Math 375: Lecture notes

Professor Monika Nitsche

September 21, 2011

Contents

Syllabus for Fall 2010

1. MATLAB (see Tutorial on web)

Lecture 1 (Mon Aug 23) : Vectors, plotting, matrix operations.
Lecture 2 (Wed Aug 25) : Scripts, label plots, printing tables/figs. If, for, while statements.
Lecture 3 (Fri Aug 27) : Functions. Timing code. Vectorizing. Memory.

2. COMPUTING FUNDAMENTALS

Lecture 4 (Mon Aug 30) : What affects execution time? Vectorizing, initializing, operation counts. Examples: Horners algorithm, Taylor series. Big-O notation. (§0.1)
Lecture 5 (Wed Sep 1) : Binary numbers. Floating point representation. (§0.2-0.3)
Lecture 6 (Fri Sep 3) : Loss of significance. (§0.4)

3. NONLINEAR EQUATIONS

Lecture 7 (Wed Sep 8) : Bisection method, §1.1
Lecture 8 (Fri Sep 10) : Fixed point iteration, §1.2
Lecture 9 (Mon Sep 13): Fixed point iteration, §1.2
Lecture 10 (Wed Sep 15): Ill-conditioned problems, §1.3. Newton's Method, §1.4
Lecture 11 (Fri Sep 17) : Newton's Method, §1.4. Secant Method. §1.5

4. SOLVING LINEAR SYSTEMS

Lecture 12 (Mon Sep 20): Gauss Elimination, §2.1
Lecture 13 (Wed Sep 22): LU Decompo, §2.2
Lecture 14 (Fri Sep 24) : EXAM 1
Lecture 15 (Mon Sep 27): PLU Decomposition, §2.4
Lecture 16 (Wed Sep 29): Conditioning, §2.3
Lecture 17 (Fri Sep 31) : Iterative methods: Example and convergence criteria, §2.5
Lecture 18 (Mon Oct 4): Iterative methods: Jacobi
Lecture 19 (Wed Oct 6): Iterative methods: Gauss-Seidel

5. INTERPOLATION

Lecture 20 (Fri Oct 8) : Polynomial interpolation. Example.
Lecture 21 (Mon Oct 11) Polynomial interpolation. Lagrange approach.
Lecture 22 (Wed Oct 13) Polynomial interpolation. Vandermonde approach.
FALL BREAK
Lecture 23 (Mon Oct 18) Polynomial interpolation. Newton approach.
Lecture 24 (Wed Oct 20) Polynomial interpolation. Interpolation error.
Lecture 25 (Fri Oct 22) : Polynomial interpolation. Runge phenomena, Chebishev points.
Lecture 26 (Mon Oct 25) Spline: linear splines, cubic splines, derivation. §3.4.
Lecture 27 (Wed Oct 27) Cubic spline: derivation, MATLAB codes. §3.4.
Lecture 28 (Fri Oct 29) : Cubic spline: MATLAB codes, examples. §3.4.

6. TRIG INTERPOLANTS AND FOURIER TRANSFORM

Lecture 29	(Mon Nov 1) :	Trig interpolation: fourier coefficients, examples.
Lecture 30	(Wed Nov 3) :	Trig interpolation: Derive DFT, IDFT. Examples. §10.2
Lecture 31	(Fri Nov 5) :	Trig interpolation: Fourier coefficients and smoothness of functions.

7. LEAST SQUARES

Lecture 32	(Mon Nov 8) :	Least Squares: normal equations. §4.1
Lecture 33	(Wed Nov 10) :	Approximating data using models. §4.2
Lecture 34	(Fri Nov 12) :	QR decomposition. §4.3

8. NUMERICAL INTEGRATION

Lecture 35	(Mon Nov 15) :	Numerical Integration: Newton-Cotes. §5.2
Lecture 36	(Wed Nov 17) :	Numerical Integration: Composite N-C. §5.3
Lecture 37	(Fri Nov 19) :	REVIEW
Lecture 38	(Mon Nov 22) :	EXAM 2
Lecture 39	(Wed Nov 24) :	Numerical Integration: Mac-Laurin Formula for Trapezoid rule error. Review
THANKSGIVING BREAK		
Lecture 40	(Mon Nov 29) :	Numerical Integration: Gauss Quadrature. §5.2

9. NUMERICAL DIFFERENTIATION

Lecture 41	(Wed Dec 1) :	Numerical Differentiation. Truncation and Roundoff. §5.1

10. ORDINARY DIFFERENTIAL EQUATIONS

Lecture 42	(Fri Dec 1) :	Euler's Method. MATLAB Algorithm. S 6.1
Lecture 43	(Mon Dec 3) :	Local and global truncation errors. §6.2
Lecture 44	(Wed Dec 5) :	Euler's Method for systems. §6.3
Lecture 45	(Fri Dec 7) :	RK Methods. §6.4

1 MATLAB Basics

1.1 Example: Plotting a function

Starting MATLAB:

```
Windows: search for MATLAB icon or link and click
Linux: % ssh linux.unm.edu
       % matlab
       or
       % matlab -nojvm
```

Sample MATLAB code illustrating several Matlab features; code to plot the graph of $y = \sin(2\pi x)$, $x \in [0,1]$: What is really going on when you use software to graph a function?

1. The function is sampled at a set of points x_k to obtain $y_k = f(x_k)$.
2. The points (x_k, y_k) are then plotted together with some interpolant of the data (piecewise linear or a smoother curve such as splines of Bezier curves).

In MATLAB you specify x_k, then compute y_k. The command `plot(x,y)` outputs a piecewise linear interpolant of the data.

```
% Set up gridpoints x(k)
x=[0.0, 0.1,0.2,0.3,0.4...
   0.5,0.6,0.7,0.8,0.9,1.0];
% Set up function values y(k)
n=length(x);
y=zeros(1,n);
for k=1:n
  y(k)=sin(2*pi*x(k));  %pi is predefined
end
% plot a piecewise linear interpolant to the points (x(k),y(k))
plot(x,y)
```

Notes:

(1) The % sign denotes the begining of a comment. Code is well commented!

(2) The continuation symbol is ...

(3) The semicolon prevents displaying intermediate results (try it, what happens if you omit it?)

(4) `length,zeros,sin,plot` are built in MATLAB functions. Later on we will write our own functions.

(5) In Matlab variables are defined when they are used. Reason for `y=zeros(1,n)`: allocate amount of memory for variable y; initialize. How much memory space is allocated to y if that line is absent, as you step through the loop? Why is `zeros` not used before defining x?

(6) In Matlab all variables are matrices. Column vectors are nx1, row vectors are 1xn, scalars are 1x1 matrices. What is output of `size(x)`?

(7) All vectors/matrices are indexed starting with 1. What is `x(1), x(2), x(10), x(0)`?

(8) Square brackets are used to define vectors. Round brackets are used to access entries in vectors.

(9) Note syntax of `for` loop.

(10) What is the output of

```
plot(x,y,'*')
plot(x,y,'*-')
plot(x,y,'*-r')
```

See Tutorial for further plotting options, or `help plot`. Lets create a second vector $z = cos(2\pi x)$ and plot both y vs x in red dashed curve with circle markers, and z in green solid curve with crosses as markers. Then use default. But first:

1.2 Scripts

MATLAB Script m-file: A file (appended with .m) stored in a directory containing MATLAB code. From now on I will write all code in scripts so I can easily modify it. To write above code in script:

```
make a folder with your name
if appropriate, make a folder in that folder for current howework/topic
in MATLAB, go to box next to "current directory" and find directory
click on "File/New/M-file", edit, name, save
or click on existing file to edit
execute script by typing its name in MATLAB command window
lets write a script containing above code
```

Save your work on Floppy or USBport, or ftp to other machine. No guarantees that your folders will be saved.

Now I'd like to modify this code and use more points to obtain a better plot by changing the line defining x, using spacing of 1/100 instead of 1/10. But how to set up x with 101 entries??

1.3 Setting up vectors

Row Vectors

- Explicit list

  ```
  x=[0 1 2 3 4 5];
  x=[0,1,2,3,4,5];
  ```

- Using a:increment:b. To set up a vector from x=a to x=b in increments of size h you can use

  ```
  x=a:h:b;
  ```

  Here you specify beginning and endpoints, and stepsize. Most natural to me. However, if (b-a) is not an integer multiple of stepsize, endpoint will be omitted. Note, if h omitted, default stepsize =1. What is?

  ```
  x=0:0.1:1;
  x=0:5;
  ```

  We already used this notation in the for loop!

- Using linspace

  ```
  x=linspace(a,b,n);
  ```

  Use linspace to set up `x=[0 1 2 3 4]`, `x=[0,0.1,0.2,...,1]`, `x=[0,0.5,1,1.5,2]`, `x=a:h:b`

- Using for loops

Column Vectors

- Explicit list

  ```
  x=[0;1;2;3;4;5];
  ```

- Transpose row vector

```
x=[0:.1:1]';
```

Matrices

```
A=[1 1 1; 2 0 -1; 3 -1 2; 0 1 -1];
A=zeros(1,4); B=zeros(5,2); C=eye(3); D=ones(2,4);   %special matrices
```

How to access entry in second row, first column? What is A(1,1), A(2,0)?

Now we have a better way to define x using $h = 1/100$! Do it.

Try `x-y` where x row, y column!!

1.4 Vector and Matrix operations

We can replace the `for` loop in the example in §1.1 by

```
y=sin(2*pi*x);
```

The sine function is applied to a vector x and applies the sine operation to each entry in the vector. This is the same as

```
y(1:n)=sin(2*pi*x(1:n));
```

and thereby the same as (!)

```
k=1:n
y(k)=sin(2*pi*x(k));
```

Note that this looks almost like the `for` loop in the example in §1.1 but it is not a loop. All entries of `y` are set at once. We can now give a short version of the MATLAB code to plot the graph of $y = \sin(2\pi x)$:

```
x=0:.01:1;
y=sin(2*pi*x);
plot(x,y)
```

Since vectors are special cases of matrices, the above operation can also be applied to matrices. The statement `B=sin(A)` applies the sine function to every entry in A.

Setting up matrices. Before further addressing matrix operations, I want to mention another possibility to set up matrices. First note that the line `A=[1 1 1; 2 0 -1; 3 -1 2; 0 1 -1];` in §1.3 can be written as

```
    A=[1 1 1
       2 0 -1
       3 -1 2
       0 1 -1
     ];
```

(the commas and semicolons are not necessary in this form). If you create, possibly using FORTRAN or C, a file that contains a matrix of numbers, and enter `A=[` and `];` before and after, you can read these numbers in as a script. For example, I created (in FORTRAN) a file called t22dat.m that contains the following lines

```
%    t        xcore      ycore     ylmb     ulmb      uf        xf      jctr  n
a=[
 0.00000  0.000000  0.924700  0.81650  0.16004  0.590545  0.000000    2  400
 0.02500  0.002659  0.924680  0.81650  0.16013  0.590451  0.014763   93  400
 0.05000  0.005320  0.924620  0.81650  0.16039  0.590168  0.029521   92  400
 0.07500  0.007981  0.924520  0.81650  0.16081  0.589698  0.044270   93  400
 0.10000  0.010643  0.924380  0.81650  0.16141  0.589041  0.059004   92  400
 0.12500  0.013301  0.924190  0.81650  0.16216  0.588201  0.073720   93  400

...

59.90000 12.920807  0.822010  0.81665  0.21526  0.216202 13.726140   73 3761
59.92500 12.926320  0.822030  0.81665  0.21526  0.216193 13.731545   72 3764
59.95000 12.931837  0.822050  0.81665  0.21526  0.216184 13.736949   73 3768
59.97500 12.937345  0.822080  0.81665  0.21526  0.216175 13.742354   72 3772
];
```

(where the dots denote the remaining 2390 lines). I can now read this matrix into MATLAB and extract the vectors `t` and `ylmb` for example as follows

```
     t22dat         %this executes all 2402 lines in the script
     t=a(:,1)       %extract vector t, note colon notation
     ylmb=a(:,4)    %extract vector ylmb
     plot(t,ylmb)   %plot one vs the other
```

Note: entries of vectors are accessed using round brackets. Vectors are defined using square brackets.

More matrix operations.

```
    A+B                   %A,B need to have same dimensions
    A*B                   %A,B need to have proper dimensions (number
                          %    of columns of A=number of rows of B)
```

If `x` is a vector, what is

```
        x*x
```

? Answer: error!, Inner matrix dimensions dont agree. If `x` and `y` are 1xn row vectors,

```
        x*y'
```

Answer: the **inner product** $\sum_{k=1}^{N} x_k y_k$. For example to compute the Euclidean norm of x, $\sqrt{\sum_{k=1}^{N} x_k^2}$ you can use the one-line code

```
        euclidnormx=sqrt(x*x');
```

What is

```
        y'*x
```

Answer: an nxn matrix called the **outer product**.

What if you instead of plotting $y = sin(x)$ you want to plot $y = x^2$? Given a vector x you want to create a vector y whose entries are the square of the entries of x. The following

```
        x=0:.01:1;
        y=x*x;
or
        y=x^2;     %the hat is MATLABs symbol for exponentiation
```

wont work. Instead, to perform componentwise operations you need to replace * by .*, $\hat{}$ by .$\hat{}$, etc. for example:

```
y=x.^2;
y=A.^2;
y=1./x;
y=x.*z;    %where x,z are vectors/matrices of same length
```

Another useful built in MATLAB function is

```
s=sum(x);
```

Use the help command to figure out what it does.

1.5 Plotting

Labelling plots. In class we used:

```
plot(x,y,'r:x')        %options for color/line type/marker type
xlabel('this is xlabel')
ylabel('this is ylabel \alpha \pi')  %using greek alphabet
title('this is title')
text(0.1,0.3,'some text')        %places text at given coordinates
text(0.1,0.3,'some text','FontSize',20)    %optional override fontsize
set(gca,'FontSize',20)      %override default fontsize
axis([0,1,-2,2])          %sets axis
axis square               %square window
axis equal                %units on x- and y-axis have same length
figure(2)                 %creates new figure or accesses existing figure(2)
```

Plotting several functions on one plot. Suppose we created `x=0:.1:1; y1=sin(x); y2=cos(x);`. Two options. First one:

```
plot(x,y1,x,y2)                %using default colors/lines/markers
plot(x,y1,'b--0',x,y2,'r-')   %customizing
legend('sin(x)','cos(x)',3)   %nice way to label, what does entry '3' do?
```

Second one: using hold command:

```
plot(x,y1)
hold on
plot(x,y2)
hold off
```

Type `help hold` to see what this does. If you dont want to save the previous plot you can view consecutive plots using pause command in your script (what happens if pause is missing from below?)

```
plot(x,y1)
pause
plot(x,y2)
```

Creating several plots.

```
subplot(3,2,1)    %creates and accesses 1st subplot of a 3x2 grid
figure(3)         %creates and accesses new window
```

1.6 Printing data

Printing tables:

```
disp(' j   x  sin(x)')
for k=1:n
   disp(sprintf('%4d %5.1f %10.4f',k,x(k),y(k)));
end
```

What does the format `%3.1f` specify? Type `help sprintf` to see how to format integers, character strings, etc.

