16}{135}K_1 + \frac{6656}{12825}K_3 + \frac{28561}{56430}K_4 - \frac{9}{50}K_5 + \frac{2}{55}K_6.$$

Assuming that the 5th order approximation is almost the exact solution, we can use the difference $|x(t+h) - \bar{x}(t+h)|$ as an estimate for the error.

Pseudo code for adaptive RK45, with time step controller, is provided below.

Given $t_0, t_f, x_0, h_0, n_{max}, e_{min}, e_{max}, h_{min}, h_{max}$

set $h = h_0, t = t_0, x_0 = x_0, k = 0,$

while ($k < n_{max}$ and $t < t_f$) do

 if $h < h_{min}$ then $h = h_{min},$

 else if $h > h_{max}$ then $h = h_{max},$

 end

 Compute RKF4, RKF5, and $e = |RKF4 - RKF5|$

 if ($e > e_{max}$ and $h > h_{min}$), then $h = h/2;$ (reject the step)

 else –(accept the step)

$k = k + 1; t = t + h; x_k = RKF5;$

 If $e < e_{min}$, then $h = 2 * h;$ end

 end

end (while)

Explicit Adam-Bashforth methods

Given

$$x' = f(t, x), \quad x(t_0) = x_0,$$

Let $t_n = t_0 + nh$. If $x(t_n)$ is given, the exact value for $x(t_{n+1})$ would be

$$x(t_{n+1}) = x(t_n) + \int_{t_n}^{t_{n+1}} x'(t) dt = x(t_n) + \int_{t_n}^{t_{n+1}} f(t, x(t)) dt.$$

Idea: Find a good approximation to the integral

$$\int_{t_n}^{t_{n+1}} f(t, x(t)) dt.$$

How? One possibility: approximate f by polynomials, i.e., polynomial interpolations.

Given

$$t_n, t_{n-1}, \dots, t_{n-k}, \quad x_n, x_{n-1}, \dots, x_{n-k},$$

we can compute

$$f_n = f(t_n, x_n), \quad f_{n-1} = f(t_{n-1}, x_{n-1}), \quad \dots, \quad f_{n-k} = f(t_{n-k}, x_{n-k}),$$

to obtain the data set

$$(t_n, f_n), \quad (t_{n-1}, f_{n-1}), \quad \dots, \quad (t_{n-k}, f_{n-k}).$$

We find interpolating polynomial $P_k(t)$ that interpolates the above data set $(t_i, f_i)_{i=n-k}^n$.

The Lagrange form for $P_k(t)$ with the cardinal functions $l_i(t)$

$$P_k(t) = f_n l_n(t) + f_{n-1} l_{n-1}(t) + \dots + f_{n-k} l_{n-k}(t).$$

We then use $P_k(t)$ as an approximation to $f(t, x(t))$, and obtain the time iteration step:

$$x_{n+1} = x_n + \int_{t_n}^{t_{n+1}} P_k(s) ds,$$

which gives

$$\begin{aligned} x_{n+1} &= x_n + \int_{t_n}^{t_n+h} (f_n l_n(t) + f_{n-1} l_{n-1}(t) + \cdots + f_{n-k} l_{n-k}(t)) dt \\ &= x_n + \int_{t_n}^{t_n+h} f_n l_n(t) dt + \cdots + \int_{t_n}^{t_n+h} f_{n-k} l_{n-k}(t) dt \\ &= x_n + f_n \int_{t_n}^{t_n+h} l_n(t) dt + \cdots + f_{n-k} \int_{t_n}^{t_n+h} l_{n-k}(t) dt \\ &= x_n + h \cdot (b_0 f_n + b_1 f_{n-1} + b_2 f_{n-2} + \cdots + b_k f_{n-k}), \end{aligned}$$

where b_0, b_1, \dots, b_k are constants,

$$b_0 = \frac{1}{h} \int_{t_n}^{t_n+h} l_n(t) dt, \quad b_1 = \frac{1}{h} \int_{t_n}^{t_n+h} l_{n-1}(t) dt, \quad \cdots \quad b_k = \frac{1}{h} \int_{t_n}^{t_n+h} l_{n-k}(t) dt.$$

Examples of some explicit Adam-Bashforth (AB) methods

Example 1. If $k = 0$, then we use only one point, i.e., (t_n, f_n) , and $P_0(t) = f_n$.

This gives

$$x_{n+1} = x_n + h \cdot f(t_n, x_n)$$

We recognize this as the forward explicit Euler's method.

Example 2. Consider $k = 1$, and we will use two points.

Given x_n, x_{n-1} , we can compute f_n, f_{n-1} as

$$f_n = f(t_n, x_n), \quad f_{n-1} = f(t_{n-1}, x_{n-1})$$

Use now linear interpolation, we get

$$x'(s) \approx P_1(s) = f_{n-1} + \frac{f_n - f_{n-1}}{h}(s - t_{n-1}).$$

Then

$$x_{n+1} = x_n + \int_{t_n}^{t_{n+1}} P_1(s) ds = x_n + \frac{h}{2}(3f_n - f_{n-1}).$$

This is the famous 2nd order Adam-Bashforth method.

One needs two initial values to start the iteration, where $x_0 = x(t_0)$ is given, and x_1 could be computed by for either Euler, Heun's or some RK4 method.

Example 3. For $k = 2$, we get a third order method:

$$x_{n+1} = x_n + h \left(\frac{23}{12} f_n - \frac{4}{3} f_{n-1} + \frac{5}{12} f_{n-2} \right).$$

For $k = 3$, we have a 4th order method:

$$x_{n+1} = x_n + h \left(\frac{55}{24} f_n - \frac{59}{24} f_{n-1} + \frac{37}{24} f_{n-2} - \frac{3}{8} f_{n-3} \right).$$

Critics on the method:

- Good sides: Simple, minimum number of $f(\cdot)$ evaluations. Fast.
- Disadvantage: Here we use interpolating polynomial to approximate a function outside the interval of interpolating points. This is called extrapolation, and it gives a bigger error.

Implicit Adam-Bashforth-Moulton (ABM) methods

The main idea here is to avoid using extrapolation, to reduce the interpolation error.

We will now find an interpolating polynomial $P_{k+1}(t)$ that interpolates

$$(f_{n+1}, t_{n+1}), \quad (f_n, t_n), \quad \dots, \quad (f_{n-k}, t_{n-k}),$$

and use it to approximate $f(t, x(t))$, to generate an iteration step

$$x_{n+1} = x_n + \int_{t_n}^{t_{n+1}} P_{k+1}(s) ds.$$

From previous discussion, using the Lagrange form for P_{k+1} , we end up with the general form for each time step:

$$x_{n+1} = x_n + h \cdot (b_{-1}f_{n+1} + b_0f_n + b_1f_{n-1} + b_2f_{n-2} + \cdots + b_kf_{n-k})$$

for some suitable constants b_{-1}, b_0, \dots, b_k .

Important observation: The function $f_{n+1} = f(t_{n+1}, x_{n+1})$ is unknown! Therefore, we get a non-linear equation to solve for x_{n+1} . We called this kind of method an **implicit method**.

One can solve the nonlinear equation by Newton or secant method. An excellent choice for initial guess would be the solution with 2nd order AB (explicit) method. Then, Newton iteration will converge in 1-2 iterations.

Examples of ABM methods

Consider $k = -1, 0, 1, 2, 3$ (which correspond to using 1, 2, 3, 4, 5 points respectively). Straight computation gives

$$k = -1 : \quad x_{n+1} = x_n + h \cdot f_{n+1}, \quad (\text{implicit backward Euler's method})$$

$$k = 0 : \quad x_{n+1} = x_n + \frac{h}{2} (f_n + f_{n+1}), \quad (\text{trapezoid rule})$$

$$k = 1 : \quad x_{n+1} = x_n + h \cdot \left(\frac{5}{12} f_{n+1} + \frac{2}{3} f_n - \frac{1}{12} f_{n-1} \right)$$

$$k = 2 : \quad x_{n+1} = x_n + h \cdot \left(\frac{3}{8} f_{n+1} + \frac{19}{24} f_n - \frac{5}{24} f_{n-1} + \frac{1}{24} f_{n-2} \right)$$

$$k = 3 : \quad x_{n+1} = x_n + h \cdot \left(\frac{251}{720} f_{n+1} + \frac{646}{720} f_n - \frac{264}{720} f_{n-1} + \frac{106}{720} f_{n-2} - \frac{19}{720} f_{n-3} \right)$$

Multi-step ABM method.

Next is the famous 2nd order ABM (Adam-Basforth-Moulton) method, which combines the explicit and implicit methods in a smart way.

- Given $x_n, x_{n-1}, f_n, f_{n-1}$, compute explicit AB solution with $k = 1$:

$$(P) \quad \begin{cases} x_{n+1}^* &= x_n + h \left(\frac{3}{2}f_n - \frac{1}{2}f_{n-1} \right), \\ f_{n+1}^* &= f(t_{n+1}, x_{n+1}^*). \end{cases}$$

- Take one step of the implicit AB solution with $k = 0$, using the value in step 1 as the value for t_{n+1} :

$$(C) \quad \begin{cases} x_{n+1} &= x_n + \frac{h}{2} (f_{n+1}^* + f_n), \\ f_{n+1} &= f(t_{n+1}, x_{n+1}). \end{cases}$$

Here step (P) is called the *predictor*, and step (C) is the *corrector*. This is called a *predictor-corrector's method*.

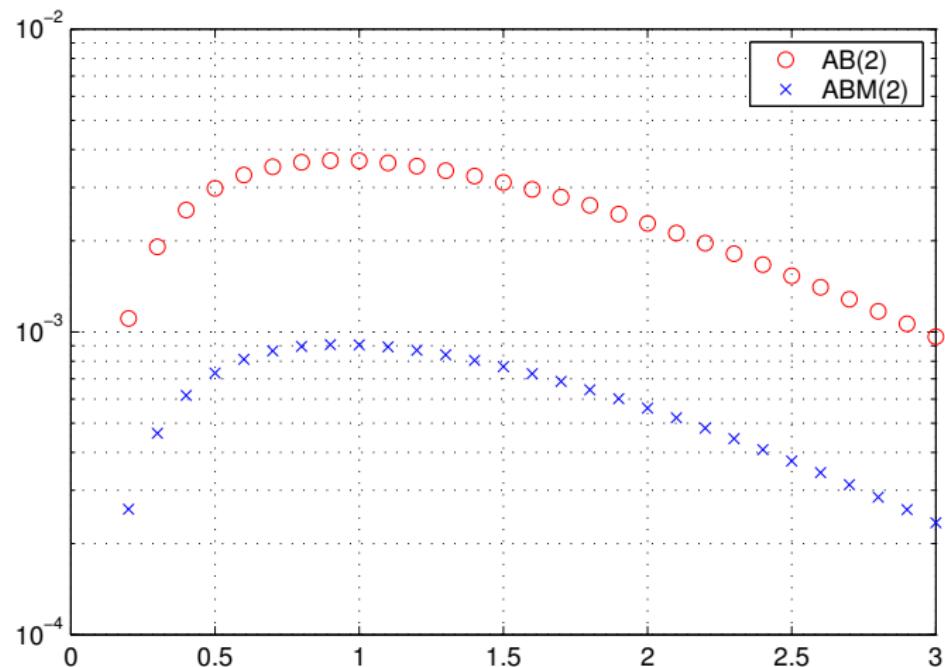
Numerical Simulations.

We consider again the ODE

$$x' = -x + e^{-t}, \quad x(0) = 0.$$

In order to convince ourselves that the corrector step really reduces the error, we solve the equation by both the explicit Adam-Bashforth method and the multi-step Adam-Bashforth-Moulton method.
Both methods are formally second order.

The errors are plotted in the Figure below.



The merit of the corrector step is apparent in the result.

Methods for first order systems of ODE

We consider

$$\vec{x}' = F(t, \vec{x}), \quad \vec{x}(t_0) = \vec{x}_0$$

Here $\vec{x} = (x_1, x_2, \dots, x_n)^t$ is a vector, and $F = (f_1, f_2, \dots, f_n)^t$ is a vector-valued function.

Write it out

$$\begin{cases} x'_1 &= f_1(t, x_1, x_2, \dots, x_n) \\ x'_2 &= f_2(t, x_1, x_2, \dots, x_n) \\ \dots \\ x'_n &= f_n(t, x_1, x_2, \dots, x_n) \end{cases}$$

Good news: All methods for scalar equation can be adapted for systems!

Taylor series methods:

$$\vec{x}(t + h) = \vec{x} + h\vec{x}' + \frac{1}{2}h^2\vec{x}'' + \cdots + \frac{1}{m!}h^m\vec{x}^{(m)}$$

Example

Consider

$$\begin{cases} x_1' &= x_1 - x_2 + 2t - t^2 - t^3 \\ x_2' &= x_1 + x_2 - 4t^2 + t^3 \end{cases}$$

We will need the high order derivatives:

$$\begin{cases} x_1'' &= x_1' - x_2' + 2 - 2t - 3t^2 \\ x_2'' &= x_1' + x_2' - 8t + 3t^2 \end{cases}$$

and

$$\begin{cases} x_1''' &= x_1'' - x_2'' - 2 - 6t \\ x_2''' &= x_1'' + x_2'' - 8 + 6t \end{cases}$$

and so on...

Runge-Kutta methods also take the same form for systems. For example, the classical RK4 becomes:

$$\vec{x}_{k+1} = \vec{x}_k + \frac{1}{6} [\vec{K}_1 + 2\vec{K}_2 + 2\vec{K}_3 + \vec{K}_4]$$

where

$$\vec{K}_1 = h \cdot F(t_k, \vec{x}_k)$$

$$\vec{K}_2 = h \cdot F(t_k + \frac{1}{2}h, \vec{x}_k + \frac{1}{2}\vec{K}_1)$$

$$\vec{K}_3 = h \cdot F(t_k + \frac{1}{2}h, \vec{x}_k + \frac{1}{2}\vec{K}_2)$$

$$\vec{K}_4 = h \cdot F(t_k + h, \vec{x}_k + \vec{K}_3)$$

Here everything is a vector instead of a scalar value.

Matlab Codes: The classical RK4 method .

```
function [t,x] = rk4(f,t0,x0,tend,N)
% f : Differential equation xp = f(t,x)
% x0 : initial condition
% t0,tend : initial and final time
% N : number of time steps

h = (tend-t0)/N;
t = [t0:h:tend];
s = length(x0); % x0 can be a vector
x = zeros(s,N+1);
x(:,1) = x0;

for n = 1:N
    k1 = feval(f,t(n),x(:,n));
    k2 = feval(f,t(n)+0.5*h,x(:,n)+0.5*h*k1);
    k3 = feval(f,t(n)+0.5*h,x(:,n)+0.5*h*k2);
    k4 = feval(f,t(n)+h,x(:,n)+h*k3);
    x(:,n+1) = x(:,n) + h/6*(k1+2*(k2+k3)+k4);
end
```

Matlab code: The multi-step second order Adams-Bashforth-Moulton method.

```
function [t,x] = abm2(f,t0,x0,x1,tend,N)
% f : Differential equation xp = f(t,x)
% x0,x1 : Starting data
% t0,tend : initial time and final time
% N : number of time steps

h = (tend-t0)/N; t = [t0:h:tend]; s = length(x0);
x = zeros(s,N+1);
x(:,1) = x0; x(:,2) = x1; % starting data for time iterations
fnm1 = feval(f,t(1),x0); fn = feval(f,t(2),x1);

for n = 2:N
    xs = x(:,n) + 0.5*h*(3*fn-fnm1); % predictor
    fnp1 = feval(f,t(n+1),xs); % predictor

    x(:,n+1) = x(:,n)+0.5*h*(fnp1+fn); % corrector
    fnm1 = fn;
    fn = feval(f,t(n),x(:,n)); % corrector
end
```

Higher order equations and systems

Consider the higher order ODE

$$u^{(n)} = f(t, u, u', u'', \dots, u^{(n-1)}), \quad \text{ICs: } u(t_0), u'(t_0), u''(t_0), \dots, u^{(n-1)}(t_0).$$

Introduce a systematic change of variables

$$x_1 = u, \quad x_2 = u', \quad x_3 = u'', \quad \dots \quad x_n = u^{(n-1)}.$$

We then have

$$\left\{ \begin{array}{lcl} x'_1 & = & u' = x_2 \\ x'_2 & = & u'' = x_3 \\ x'_3 & = & u''' = x_4 \\ \vdots & & \\ x'_{n-1} & = & u^{(n-1)} = x_n \\ x'_n & = & u^{(n)} = f(t, x_1, x_2, \dots, x_n) \end{array} \right.$$

This is a system of 1st order ODEs, with initial data given at

$$x_1(t_0) = u(t_0), \quad x_2(t_0) = u'(t_0), \quad \dots, \quad x_n(t_0) = u^{(n-1)}(t_0).$$

Example

Consider the ODE

$$u'' + 4uu' + t^2u + t = 0, \quad u(0) = 1, \quad u'(0) = 2.$$

By the variable change

$$x_1 = u(t), \quad x_2 = u'(t)$$

we have

$$\begin{cases} x'_1 &= x_2 \\ x'_2 &= -4x_1x_2 - t^2x_1 - t \end{cases}$$

with initial conditions

$$\begin{cases} x_1(0) &= u(0) = 1 \\ x_2(0) &= u'(0) = 2 \end{cases}$$

Systems of high-order equations are treated in the same way. We can write each higher order ODE into a system of first order ODEs, and we go through all the equations.

Example

Consider the system of 2nd order ODEs

$$\begin{aligned} u'' &= u(1 - v') + t, \\ v'' &= v^2 - u'v' + tu^2, \end{aligned}$$

with the initial conditions

$$u(1) = 1, \quad u'(1) = 2, \quad v(1) = 3, \quad v'(1) = 4.$$

Introduce the variable change

$$x_1 = u, \quad x_2 = u', \quad x_3 = v, \quad x_4 = v',$$

we have the 4×4 system of first order equations

$$\left\{ \begin{array}{lcl} x'_1 & = & x_2 \\ x'_2 & = & x_1(1 - x_4) + t \\ x'_3 & = & x_4 \\ x'_4 & = & x_3^2 - x_2x_4 + tx_1^2 \end{array} \right. \quad \text{IC.} \quad \left\{ \begin{array}{lcl} x_1(1) & = & u(1) = 1 \\ x_2(1) & = & u'(1) = 2 \\ x_3(1) & = & v(1) = 3 \\ x_4(1) & = & v'(1) = 4 \end{array} \right.$$

Stiff systems

A scalar equation. We first consider a simple scalar equation

$$x' = -ax, \quad x(0) = 1$$

where $a > 0$ is a constant, possibly very large. The exact solution is

$$x(t) = e^{-at}.$$

This is an exponential decay. We see that

$$x \rightarrow 0 \quad \text{as} \quad t \rightarrow +\infty. \tag{1}$$

Furthermore, the larger the value a , the faster the decay.