Printing figure:

```
print              %prints current figure to printer
print -deps name   %creates name.eps file (encapsulated postscript)
                   % of current figure and stores in directory
print -depsc name  %creates name.eps file in color of current figure
print -dpsc name   %creates name.ps (postscript) file in color
```

1.7 For loops

```
% The command for repeats statements for a specific number of times.
% The general form of the while statement is
```

```
FOR variable=expr
   statements
END

% expr is often of the form i0:j0 or i0:l:j0.
% Negative steps l are allowed.
% Example : What does this code do?

n = 10;
for i=1:n
   for j=1:n
      a(i,j) = 1/(i+j-1);
   end
end
```

1.8 While loops

```
% The command while repeats statements an indefinite number of times,
% as long as a given expression is true.
% The general form of the while statement is

WHILE expression
   statement
END

% Example 1: What does this code do?
x = 4;
y = 1;
n = 1;
while n<= 10;
   y = y + x^n/factorial(n);
   n = n+1;
end

% Remember to initialize $n$ and update its value in the loop!
```

1.9 Timing code

```
tic   % starts stopwatch
  statements
toc   % reads stopwatch
```

Exercise: Compare the following runtimes. What do you deduce?

```
tic; clear, for j=1:10000, x(j)=sin(j); end, toc
tic; clear, j=1:10000; x(j)=sin(j); toc
tic; for j=1:10000, x(j)=sin(j); end, toc
```

Exercise: Compare the following runtimes. What do you deduce?

```
clear
tic; for j=1:10000, sin(1.e8); end, toc

format long, pi2=2*pi; 1.e8/pi2
clear, alf = 1.e8-1.5915494e7;
tic; for j=1:10000, sin(alf); end, toc

clear
tic; for j=1:10000, sin(0.1); end, toc
```

1.10 Functions

MATLAB Functions are similar to functions in Fortran or C. They enable us to write the code more efficiently, and in a more readable manner.

The code for a MATLAB function must be placed in a separate `.m` file having the same name as the function. The general structure for the function is

```
function <Output parameters>=<Name of Function><Input Parameters>
% Comments that completely specify the function

<function body>
```

When writing a function, the following rules must be followed:

- Somewhere in the function body the desired value must be assigned to the output variable!
- Comments that completely specify the function should be given immediately after the `function` statement. The specification should describe the output and detail all input value assumptions.
- The lead block of comments after the `function` statement is displayed when the function is probed using `help`.
- All variables inside the function are local and are not part of the MATLAB workspace

Exercise 1: Write a function with input parameters x and n that evaluates the nth order Taylor approximation of $e^x$. Write a script that calls the function for various values of n and plots the error in the approximation.

Solution: The following code is written in a file called `ApproxExp.m`:

```
function y=ApproxExp(x,n);
% Output parameter: y (nth order Taylor approximation of $e^x$)
% Input parameters: x (scalar)
%                   n (integer)

sumo = 1;
for k=1:n
   sumo = sumo + x^k/factorial(k);
end
y = sumo;
```

(What does this code do? First, set k=1. Then k=2. Then k=3. etc. Write out the result after each time through loop.) A script that references the above function and plots approximation error is:

```
x=4;
for n=1:10
   z(n) =ApproxExp(x,n)
end
exact=exp(4)
plot(abs(exact-z))
```

Exercise 2: Do the same as Exercises 1, but let `x` and `y` be vectors.

Example: An example of a function that outputs more than one variable. The function computes the approximate derivative of function fname, the error in the approximation, and the estimated error. The following code is written in a file called `MyDeriv.m`:

```
function [d,err]=MyDeriv(fname,dfname,a,h);
% Output parameter: d (approximate derivative using
%                      finite difference (f(h+h)-f(a))/h)
%                   err (approximation error)
% Input parameters: fname (name of function)
%                   dfname (name of derivative function)
%                   a (point at which derivative approx)
%                   h (stepsize)

d = ( fname(a+h)-fname(a) )/h;
err = abs(d-dfname(a));
```

(Note: this works using MATLAB versions 7.0 but not 6.1. For an older MATLAB version you may have to use the feval command. See help, Daishu or me.)

A script that references the above function and plots the approximation error:

```
a=1;
h=logspace(-1,-16,16);
n=length(h);
for i=1:n
   [d(i),err(i)]=MyDeriv(@sin,@cos,a,h(i));
end
loglog(h,err)
```

Exercise: What happens if you call

```
      d=MyDeriv(@sin,@cos,a,h)
or simply
      MyDeriv(@sin,@cos,a,h)
```

You can replace `sin` and `cos` by user defined functions, for example `'f1'` and `'df1'`. Do it. That is, you need to write the function that evaluates f1 and df1 at x (in files f1.m and df1.m).

2 Computing Fundamentals

2.1 Vectorizing, timing, operation counts, memory allocation

2.1.1 Vectorizing for legibility and speed

MATLAB allows you to write the code in vectorized form. For example you can assign the vector x using

```
  x=0:0.01:1;
```

instead of the for loop

```
  for j=1:101, x(j) = (j-1)*0.01; end
```

The first statement is more concise and more legible. MATLAB also evaluates it faster. The following code times the two statements

```
m=10^4;

tic
for i=1:m
  x=0:0.01:1;
end
time1=toc;

for i=1:m
  for j=1:101, x(j) = (j-1)*0.01; end
end
time2=toc;


times=[time1, time2]
ratios=[time2/time1]
```

On my machine the second version takes 1.9 times more time than the first.

2.1.2 Memory allocation

If your code requires much memory, it helps to preallocate memory before assigning large vectors or matrices. For example, if the entries of a vector are assigned in a for loop (as is often unavoidable), we preassign memory to the vector by initializing it using the commands `zeros, ones, eye`. Example:

```
x=zeros(1,n+1);
for j=1:n+1, x(j) = (j-1)*h; end
```

Preallocating memory reduces coding errors, and can significantly reduce execution times.

2.1.3 Counting operations: Horner's algorithm

The way you write a code to perform certain operations affects how fast it runs. And if it makes a difference between 1 hour and 1 day, it matters! As we saw, the runtime depends on vectorization. But the runtime depends mainly on how many **floating point operations** the computer has to perform. A floating point operation is a multiplication, addition or subtraction of real numbers. These are much more costly than **integer operations** (adding/multiplying integers), therefore they dominate the runtime.

Lets look at examples on how implementation affects the total number of operation counts. The following are three ways to evaluate the sample polynomial of degree 4, $y = 2 + 4x - 3x^2 - x^3 + x^4$:

```
y= 2 + 4*x - 3*x*x  - x*x*x + x*x*x*x
y= 2 + 4*x - 3*x.^2 - x.^3  + x.^4
y= 2 + x*(4 + x.*(-3 + x.*(-1 + 1*x)))
```

More generally, if the polynomial is of degree N (assume `a0,a1,...,amin1,an` are constants that have already been defined in the code, and `x` is a vector of length m):

```
y= a0 + a1*x + a2*x*x  + a3*x*x*x + ... + an*x*x*x*...*x
y= a0 + a1*x + a2*x.^2 + a3*x.^3  + ... + an*x.^n
y= a0 + x*(a1 + x.*(a2 + x.*(a3 + x.*(a4 ..... + x.*(amin1 + am*x)..))))
```

Lets count the number of operations for each entry of the vector `x`. The first approach contains N additions and $1 + 2 + 3 + \cdots + N = \sum_{k=1}^{N} k = N(N+1)/2$ multiplications for a total of

$$N^2/2 + 3N/2$$

floating point operations. For the second approach lets assume that one integer power operation takes as much as 2 multiplications (this is approximately true for some computers; others take much longer). Since the second approach contains N additions, N multiplications and $N-1$ powers (with our assumption) it takes as much as

$$4N-2$$

floating point operations. The third approach takes only N multiplications and N additions for a total of

$$2N$$

floating point operations. This is the fastest and best approach, specially if N is large, and if the polynomial evaluation is a significant portion of your code (performed possibly millions of times). This approach is called **Horner's rule** or **nested multiplication**.

I compared the timings for an example polynomial of degree 20 and found that MATLAB uses about the same amount of time for the second and third approach, probably due to internal optimization when it generates machine code.

2.1.4 Counting operations: evaluating series

Let's implement the function `ApproxExp`

```
function y=ApproxExp(x,n);
% Output parameter: y (nth order Taylor approximation of $e^x$)
% Input parameters: x (scalar)
%                   n (integer)

sumo = 1;
for k=1:n
   sumo = sumo + x^k/factorial(k);
end
y = sumo;
```

in a more efficient way using less floating point operations, as follows:

```
function y=ApproxExp2(x,n);
% Output parameter: y (nth order Taylor approximation of $e^x$)
% Input parameters: x (scalar)
%                   n (integer)

temp = 1;
```

```
    sumo = temp;
    for k=1:n
       temp = temp*x/k;
       sumo = sumo + temp;
    end
    y = sumo;
```

Note that the second code replaced a power and a factorial by a multiplication and a division, which should be more efficient. The script

```
  n=100;
  x=1;
  tic
  ApproxExp(x,n);
  t1=toc

  tic
  ApproxExp2(x,n);
  t2=toc

  t1/t2
```

shows that the second approach is 80 times faster than the first, depending on the values of n and on the machine.

2.2 Machine Representation of real numbers, Roundoff errors

2.2.1 Decimal and binary representation of reals

Reals can be represented in base 10 (decimal representation) using digits 0,...,9, or base 2 (binary representation), using digits 0,1, or any other base. Examples next.

Example: What number is $(1534.4141)_{10}$?

Example: Find the base 10 representation of $(1011.01101)_2$

Example: Find the first 10 digits of the base 2 representation of 53.7_{10}. (Use common sense, no need to learn an algorithm here.)

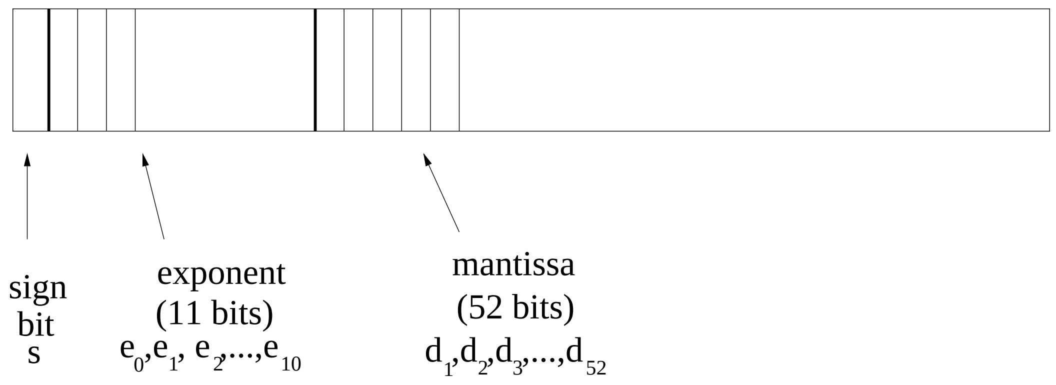


Figure 1: Machine representation of double precision real numbers in the IEEE standard. Number is stored in 64 bits: the first is the sign bit s, the following 11 ones are the exponent bits $e_0, \ldots, e_{10}$, the last 52 bits are the mantissa $d_1, \ldots, d_{52}$.

2.2.2 Floating point representation of reals

Machines store numbers using their binary representation (as opposed to the decimal representation). The coefficients in the binary representation are either 0 or 1 and these can easily be stored electronically. One 0 or 1 coefficient is called a **bit**. Eight bits are a **byte**. The amount of storage in a device is measured either in **kilobyte** (10^3 bytes), **megabyte** (10^6 bytes), **gigabyte** (10^9 bytes) or **terrabytes** (10^{12} bytes).

The IEEE standard uses 64 bits (8 bytes) to store one real number in double machine precision. See Figure 1. The first bit is the sign bit s. The following 11 ones are the exponent bits $e_0 \ldots e_{10}$. The remaining 52 bits form the mantissa $d_1, \ldots, d_{52}$. What number do these 64 digits represent? First we have to determine the exponent

$$F = (e_{10}e_9 \ldots e_1 e_0)_2 = e_{10}2^{10} + e_9 2^9 + \ldots e_1 2^1 + e_0 2^0$$

which is an integer. Note that the largest exponent corresponds to $e_0 = e_1 = \cdots = e_{10} = 1$ and equals $\sum_{k=0}^{10} 2^k = 2^{11} - 1 = 2047$ and the smallest one corresponds to $e_0 = e_1 = \cdots = e_{10} = 0$ and equals 0. Thus

$$0 \leq F \leq 2047$$

To determine which number is represented by the 64 bits one of 4 definitions is used, depending on the value of F:

1. **Normalized numbers.** If $1 \leq F \leq 2046$ the represented real number is

   $$V = (-1)^s 2^E (d_0.d_1 d_2 d_3 \ldots d_{52})_2$$

   where $E = F - 1023$, $d_0 = 1$. Thus $-1022 \leq E \leq 1023$. The smallest normalized number is $\pm 2^{-1022} \times 1.000\ldots 0 = \pm 2^{-1022}$ The unnormalized numbers enable representation of even smaller numbers.

2. **Unnormalized numbers.** If $F = 0$

$$V = (-1)^s 2^{-1022}(0.d_1 d_2 d_3 \dots d_{52})_2$$

3. **NaN.** If $F = 2047$ and mantissa $\neq 0$:

$$V = NaN \text{ (not-a-number)}$$

   Invalid operations such as $0/0$ or $\infty - \infty$ lead to NaNs.

4. **Infinity.** If $F = 2047$ and mantissa $= 0$:

$$V = (-1)^s \infty$$

   If a valid operation leads to x too large, the result is set to ∞ with the appropriate sign.

Largest and smallest numbers. The largest number that can be representd is the largest normalized number

$$V_{max} = \pm 2^{1023}(1.1111\dots 1)_2 = \pm 2^{1023}\sum_{k=0}^{52}(1/2)^k = \pm 2^{1023}(2 - 2^{-52}) \approx 1.7975 \times 10^{308}$$

The smallest number that can be representd is the smallest unnormalized number

$$V_{min} = \pm 2^{-1022}(0.000\dots 01)_2 = \pm 2^{-1074} \approx 4.9407 \times 10^{-324}$$

What is result of `x=1.797e308; 2*x`? And of `x=4.941e-324; x/2`? They result in **overflow** or **underflow** exceptions respectively.

2.2.3 Machine precision, IEEE rounding, roundoff error

In double machine precision, the computer stores only the first 53 binary digits (or 52 if unnormalized) of a real number x (this is equivalent to roughly 16 decimal digits). The remaining digits that are present if x has a longer binary expansion are truncated or rounded. Thus the machine representation of x is not the same as x! We call the floating point representation of x

$$fl(x)$$

Thus: $fl(x) - x \neq 0$ and define

$$fl(x) - x \text{ as the } \textbf{absolute roundoff error}.$$

$$\frac{fl(x) - x}{x} \text{ as the } \textbf{relative roundoff error}.$$

Note: What is the result of computing `9.4-9-0.4` in MATLAB?

IEEE addition and rounding. When two numbers are added they are first placed out of storage into a register in which addition is performed using *higher than machine precision.* The result is then rounded to 52 digits after decimal (with at most 1 nonzero digit before decimal) and stored. IEEE rounding consists of rounding down (truncating) if the 53rd digit equals 0. Otherwise, if the 53rd digit equals 1, it rounds up UNLESS all digits to the right of the 53rd and the 52nd digit are zero.

all 52 digits in the mantissa are zeros, the number is rounded down (truncated). Thus:

$$1 + 2^{-52} + 2^{-54} \text{ is rounded down to } 1 + 2^{-52}$$

$$1 + 2^{-52} + 2^{-53} \text{ is rounded up to } 1 + 2^{-51}$$

$$1 + 2^{-53} \text{ is rounded down to } 1$$

$$1 + (2^{-53} + 2^{-54}) \text{ is rounded up to } 1 + 2^{-52}$$

$$1 + 2^{-53} + 2^{-54} \text{ is rounded down to } 1$$

$$1 + 2^{-51} + 2^{-53} \text{ is rounded down to } 1 + 2^{-51}$$

To see for example the last three, type `(1+(2^-53+2^-54))-1`, `(1+2^-53+2^-54)-1`, and `(1+(2^-51+2^-53))-1`.