We now solve it by forward Euler's method:

$$x_0 = 1, \quad x_{n+1} = x_n - ahx_n = (1 - ah)x_n, \quad n \geq 1.$$

Simple induction argument shows that

$$x_n = (1 - ah)^n x_0 = (1 - ah)^n.$$

We expect that the numerical solution should preserve the important property (1), i.e.,

$$x_n \rightarrow 0, \quad \text{as} \quad n \rightarrow +\infty.$$

We must require

$$|1 - ah| < 1, \quad \Rightarrow \quad h < \frac{2}{a}.$$

This gives a restriction to the time step size h , i.e., h must be sufficiently small. The larger the value of a , the smaller h must be, even though the solution is almost 0 after a very short time!

To improve the stability, we now use the implicit Euler step:

$$x_0 = 1, \quad x_{n+1} = x_n - ahx_{n+1}, \quad n \geq 1.$$

This implies

$$x_{n+1} = \frac{1}{1 + ah} x_n.$$

Simple induction argument shows that for all $n \geq 0$, we have

$$x_n = \left(\frac{1}{1 + ah} \right)^n.$$

Since $ah > 0$, we have

$$0 < \frac{1}{1 + ah} < 1$$

leading to

$$\lim_{n \rightarrow +\infty} x_n = 0$$

for any values of h . This is called **unconditionally stable**.

Remark. Stability condition occurs also for nonlinear equations. But if one applies an implicit method, it becomes unconditionally stable, but at a price.

Consider the general equation

$$x' = f(t, x), \quad x(t_0) = x_0.$$

where $f(t, x)$ is nonlinear in x . The implicit Euler step becomes

$$x_{n+1} = x_n + h \cdot f(t_{n+1}, x_{n+1}).$$

We see that this becomes a non-linear equation for x_{n+1} , which may or may not have solutions, or multiple solutions. An approximate solution could be obtained using a possible Newton iteration or some varieties of it. It can be very time consuming.

System of ODEs.

The problem is more annoying for systems. We now consider a system

$$\begin{cases} x' = -20x - 19y \\ y' = -19x - 20y \end{cases} \quad \begin{cases} x(0) = 2 \\ y(0) = 0 \end{cases}$$

We can re-write the system in vector and matrix form as

$$\vec{x}'(t) = A\vec{x}, \quad A = \begin{pmatrix} -20 & -19 \\ -19 & -20 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} x \\ y \end{pmatrix}$$

For the coefficient matrix A , we have the following eigenvalues and condition number

$$\lambda_1(A) = -1, \quad \lambda_2(A) = -39, \quad \text{cond}(A) = 39.$$

Note that the condition number is rather large.

The exact solution is

$$\begin{cases} x(t) = e^{-39t} + e^{-t} \\ y(t) = e^{-39t} - e^{-t} \end{cases}$$

One can easily verify it by plugging it in the ODEs and the ICs.
The exact solution has the following decay property

$$x \rightarrow 0, \quad y \rightarrow 0 \quad \text{as} \quad t \rightarrow +\infty. \quad (1)$$

We have the following further observations:

- Two components in the solution, e^{-39t} and e^{-t} ;
- The two eigenvalues give exact the two decay rates in the solution.
- The condition number of A is rather large, indicating two very different rates of decay in the solution. The term e^{-39t} tends to 0 much faster than the term e^{-t} ;
- For large values of t , the term e^{-t} dominate.
- Therefore, e^{-39t} is called the *transient term*.

We solve the system with forward Euler's method:

$$\begin{cases} x_{n+1} = x_n + h \cdot (-20x_n - 19y_n), \\ y_{n+1} = y_n + h \cdot (-19x_n - 20y_n), \end{cases} \quad \begin{cases} x_0 = 2, \\ y_0 = 0. \end{cases}$$

One can show by induction that

$$\begin{cases} x_n = (1 - 39h)^n + (1 - h)^n, \\ y_n = (1 - 39h)^n - (1 - h)^n. \end{cases}$$

We must require that the numerical approximation preserves the property (1), i.e.,

$$x_n \rightarrow 0, \quad y_n \rightarrow 0 \quad \text{as} \quad n \rightarrow +\infty.$$

This gives the conditions

$$|1 - 39h| < 1 \quad \text{and} \quad |1 - h| < 1$$

which implies

$$(1) : h < \frac{2}{39} \quad \text{and} \quad (2) : h < 2$$

We see that condition (1) is much stronger than condition (2), therefore it must be satisfied.

Condition (1) corresponds to the term e^{-39t} , which is the transient term and it tends to 0 very quickly as t grows. Unfortunately, time step size is restricted by this transient term.

Why is this system stiff? Because the condition number is very large, so the system has two components with very different varying rates.

Stiff System: Implicit method

We now propose a more stable method, the implicit Backward Euler method:

$$\begin{cases} x_{n+1} = x_n + h \cdot (-20x_{n+1} - 19y_{n+1}), \\ y_{n+1} = y_n + h \cdot (-19x_{n+1} - 20y_{n+1}), \end{cases} \quad \begin{cases} x_0 = 2, \\ y_0 = 0. \end{cases}$$

Let

$$A = \begin{pmatrix} -20 & -19 \\ -19 & -20 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} x \\ y \end{pmatrix}, \quad \vec{x}_n = \begin{pmatrix} x_n \\ y_n \end{pmatrix}.$$

We can write

$$\vec{x}_{n+1} = \vec{x}_n + hA \cdot \vec{x}_{n+1} \Rightarrow (I - hA) \vec{x}_{n+1} = \vec{x}_n \Rightarrow \vec{x}_{n+1} = (I - hA)^{-1} \vec{x}_n.$$

$$\vec{x}_{n+1} = (I - hA)^{-1} \vec{x}_n.$$

Take some vector norm $\|\cdot\|$ on both sides, we get

$$\|\vec{x}_{n+1}\| = \|(I - hA)^{-1} \vec{x}_n\| \leq \|(I - hA)^{-1}\| \cdot \|\vec{x}_n\|.$$

We see that if $\|(I - hA)^{-1}\| < 1$, then $\vec{x}_n \rightarrow 0$ as $n \rightarrow +\infty$.

Let's check this condition using the l_2 norm:

$$\|(I - hA)^{-1}\|_2 = \max_i |\lambda_i(I - hA)^{-1}| = \max_i \frac{1}{|(1 - h \cdot \lambda_i(A))|}.$$

We have

$$\lambda_1(A) = -1, \quad \lambda_2(A) = -39$$

They are both negative, therefore $1 - h\lambda_i > 1$ for $i = 1, 2$, implying

$$\|(I - hA)^{-1}\|_2 < 1$$

independent of the value of h .

Therefore, this implicit method is called *unconditionally stable*.

Some critics on the implicit method:

- Advantage: One can choose large h , and the method is always stable. This is particularly suitable for stiff systems.
- Disadvantage: One must solve a system of linear equations at each time step

$$(I - hA)\vec{x}_{n+1} = \vec{x}_n$$

Longer computing time for each step.

In general, for nonlinear systems, implicit method leads to a system of nonlinear equations to solve at every time step. One must then use Newton/secant method. Very expensive to compute.

Therefore the implicit method is NOT recommended if the system is not stiff.

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Two-point boundary value problems

We now consider a second order ODE in the form

$$y''(x) = f(x, y(x), y'(x)), \quad y(a) = \alpha, \quad y(b) = \beta.$$

Here $y(x)$ is the unknown function defined on the interval $a \leq x \leq b$. The values of y at the boundary points $x = a, x = b$ are given.

Such differential equations arise in many physical models.

For example, the model for an elastic string:

$$y'' = ky + mx(x - L), \quad y(0) = 0, \quad y(L) = 0.$$

Note that this equation is linear.

One can also have non-linear equations. For example:

$$-(y')^2 - 2b(x)y + 2yy'' = 0, \quad y(0) = 1, \quad y(1) = a.$$

We study two numerical methods for this two-point boundary value problem:

- Shooting method: based on ODE solvers;
- Finite Difference Method (FDM).

Shooting method

Given some two-point boundary value problem on $a \leq x \leq b$.

Main algorithm:

- Solve two-point boundary value problem as an initial value problem, with initial data given at $x = a$ (a guess).
- Compute the solution and the value in the solution at $x = b$.
- Compare this with the given boundary condition at $x = b$. Then adjust your guess at $x = a$ and iterate if needed.

It makes a difference if the differential equation is linear and nonlinear.

The linear case is simpler.

Linear Shooting.

Let's consider the linear problem in the general form:

$$y''(x) = u(x) + v(x)y(x) + w(x)y'(x), \quad y(a) = \alpha, \quad y(b) = \beta. \quad (1)$$

Let \bar{y} solve the same equation, but with initial conditions:

$$\bar{y}''(x) = u(x) + v(x)\bar{y}(x) + w(x)\bar{y}'(x), \quad \bar{y}(a) = \alpha, \quad \bar{y}'(a) = 0. \quad (2)$$

Note that $\bar{y}'(a) = 0$ is the “guess” we make.

Let \tilde{y} solve the same equation, but with different initial conditions:

$$\tilde{y}''(x) = u(x) + v(x)\tilde{y}(x) + w(x)\tilde{y}'(x), \quad \tilde{y}(a) = \alpha, \quad \tilde{y}'(a) = 1. \quad (3)$$

Note that $\tilde{y}'(a) = 1$ is the other “guess” we make.

As we will see later, it doesn't matter with guesses we make here. Any numbers will work, as long as they are different for \bar{y} and \tilde{y} .

Note that both equations (2) and (3) can be written into a system of first order ODEs, and can be solved by Matlab ODE solver.

Assume now both equations (2) and (3) are now solved on the interval $x \in [a, b]$, (say, by some ODE Solver in Matlab), such that the values $\bar{y}(b)$ and $\tilde{y}(b)$ are computed.

Now let

$$y(x) = \lambda \cdot \bar{y}(x) + (1 - \lambda) \cdot \tilde{y}(x) \quad (4)$$

where λ is a constant to be determined, such that $y(x)$ in (4) becomes the solution for (1).

We now check which equation the y in (4) solves. We have

$$\begin{aligned} y'' &= \lambda \cdot \bar{y}''(x) + (1 - \lambda) \cdot \tilde{y}''(x) \\ &= \lambda(u + v\bar{y} + w\bar{y}') + (1 - \lambda)(u + v\tilde{y} + w\tilde{y}') \\ &= u + v(\lambda\bar{y} + (1 - \lambda)\tilde{y}) + w(\lambda\bar{y}' + (1 - \lambda)\tilde{y}') \\ &= u + vy + wy'. \end{aligned}$$

We see that this y solve the equation (1) for any choices of λ .

We now check the boundary conditions. At $x = a$, we have

$$y(a) = \lambda\bar{y}(a) + (1 - \lambda)\tilde{y}(a) = \lambda\alpha + (1 - \lambda)\alpha = \alpha.$$

The boundary condition is satisfied for any choices of λ .

At $x = b$, we have

$$y(b) = \lambda \bar{y}(b) + (1 - \lambda) \tilde{y}(b).$$

Since we must require $y(b) = \beta$, this gives us a equation to find λ ,

$$\lambda \bar{y}(b) + (1 - \lambda) \tilde{y}(b) = \beta, \quad \Rightarrow \quad \lambda = \frac{\beta - \tilde{y}(b)}{\bar{y}(b) - \tilde{y}(b)}. \quad (5)$$

Conclusion. The $y(x)$ given as

$$y(x) = \lambda \cdot \bar{y}(x) + (1 - \lambda) \cdot \tilde{y}(x)$$

with λ given as

$$\lambda = \frac{\beta - \tilde{y}(b)}{\bar{y}(b) - \tilde{y}(b)}$$

is the solution of the BVP in (1).

Practical issues.

One can solve for \bar{y} and \tilde{y} as initial value problems, even simultaneously.
Let

$$y_1 = \bar{y}, \quad y_2 = \bar{y}', \quad y_3 = \tilde{y}, \quad y_4 = \tilde{y}',$$

then

$$\begin{pmatrix} y'_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{pmatrix} = \begin{pmatrix} y_2 \\ u + vy_1 + wy_2 \\ y_4 \\ u + vy_3 + wy_4 \end{pmatrix}, \quad \text{IC: } \begin{pmatrix} y_1(a) \\ y_2(a) \\ y_3(a) \\ y_4(a) \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \\ \alpha \\ 1 \end{pmatrix}.$$

This is a 4×4 system of first order ODEs, which could be solved in Matlab with efficient ODE solvers. The values $\bar{y}(b), \tilde{y}(b)$ will be the last element in the vector y_1, y_3 , respectively.

Some extensions of Linear Shooting

Case 1. We now consider the effect of different boundary conditions.

$$y''(x) = u(x) + v(x)y(x) + w(x)y'(x), \quad y(a) = \alpha, \quad y'(b) = \beta.$$

A same shooting method can be designed, with minimum adjustment for the boundary condition at $x = b$.

The function y in the general algorithm will satisfy the differential equation as well as the boundary condition at $x = a$.

For the boundary condition at $x = b$, we must require

$$y'(b) = \lambda \bar{y}'(b) + (1 - \lambda) \tilde{y}'(b) = \beta, \quad \Rightarrow \quad \lambda = \frac{\beta - \tilde{y}'(b)}{\bar{y}'(b) - \tilde{y}'(b)}.$$

Case 2. Consider a higher order linear equation

$$y''' = f(x, y, y', y''), \quad y(a) = \alpha, \quad y'(a) = \gamma, \quad y(b) = \beta. \quad (1)$$

Here $f(x, y, y', y'')$ is an affine function in y, y', y'' .

A shooting method can be designed as follows. Let \bar{y} and \tilde{y} solve the same equation (1), but with initial conditions:

$$\bar{y}(a) = \alpha, \quad \bar{y}'(a) = \gamma, \quad \bar{y}''(a) = 0. \quad (2)$$

$$\tilde{y}(a) = \alpha, \quad \tilde{y}'(a) = \gamma, \quad \tilde{y}''(a) = 1. \quad (3)$$

Assume now we solved both equations (2) and (3), and the values $\bar{y}(b)$ and $\tilde{y}(b)$ are computed. Let

$$y(x) = \lambda \cdot \bar{y}(x) + (1 - \lambda) \cdot \tilde{y}(x) \quad (4)$$

where λ is a constant to be determined, such that $y(x)$ in (4) becomes the solution for (1).

It is easy to check that y solves the equation in (1), and satisfies the boundary conditions $y(a) = \alpha, y'(a) = \gamma$, due to the linear properties. It remains to check the last boundary condition at $x = b$.

At $x = b$, we have

$$y(b) = \lambda \bar{y}(b) + (1 - \lambda) \tilde{y}(b) = \beta.$$

which give the same formula to compute λ , i.e,

$$\lambda = \frac{\beta - \tilde{y}(b)}{\bar{y}(b) - \tilde{y}(b)}.$$

Non-linear Shooting.

We now consider the general nonlinear equation

$$y'' = f(x, y, y'), \quad y(a) = \alpha, \quad y(b) = \beta$$

Let \tilde{y} solve the IVP

$$\tilde{y}'' = f(x, \tilde{y}, \tilde{y}'), \quad \tilde{y}(a) = \alpha, \quad \tilde{y}'(a) = z. \quad (1)$$

Note that the condition $\tilde{y}'(a) = z$ is our guess.

The solution of (1) depends on z . Denote

$$\tilde{y}(b) \doteq \phi(z),$$

where ϕ is a non-linear function denoting the relation on how the value $\tilde{y}(b)$ depend on z . We need to find the value z such that

$$\phi(z) = \beta, \quad \Rightarrow \quad \phi(z) - \beta = 0.$$

Since $\phi(z)$ is a non-linear function, we need to find a root for the above nonlinear equation. One can use secant method!

The algorithm goes as follows.

- (1). Choose some initial guess z_1, z_2 , and compute the values

$$\phi_1 = \phi(z_1), \quad \phi_2 = \phi(z_2)$$

- (2) Then, the next value z_3 could be computed by a secant step:

$$z_3 = z_2 + (\beta - \phi_2) \cdot \frac{z_2 - z_1}{\phi_2 - \phi_1}.$$

- (3). One can then iterate and get values z_4, z_5, \dots until converges, for example, until $|\phi(z_n) - \beta| \leq \text{tol}$.

Remark:

The nonlinear shooting could be used on linear problem. In that case, one iteration is enough.

Extensions to other types of boundary conditions, as well as higher order nonlinear equations, can be made in a similar way as we did for the linear case. One needs to adopt a secant iteration around the shooting. Students are encouraged to working out the details on their own.

FDM: Finite Difference Methods

We consider the linear problem, in the general form, with Dirichlet boundary condition

$$y''(x) = u(x) + v(x)y(x) + w(x)y'(x), \quad y(a) = \alpha, \quad y(b) = \beta. \quad (1)$$

Discretize the domain: Choose n , make a uniform grid:

$$h = \frac{b - a}{n}, \quad x_i = a + ih, \quad i = 0, 1, 2, \dots, n, \quad x_0 = a, \quad x_n = b$$

Goal: Find approximations $y_i \approx y(x_i)$.

Tool: finite difference approximation to the derivatives:

$$y'(x_i) \approx \frac{y(x_{i+1}) - y(x_{i-1})}{x_{i+1} - x_{i-1}} = \frac{y_{i+1} - y_{i-1}}{2h},$$

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}.$$

Plug these into the ODE $y''(x) = u(x) + v(x)y(x) + w(x)y'(x)$, we get

$$\frac{1}{h^2} (y_{i+1} - 2y_i + y_{i-1}) = u_i + v_i y_i + \frac{w_i}{2h} (y_{i+1} - y_{i-1}),$$

for $i = 1, 2, \dots, n-1$, where we used the notation

$$u_i = u(x_i), \quad v_i = v(x_i), \quad w_i = w(x_i)$$

We can clean up a bit, and get

$$-(1 + \frac{h}{2}w_i)y_{i-1} + (2 + h^2v_i)y_i - (1 - \frac{h}{2}w_i)y_{i+1} = -h^2u_i. \quad (2)$$

Calling

$$a_i = -(1 + \frac{h}{2}w_i), \quad d_i = (2 + h^2v_i), \quad c_i = -(1 - \frac{h}{2}w_i), \quad b_i = -h^2u_i;$$

discrete equations (2) can be written in a simpler way

$$a_i y_{i-1} + d_i y_i + c_i y_{i+1} = b_i, \quad i = 1, 2, \dots, n-1. \quad (3)$$

By the boundary conditions $y_0 = \alpha, y_n = \beta$, the first and last equation in (3) become

$$\begin{aligned} d_1 y_1 + c_1 y_2 &= b_1 - a_1 \alpha, \\ a_{n-1} y_{n-2} + d_{n-1} y_{n-1} &= b_{n-1} - c_{n-1} \beta. \end{aligned}$$

The discrete equations

$$a_i y_{i-1} + d_i y_i + c_i y_{i+1} = b_i, \quad i = 1, 2, \dots, n-1,$$

lead to a tri-diagonal system of linear equations.

$$A\vec{y} = \vec{b}$$

with

$$\begin{pmatrix} d_1 & c_1 & & & \\ a_2 & d_2 & c_2 & & \\ \ddots & \ddots & \ddots & \ddots & \\ & a_{n-2} & d_{n-2} & c_{n-2} & \\ & a_{n-1} & d_{n-1} & & \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} b_1 - a_1 \alpha \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} - c_{n-1} \beta \end{pmatrix}$$

One desirable property to have is that the coefficient matrix A being diagonal dominant. In this case, Gaussian elimination without pivoting can be used to solve the linear system. Also, any of the iterative solvers we have learned will converge with any initial guess.