Machine precision. Machine precision, or machine ϵ, measures the amount of relative precision you have in the floating point representation of a number. It is defined as

machine ϵ: distance between 1 and the smallest floating point number greater than 1. That is, smallest number such that

$$fl(1+\epsilon) \neq 1$$

In the IEEE standard

$$\epsilon = 2^{-52} \approx 2.2 \times 10^{-16} \quad \text{(double machine precision)}$$

$$\epsilon = 2^{-23} \approx 1.2 \times 10^{-7} \quad \text{(single machine precision)}$$

since any floating point number not equal to one has to have at least one of $d_1, \ldots, d_{52}$ nonzero. The smallest such number bigger than 1 is the one with $d_{52} = 1$ and all other $d_1 = \cdots = d_{51} = 0$. In MATLAB, it is defined as the variable `eps`. The relative roundoff error in representing a number is

$$\frac{fl(x) - x}{x} \leq \frac{\epsilon}{2}.$$

Be sure to understand the difference between the smallest representable number 2^{-2074} and machine ϵ. Many numbers below ϵ are representable, but adding them to 1 has no effect.

2.2.4 Loss of significant digits during subtraction

While the relative precision in the numerical representation of a number is at most $\epsilon/2$, precision can be lost in particular during subtraction of almost equal numbers. Imagine for simplicity that we are computing using 5 significant decimal digits.Then consider subtracting $300.\overline{3} - 300$. In exact arithmetic

$$300.\overline{3} - 300 = 1/3$$

In floating point arithmetic keeping 5 significant digits

$$fl(300.\overline{3}) - fl(300) = 300.33 - 300 = 0.33$$

Thus the answer is represented with only 2 significant digits of accuracy. This loss of significance due to subtraction of almost equal numbers can be a problem that can sometimes be remedied.

Example: Consider computing

$$\frac{1 - \cos x}{\sin^2 x}$$

for small x. In that case, the numerator will lead to loss of significance due to subtraction of almost equal numbers. The resulting error is then amplified by dividing by a small number. The alternative formulation

$$\frac{1}{1 + \cos x}$$

(which is equal in exact arithmetic) does not lead to loss of significance. See our textbook, p 18 for a comparison of the two formulations in MATLAB. Plot both expressions vs `h=logspace(-1,-16,16)`.

Example: The quadratic formula to solve $ax^2 + bx + c = 0$,

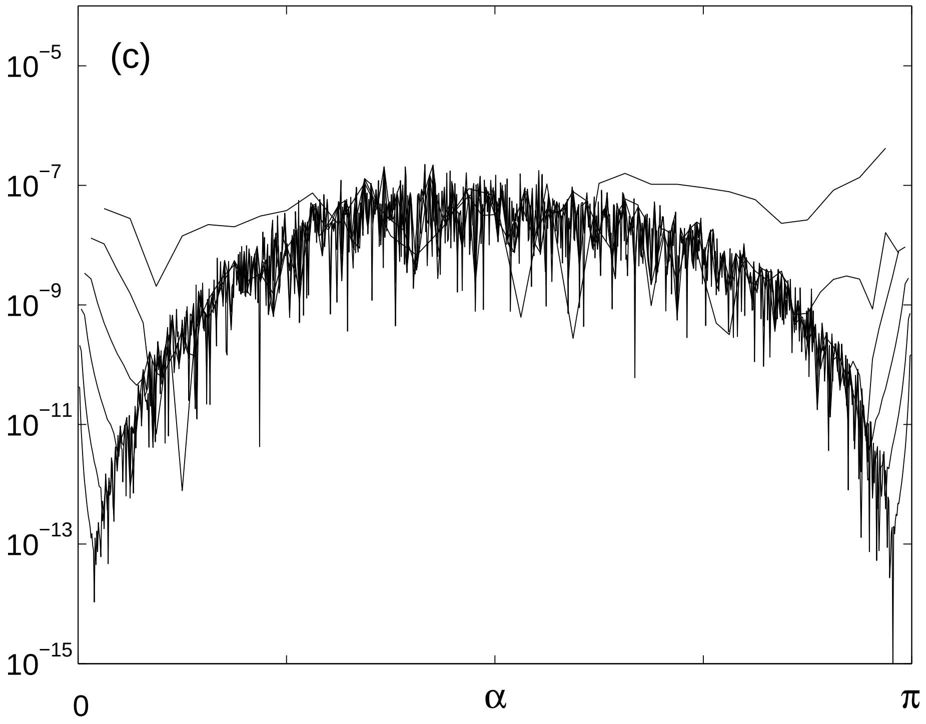
$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad , \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

leads to loss of significance in x_1 if $4ac \ll b^2$ (much less than), and $b > 0$. One avoids this loss of significance by rewriting (multiply top and bottom by conjugate and simplify) the formula for x_1 as:

$$x_1 = \frac{-c}{b + \sqrt{b^2 - 4ac}}$$

Example: Plot $(x-1)^3$ and $x^3 - 3x^2 + 3x - 1$ vs $x - 1$ for $x = linspace(1 - h, 1 + h, 1000)$ for $h = 10^{-5}$.

Example: This issues can be relevant in applications. Figure 2 illustrates the loss of significance when computing the velocity of a bubble in viscous fluid using a straight forward implementation of the governing equation (top) and after cleverly rewriting them to remove subtraction of large numbers (bottom).



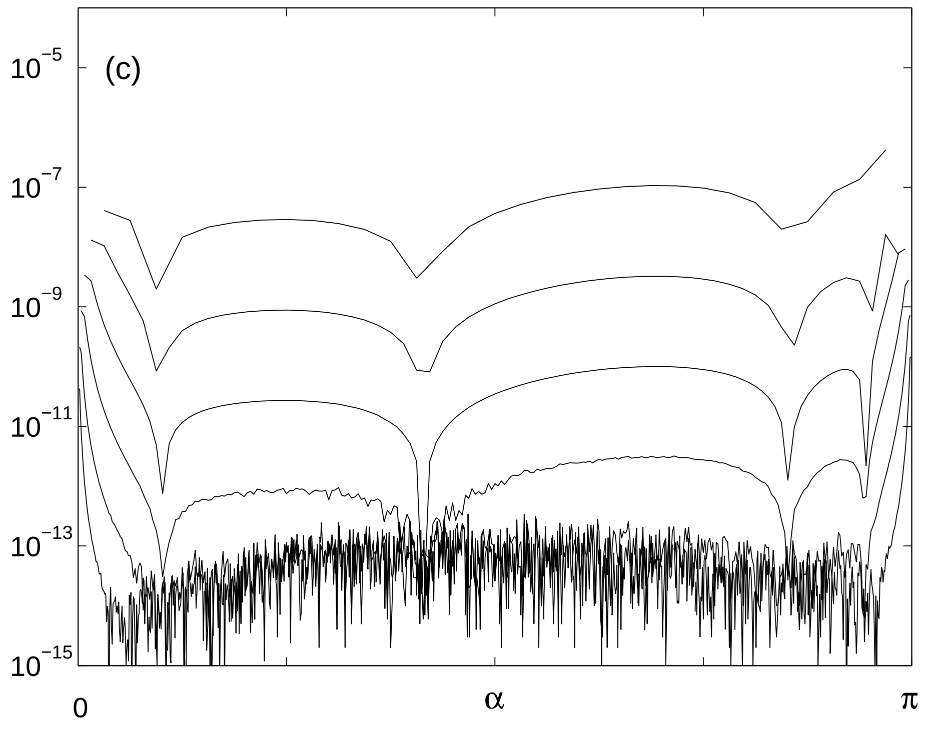


Figure 2: Effect of roundoff error on velocity of a bubble in the limit of infinite viscosity (Stokes flow). Top: subtracting large numbers introduces errors of about 10^9 times machine precision. Bottom: rewriting the differences using algebraic manipulations (determining dominant terms in the large numbers and removing them algebraically) reduces the error by a factor of 10^6. The different curves correspond to using increasing number of points to represent the bubble, with the goal of reducing the discretization error. (From Nitsche et al 2010, preprint.)

2.3 Approximating derivatives, Taylor's Theorem, plotting $y = h^p$

Another example of loss of significance due to subtraction comes in approximating derivatives, as we did earlier (MATLAB Tutorial, page 16). This example also gives us a reason to review Taylors theorem (which is fundamental for all that we will do in this class!) and go over plots of $y = h^p$ on a log-log scale.

Taylor's Theorem for functions of one variable. If the $n+1$st derivative of f exists and is continuous on $[x, x+h]$ then

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots + \frac{f^{(n)}(x)}{n!}h^n + \frac{f^{(n+1)}(\xi)}{n!}h^{n+1}$$

for some value $\xi \in [x_0, x]$. If M is an upper bound for $f^{(n+1)}$ on this interval we see that the last term is $\leq Mh^{n+1}$ and thus

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots + \frac{f^{(n)}(x)}{n!}h^n + O(h^n)$$

using the Big-O notation defined in next section.

Approximating derivatives. Using Taylor's theorem with $n = 1$ we can show that in exact arithmetic

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

Thus the approximation of the first derivative by the left hand side makes an $O(h^1)$ error, we call it a first order approximation.

Approximating derivatives in IEEE arithmetic. If h is small, then the numerator of

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

leads to loss of significance due to subtraction of almost equal numbers. In particular:

$$\frac{fl(f(x+h)) - fl(f(x))}{fl(h)} = \frac{f(x+h) + \epsilon_1 - (f(x) + \epsilon_2)}{h + \epsilon_3} \approx \frac{f(x+h) - f(x)}{h} + \frac{\epsilon_4}{h} = O(h) + O(1/h)$$

where $\epsilon_{1,2,3,4}$ are of the order of $\epsilon f(x)$. If h is large, the first term will dominate, if h is small the second term will dominate!. This is what we saw in the plot obtained in section 1.9.

Plotting $y = h^p$ on log-log scale. Note that if $y = Ch^p$ then $\log y = \log C + p \log h$. Thus plotting $\log y$ vs $\log h$ (ie, using log-log scale) we obtain a *linear plot* with slope p! For example when we plotted the error above of the form $O(h) + O(1/h)$ we obtained a portion of the curve with slope 1 (where $O(h)$ term dominates) and a portion with slope -1 (where $O(1/h)$ term dominates).

2.4 Big-O Notation

Let E be a function of h. We say that $E(h) = O(h^p)$ as $h \to 0$ (Read as "E is big-O of h to the p as h goes to 0", or "E is of order h to the p") if there exists constants ϵ and C such that

$$\|E(h)\| < Ch^\alpha \qquad \forall h \in [0, \epsilon]$$

This is of interest since it gives an upper bound on how fast a function approaches zero in the limit as h goes to zero. Similarly $E(N) = O(N)$ as $N \to \infty$ means that there exists constants C and N_0 such that

$$\|E(N)\| < CN \qquad \forall N \geq N_0$$

and as a result, this notation gives a bound on how fast E can grow as $N \to \infty$ (in this case, at most linearly). (Usually, we use h or ϵ to denote small numbers and N to denote large numbers.)

3 Solving nonlinear equations $f(x) = 0$

Problem Statement: Find root of a function $f(x)$ (linear or nonlinear). Equivalently, the problem is to solve an equation (linear or nonlinear).

Definition:A function $f(x)$ has a *root* r if $f(r) = 0$.

Note: Why are the two problems stated above equivalent? Because we can write the solution to any equation as the root of a function. For example, solution to $\cos^3 x = x + 1$ is the root of $f(x) = \cos^3 x - x - 1$.

Intermediate Value Theorem (IVT): The IVT is a theorem that guarantees that f has a root if it satisfies certain conditions. The theorem says:

If $f(a)f(b) < 0$ and f is continuous on $[a, b]$ then f has a root in (a, b).

Note that no conclusions can be drawn if the conditions of the theorem does not apply. For example, if f is not continuous on $[a, b]$, or if $f(a)f(b) > 0$ we CANNOT conclude anything about the roots of f in (a, b). But if the conditions apply, we know f has a root and we want a numerical method to find it. Moreover, the method should be as accurate as desired (if possible), and fast.

3.1 Bisection method to solve $f(x) = 0$ (§1.1)

- **Basic idea:** If f is known to have a root on $[a, b]$ by IVT, bisect interval into two subintervals, choose the one containing the root (using IVT), repeat. The length of the interval decreases by half at each step, the root is known to within that precision, and can be found to specified precision by taking sufficiently many steps. In theory.

- **The algorithm.** Outline of algorithm for `function root=bisect(f,a,b,tol)`

```
while abs(a-b)/2>tol
   c=(a+b)/2
   if f(c)==0 done
   if f(a)*f(c)<0
      b=c
   else
      a=c
   end
end
root = c
```

Problems: what if tol too small? Two function evaluations per loop. Filling in the details. Explain modification to book algorithm.

```
function [root,i]=bisect(f,a,b,tol)
fa=f(a); fb=f(b);
if (fa*fb >= 0 )
   error('f(a)f(b)<0 not satisfied')
endif

i=0;
disp(sprintf('  %3d  %20.15f  ',i,a))
while abs(a-b)/2 > tol
   c= (a+b)/2; fc=f(c);
   if fc==0
      break
   end
   if fa*fc<0
      b=c; fb=fc;
   else
      a=c; fa=fc;
   end
   i=1+1;
   disp(sprintf('  %3d  %20.15f  ',i,a))
end
root = (a+b)/2;
```

Example: Apply to solve $3x^3 + x^2 = x + 5$ to within 7 decimal places.

Definition: A solution is correct within p decimal places if the error is less than 0.5×10^{-p}. Note, this does not mean that the pth digit is exact!

Example: Apply to approximate the cube root of 5 to within 10 decimal places.

Example: What happens if you type

```
[xc,niter]=bisect(@sin,3,4,10^-20)
```

? Fix the problem. (Replace `tol` by `tol+eps*max(abs(a),abs(b))`).

- **Convergence rate.** Rate of reduction of error.

  The method builds a sequence of approximate solutions x_k, given by the midpoints of the relevant interval. We know that, up to roundoff, the sequence of midpoints x_k computed by the bisection method converges to the exact root. How fast does it converge? That is, how fast does the error decrease in each step? We know that after n steps, the root r lies in the $n+1$st subinterval, with length $(b-a)/2^{n+1}$. So

  $$|x_n - r| \leq \frac{b-a}{2^{n+1}} \,.$$

  That is, the error goes down by 1/2 in each step. This is called **Linear convergence** which we will define more precisely later.

  With this result we can estimate how many steps it takes to get within a prescripbed tolerance.

  Example: If $|b-a| = 1$, tolerance $tol = 5 \cdot 10^{-8}$, the error is guaranteed to be $< tol$, after $n = 24$ steps.

3.2 Fixed point iteration to solve $x = g(x)$ (FPI, §1.2)

3.2.1 Examples

- Starting with any number x_0 of your choice, compute $x_{k+1} = \cos(x_k)$ for approximately $0 \leq k \leq 20$ (by repeatedly hitting the cosine button). What do you observe?
- Method of the Babylonians (1750 BC!) to approximate $\sqrt{2}$: Note that if $x > \sqrt{2}$, then $2/x < 2/\sqrt{2} = \sqrt{2}$. The Babylonians proposed to, starting with any initial guess x_0, obtain a better approximation to $\sqrt{2}$ by averaging x and $2/x$. This leads to the iteration
$$x_{k+1} = \frac{x_k + \frac{2}{x_k}}{2}$$
Try it out with an initial guess of your choice ($\neq 0$), and see how quickly the iterates approach $\sqrt{2}$.