We see that A is strictly diagonal dominant if $|d_i| > |a_i| + |c_i|$, i.e., if

$$|2 + h^2 v_i| > |1 + hw_i/2| + |1 - hw_i/2|$$

which holds if we assume that every term in the absolute value sign above is positive.

We may now require

$$v(x) > 0, \quad h \leq \frac{2}{\max_x |w(x)|}.$$

Example

Set up the FDM for the problem

$$y'' = -4(y - x), \quad y(0) = 0, \quad y(1) = 2.$$

Note that the exact solution is $y(x) = (1/\sin 2) \sin 2x + x$.

Answer. Fix an n , we make a uniform grid:

$$h = \frac{1}{n}, \quad x_i = ih, \quad i = 0, 1, 2, \dots, n.$$

Central Finite Difference for the second derivative $y''(x_i)$ gives us

$$y''(x_i) \approx \frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) = -4y_i + 4x_i.$$

After some cleaning up, we get

$$y_{i-1} - (2 - 4h^2)y_i + y_{i+1} = 4h^2x_i, \quad i = 1, 2, \dots, n-1,$$

with boundary conditions

$$y_0 = 0, \quad y_n = 2.$$

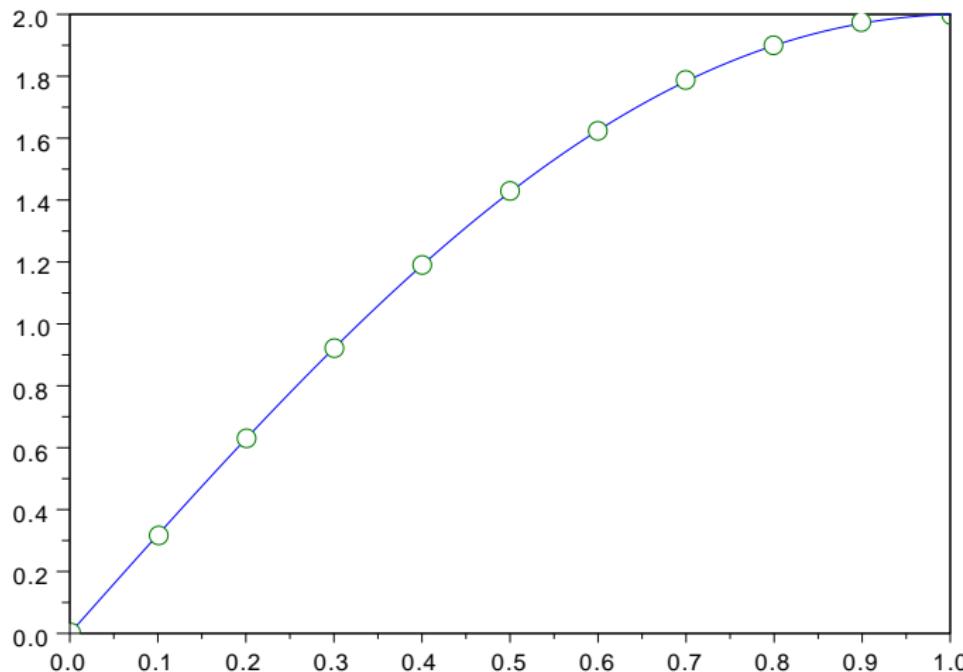
We end up with the tri-diagonal system $A\vec{y} = \vec{b}$:

$$A = \begin{pmatrix} -2 + 4h^2 & 1 & & & \\ 1 & -2 + 4h^2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 + 4h^2 & 1 \\ & & & 1 & -2 + 4h^2 \end{pmatrix}$$

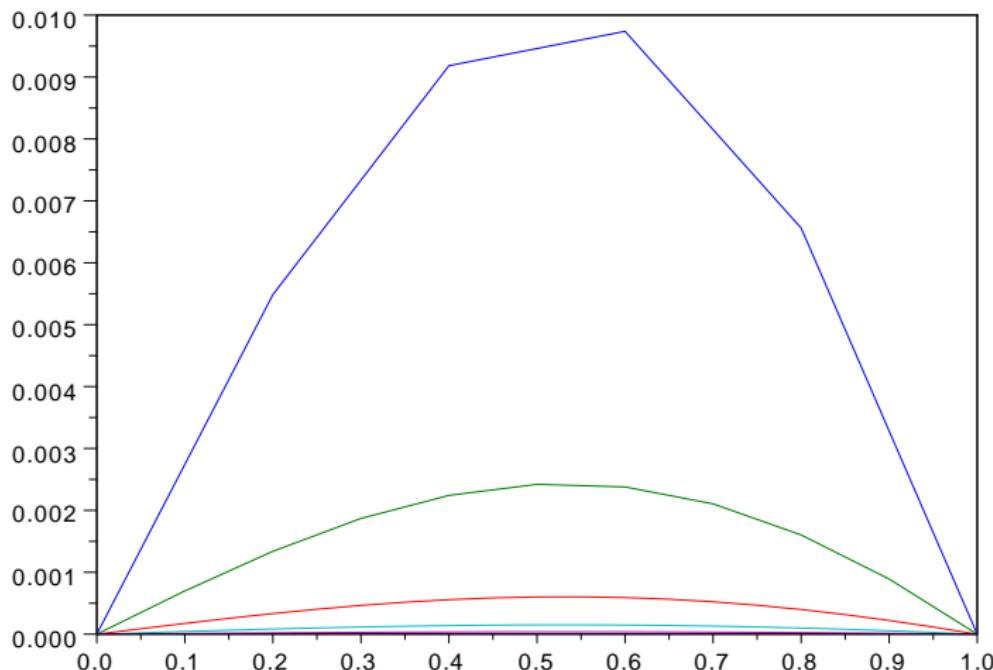
$$\vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 4h^2x_1 - a \\ 4h^2x_2 \\ \vdots \\ 4h^2x_{n-2} \\ 4h^2x_{n-1} - b \end{pmatrix}.$$

The system is “almost” diagonal dominant. It can be solved by Gaussian Elimination efficiently. It could also be solved by any of our iterative methods, and they will all converge.

The graph of the approximate solution with $n = 10$, together with the exact solution:

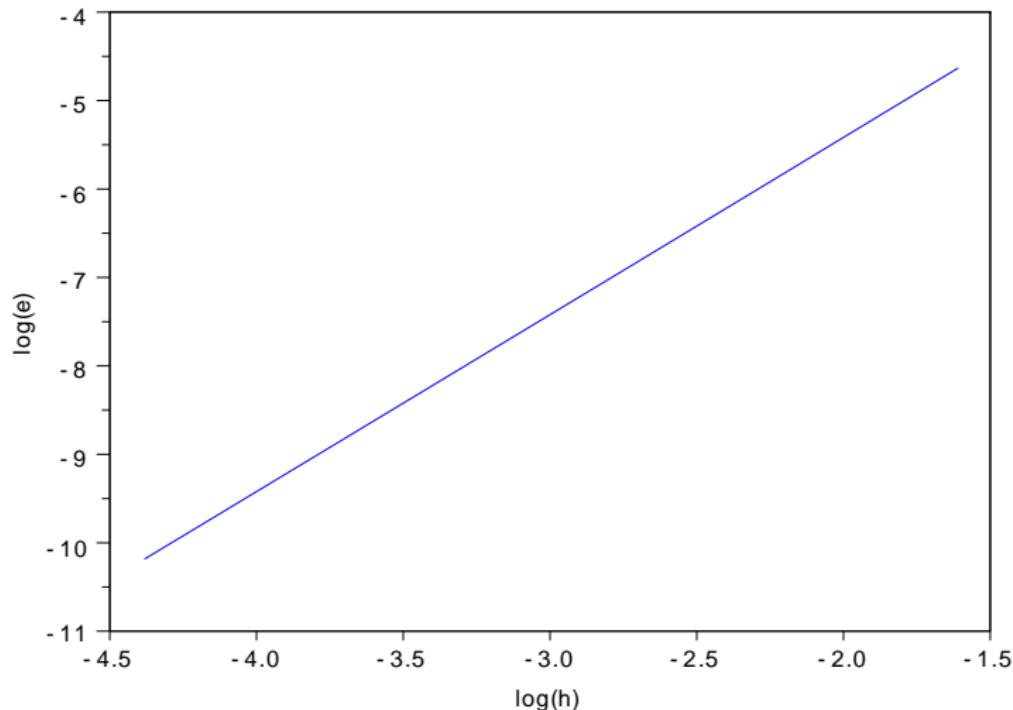


Plot of the error $e_i = y_i - y(x_i)$, for $n = 5, 10, 20, 40, 80$:



We see clearly that error decreases as n increases.

Plot for the $\log(\max(e_i))$ against $\log(h)$:



We see that we get a straight line with slope 2.

A straight line with slope 2 in a $\log(e) - \log(h)$ plot indicates a second order method.

Indeed, assume the method is of order m , then the error is approximately

$$e = Ch^m$$

for some constant C .

Taking log on both sides, we get

$$\log(e) = \log(C) + m \log(h),$$

where m is exactly the slope in the $\log(e)$ vs $\log(h)$ plot.

Neumann Boundary conditions.

Neumann Boundary condition is when the derivative of the unknown is given at the boundary. For example, we consider the Poisson equation in 1D:

$$u''(x) = f(x), \quad u'(0) = a, \quad u(1) = b. \quad (1)$$

Note the condition at $x = 0$ is given as the derivative of the unknown $u(x)$.

Uniform grid:

Fix an N , let $h = 1/N$ and $x_i = ih$ for $i = 0, 1, 2, \dots, N$, and let $u_i \approx u(x_i)$ be the approximation.

We now have N unknowns, namely u_0, u_1, \dots, u_{N-1} . We also have $u_N = b$ which is the Dirichlet boundary condition.

We set up the finite difference scheme

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = f(x_i), \quad \Rightarrow \quad u_{i-1} - 2u_i + u_{i+1} = h^2 f(x_i), \quad (2)$$

which holds for $i = 1, 2, \dots, N - 1$.

Discussion:

Since the central finite difference approximation to $u''(x)$ is second order, we want also to approximate the boundary condition $u'(0) = a$ with a second order finite difference.

The central finite difference for $u'(0)$ is second, but it requires information at $x = -h$.

To handle this, we add an additional grid point outside the domain, $x_{-1} = x_0 - h = -h$.

This point is called *ghost boundary*.

Writing $u_{-1} \approx u(x_{-1})$, we now write out the central finite difference for the boundary condition:

$$\frac{u_1 - u_{-1}}{2h} = a, \quad \Rightarrow \quad u_{-1} = u_1 - 2ha. \quad (3)$$

We also write out (2) at $i = 0$:

$$u_{-1} - 2u_0 + u_1 = h^2 f(x_0). \quad (4)$$

Plugging (3) into (4), we get the discrete equation for $i = 0$

$$u_1 - 2ha - 2u_0 + u_1 = h^2 f(x_0), \quad \Rightarrow \quad -2u_0 + 2u_1 = h^2 f(x_0) + 2ha. \quad (5)$$

The equation $i = N - 1$ is slightly different due to the boundary condition $u_N = b$:

$$u_{N-2} - 2u_{N-1} = h^2 f(x_{N-1}) - b. \quad (6)$$

Collecting all the equation with $i = 0, 1, 2, \dots, N - 1$, we obtain the following tri-diagonal system of linear equations:

$$\begin{pmatrix} -2 & 2 & & & \\ 1 & -2 & 1 & & \\ \ddots & \ddots & \ddots & & \\ & 1 & -2 & 1 & \\ & & 1 & -2 & \end{pmatrix} \cdot \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} = \begin{pmatrix} h^2 f(x_0) + 2ha \\ h^2 f(x_1) \\ \vdots \\ h^2 f(x_{N-2}) \\ h^2 f(x_{N-1}) - b \end{pmatrix}.$$

Remark.

A Neumann boundary condition at $x = 1$ would be treated in a completely similar way, by adding a ghost boundary point $x_{N+1} = x_N + h$. We omit the details. Students are encouraged to work out the details.

Robin Boundary conditions

For some problem, one might have boundary conditions involving both the unknown and its derivative. These are called **Robin boundary conditions**.

Consider again the 1D Poisson, but with Robin boundary condition at $x = 0$:

$$u''(x) = f(x), \quad u'(0) - u(0) = \gamma, \quad u(1) = b. \quad (1)$$

Using again the ghost boundary x_{-1} , we can approximate the Robin condition by a second order central finite difference

$$\frac{u_1 - u_{-1}}{2h} - u_0 = \gamma, \quad \Rightarrow \quad u_{-1} = u_1 - u_0 - 2h\gamma.$$

Plugging this into the finite difference scheme at $i = 0$, we get the discrete equation for $i = 0$

$$(u_1 - u_0 - 2h\gamma) - 2u_0 + u_1 = h^2 f(x_0), \quad \Rightarrow \quad -3u_0 + 2u_1 = h^2 f(x_0) + 2h\gamma.$$

The rest of the equations remain unchanged.

We end up with the following tri-diagonal system of linear equations.

$$\begin{pmatrix} -3 & 2 & & \\ 1 & -2 & 1 & \\ & \ddots & \ddots & \ddots \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix} \cdot \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} = \begin{pmatrix} h^2 f(x_0) + 2h\gamma \\ h^2 f(x_1) \\ \vdots \\ h^2 f(x_{N-2}) \\ h^2 f(x_{N-1}) - b \end{pmatrix}.$$

CMPSC/Math 451, Numerical Computation

Wen Shen

Department of Mathematics, Penn State University

Finite Difference Methods for Some Partial Differential Equations

In this chapter we consider several well-known second order linear partial differential equations, and study finite difference approximations.

These include:

- Laplace Equation in 2D, Poisson Equation in 2D, with Dirichlet and/or Neumann boundary conditions.
- Heat equation in 1D, or other simple parabolic type PDE in 1D, with various boundary conditions.

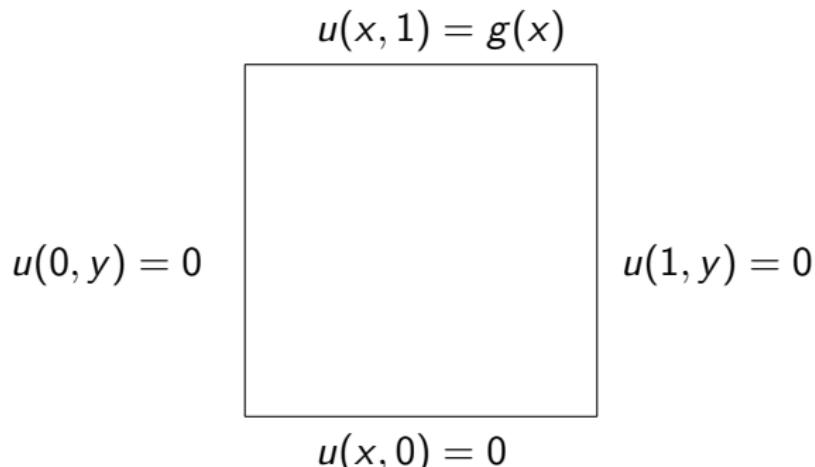
Laplace Equation in 2D: Finite Difference Methods

We consider the Laplace equation on a unit square

$$u_{xx} + u_{yy} = 0, \quad (0 < x < 1, \quad 0 < y < 1),$$

with boundary Dirichlet conditions, homogeneous on three sides, namely

$$\begin{aligned} u(0, y) &= 0, & u(1, y) &= 0, & (0 \leq y \leq 1) \\ u(x, 0) &= 0, & u(x, 1) &= g(x), & (0 \leq x \leq 1) \end{aligned}$$



We use a uniform grid in both x and y direction.

Choose N , the number of intervals in x and y directions, and let

$$h = 1/N, \quad x_i = ih, \quad y_j = jh, \quad i, j = 0, 1, 2, \dots, N.$$

Here h is the grid size.

Our goal is to find approximate solution at the grid points only, i.e., we want to find $u_{i,j} \approx u(x_i, y_j)$.

Finite differences approximations to the partial derivatives at the grid point (x_i, y_j) are

$$u_{xx}(x_i, y_j) \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2},$$

$$u_{yy}(x_i, y_j) \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2}.$$

Plugging these approximations into the Laplace equation at the point (x_i, y_j) , we get

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = 0.$$

After some simplification, we get

$$u_{i-1,j} - 4u_{i,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = 0, \quad (1)$$

We see that the discrete equation (1) holds at every grid point (x_i, y_j) that is not on the boundary, i.e., $i = 1, 2, \dots, N-1, j = 1, 2, \dots, N-1$.

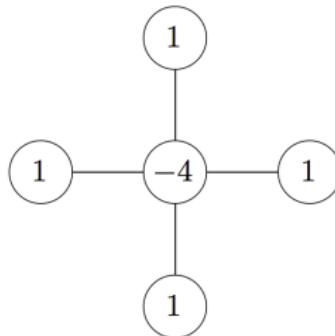
This gives us $(N-1) \times (N-1)$ equations.

The discrete boundary conditions are

$$u_{0,j} = 0, \quad u_{N,j} = 0, \quad 0 \leq j \leq N$$

$$u_{i,0} = 0, \quad u_{i,N} = g_i = g(x_i), \quad 0 \leq i \leq N.$$

Each discrete equation in (1) involves 5 grid points, which forms a **computational stencil**:



Note that only the interior points will be the unknowns, and the number of unknowns is $(N - 1)^2$. which matches the number of the equations.

This leads to a $(N - 1)^2 \times (N - 1)^2$ system of linear equations. – next video.

System of Linear Equations for Discrete Laplace Equation

The $(N - 1)^2 \times (N - 1)^2$ system of linear equations can be written as $A\vec{v} = \vec{b}$ where $\vec{v} = (v_1, v_2, \dots, v_{(N-1)^2})^T$.

To form the unknown vector \vec{v} out of the double indexed data u_{ij} , we go through a process called **natural ordering**.