3.2.2 The FPI

- A fixed point iteration is of the form
$$x_{k+1} = g(x_k) \ , \qquad \text{with } x_o \text{ given}$$
IF the iteration converges, that is, $\lim_{k\to\infty} x_k = r$ for some r, as in the above examples, then r satisfies
$$r = g(r)$$
and is called a **fixed point** of g. Geometrically, this point is the x-coordinate of the intersection of the graphs of $y = g(x)$ and $y = x$. (Show that the fixed point of the Babylonian map satisfies $x^2 = 2$.)
- The iterates of FPI can be visualized by a **cobweb diagram**. Starting with the point $(x_0, 0)$, get to x_1 geometrically by going vertically up to the point $(x_0, g(x_0))$ on the curve $y = g(x)$, and then moving over horizontally to point $(g(x_0), g(x_0))$ on the line $y = x$. This new point has x-coordinate x_1. Repeat this process. Note, to obtain an accurate cobweb diagram, the line $y = x$ must be drawn correctly. This is easiest if, by hand and in matlab, we use a **1-1 scale** (in matlab, use `axis equal`).
- The cobweb diagrams for the cosine map above show that the iterates converge to the root of $r = \cos r$, for any x_0. In this case we say the iteration is **globally convergent**. The cobweb diagram for the Babylonian map shows that for $x_0 > 0$, the iterates converge to $r = \sqrt{2}$, while for $x_0 < 0$, the iterates converge to $r = -\sqrt{2}$. It also shows that the iterates converge very fast to r ones they are close to r, because $g'(r) = 0$.
- Can always rewrite $f(x) = 0$ as $g(x) = x$, but not uniquely! For example: can rewrite $x^3 + x - 1 = 0$ in three ways: as $x = g_1(x) = 1 - x^3$, $x = g_2(x) = (1 - x)^{1/3}$, $x = g_3(x) = \frac{1+2x^2}{1+3x^2}$.

3.2.3 Implementing FPI in Matlab

- Pseudocode 1:

```
initialize vector x   %to alot memory to it
initialize x(1)       %to the given initial guess
for k=1:iter
  x(k+1)=g(x(k))
end
```

  where `iter` is the total number of iterations performed. This code saves all the iterates of the iteration in a vector x. We are mostly interested only in the last iterate and dont want to save all the intermediate steps, in which case we simply overwrite the current iterate:

  Pseudocode 2:

```
initialize x       %to the given initial guess
for k=1:iter
  x=g(x)
end
```

- Using a function that implements either of the above, we now apply FPI to solve $x^3 + x - 1 = 0$ in three ways. We find that (1) Diverges, (2) converges, (3) converges much faster.

- The cobweb diagram show that (1) diverges because the slope of g at the root is > 1 in magnitude!, $|g_1'(r)| > 1$. (2) converges to r (provided x_0 is sufficiently close to r) since $|g_2'(r)| < 1$ and (3) converges faster since $|g_3'(r)| \ll |g_2'(r)| < 1$. Suggestion: convergence depends on local slope.

3.2.4 Theoretical Results

- **Theorem:** If $|g'(r)| < 1$ and x_0 is sufficiently close to r, then the fixed point iteration converges to r. Furthermore, the error $e_k = x_{k+1} - r$ satisfies

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k} = g'(r)$$

  Proof: If $|g'(r)| < 1$ then $|g'(r)| \le S < 1$ in some neighbourhood of r. If x_0 is in this neighbourhood, then, by Taylor's Theorem,

$$\begin{aligned}
|x_1 - r| &= |g'(\xi_0)||x_0 - r| \le S|x_0 - r| && (1)\\
|x_2 - r| &= |g'(\xi_1)||x_1 - r| \le S|x_1 - r| \le S^2|x_0 - r| && (2)\\
&\dots && (3)
\end{aligned}$$

where ξ_0 is some point between x_0 and r, ξ_1 is some point between x_2 and r, etc. By iterating this process we find that

$$|x_k - r| \leq S^k |x_0 - r| \quad \text{or} \quad |e_k| \leq S^k |e_0|$$

Note that the right hand side $\to 0$ as $k \to \infty$ because $S < 1$, and thus, $\lim_{k\to\infty} x_k = r$, or $\lim_{k\to\infty} e_k = 0$. and we proved the first part.
Now note that

$$(x_{k+1} - r) = g'(\xi_k)(x_k - r) \quad \text{or} \quad \frac{e_{k+1}}{e_k} = g'(\xi_k)$$

where ξ_k is between x_k and r. Since $x_k \to r$ as $k \to \infty$, also $\xi_k \to r$. Thus

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k} = g'(r)$$

and we proved the second part.

Note: This implies that if k is sufficiently large, $e_{k+1} \approx g'(r)e_k$. The error decays by a factor of $g'(r)$ at each step. If $g'(r)$ is small, it will decay very fast. If $g'(r) \approx 1$, it will decay very slowly.

- Example: For the cosine map, print $k, x_k, e_k, e_{k+1}/e_k$. Compare to $g'(r)$. Run the following matlab code.

```
clear
g=inline('cos(x)'); gp=inline('sin(x)'); f=inline('cos(x)-x');
%g=inline('(x+2./x)/2');gp=inline('(1-2./x.^2)/2');
%f=inline('x.^2-2');fp=inline('2*x');fpp=inline('2');

a=1; kmax=20;
x=fixedpt2(g,a,kmax);  %this function returns a vector of all iterates, x

r=fzero(f,a);
n=length(x);
e=abs(x-r);
disp('   k          x            error        ratio ')
for k=1:n-1
  disp(sprintf('%3d %15.10f %14.10f %10.6f', k,x(k),e(k),e(k+1)/e(k)) )
%  disp(sprintf('%3d %15.10f %13.10f %10.6f', k,x(k),e(k),e(k+1)/e(k)^2) )
end
disp(sprintf('\n Compare limit of ratios with g''(r)= %9.6f', gp(r)))
%disp(sprintf('\n Compare limit of ratios with g''(r)/(2g''(r))= %9.6f',...
%    fpp(r)/(2*fp(r))))
```

- Example: Repeat for the two converging maps for $x^3 + x - 1 = 0$.

3.2.5 Definitions

- **Local vs global convergence.** An iterative scheme is **locally convergent** if $x_k \to r$ as $k \to \infty$ provided x_0 is sufficiently close to r. FPI is locally convergent.
  An iterative scheme is **globally convergent** if $x_k \to r$ as $k \to \infty$ for any x_0. Some FPI, such as cosine map, are globally convergent. We can deduce this from the cobweb diagram.

- **Rates of convergence.**
  If $\lim_{k\to\infty} \frac{e_{k+1}}{e_k} = S$, with $0 < S < 1$, then the iteration x_k is **linearly convergent**. This implies that for sufficiently large k, $e_{k+1} \approx Se_k$ (the error gets reduced by a factor S). If $0 < |g'(r)| < 1$, FPI is locally convergent.

  If $\lim_{k\to\infty} \frac{e_{k+1}}{e_k^2} = S$, with $0 < S$, then the iteration x_k **converges quadratically**. This implies that for sufficiently large k, $e_{k+1} \approx Se_k^2$. Once e_k is small, this convergence is much better than linear.

  If $\lim_{k\to\infty} \frac{e_{k+1}}{e_k^p} = S$, with $0 < S$, then the iteration x_k **converges to order p**.

- What is convergence rate for Babylonian method? Hint: run the matlab code above after replacing the three lines starting with "g...", and "disp..." by the commented lines below them. We will see the reason for the last line in this code in the next section.

3.2.6 Stopping criterion

- Implement stopping criterion: iterate FPI until $|x_{k+1} - x_k| < tol$, for some given tolerance.
  How do you do this in Pseudocode 1? Pseudocode 2?
  Note that even if $|x_{k+1} - x_k| < tol$, we dont know what the actual error $|x_{k+1} - r|$ after that step is.

3.3 Newton's method to solve $f(x) = 0$ (§1.4)

3.3.1 The algorithm

- Picture and derivation of algorithm. Stopping criteria.
- Write out Newton's algorithm to solve $x^2 = 2$ (!)
  Write out Newton's algorithm to solve $x^3 + x - 1$. Combine terms. (!)
- What can go wrong? $f'(x_k) = 0$, oscillation between two points.
  If $f'(x_k) \approx 0$, move far away from initial condition, and may converge to a far away root.
  What happens if $f'(r) = 0$?
- Note: Newton's is a special type of fixed point iteration.

3.3.2 Matlab implementation

Again, two choices: either return vector of solutions or overwrite old iterate by current

- Pseudocode 1:

```
initialize vector x
initialize x(1)
for k=1:kmax
  x(k+1) = x(k)-f(x(k))/fp(x(k));
  if abs(x(k+1)-x(k)) < tol exit loop
end
```

- Pseudocode 2:

```
initialize x
set xold=x
for k=1:kmax
  x = x-f(x)/fp(x);
  if abs(x-xold) < tol exit loop
  xold=x
end
```

- Apply to $x^3 + x - 1 = 0$. Plot e_{k+1}/e_k^2.

3.3.3 Theoretical results

- Prove: if $f'(r) \neq 0$ then Newton's method is locally convergent. (to prove, view Newtons as fixed point iteration for some g. Find $g'(r)$)
- Prove: if $f'(r) \neq 0$, Newton's method is quadratically convergent. (Hint: Use Talor polynomial of order 1 for f, with remainder.)
- State: if $f'(r) = 0$, method still converges but slower: linear convergence, with $\lim_{k\to\infty} e_{k+1}/e_k = (m-1)/m$. (To see this, apply Newtons method to solve x^m)
  State modified Newton's method for multiple roots, with quadratic convergence.
- Summary. Ups: fast convergence. Downs: requires f'. Only locally convergent (but this is true for most methods).

3.4 Secant method (§1.5)

- State Secant Method

3.5 How things can go wrong. Conditioning. (§1.3)

Example: Apply bisection algorithm to solve

$$x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27} = 0$$

(which equals $(x - \frac{2}{3})^3 = 0$) to within 4 digits, 6, 10, 20 digits. Compare approximate and exact roots. How big is the error? Repeat with `fzero`. Plot $y = f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27}$, zoom into region near triple root. Repeat for $f(x) = (x - \frac{2}{3})^3$. What is the problem? Why does it occur in one formulation, and not the other? (Answer: loss of significance due to cancellation in one case, not the other.)

3.5.1 Multiple roots

Definition:A function $f(x)$ has a *root of multiplicity* n at $x = r$ if $f(r) = f'(r) = f''(r) = \cdots = f^{(n-1)}(r) = 0$ but $f^{(n)}(r) \neq 0$. Alternatively, the root r has multiplicity n if the function's Taylor series about $x = r$ has leading order term

$$f(x) \sim C(x-r)^n + \text{ smaller terms of higher order}$$

For example it is easy to check using either criterion that $f(x) = \sin(x) - x$ has a triple root at $x = 0$. It is easy to check using Taylor series that $\sin(x^{100})$ has a root of multiplicity 100 at the origin.

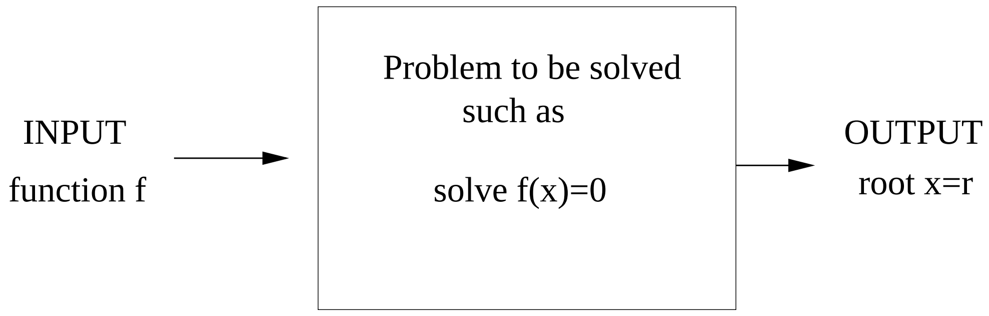


Figure 3: Input and output of a problem to be solved.

We see that in the above example, the large multiplicity of the root causes a problem: for a relatively large range of values of x away from the root the floating point values of the function are near 0 and thus the bisection algorithm (or `fzero` or any other algorithm) finds many incorrect roots. In other words, here,

large changes in x yield small changes in f

The problem occurs only in one formulation and not in the other because the relative size of roundoff error differs in the two cases.

3.5.2 Forward and Backward error. Error magnification.

One can view the problem to be solved as having input values (in our case f) and output values (in our case the root r). See figure 3.

The numerical method outputs an approximate solution, call it x_c. Define

Forward error: Error in output, in our case $|x_c - r|$

Backward error: Error in input (introduced for example by machine precision, noisy measurements, etc), in our case $|f(x_c) - f(r)| = |f(x_c) - 0|$

So the problem is that the forward error is very large relative to the backward error! That is, the

$$\text{error magnification} = \frac{\text{forward error}}{\text{backward error}} \tag{4}$$

is very large. The input is represented accurately numerically, but since the error magnification is large, the output is much less accurate. Problems for which this is true are said to be **ill-conditioned**. For such problems we cannot obtain good answers numerically, no matter which method we use! Note that the conditioning is inherent to the problem, not the method.

Why is this a problem? In general we do not know the forward error since we dont have the exact solution (bisection algorithm is an exception!), and we can only compute the backward error. But we would like, based on the backward, to know the forward error. If we knew the error magnification factor we could do that. In some cases an estimate for the error magnification factor can be found.

3.5.3 Other examples of ill-conditioned problems

Multiple roots are not the only cases yielding ill-conditioned problems.

Example: Solve $x + y = 2, 1.00001x + y = 2.00001$ using Matlabs $A\backslash b$ solver of linear systems. Note: exact solution $x = 1, y = 1$. Numerical solution relatively far from it. Why? Geometric interpretation.

Example: Wilkinsons polynomial

$$W(x) = (x-1)(x-2)\dots(x-20) = x^{20} - 210x^{19} + 20615x^{18} + \dots 2432902008176640000$$

has simple roots, yet, if implemented in the formulation on the right hand side,

$$\texttt{fzero(wilkpoly, 16)} = \texttt{16.014...}$$

The problem here is again loss of precision due to subtraction of almost equal numbers. For x near 16, each term in the sum is large, and they need to cancel to yield the resulting value near 0. The relative errors in each number, due to finite machine precision, is amplified, yielding very large relative error (only 2 correct digits since the relative error is only less than $0.5\dot{1}0^{-2}$))

Thus, small relative errors (of 1.e-16) in the coefficients yield large relative errors in the solution $\Rightarrow$ ill-conditioned.

3.5.4 The condition number

Whether a problem is ill-conditioned or well-conditioned is based on the magnification of the relative errors (as opposed to the absolute errors considered in Eq 4).

Definition: The **condition number** of a problem is

$$\text{cond} = \text{maximum } \frac{\text{relative forward error}}{\text{relative backward error}} \tag{5}$$

over all changes in input. For example, in the Wilkinson polynomial case, the relative changes in the coefficients of f are of order 10^{-16} and the relative change in the output is

10^{-2}, yielding condition number of at least 10^{14}. The maximal amplification of the relative errors can sometimes be found precisely. More later. Note that if we know the condition number and we know the backward error we can estimate the forward error.

4 Solving linear systems

4.1 Gauss Elimination (§2.1)

- Solve sample 2x2 linear system $x + y = 3, 3x - 4y = 2$, geometrically and algebraically
- Solve sample 3x3 linear system

$$\begin{aligned} 2x + y - 4z &= -7 \\ x - y + z &= -2 \\ -x + 3y - 2z &= 6 \end{aligned}$$

  by hand (using Tableau form), doing Gauss Elimination, then back substitution. Define pivots. Keep track of the multipliers, and carefully do all algebra.

- Write GE algorithm (no pivoting). Note that we only have to update the nonzero entries a_{ik}, $i, k \geq j + 1$, that is, the entries below and to the right of a_{jj}.

```
[n,n]=size(A);
for j=1:n-1         %go over all columns j except the last one
  for i=j+1:n       %eliminate all entries A(i,j) below A(j,j), i>j
     m=A(i,j)/A(j,j);  % compute multipliers to eliminate entry
     for k=j+1:n       % replace all entries A(i,k) in Row i to the
        A(i,k)=A(i,k)-m*A(j,k);                  % right of column j
     end
     b(i)=b(i)-m*b(j);  % do row operation on rhs b
  end
end
```

- Count number of operations for GE: $\sum_{j=1}^{n-1}[2(n-j)+3](n-j)$. Use

$$\sum_{j=1}^{n} 1 = n \ , \quad \sum_{j=1}^{n} j = \frac{n(n+1)}{2} \ , \quad \sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6}$$

  to evaluate.

- Backward substitution to solve $Ux = b$, algorithm and operation count.

```
for i=n:-1:1
  for j=i+1:n
    b(i)=b(i)-a(i,j)*x(j)
  end
  x(i)=b(i)/a(i,i);
end
```

- Operation counts:

$$\begin{array}{ll} \text{Gauss Elimination} & : \quad \frac{2}{3}n^3 + l.o.t \\ \text{Upper Triangular system :} & n^2 \\ \text{Lower Triangular system :} & n^2 \end{array}$$

4.2 LU decomposition (§2.2)

- Today we'll show that GE is equivalent to factoring $A = LU$, where L is lower triangular, U is upper triangular. Why is such a factorization useful? Suppose you need to solve $A\mathbf{x}_k = \mathbf{b}_k$ for many $\mathbf{b}_k$, suppose $k = 1, 1000$. One option is to use GE each time for a total cost of

$$1000(2/3)n^3$$

The alternative is to find the LU factorization once (at a cost of 2/3 n^3) and then solve $LU\mathbf{x}_k = \mathbf{b}_k$ in two steps:

$$\begin{array}{llll} (1) & \text{Solve} & L\mathbf{y} = \mathbf{b}_k & \text{for } \mathbf{y} \\ (2) & \text{Solve} & U\mathbf{x}_k = \mathbf{b}_k & \text{for } \mathbf{x}_k \end{array}$$

for a cost of n^2 each step, and a total cost of

$$(2/3)n^3 + 2000n^2 \ .$$

The $O(n^3)$ operation can be viewed as an initialization cost, and all subsequent steps are $O(n^2)$ instead of $O(n^3)$. Big savings.

- To show that GE is equivalent to factoring $A = LU$: Every step in GE consists of adding a multiple of $-m_{ij}$ of the jth row to the ith row.

This is an elementary row operation. There are 3 elementary row operations:

$$\begin{array}{l} \text{Replacing } Row_i \text{ by } Row_i - m \cdot Row_j \\ \text{Replacing } Row_i \text{ by } m \cdot Row_i \\ \text{Exchanging } Row_i \text{ and } Row_j \end{array}$$

Each one is equivalent to premultiplying A by an *elementary matrix*. The first operation, replacing Row_i by $Row_i - m \cdot Row_j$ is equivalent to premultiplying A by L_{ij}, where L is the lower triangular matrix which equals the identity except that it contains $-m_{ij}$ in its (ij)th position. (That is, it is the matrix obtained from the identity matrix by performing the desired row operation on it.)

Thus at the end of GE we have performed the following operations

$$L_{n,n-1} \dots L_{3,2}L_{n,1} \dots L_{3,1}L_{2,1}Ax = L_{n,n-1} \dots L_{3,2}L_{n,1} \dots L_{3,1}L_{2,1}\mathbf{b}$$

and

$$L_{n,n-1}\dots L_{3,2}L_{n,1}\dots L_{3,1}L_{2,1}A = U$$

Now note that L_{ij} are invertible with simple inverse and compute

$$L_{2,1}^{-1}L_{3,1}^{-1}\dots L_{n,1}^{-1}L_{3,2}^{-1}\dots L_{n,n-1}^{-1} = L$$

where L contains the m_{ij} below the diagonal, and ones on the diagonal. Thus, after GE have $A = LU$ decomposition.

- Change *GE* algorithm so as to return L, U. Test for an example.

  ```
  [n,n]=size(A);
  for j=1:n-1
    for i=j+1:n
       A(i,j)=A(i,j)/A(j,j);  % overwrite A(i,j) by multiplier
       for k=j+1:n
          A(i,k)=A(i,k)-A(i,j)*A(j,k);
       end
    end
  end
  L=eye(n,n)+tril(A,-1);
  U=triu(A);
  ```

  Write main portion of above algorigthm more concisely in matlab (just for curiosity).

  ```
  for j=1:n-1
    A(j+1:n,j)=A(j+1:n,j)/A(j,j);
    A(j+1:n,j+1:n)=A(j+1:n,j+1:n)-A(j+1:n,j)*A(j,j+1:n);
  end
  ```

- Find *LU*-factorization of A by hand for

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

4.3 Partial Pivoting (§2.3, §2.4)

- What if pivots $a_{jj} = 0$? Then the LU decomposition does not exist

  Example: Show that $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ has no LU decomposition.

  What if pivots a_{jj} are small? Then multipliers a_{ij}/a_{jj} are large, which can lead to loss of precision when computing $a_{ik} - ma_{jj}$.

  Example: Show that using GE to solve $\begin{bmatrix} 10^{-20} & 1 \\ 1 & 2 \end{bmatrix}\mathbf{x} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$ leads to large amplification of small errors introduced by roundoff. $\Rightarrow$ GE without pivoting is UNSTABLE

- Solution: pivoting (exchanging rows so that maximal entry a_{ij} in jth column, with $i \geq j$, moves into the pivot position). As a result the multipliers a_{ij}/a_{jj} will always be < 1.
- Describe partial pivoting. Use it to solve $\begin{bmatrix} 10^{-20} & 1 \\ 1 & 2 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$ to show roundoff errors stay small. $\Rightarrow$ GE with partial pivoting is STABLE (small erros are not amplified, provided the problem we are trying to solve is WELL-POSED.) we have not proven this, only illustrated by example, but it can be proven to be true. Note that if the problem we are trying to solve is ILL-POSED, than no method can give a good result: small errors are amplified by the problem, even if the method is stable.
- Switching rows is equivalent to premultiplying by elementary matrices P_k. (What happens if you postmultiply A by P_k?) Show that it leads to decomposition $PA = LU$. (For this we need to know how to write $P_k L_{ij} = \tilde{L}_{ij} P_k$.
- Find the PLU factorization P, L, U for a sample 3x3 matrix A, by hand. Check result by confirming that $PA = LU$.
- Check result using MATLABs `lu(A)` function.
- How to use PLU decomposition to solve $A\mathbf{x} = \mathbf{b}$ for many b.
- What is full pivoting?

4.4 Conditioning of linear systems (§2.3)

- Consider $Ax = b$

$$\begin{aligned} 0.835x + 0.667y &= 0.168 \\ 0.333x + 0.266y &= 0.067 \end{aligned}$$

Has exact solution $\mathbf{x} = (1, -1)$. If $\mathbf{b}$ is changed to $\tilde{\mathbf{b}} = (0.168, 0.066)$ the exact solution is $\tilde{\mathbf{x}} = (-666, 834)$! That is, the forward error (change in output x) is much larger than the small backward error (change in input A,b). This is a symptom of an illconditioned problem: small perturbations, such as introduced by roundoff or measurement errors, can be amplified significantly.

For a 2x2 linear system this illconditioning can be explained geometrically: graph the lines represented by each of the two equations and find their slopes are almost identical, so small changes in the line (input) cause large changes in the intersection (output).

For general linear system, the amplification factor of the relative forward error is measured by the condition number. The exact statement follows next.

- To measure the changes in vectors and matrices we need vector and matrix **norms**. The vector norm you are probably most familiar with is the Euclidean 2-norm, for example

$$\begin{aligned} ||\langle a, b\rangle||_2 &= \sqrt{a^2+b^2} \\ ||\langle x_1, x_2, \dots, x_n\rangle||_2 &= \sqrt{x_1^2+x_2^2+\cdots+x_n^2} \end{aligned}$$

  A compatible matrix norm needs to satisfy certain properties (too learn more about this, take Math 464). The matrix norm compatible with the vector 2-norm is given in terms of the largest absolute singular value of the matrix.

  Instead, we will use a simpler vector- and matrix- norm. (For a rule to be a norm, it must satisfy certain properties. Again, more on this in Math 464.) We will use the inf-norm. For a $n \times 1$ vector $\mathbf{x}$ and an $n \times n$ matrix A it is defined as

  vector inf-norm $\quad ||\mathbf{x}||_\infty = \max\limits_{1\le j\le n} |x_j|$

  matrix inf-norm $\quad ||A||_\infty = \max\limits_{1\le i\le n} \sum\limits_{j=1}^{n} |A_{ij}|$ (maximum absolute row sum)

- Define relative forward error $||\mathbf{x}-\tilde{\mathbf{x}}||/||\mathbf{x}||$

  relative backward error $||\mathbf{b}-\tilde{\mathbf{b}}||/||\mathbf{b}||$

  The backward error is the **residual** $A\tilde{\mathbf{x}}-\mathbf{b}$. The example shows that residual can be small even if the error in the solution is very large. The next theorem shows that this amplification factor can be as large as the **condition number** of the matrix.

- Theorem:

$$\frac{||\mathbf{x}-\tilde{\mathbf{x}}||}{||\mathbf{x}||} \le K(A)\frac{||\mathbf{b}-\tilde{\mathbf{b}}||}{||\mathbf{b}||} \text{ where } K(A) = ||A||||A^{-1}||$$

  with equality attainable for some $\mathbf{b}$. (This theorem holds using any matrix norm.)

  Note: the condition number is a property of the matrix, not of any numerical method used to solve the system. In MATLAB, you can use the function `cond` to obtain the condition number of a matrix, using a norm of your choice. You can also use `norm` to compute norms of matrices and vectors.

  One can expect that roundoff error introduces errors of size of machine epsilon ϵ in the matrix A and the right hand side b. Then the relative forward error can be expected to be

$$\frac{||\mathbf{x}-\tilde{\mathbf{x}}||}{||\mathbf{x}||} \le K(A)\epsilon$$

So if, for example, $K(A) \approx 10^9$, then you loose 9 digits of accuracy in solving the linear system $A\mathbf{x} = \mathbf{b}$.

Example: Hilbert matrix (example 2.12)

4.5 Iterative methods (§2.5)

- GE with pivoting solves the problem $A\mathbf{x} = \mathbf{b}$ to within machine precision (times condition number, see §3.5). Why do we need to study other methods to solve this problem? After obtaining the LU decomposition once, GE is fast ($O(n^2)$) if $A\mathbf{x} = \mathbf{b}$ has to be solve repeatedly for many right hand sides. However, if $A\mathbf{x} = \mathbf{b}$ has to be solved once (for example, new A and b at every timestep in a time-evolution problem) then it is expensive ($O(n^3)$). Need faster methods.
- Iterative methods to solve $A\mathbf{x} = \mathbf{b}$ are of the form
$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{c} \tag{6}$$
with some initial guess $\mathbf{x}_0$. (This is similar to the fixed point iteration which we studied for nonlinear scalar problems.) Under what conditions do such iterative methods for linear nxn systems converge?
- Answer (which can be motivated using the fixed point iteration result): The iteration $\mathbf{x}_{k+1} = B\mathbf{x}_k + \mathbf{c}$ converges if and only if the spectral radius $\rho(B) < 1$. The spectral radius is defined to be the largest absolute eigenvalue of B: $\rho(B) = \max\limits_{1\le j\le n} |\lambda_j|$.
- **Jacobi method. Example.** This method consists of solving the jth equation in $A\mathbf{x} = \mathbf{b}$ for x_j, that is, which yields a specific equivalent system $\mathbf{x} = B\mathbf{x} + \mathbf{c}$, and then performing the fixed point iteration.

  Go through examples 2.19 and 2.20. Show how to get the system $\mathbf{u} = f(\mathbf{u}) = B\mathbf{u} + \mathbf{c}$, where $\mathbf{u} = \langle u, v\rangle$ by solving first equation for u, second one for v. Write a MATLAB function `f(u)` and iterate
$$\mathbf{u}^{(k+1)} = f(\mathbf{u}^{(k)})$$
  (we dont need save all iterates of the vector $\mathbf{u}$, but simply overwrite the old $\mathbf{u}$. Note that:
  - The resulting system $\mathbf{u} = B\mathbf{u} + \mathbf{c}$ depends on the order of the original equation
  - Using one order, the method converges, in the other, the method does not converge. Can you explain this fact by looking at the eigenvalues of the corresponding matrix B?
- **Jacobi method for general Ax=b**. Solving the jth equation for x_j yields the (linear) system of equations
$$x_j = \frac{1}{a_{jj}} \left(b_j - \sum_{\substack{l=1 \\ l\neq j}} a_{jl} x_l \right)$$

(Note: this system clearly depends on the order of the equations. For example, need $a_{jj} \neq 0$!) Jacobi's method consists of the fixed point iteration

$$x_j^{(k+1)} = \frac{1}{a_{jj}} \left(b_j - \sum_{l \neq j} a_{jl} x_l^{(k)} \right) \tag{7}$$

using some initial guess for $x_j^{(0)}$.

Let's write the method (7) in matrix form and then state under which conditions it converges. Write $A = L + D + U$ where D contains the diagonal elements of A, and L and U the elements below and above the diagonal respectively. Then we can solve the jth equation for x_j (assuming D is invertible) as follows:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \qquad \Longleftrightarrow \\ L\mathbf{x} + D\mathbf{x} + U\mathbf{x} &= \mathbf{b} \qquad \Longleftrightarrow \\ D\mathbf{x} &= \mathbf{b} - (L+U)\mathbf{x} \qquad \Longleftrightarrow \\ \mathbf{x} &= D^{-1}(\mathbf{b} - (L+U)\mathbf{x}) \\ \mathbf{x} &= D^{-1}\mathbf{b} - D^{-1}(L+U)\mathbf{x} \end{aligned}$$

Jacobi's method consist of the iteration

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - (L+U)\mathbf{x}^{(k)}) \tag{8}$$

where $\mathbf{x}^{(0)}$ is some initial guess. Typically we let $\mathbf{x}^{(0)} = \mathbf{0}$, unless there is information to make a better initial choice. Thus Jacobi is of the form 6 with

$$B_{Jac} = -D^{-1}(L+U), \mathbf{c} = D^{-1}\mathbf{b}$$

From the theorem above it follows that Jacobi converges if $\rho(B_{Jac}) < 1$.

- **An easier convergence criterium.**

  Definition : A matrix is *strictly diagonally dominant* if

  $$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \ .$$

  Theorem: Jacobi method converges if A is strictly diagonally dominant. (One can show that this implies $\rho(B) < 1$.)

  Example: go back to examples 2.19, 2.20. Can we determine convergence based on this criterium?

- **A better implementation. Stopping criteria.** We can rewrite Jacobi's method by noting that $D^{-1}(\mathbf{b} - (L+U)\mathbf{x}) = D^{-1}(\mathbf{b} - (L+U+D)\mathbf{x} + D\mathbf{x}) = \mathbf{x} + D^{-1}(\mathbf{b} - A\mathbf{x})$ and thus (8) is equivalent to

  $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \tag{9}$$

In practice, we stop iterating when the *residual* $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$ (which is the same as the backward error), has norm

$$||\mathbf{r}^{(k)}||_\infty = \max_j |r_j^{(k)}| < tol \tag{10}$$

for some chosen tolerance *tol*. In the homework you are asked to stop when

$$||\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}|| < tol \tag{11}$$

Note however that in view of equation (9),

$$\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} = D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$$

is basically the residual, up to multiplication by D^{-1}. That is the two stopping criteria (10) and (11) are not that different.

- **Numerical cost:** How expensive is Jacobi? If A is full, how many flops does it take to compute $A\mathbf{x}$? How many iterations did it take in you homework? How does Jacobi then compare to GE?
- **Faster iterative methods:** Gauss-Seidel, SOR, Conjugate gradient, GMRES (briefly describe their uses)

5 Interpolation

Problem statement: Given n data points $(x_j, y_j), j = 1, \ldots, n$, find a function $g(x)$ such that $g(x_j) = y_j$. (See Figure 3.1, page 140)

Some interpolating functions may do a much better job at approximating the actual function than others. We will consider 3 possible types of interpolating functions $f(x)$:

- polynomials
- piecewise polynomials
- trigonometric polynomials

5.1 Polynomial Interpolants

Theorem: Given n points $(x_j, y_j), j = 1, \ldots, n$, there exists a unique polynomial

$$p(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}$$

of degree $n-1$ such that $p(x_j) = y_j$. (Note, the n conditions $p(x_j) = y_j$ will be used to determine the n unknowns c_j.)