We sweep through x -direction, then y -direction, as follows:

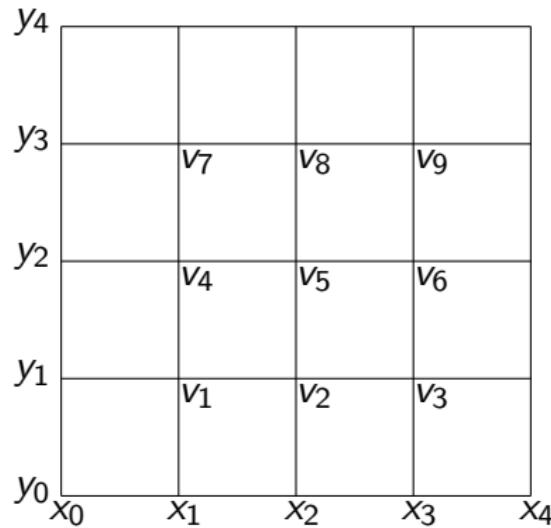
$$\begin{aligned}\vec{v} &= (v_1, v_2, \dots, v_{(N-1)^2})^T \\ &= (u_{1,1}, u_{2,1}, \dots, u_{N-1,1}, u_{1,2}, u_{2,2}, \dots, u_{N-1,2}, \dots, u_{N-1,N-1})^T\end{aligned}$$

Take for example $N = 4$. Total number unknowns is 9, and the unknown vector is

$$v_1 = u_{1,1}, \quad v_2 = u_{2,1}, \quad v_3 = u_{3,1},$$

$$v_4 = u_{1,2}, \quad v_5 = u_{2,2}, \quad v_6 = u_{3,2},$$

$$v_7 = u_{1,3}, \quad v_8 = u_{2,3}, \quad v_9 = u_{3,3}.$$



We can form the equations by using the computational stencil, placing it over each interior grid point, and write out the equation.

We must take consideration of the boundary conditions, and move them to the righthand side.

We obtain 9 equations:

$$\begin{aligned}-4v_1 + v_2 + v_4 &= 0 \\v_1 - 4v_2 + v_3 + v_5 &= 0 \\v_2 - 4v_3 + v_6 &= 0 \\v_1 - 4v_4 + v_5 + v_7 &= 0 \\v_2 + v_4 - 4v_5 + v_6 + v_8 &= 0 \\v_3 + v_5 - 4v_6 + v_9 &= 0 \\v_4 - 4v_7 + v_8 &= -g(x_1) \\v_5 + v_7 - 4v_8 + v_9 &= -g(x_2) \\v_6 + v_8 - 4v_9 &= -g(x_3)\end{aligned}$$

We can write them out in the following more organized way:

$$\begin{array}{ccc|ccc|ccc}
 -4v_1 & +v_2 & & +v_4 & & & & & = & 0 \\
 v_1 & -4v_2 & +v_3 & & +v_5 & & & & = & 0 \\
 & v_2 & -4v_3 & & +v_6 & & & & = & 0 \\
 \hline
 v_1 & & & -4v_4 & +v_5 & & +v_7 & & = & 0 \\
 & v_2 & & +v_4 & -4v_5 & +v_6 & +v_8 & & = & 0 \\
 & & v_3 & +v_5 & -4v_6 & & +v_9 & = & 0 \\
 \hline
 & & & v_4 & & -4v_7 & +v_8 & = & -g(x_1) \\
 & & & & v_5 & +v_7 & -4v_8 & +v_9 & = & -g(x_2) \\
 & & & & & v_6 & +v_8 & -4v_9 & = & -g(x_3)
 \end{array}$$

This gives us a linear system $A\vec{v} = \vec{b}$, where

$$A = \left(\begin{array}{ccc|ccc|ccc} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{array} \right), \quad \vec{b} = \left(\begin{array}{c} 0 \\ 0 \\ 0 \\ \hline 0 \\ 0 \\ 0 \\ \hline -g(x_1) \\ -g(x_2) \\ -g(x_3) \end{array} \right)$$

Note the tri-diagonal block-structure of the A matrix!
The matrix A is symmetric, and diagonal dominant.

In the general case, the same block-structure is preserved, if we form the unknown vector using natural ordering.

Let D be an $(N - 1) \times (N - 1)$ tri-diagonal matrix with -4 on the diagonal and 1 on the sup and sub diagonal.

Let I be an $(N - 1) \times (N - 1)$ identity matrix.

Then, the A matrix is block-tridiagonal, $(N - 1) \times (N - 1)$, block structured:

$$A = \begin{pmatrix} D & I & & & \\ I & D & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & D & I \\ & & & I & D \end{pmatrix}$$

The final dimension of A is $(N - 1)^2 \times (N - 1)^2$.

In the load vector b we will collect all boundary conditions.

Laplace equation with non-homogeneous Dirichlet BCs

Consider the Laplace equation with non-homogeneous Dirichlet boundary conditions on all 4 sides of the unit squares:

$$u(x, 1) = g(x)$$

$$u(0, y) = m(y)$$

$$u(1, y) = f(y)$$

$$u(x, 0) = h(x)$$

Since the boundary conditions only enter the load vector \vec{b} , the coefficient matrix A remains unchanged.

One may go through the grid again with the computational stencil, and collect all the boundary terms and move them to the load vector.

With $N = 4$, this gives us

		$g(x_1)$	$g(x_2)$	$g(x_3)$	
$m(y_3)$		v_7	v_8	v_9	$f(y_3)$
$m(y_2)$		v_4	v_5	v_6	$f(y_2)$
$m(y_1)$		v_1	v_2	v_3	$f(y_1)$
		$h(x_1)$	$h(x_2)$	$h(x_3)$	

$$\vec{b} = \begin{pmatrix} -h(x_1) - m(y_1) \\ -h(x_2) \\ -h(x_3) - f(y_1) \\ \hline -m(y_2) \\ 0 \\ -f(y_2) \\ \hline -g(x_1) - m(y_3) \\ -g(x_2) \\ -g(x_3) - f(y_3) \end{pmatrix}$$

$$N = 5 : \quad \vec{b} = \begin{pmatrix} -h(x_1) - m(y_1) \\ -h(x_2) \\ -h(x_3) \\ -h(x_4) - f(y_1) \\ \hline -m(y_2) \\ 0 \\ 0 \\ 0 \\ \hline -f(y_2) \\ -m(y_3) \\ 0 \\ 0 \\ 0 \\ \hline -f(y_3) \\ \hline -g(x_1) - m(y_4) \\ -g(x_2) \\ -g(x_3) \\ -g(x_4) - f(y_4) \end{pmatrix}.$$

The pattern shall be clear by now.

Poisson equation on a unit square with Dirichlet boundary condition

We now consider the Poisson equation on a unit square

$$u_{xx} + u_{yy} = \phi(x, y)$$

with Dirichlet boundary condition, given as

$$u(x, 1) = g(x)$$

$$u(0, y) = m(y)$$

$$u(1, y) = f(y)$$

$$u(x, 0) = h(x)$$

Finite difference discretization gives

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = \phi(x_i, y_j) = \phi_{ij} \quad (1)$$

so

$$u_{i-1,j} + u_{i+1,j} - 4u_{i,j} + u_{i,j-1} + u_{i,j+1} = h^2\phi_{ij}, \quad i, j = 1, 2, \dots, N-1 \quad (2)$$

We see that the left-hand side of the discrete equation is unchanged. The source term $h^2\phi_{ij}$ will only enter the load vector.

Take for example $N = 4$, we get

$$\vec{b} = \begin{pmatrix} h^2\phi_{1,1} - h(x_1) - m(y_1) \\ h^2\phi_{2,1} - h(x_2) \\ h^2\phi_{3,1} - h(x_3) - f(y_1) \\ \hline h^2\phi_{1,2} - m(y_2) \\ h^2\phi_{2,2} \\ h^2\phi_{3,2} - f(y_2) \\ \hline h^2\phi_{1,3} - g(x_1) - m(y_3) \\ h^2\phi_{2,3} - g(x_2) \\ h^2\phi_{3,3} - g(x_3) - f(y_3) \end{pmatrix}.$$

Laplace Equation with Neumann Boundary Condition

Consider the Laplace equation

$$u_{xx} + u_{yy} = 0$$

on a unit square, with boundary conditions

$$u(x, 1) = g(x)$$

$$u_x(0, y) = a(y)$$

$$u(1, y) = f(y)$$

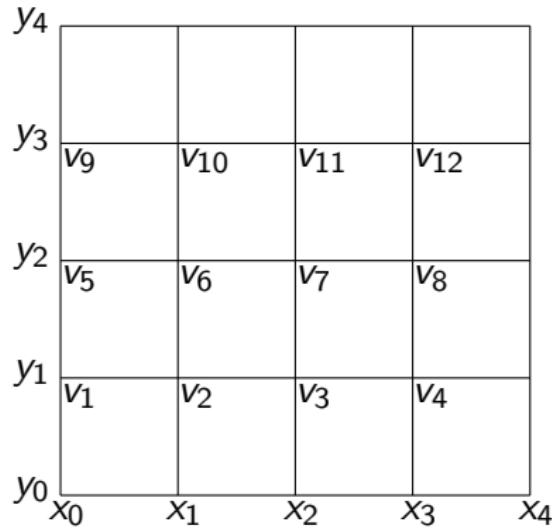
$$u(x, 0) = h(x)$$

We assign Neumann boundary condition on one side of the square.

This changes quite a bit the discretization.

Since now $u_{0,j}$ are also unknowns, we need to take them into the ordering.

The natural ordering is illustrated below:



In order to use a second order central finite difference for the Neumann boundary condition, we add a layer of ghost boundary, i.e.,

$$u_{-1,j}, \quad \text{for } j = 1, 2, \dots, N-1.$$

A central finite difference for the boundary condition can be

$$u_x(0, y_j) \approx \frac{u_{1,j} - u_{-1,j}}{2h} = a(y_j) = a_j, \quad \Rightarrow \quad u_{-1,j} = u_{1,j} - 2ha_j.$$

The discrete Laplace equation at $i = 0$ for any j gives

$$u_{-1,j} - 4u_{0,j} + u_{1,j} + u_{0,j-1} + u_{0,j+1} = 0.$$

Eliminating the ghost $u_{-1,j}$ from the previous two equations, we get

$$u_{1,j} - 2ha_j - 4u_{0,j} + u_{1,j} + u_{0,j-1} + u_{0,j+1} = 0,$$

$$\Rightarrow -4u_{0,j} + 2u_{1,j} + u_{0,j-1} + u_{0,j+1} = 2ha_j.$$

This leads to a system of linear equations $A\vec{v} = \vec{b}$.