Example: Find the cubic polynomial that interpolates the four points $(-2, 10)$, $(-1, 4)$, $(1, 6)$, $(2, 3)$. Set up the linear system that determines the coefficients c_j. Solve using Matlabs backslash, print polynomial on a finer mesh.

Example: Find the linear polynomial through (1,2), (2,3) using different bases.

Two Questions:

1. **How good is the method?** There are many methods, depending on how we represent the polynomial. There are many ways to represent a polynomial $p(x)$. For example, you can represent the quadratic polynomial interpolating $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ in the following ways

$$\begin{aligned}
p(x) &= c_1 + c_2 x + c_3 x^2 \quad \text{Vandermonde approach} \\
p(x) &= \tilde{c}_1 + \tilde{c}_2 (x - x_1) + \tilde{c}_3 (x - x_1)(x - x_2) \quad \text{Newton interpolant} \\
p(x) &= c_1' \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + c_2' \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + c_3' \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} \\
& \qquad \text{Lagrange interpolant}
\end{aligned}$$

were the coefficients $c_k, \tilde{c}_k, \hat{c}_k, c'_k$ are chosen to satisfy that $p(x_k) = y_k, k = 1, 2, 3$. All we've done is used different **bases** to represent the polynomial $p(x)$. In exact arithmetic, the result with any of these bases is the same: $p(x)$. But numerically, we will see that it will make a difference which basis we choose, in terms of accuracy and/or speed.

2. **How good is the problem?** Given a set of data $\{x_k, y_k\}$, where $y_k = f(x_k)$, there is an interpolant. How good does the interpolant approximate the underlying function $f(x)$?? We will see that for certain classes functions and points, certain types of interpolants are bad. This is why we also consider other types of interpolants.

5.1.1 Vandermonde approach

Let

$$p(x) = c_1 + c_2 x + \cdots + c_n x^{n-1}$$

where c_k are determined by the conditions that $p(x_j) = y_j, j = 1, \ldots, n$. Write out these conditions as a linear system $V\mathbf{c} = \mathbf{b}$. What is V? What is $\mathbf{b}$? MATLAB code to find the coefficients:

```
function c=interpvan(x,y)
n=length(x);
V(:,1)=ones(n,1);
for i=2:n
  V(:,i)=x'.*V(:,i-1);
end
c=V\y';
```

Note that V is full. That is, solving $V\mathbf{c} = \mathbf{y}$ for the coefficients is an $O(n^3)$ operation. Furthermore, in class, we found the condition number of V for n equally spaced points, and found that already for $n = 10$ it is 10^7 (exact values depend on range of $\mathbf{x}$), and becomes much larger as n increased.

If you now want to evaluate the polynomial at an arbitrary set of values x (different than the originally given data points), the most efficient way (using O(n) flops) to do this is to use Horner's rule. For example, for $n = 5$ this consists of rewriting p as

$$p(x) = c_1 + x(c_2 + x(c_3 + x(c_4 + x(c_5)))) \ .$$

A MATLAB implementation is:

```
function p=evalpvan(c,x)
%evaluates Vandermonde polynomial coefficients c using Horner's algorithm
```

```
%Input x: row vector
%Output p: row vector
n=length(c); m=length(x);
p=c(n)*ones(size(x));
for k=n-1:-1:1
  p=p.*x+c(k);
end
```

So, for this method:

Cons: **1.** large condition numbers leading to inaccuracies
2. $O(n^3)$ amount or work to invert linear system

Pros: **1.** O(n) algorithm to evaluate polynomial once coefficients are known

Example: Use above to find interpolant through $(-2, 10)$, $(-1, 4)$, $(1, 6)$, $(2, 3)$. Plot polynomial on $x \in [-3, 3]$ using a fine mesh.

```
xx=[-2,-1,1,2];
yy=[10,4,6,3];
c=interpvan(xx,yy);
x=-3:.05:3;
y=evalpvan(c,x);
plot(xx,yy,'r*',x,y,'b-')
```

5.1.2 Lagrange interpolants

The polynomial interpolant through $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ is

$$p(x) = c_1 L_1(x) + c_2 L_2(x) + \ldots c_n L_n(x)$$

where

$$L_k(x) = \frac{(x - x_1) \ldots (x - x_{k-1})(x - x_{k+1}) \ldots (x - x_n)}{(x_k - x_1) \ldots (x_k - x_{k-1})(x_k - x_{k+1}) \ldots (x_k - x_n)}$$

and the $c_j = y_j$. This follows since the basis function L_k satisfy

$$L_k(x_j) = 0, j \neq k \ , L_k(x_k) = 1 \ .$$

Thus, no work needed to find c's! Find operation count to evaluate $p(x)$.

Pros: Explicit representation (no need to solve for c_ks).

Cons: $O(n^2)$ operations to evaluate polynomial

Good for small n and number m at which to evaluate. Good as theoretical tool. Can use to prove existence of unique poly interpolating n points. Outline: 1. Existence: here is a formula 2. Uniqueness: assume another poly q interpolates same points. Then difference $p - q$ is poly of degree n-1 that is zero at n points. By fund thm of algebra: must be zero poly.

5.1.3 Newton's divided differences

We represent the polynomial interpolant through $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ by

$$p(x) = c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + \cdots + c_n(x - x_1)(x - x_2) \ldots (x - x_{n-1})$$

where coefficients c_k are chosen such that $p(x_j) = y_j, j = 1, \ldots, n$. For example, with $n = 4$, these n equations determining the unkowns c_k are

$$\begin{aligned}
&c_1 &&= y_1\\
&c_1 + c_2(x_2 - x_1) &&= y_2\\
&c_1 + c_2(x_3 - x_1) + c_3(x_3 - x_1)(x_3 - x_2) &&= y_3\\
&c_1 + c_2(x_4 - x_1) + c_3(x_4 - x_1)(x_4 - x_2) + c_4(x_4 - x_1)(x_4 - x_2)(x_4 - x_3) &&= y_4
\end{aligned}$$

You can see that the resulting linear system for c_k is triangular and can be evaluated in $O(n^2)$ operations. In class we solved the system with $n = 3$ and saw that solution is obtained in terms of divided differences. The book gives an algorithm for computing these differences and obtaining the result by hand. We'll skip this and write a MATLAB algorithm to solve the problem. To arrive at the MATLAB algorithm lets go back to the case $n = 4$ and do one step of Gauss Elimination, then divide each equation by leading coefficient to obtain

$$\begin{bmatrix}
1 & 0 & 0 & 0 & | & y_1\\
1 & x_2 - x_1 & 0 & 0 & | & y_2\\
1 & x_3 - x_1 & (x_3 - x_1)(x_3 - x_2) & 0 & | & y_3\\
1 & x_4 - x_1 & (x_4 - x_1)(x_4 - x_2) & (x_4 - x_1)(x_4 - x_2)(x_4 - x_3) & | & y_4
\end{bmatrix}$$

$$\rightarrow \begin{bmatrix}
1 & 0 & 0 & 0 & | & y_1\\
0 & x_2 - x_1 & 0 & 0 & | & y_2 - y_1\\
0 & x_3 - x_1 & (x_3 - x_1)(x_3 - x_2) & 0 & | & y_3 - y_1\\
0 & x_4 - x_1 & (x_4 - x_1)(x_4 - x_2) & (x_4 - x_1)(x_4 - x_2)(x_4 - x_3) & | & y_4 - y_1
\end{bmatrix}$$

$$\rightarrow \begin{bmatrix}
1 & 0 & 0 & 0 & | & y_1\\
0 & 1 & 0 & 0 & | & (y_2 - y_1)/(x_2 - x_1)\\
0 & 1 & x_3 - x_2 & 0 & | & (y_3 - y_1)/(x_3 - x_1)\\
0 & 1 & x_4 - x_2 & (x_4 - x_2)(x_4 - x_3) & | & (y_4 - y_1)/(x_4 - x_1)
\end{bmatrix}$$

Note that the 3x3 lower right subsystem is of identical form as the original one except with fewer points and a right hand side replaced by divided differences. Now repeat this process $n - 1$ times. At end the right hand side will contain the solution c_k. In class we derived the resulting algorithm:

```
function c=interpnew(x,y)
n=length(x);
for k=1:n-1
   y(k+1:n)=(y(k+1:n)-y(k))./(x(k+1:n)-x(k));
end
c=y;
```

So, coefficients can be obtained in $O(n^2)$ flops. (how many exactly?) Another advantage of this approach is that the polynomials in Newton representation can be evaluated in $O(n)$ flops using a nested algorithm similar to the one for monomials. Write out Horner's rule for the Newton polynomial with $n = 5$. Deduce the following algorithm:

```
function p=evalpnew(c,xx,x)
n=length(c);
p=c(n)*ones(size(x));
for k=n-1:-1:1
   p=p.*(x-xx(k))+c(k);
end
```

Pros: 1. $O(n^2)$ algorithm to find c_k (faster than Vandermonde approach)
2. $O(n)$ nested algorithm to evaluate polynomial

Cons: condition numbers still large

Example: Use above to find interpolant through $(0,0),(1,1),(3,-1),(4,-1)$. Plot polynomial on $x \in [0,5]$ using a fine mesh.

5.2 Accuracy of Polynomial interpolation

Here we get to the second question: How good is the polynomial interpolant? Suppose the points (x_j, y_j), $j = 1, \ldots, n$ satisfy $y_j = f(x_j)$, where f may or may not be known a priori. Now you interpolate them by a polynomial $p_n(x)$, as discussed in previous section. This polynomial equals f at the gridpoints. But how close is it to f away from the gridpoints??

That is, how well does the $p(x)$ approximate $f(x)$ for all x? So we want to know something about the maximum error difference between f and p_n at any given point x.

Theorem: If p_n is the (unique) polynomial of degree $n-1$ that interpolates f at x_j, $j = 1, \ldots, n$ (that is, $f(x_j) = p(x_j)$) then

$$f(x) - p_n(x) = \frac{(x-x_1)(x-x_2)\ldots(x-x_n)}{n!} f^{(n)}(c)$$

for some $c \in (x_1, x_n)$ (assuming the x_js are in increasing order).

There are a few items to deduce from this

- If we know an upper bound $|f^{(n)}(x)| \leq M_n$ for all $x \in (x_1, x_n)$, then we know a bound for the error
$$|f(x) - p_n(x)| = \frac{M_n}{n!}|(x-x_1)(x-x_2)\dots(x-x_n)|$$
- The error is a product of a terms that depend solely on the points x_k, and a term that depends solely on the derivative of f.
- If we can choose the points x_k, we can find the best polynomial p_n for any given f, by finding the points x_k that minimize the maximum of
$$|(x-x_1)(x-x_2)\dots(x-x_n)|$$

Example: Investigate the error graphically for $f(x) = \frac{1}{1+25x^2}$ for $x \in [-1, 1]$, using n uniformly spaced points. Plot f, f', f''. Plot $(x-x_1)\dots(x-x_n)$, compare to $f - p_n$.

5.2.1 Uniform points vs Tschebischeff points

Theorem: The set of points $x_1, \dots, x_n$ that minimizes the maximum of $|(x-x_1)(x-x_2)\dots(x-x_n)|$ are the Tschebischeff points
$$x_j = \cos\left(\frac{\pi(2j-1)}{2n}\right) \quad , \quad j = 1, \dots, n$$
Note that the argument of the cosine is odd multiples $1, 3, \dots, (2n-1)$ of $\pi/(2n)$, so it ranges from just a little above 0 to just below π. Therefore the x_j range from just below 1 to just above -1. Visualize the points as in book, by plotting semi-circle and marking x-coordinates of points on semicircle at equally spaced arclength. Alternatively, figure (4) plots the cos values at equal spaced arguments, showing that they buch up at ± 1. For an interval [a,b], the corresponding points would be
$$x_j = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{\pi(2j-1)}{2n}\right) \quad , \quad j = 1, \dots, n$$

[Side note : The way to prove this theorem requires properties of the **Tschebyscheff** (or **Chebichev**) **polynomials**
$$T_n(x) = \cos(n \arccos x)$$
Pafnuti Lwowitsch Tschebyschow was an important Russian Mathematician in the 19th centure, 1821-1894. Some of the properties of T_n: $T_n(x)$ is the unique polynomial of degree

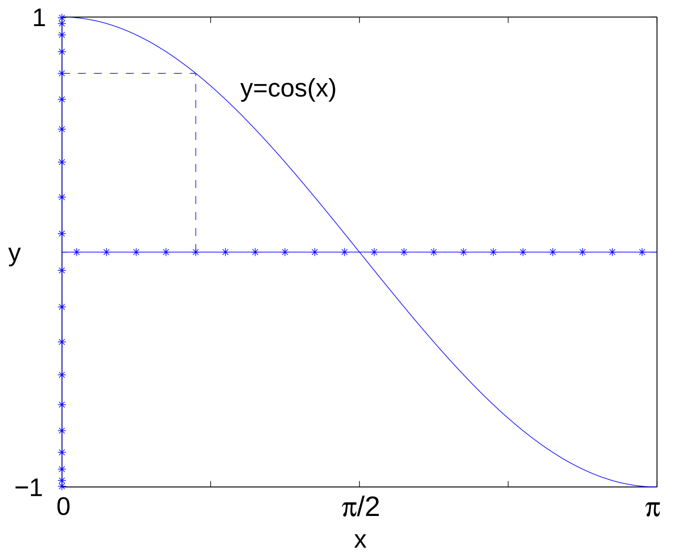


Figure 4: Image of equally spaced points under cosine map, showing how they bunch up near ± 1. These are the Tschebischeff points.

n that satisfies $T_n(x_j) = 0$, where x_j $j = 1, \ldots, n$, are the Tschebyscheff points given above, and $T_n(1) = 1$. The fact that it is a polynomial of degree n follows from the recurrence relation

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

which can be deduced using trigonometric identities. Its values at x_j and at 1 can be found by inspections. The leading coefficient of x^n is $2^{(n-1)}$, which also follows from the recursive formula. It thus follows that

$$T_n(x) = 2^{n-1}(x - x_1)(x - x_2) \ldots (x - x_n) \ .$$

Finish side note.]

Example: Investigate the error $f(x) - p_n(x)$ graphically, for $f(x) = \frac{1}{1+25x^2}$ and $x \in [-1, 1]$, using n Tschebischeff points.

One can show that

- With equally spaced points, $|(x - x_1)(x - x_2) \ldots (x - x_n)| \leq (b - a)^n \dfrac{(n-1)!}{(n-1)^n}$
  (Does this term grow or decrease as $n \to \infty$?)
- With Tschebischeff points, $|(x - x_1)(x - x_2) \ldots (x - x_n)| \leq \left(\dfrac{b-a}{2}\right)^n \dfrac{1}{2^{n-1}}$
  (Does this term decrease any faster as $n \to \infty$?)

5.3 Piecewise Polynomial interpolants

Here we discuss a different approach. Suppose we are given n points $(x_j, y_j), j = 1, \ldots, n$, and we want to find an interpolant through the points. Instead of finding one function that interpolates all n points, we find a function on each interval that interpolates two consecutive points, with some matching conditions connecting the pieces from one interval to the next. The result is a piecewise interpolant

$$g(x) = \begin{cases} S_1(x) \ , & x_1 \le x < x_2 \\ S_2(x) \ , & x_2 \le x < x_3 \\ \vdots & \\ S_{n-1}(x) \ , & x_{n-1} \le x < x_n \end{cases}$$

5.3.1 Piecewise linear interpolant

The simplest case is when each $S_j(x)$ is a linear function interpolating (x_j, y_j) and (x_{j+1}, y_{j+1}). This is what MATLAB plots when you type `plot(x,y)`. That is,

$$S_j(x) = a_j + b_j(x - x_j) \ , \quad x \in [x_j, x_{j+1}]$$

where the two unknowns a_j, b_j are determined by the two conditions $S(x_j) = y_j, S(x_{j+1}) = y_{j+1}$.

5.3.2 Cubic splines

A cubic spline interpolant of the data $(x_j, y_j), j = 1, \ldots, n$ is a piecewise cubic function defined by

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3 \ , \quad x \in [x_j, x_{j+1}] \ , \quad j = 1, \ldots, n-1$$

such that

$$\begin{array}{lcl} (0) & S_j(x_j) = y_j & j = 1, \ldots, n-1 \\ (1) & S_j(x_{j+1}) = y_{j+1} & j = 1, \ldots, n-1 \\ (2) & S'_j(x_{j+1}) = S'_{j+1}(x_{j+1}) & j = 1, \ldots, n-2 \\ (3) & S''_j(x_{j+1}) = S''_{j+1}(x_{j+1}) & j = 1, \ldots, n-2 \end{array}$$

Conditions (0),(1) ensure that the polynomials interpolate the data, while conditions (2),(3) enforce continuity of g', g'' at the interior data points, thereby ensuring that the interpolant g is not only continuous, but has continuous slope and curvature. This leads to a smooth-looking curve to the eye.

Condition (0) is trivially satisfied by setting $a_j = y_j$, leaving us with $3n-3$ unknowns $b_j, c_j, d_j, j = 1, \ldots, n-1$. Conditions (1)-(3) give $3n-5$ equations, so that we still need 2 more equations to uniquely solve for the $3n-3$ unknowns. Different conditions can be chosen. The **natural spline** consists of specifying zero curvature at the endpoints.

$$\begin{array}{cc} (4a) & S_1''(x_1) = 0 \\ (4b) & S_{n-1}''(x_n) = 0 \end{array}$$

The **clamped spline** consists of prescribing the derivatives at the endpoints.

$$\begin{array}{cc} (4a') & S_1'(x_1) = \alpha \\ (4b') & S_{n-1}'(x_n) = \beta \end{array}$$

The **not-a-knot spline** consists of prescribing continuous third derivative at the two interior points closest to the end.

$$\begin{array}{cc} (4a\text{''}) & S_1'''(x_2) = S_2'''(x_2) \\ (4b\text{''}) & S_{n-2}'''(x_{n-1}) = S_{n-1}'''(x_{n-1}) \end{array}$$

We now derive the equations for the coefficients of the natural spline. First, note that

$$\begin{array}{rcl} S_j(x) & = & a_j + b_j(x-x_j) + c_j(x-x_j)^2 + d_j(x-x_j)^3 \\ S_j'(x) & = & b_j + 2c_j(x-x_j) + 3d_j(x-x_j)^2 \\ S_j''(x) & = & 2c_j + 6d_j(x-x_j) \end{array}$$

Conditions (1)-(3) are

$$\begin{array}{ccc} (1) & b_j\Delta x_j + c_j\Delta x_j^2 + d_j\Delta x_j^3 = \Delta y_j & j = 1, \ldots, n-1 \\ (2) & b_j + 2c_j\Delta x_j + 3d_j\Delta x_j^2 = b_{j+1} & j = 1, \ldots, n-2 \\ (3) & 2c_j + 6d_j\Delta x_j = 2c_{j+1} & j = 1, \ldots, n-2 \end{array}$$

where $\Delta x_j = x_{j+1} - x_j$, $\Delta y_j = y_{j+1} - y_j$. The goal now is to do some algebra by hand to come up with a system of less unknowns that we then solve numerically. First, we introduce a new variable $c_n = S_{n-1}''(x_n)/2$. As we showed in class, it follows that condition (3) above holds for $j = 1, \ldots, n-1$. Furthermore, conditions (4ab) can be simply stated as

$$\begin{array}{cc} (4a') & c_0 = 0 \\ (4b') & c_n = 0 \end{array}$$

We now solve (3) for d_j, to get d_j in terms of c_js:

$$(3') \qquad d_j = \frac{c_{j+1} - c_j}{3\Delta x_j} \qquad j = 1, \ldots, n-1$$

That is, using algebra we have solved for all the unknowns d_j. Then, we substitute this formula for d_j into (1) and solve (1) for b_j, so that we have b_j and d_j in terms of c_js:

$$(1') \qquad b_j = \frac{\Delta y_j}{\Delta x_j} - \frac{\Delta x_j}{3}(2c_j + c_{j+1}) \qquad j = 1, \dots, n-1$$

Finally we substitute the formulas for d_j and b_j into (2) and obtain an equation for the c_js, that depends only on the n unknowns c_j:

$$(2') \qquad \frac{\Delta y_j}{\Delta x_j} - \frac{\Delta x_j}{3}(2c_j + c_{j+1}) + 2c_j\Delta x_j + (c_{j+1} - c_j)\Delta x_j = \frac{\Delta y_{j+1}}{\Delta x_{j+1}} - \frac{\Delta x_{j+1}}{3}(2c_{j+1} + c_{j+2})$$

which simplifies to

$$(2'') \qquad c_j\Delta x_j + 2c_{j+1}(\Delta x_j + \Delta x_{j+1}) + c_{j+2}\Delta x_{j+1} = 3[\frac{\Delta y_{j+1}}{\Delta x_{j+1}} - \frac{\Delta y_j}{\Delta x_j}] \qquad j = 1, \dots, n-2$$

Together with (4a), (4b), these equations are written as a system as follows:

$$\begin{bmatrix} 1 & & & & & & \\ \Delta x_1 & 2(\Delta x_1 + \Delta x_2) & \Delta x_2 & & & & \\ & \Delta x_2 & 2(\Delta x_2 + \Delta x_3) & \Delta x_3 & & & \\ & \ddots & \ddots & \ddots & & & \\ & & & \Delta x_{n-2} & 2(\Delta x_{n-2} + \Delta x_{n-1}) & \Delta x_{n-1} \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 3[\frac{\Delta y_2}{\Delta x_2} - \frac{\Delta y_1}{\Delta x_1}] \\ 3[\frac{\Delta y_3}{\Delta x_3} - \frac{\Delta y_2}{\Delta x_2}] \\ \vdots \\ 3[\frac{\Delta y_{n-1}}{\Delta x_{n-1}} - \frac{\Delta y_n}{\Delta x_n}] \\ 0 \end{bmatrix}$$

MATLAB function `compnatspline` that returns the coefficients $a_j, b_j, c_j, d_j, j = 1, \dots, n-1$ for the natural spline:

```
function [a,b,c,d]=compnatspline(x,y)
% function [a,b,c,d]=compnatspline(x,y)
% Computes coefficients a,b,c,d of the natural spline through
% the data (x_j,y_j), j=1,\dots n

n=length(x);
delx=(x(2:n)-x(1:n-1))';
dely=(y(2:n)-y(1:n-1))';
a=zeros(n,n);
a(1,1)=1; a(n,n)=1;
```

```
r(1)=0; r(n)=0;
for row=2:n-1;
   r(row)=3*(dely(row)/delx(row)-dely(row-1)/delx(row-1));
   a(row,row-1)=delx(row-1);
   a(row,row)=2*(delx(row-1)+delx(row));
   a(row,row+1)=delx(row);
end
c=a\r';
clear a
b(1:n-1)=dely./delx -delx/3.*(2*c(1:n-1)+c(2:n));
d(1:n-1)=(c(2:n)-c(1:n-1))./(3*delx(1:n-1));
a(1:n-1)=y(1:n-1);
c=c(1:n-1);
```

MATLAB function that plots the spline as well as the data using m points on each of the $n-1$ intervals $[x_j, x_{j+1}], j = 1, n-1$, using Horner's nested polynomial evaluation:

```
function plotspline(a,b,c,d,x,y,m)
% function plotspline(a,b,c,d,x,m)
% plots m points of spline through x_k, coeff a,b,c,d, on each
% interval

plot(x,y,'*r')
hold on
n=length(x);
for j=1:n-1;
  z=linspace(x(j),x(j+1),m)
  arg=z-x(j);
  y=a(j)+arg.*(b(j)+arg.*(c(j)+arg*d(j)));
  plot(z,y)
end
hold off
```

Script that plots the 6 points $(0,3),(1,1),(2,4),(3,1),(4,2),(5,0)$ and the cubic natural spline on a fine mesh that has 10 points on each subinterval:

```
x=0:5; y=[3 1 4 1 2 0];
[a,b,c,d]=compnatspline(x,y);
plotspline(a,b,c,d,x,y,10)
```

5.4 Trigonometric interpolants

What if the data is viewed to be periodic? Then we want to use periodic basis functions ($\cos kx, \sin kx$). It turns out that this periodic interpolation works well if the data is uniformly spaced in x, and the periodic extension of the data is smooth. (Note: any set of data may be viewed to be periodic. The only problem is that the periodic extension may not correspond to a smooth function.)

For what follows, the total number of points n **must be even**.

So, let's consider n points (x_j, y_j), $j = 0, \dots, n-1$, where the x_j are uniformly distributed in an interval $[0, T]$, n is even, and the y_j correspond to a periodic function with period T. To begin with, we will assume $T = 2\pi$. That is, we are given data (x_j, y_j)

$$x_j = j\frac{2\pi}{n} \ , \quad y_j = f(x_j) \ , \qquad j = 0, \dots, n-1$$

where $f(x+2\pi) = f(x)$. Then it is natural to fit data using periodic basis functions, such as $1, \cos kx, \sin kx$. Note that these functions have period $T = 2\pi$ with k oscillations per period. We call k the **wavenumber** and k/T the **oscillation frequency** (number of oscillations per unit time). The **oscillation period** is T/k.

5.4.1 Using a basis of sines and cosines

As we did in §4.1 (and unlike the splines in §4.3) we will try to fit *all* the data using *one* linear combination of n basis functions. Here we use the basis

$$1, \cos x, \cos 2x, \dots, \cos\frac{n}{2}x, \sin x, \sin 2x, \dots, \sin(\frac{n}{2}-1)x \ .$$

(The next sine mode, $\sin\frac{n}{2}x$, is zero at the gridpoints, and thus is no different from the zero function and makes no contribution on the grid. Hence, no loss by excluding it, no gain from including it.) Thus we want to approximate $f(x)$ by a **trigonometric polynomial**

$$g(x) = \sum_{k=0}^{n/2} a_k \cos(kx) + \sum_{k=1}^{n/2-1} b_k \sin(kx) \tag{12}$$

where the n coefficients a_k, b_k as chosen such that $g(x_j) = f(x_j)$.

5.4.2 Using a basis of complex exponentials

Note that we can write $\cos kx$ and $\sin kx$ in terms of complex exponential functions e^{ikx} using Euler's formula

$$\cos(kx) = \frac{e^{ikx} + e^{-ikx}}{2} \ , \quad \sin(kx) = \frac{e^{ikx} - e^{-ikx}}{2i}$$

and similarly, we can write e^{ikx} in terms of sines and cosines

$$e^{ikx} = \cos(kx) + i\sin(kx) \ .$$

Hence, *at the gridpoints*, the basis of n sines and cosines is equivalent to the basis of n exponentials

$$e^{ikx} \ , \quad k = -n/2, \ldots, n/2-1$$

It turns out to be slightly more direct to use this basis for our trigonometric polynomials. So, instead of finding g as in (12) we will find

$$T(x) = \sum_{k=-n/2}^{n/2-1} c_k e^{ikx}$$

such that $T(x_j) = f(x_j)$, $j = 0, \ldots, n-1$. These n equations determine the n coefficients c_k, which are called the **Fourier coefficients**.

In class we show that *at the gridpoints* we can rewrite $T(x_j)$ (using the change of summation index $l = k+n$ for one part of the sum) as

$$\begin{aligned}
T(x_j) &= \sum_{k=-n/2}^{n/2-1} c_k e^{ikx_j} = \sum_{k=-n/2}^{-1} c_k e^{ikj2\pi/n} + \sum_{k=0}^{n/2} c_k e^{ikj2\pi/n} \\
&= \sum_{l=n/2}^{n-1} c_{l-n} e^{i(l-n)j2\pi/n} + \sum_{k=0}^{n/2} c_k e^{ikj2\pi/n} = \sum_{l=n/2}^{n-1} c_{l-n} e^{ilj2\pi/n} e^{-ij2\pi} + \sum_{k=0}^{n/2} c_k e^{ikx_j} \\
&= \sum_{l=n/2}^{n-1} c_l e^{ilj2\pi/n} + \sum_{k=0}^{n/2} c_k e^{ikj2\pi/n} = \sum_{k=n/2}^{n-1} c_k e^{ikj2\pi/n} + \sum_{k=0}^{n/2} c_k e^{ikj2\pi/n} \\
&= \sum_{k=0}^{n-1} c_k e^{ikx_j}
\end{aligned}$$

provided the coefficients are taken to be periodic, $c_{l-n} = c_l$, any l. In that case the equations $T(x_j) = f(x_j)$, or

$$f_j = \sum_{k=0}^{n-1} c_k e^{2\pi ikj/n} \ , \quad j = 0, \ldots, n-1$$

where $f_j = f(x_j)$, can be written as $F\mathbf{c} = \mathbf{f}$ where $F = F(\omega)$ is the Fourier matrix, given in terms of nth unit root $\omega = e^{2\pi i/n}$ as

$$F = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \ldots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ldots & \vdots & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \omega^{3(n-1)} & \ldots & \omega^{(n-1)^2} \end{bmatrix}$$

This matrix is invertible. It is easy to check that

$$F^{-1} = \frac{1}{N} F(\omega^{-1}) \;.= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \dots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots & \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{3(n-1)} & \dots & \omega^{-(n-1)^2} \end{bmatrix} \tag{13}$$

As a result we get a formula for the coefficients $\mathbf{c} = F^{-1}\mathbf{f}$, or

$$c_j = \frac{1}{N} \sum_{k=0}^{n-1} f_k e^{-2\pi i k j/n} \;, \quad j = 0, \dots, n-1$$

which also shows that they are indeed periodic (check that $c_{j+n} = c_j$).

The process of computing the Fourier coefficients from given values of f_k is called the **Fourier transform** and consists of evaluating a sum of the given form. The process of computing the values f_k from the coefficients c_k is called the **inverse Fourier transform** and consists of evaluating a very similar sum, where the negative sign in the exponent is replaced by a plus.

5.4.3 Using MATLABs `fft` and `ifft`

MATLABs functions `fft` and `ifft` compute these sums in a fast way using the Fast Fourier Transform, which evaluates the sums in $O(n \log n)$ instead of $O(n^2)$ operations. We read the online documentation to check this. In particular, `fft(f,n)` returns the vector of length n with entries $\sum_{k=0}^{n-1} f_k e^{-2\pi i j k/n}$, $\quad j = 0, \dots, n-1$. That is, it returns the Fourier coefficients up to a factor of $1/n$.

To illustrate: Let $f(x) = \sin 3x = \dfrac{e^{i3x} - e^{-i3x}}{2i}$. It has

$$c_3 = \frac{1}{2i} \;, \quad c_{-3} = c_{N-3} = -\frac{1}{2i} \;, \quad c_j = 0 \text{ for } j \neq \pm 3$$

The following matlab script computes the Fourier coefficients using $N = 64$ points. It then plots both the function (sampled at the 64 points) and the absolute value of the Fourier coefficients. You may want to print `c(4)` and `c(62)` to make sure they are the correct values. (Why 4 and 62 instead of 3 and -3 or 61?)

```
clear;clf                          % clear all data and the figure
n=64;                              % set n
x=[0:n-1]*2*pi/n; y=sin(3*x);     % set x and y at gridpoints
c=fft(y,n)/n;         % remember, need to divide by N
```

```
    axis([0,2*pi,-2,2]); ylabel(' f_j')
    plot(0:n-1,abs(c))
    axis([0,n-1,-0.1,0.6]); ylabel('abs(c_j)')
```

We did this for various examples in class.