For $N = 4$, we get

$$A = \left(\begin{array}{cccc|cccc|cccc} -4 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & -4 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 \end{array} \right)$$

We see that A still preserves the blocked tri-diagonal structure.

The load vector

$$\vec{b} = \begin{pmatrix} -h(x_0) + 2ha_1 \\ -h(x_1) \\ -h(x_2) \\ \frac{-h(x_3) - f(y_1)}{2ha_2} \\ 0 \\ 0 \\ \frac{-f(y_2)}{-g(x_0) + 2ha_3} \\ -g(x_1) \\ -g(x_2) \\ -g(x_3) - f(y_3) \end{pmatrix}.$$

The pattern for A and \vec{b} with larger value of N is now clear.

Heat Equation in 1D

We consider a simple heat equation, where $u(t, x)$ is the temperature in a rod:

$$u_t = u_{xx}, \quad 0 \leq x \leq 1$$

with boundary condition

$$u(t, 0) = 0, \quad u(t, 1) = 0, \quad t \geq 0$$

and initial condition

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

Set up the grid in x :

$$\Delta x = \frac{1}{M}, \quad x_j = j\Delta x, \quad j = 0, 1, 2, \dots, M$$

and the grid in t :

$$t_0 = 0, \quad t_n = n\Delta t, \quad n = 0, 1, 2, \dots$$

Goal: Find approximation to solution $u_j^n \approx u(t_n, x_j)$.

Tool: finite differences.

$$u_t(t_n, x_j) \approx \frac{u_j^{n+1} - u_j^n}{\Delta t}, \quad u_{xx}(t_n, x_j) \approx \frac{u_{j-1}^n - 2u_j^n + u_{j+1}^n}{\Delta x^2}$$

Forward-Euler (Explicit Euler):

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

We can clean up a bit. Writing $\gamma = \Delta t / \Delta x^2$, we get

$$u_j^{n+1} = \gamma u_{j-1}^n + (1 - 2\gamma)u_j^n + \gamma u_{j+1}^n, \quad (1)$$

with initial and boundary conditions

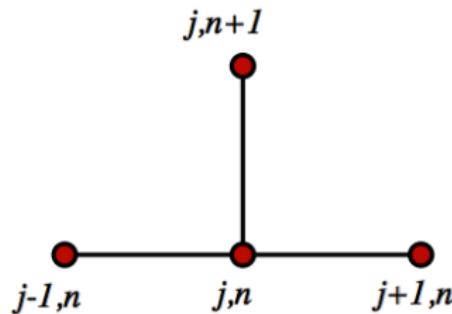
$$u_j^0 = f(x_j), \quad j = 0, 1, 2, \dots, M$$

and boundary conditions

$$u_0^n = 0, \quad u_M^n = 0, \quad n \geq 0.$$

The method is first order in time and second order in space, i.e., $\mathcal{O}(\Delta t, \Delta x^2)$.

The computational stencil:



Stability condition for explicit scheme

Maximum Principle of Heat Equation. The exact solution $u(t, x)$ of the heat equation satisfies the following maximum principle:

$$\min_{0 \leq y \leq 1} u(t_1, y) \leq u(t_2, x) \leq \max_{0 \leq y \leq 1} u(t_1, y) \quad (1)$$

for any $t_2 \geq t_1$.

In particular, (1) implies

$$\max_{0 \leq x \leq 1} |u(t_2, x)| \leq \max_{0 \leq x \leq 1} |u(t_1, x)|, \quad (2)$$

for any $t_2 \geq t_1$.

\Rightarrow The maximum value of $|u(t, x)|$ over x is non-increasing in time t .

Discrete Maximum principle.

It is desirable to require a discrete version of the maximum principle for our approximate solution, i.e.,

$$\max_j |u_j^{n+1}| \leq \max_j |u_j^n|, \quad \text{for every } n. \quad (3)$$

We now provide a sufficient condition for (3). Assume now

$$1 - 2\gamma \geq 0, \quad \text{i.e.} \quad \gamma \leq \frac{1}{2}, \quad \Rightarrow \quad \Delta t \leq \frac{1}{2} \Delta x^2. \quad (4)$$

This is called the **CFL stability condition**.

Then

$$\begin{aligned} |u_j^{n+1}| &\leq \gamma |u_{j-1}^n| + (1 - 2\gamma) |u_j^n| + \gamma |u_{j+1}^n| \\ &\leq \gamma \max_i |u_i^n| + (1 - 2\gamma) \max_i |u_i^n| + \gamma \max_i |u_i^n| \\ &= \max_i |u_i^n|. \end{aligned}$$

This means

$$|u_j^{n+1}| \leq \max_i |u_i^n|, \quad \text{for every } j.$$

Since this holds for all j , we conclude

$$\max_j |u_j^{n+1}| \leq \max_j |u_j^n|, \quad \text{for every } n.$$

The CFL stability condition

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

It actually puts a very strict constraint on the time step size Δt , especially when space grid size Δx is small.

For example, if $\Delta x = 10^{-3}$, then the time step size Δt must be $\Delta t \leq (\frac{1}{2})10^{-6}$, which is extremely small!

We are now forced to take many many time steps!

Backward-Euler (Implicit-Euler)

We instead use:

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

Writing $\gamma = \Delta t / \Delta x^2$, we can write

$$-\gamma u_{j-1}^{n+1} + (1 + 2\gamma)u_j^{n+1} - \gamma u_{j+1}^{n+1} = u_j^n, \quad (1)$$

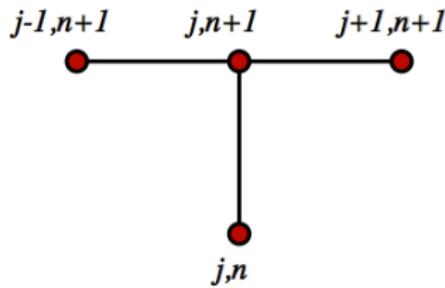
with initial and boundary conditions

$$u_j^0 = f(x_j), \quad u_0^n = 0, \quad u_M^n = 0.$$

(1) gives a tri-diagonal system to solve at every time step.

The method is first order in time and second order in space, i.e., of $\mathcal{O}(\Delta t, \Delta x^2)$.

Computational stencil:



Discrete Maximum Principle:

Scheme (1) could also be written as

$$(1 + 2\gamma)u_j^{n+1} = u_j^n + \gamma u_{j-1}^{n+1} + \gamma u_{j+1}^{n+1}.$$

$$\begin{aligned}(1 + 2\gamma)|u_j^{n+1}| &\leq |u_j^n| + \gamma |u_{j-1}^{n+1}| + \gamma |u_{j+1}^{n+1}| \\&\leq \max_i |u_i^n| + \gamma \max_i |u_i^{n+1}| + \gamma \max_i |u_i^{n+1}| \\&= \max_i |u_i^n| + 2\gamma \max_i |u_i^{n+1}|\end{aligned}$$

Since this holds for all j , it also holds when the left reaches the max. We conclude

$$\begin{aligned}(1 + 2\gamma) \max_j |u_j^{n+1}| &\leq \max_j |u_j^n| + 2\gamma \max_j |u_j^{n+1}| \\&\rightarrow \max_j |u_j^{n+1}| \leq \max_j |u_j^n|.\end{aligned}$$

Note that the Maximum Principle is satisfied for any choices of γ , i.e., any choices of grid size $\Delta t, \Delta x$!

This is called *unconditionally stable*.

Discussion:

- + Using implicit time step, we can take larger time step Δt , even though at each step it takes longer time to solve. In the end, it may still save us time!
- The method is of order $\mathcal{O}(\Delta t, \Delta x^2)$, meaning error $\approx C_1 \Delta t + C_2 \Delta x^2$. With large Δt , the error is larger.

Wish: A second order method both in time and space, and unconditionally stable.

Crank-Nicholson time step

Both explicit and implicit Euler are first order in time. Since the space discretization is second order, it would be desirable to get a method that is also second order in time.

Here we introduce a method that is also second order in time. We use

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

This is like central finite difference for the time derivative.

Writing $r = \Delta t / (2\Delta x^2)$, (note this is different from γ !), and cleaning up a bit, we get

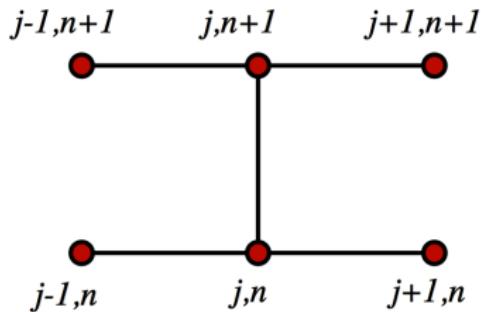
$$-ru_{j-1}^{n+1} + (1 + 2r)u_j^{n+1} - ru_{j+1}^{n+1} = ru_{j-1}^n + (1 - 2r)u_j^n + ru_{j+1}^n \quad (1)$$

with initial and boundary conditions

$$u_j^0 = f(x_j), \quad u_0^n = 0, \quad u_M^n = 0.$$

Note that scheme (1) gives a tri-diagonal system to solve at every time step.

Computational stencil:



This method is second order in both time and space, i.e., of $\mathcal{O}(\Delta t^2, \Delta x^2)$.

One can prove that Crank-Nicolson's method is *unconditionally stable*, although the stability is NOT with respect to the discrete Maximum Principle.