Examples: Look at c_k and plot $|c_k|$ for: $f(x) = 1, \sin x, \cos 8x, \cos^2 x, 1/(1+\sin^2 x), x$ using $N = 64$.

What happens when data is not periodic? Answer: decay rate of coefficients says something about the smoothness of the periodic extension of the function f.

To plot the trigonometric interpolant $T(x) = \sum_{k=-n/2}^{n/2-1} c_k e^{ikx}$ (on a fine mesh in between gridpoints) we need to correlate the Fourier coefficients MATLAB returns with indeces $1, \ldots, n$ to the actual power in the exponential given by $k = -n/2, \ldots, n/2-1$. This can be done in MATLAB by specifying the correct mode k corresponding to a given MATLAB coefficient. For example, the coefficient indexed by 1 corresponds to mode $k = 0$, the coefficient indexed by n/2 corresponds to mode $k = n/2 - 1$, the coefficient indexed by n/2+1 corresponds to mode $k = -n/2$, the coefficient indexed by n corresponds to mode $k = -1$:

```
function trigplot(x,y,m)
%function trigplot(x,y,m)
%plots trigonometric interpolant of data x,y
%assumed to have period 2pi, on [-pi,3pi]
%using m points per interval [x_i,x_{i+1}]
n=length(x);
c=fft(y,n)/n;
k(1:n/2)=0:(n/2-1);
k(n/2+1:n)=-n/2:-1;
z=linspace(-pi,3*pi,2*m*n);
T=c(1)*ones(size(z));
for ind=2:n;
  T = T+c(ind)*exp(j*k(ind)*z);
end
plot(x,y,'r*',z,real(T),'-')
```

Example: Plot $|c_k|$, f and interpolant, for $\sin 4x$, $n = 6, 8, 10, 16$. (See fig 5.4.3)

Example: Plot $|c_k|$, f and interpolant, for $f(x) = 1/(1 + \sin^2 x)$, $n = 4, 8, 16, 32$. (See fig 5.4.3)

Example: Plot $|c_k|$, f and interpolant, for $f(x) = x$, $n = 8, 16, 32, 64$. (See fig 5.4.3)

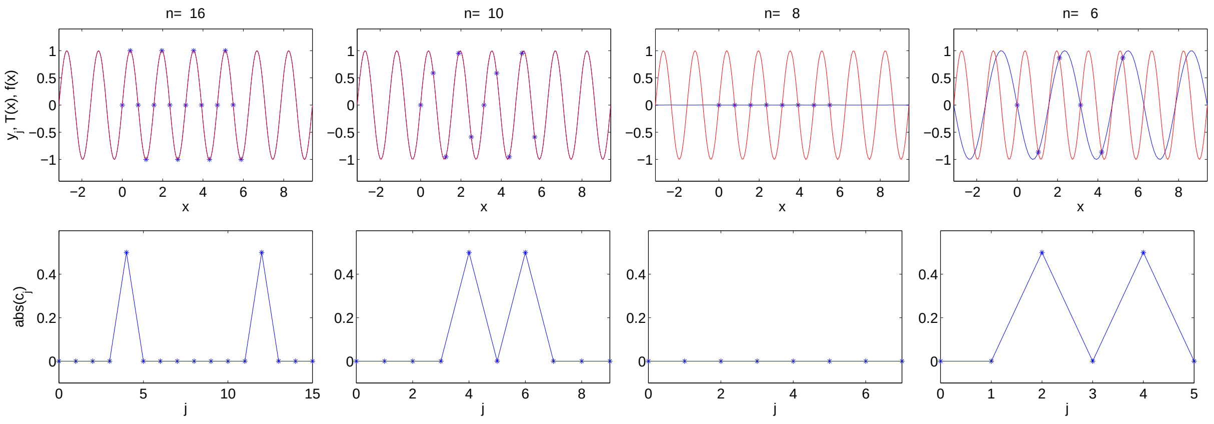


Figure 5: Coefficients $|c_k|$, function f, data, and interpolant, for $\sin 4x$, $n = 16, 10, 8, 6$. This function has only one sine mode $k = 8$. Here we see, first that if $n/2 > k$ the function is identical to the trigonometric interpolant, since the interpolant captures all of the functions nonzero modes. However, if $n/2 \le k$ then the kth mode is **undersampled** and it shows up as a lower order Fourier coefficients. This is called **aliasing error** and occurs when a function has modes bigger than $n/2$ that are not resolved by the mesh of n points. These modes show up as lower order Fourier coefficients!

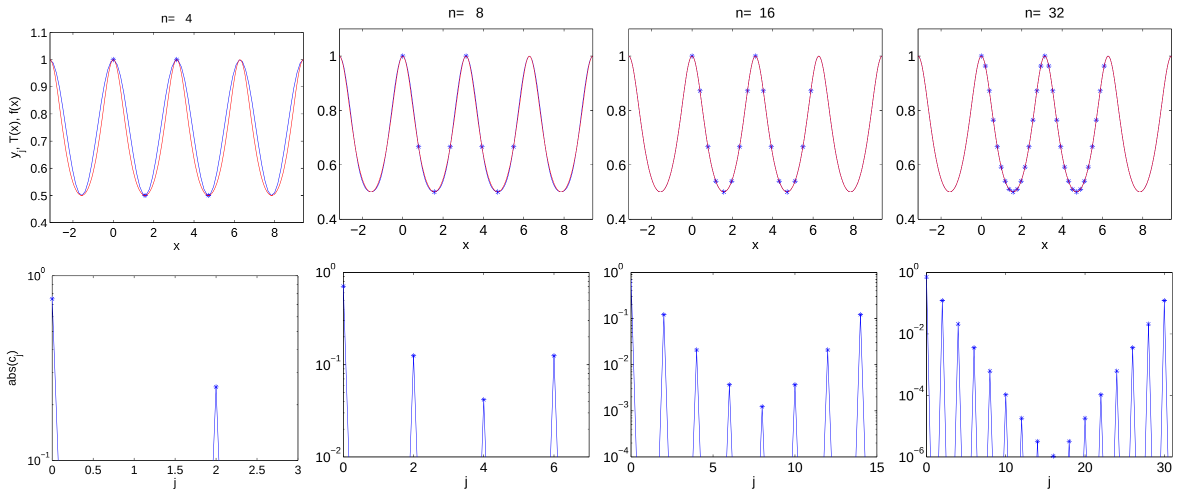


Figure 6: Coefficients $|c_k|$, data, function f and interpolant, for $f(x) = 1/(1 + \sin^2 x)$, $n = 8, 16, 32, 64$. In this example the function is not represented exactly by a finite trigonometric polynomial since the exact function has infinitely many modes. However the trigonometric interpolants are increasingly better approximations as the number of points n increases.

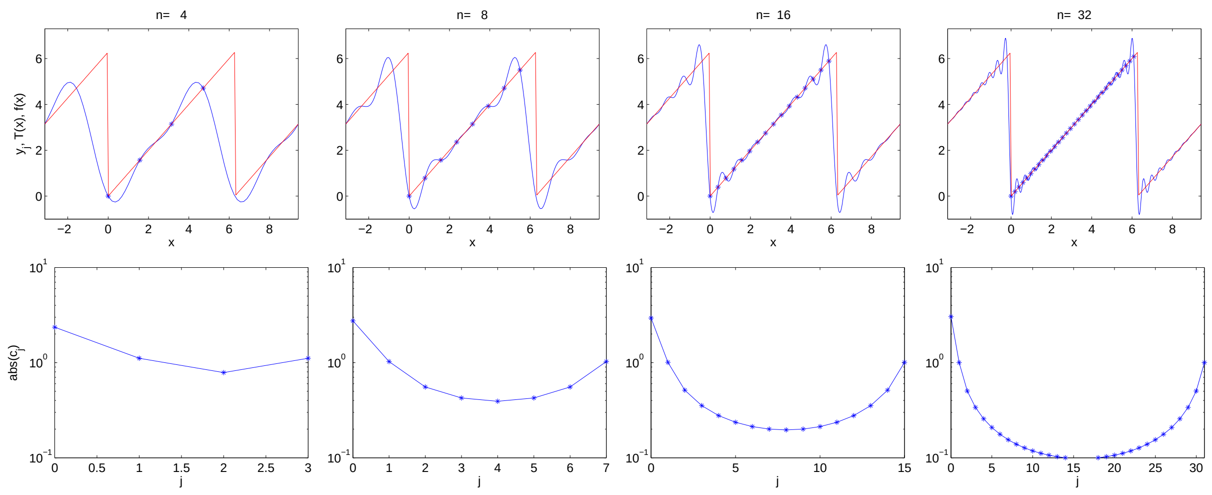


Figure 7: Coefficients $|c_k|$, data, function f and interpolant, for $f(x) = x$, $n = 8, 16, 32, 64$. In this example the periodic extension of f is **discontinuous**. As a result the coefficient decay very slowly and the trigonometric interpolant contains oscillations near the jump that *do not disappear* as n increases. This phenomenon is called **Gibbs phenomena**. Gibbs showed that the oscillations do not disappear as $n \to \infty$ and found the precise size of the oscillations in this limit. So if the function is discontinuous then the trig interpolant is not a good approximation of the function near jump.

Example: (homework) Plot $|c_k|$ for $f(x) = 1/(1 + \sin^2 x)$, $N = 8, 64$. Plot $|c_k|$ for hat function. Plot $|c_k|$ for $f(x) = x$. Compare decay rates of coefficients. Note that the smoother the funtion, the faster the decay rate as k increases.

Example: Plot polynomial interpolant and trig interpolant on uniform (periodic) data. (See Figure 8

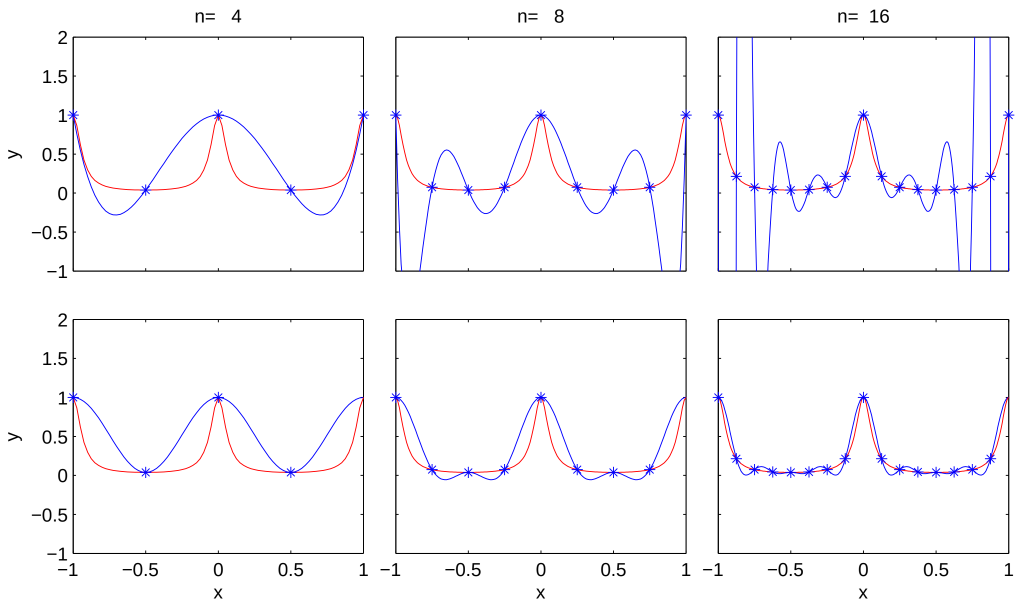


Figure 8: Function $f(x) = 1/(1+25\sin^2 x)$ (red) sampled at n equally spaced points (blue stars), and the polynomial interpolant (top row, in blue), and trigonometric interpolant (bottom row, in blue). Note: if periodic extension of f smooth, then as n increases trig interpolants are increasingly good approximations of f, whereas poly interpolant shows runge phenomena.

Time permitting: Prove that F^{-1} is as listed.

Summary:

- Isolated modes $k < n/2$ are recovered exactly by trig interpolant.
- Modes $k > n/2$ show up as lower order modes in trig interpolant and contaminate low modes, $\Rightarrow$ **aliasing error**
- The smoothness of the periodic extension of f is reflected in the decay rate of the Fourier modes.
- If the periodic extension of f is discontinuous, the coefficients decay slowly and the interpolant is oscillatory near discontinuity.
- For large n, trig interpolants good for uniform points **if** periodic extension of f is smooth.
- For large n, polynomial interpolants bad for uniform points (Runge phenomena), but good for Tschebichef points.

5.4.4 What if period $\tau \neq 2\pi$?

Then need basis functions with period τ, such as

$$\sin(k?x), \cos(k?x), e^{ik?x}$$

So then find a trig interpolant of the form

$$T(x) = \sum_{k=-n/2}^{n/2-1} c_k e^{ik?x}$$

and $x_j =?$, $j = 0, \ldots, n-1$ (they must be equally spaced in $[0, \tau]$) such that

$$T(x_j) = y_j$$

What do these equations look like? What is the resulting linear system for c_j? What changes between this case and the case $\tau = 2\pi$ we already discussed?

5.4.5 Music and Compression

Examples shown in class illustrate the following main points.

- We hear oscillatory pressure waves in the air hitting our ear. An oscillation of a given frequency corresponds to a clean tone. Frequency of 440 Hz (number of oscillation per second) corresponds to a pure A. An A an octave higher would be at 880 Hz.
- Usually what we hear is a sum of various frequencies. Compare pure tone A and tuning fork, which is much richer with many frequencies accompanying the dominant one.
- The component at a certain frequency is captured by the Fourier coefficient for that wavenumber. Its magnitude is typically measured in decibels, wich equals the logarithm of the magnitude relative to some basemeasure.
- For compression, we want to capture the dominant frequency content in a small time interval and represent it by as few coefficients as possible.
- How do we isolate a small time interval? Chopping it off (multiplying by hat function) introduces large high mode frequencies corresponding to discontinuous functions that cant be ignored (the reconstruction would not sound like the origingal if not ignored). Thus not good compression.
- Sudden onset of sound (such in castanets) have similar problem. Exact fourier spectrum has many large modes, and simply ignoring them distorts reconstruction significantly. Again, we need to isolate a small time interval. (This is a classic testcase for compression algorithms: if they reconstruct this signal well, then pretty good.)

- Multiplying by a smooth function that goes from 0 to 1 and back to zero gives you a relatively nice /compact fourier mode spectrum, but is not invertible (cannot recover signal from these modes since basis not orthogonal)
- There is a "modified cosine transform" that does the trick. Wavelets are an alternative but not used for sound, bur rather to compress images.

6 Least Squares Solutions to $A\mathbf{x} = \mathbf{b}$

Example: The systems

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \quad , \qquad \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \ , \tag{14}$$

obviously don't have a solution. Yet Matlab gives a solution to the first. What is it? It does not give solution to second. Why not?

6.1 Least squares solution to $A\mathbf{x} = \mathbf{b}$

If $A_{n\times m}\mathbf{x}_{m\times 1} = \mathbf{b}_{n\times 1}$ has no solution, then the vector $\mathbf{b}$ is not in the set of all $A\mathbf{x}$, for any $\mathbf{x}$. What does the set of all possible $A\mathbf{x}$ look like?

Note that

$$A\mathbf{x} = \mathbf{a}_1 x_1 + \mathbf{a}_2 x_2 + \cdots + \mathbf{a}_n x_n \in \Re^n$$

where $\mathbf{a}_j$ are the columns of A. So the set of all possible $A\mathbf{x}$ is the set of all linear combinations of the columns of A. Called the Span of A.

Definition: $span(A) = \{A\mathbf{x} : \text{ all } \mathbf{x} \in \Re^m\} \subseteq \Re^n$ Note:

- The span of A is a vector space (contains origin, and any linear combination of vectors in this space is in the space).
- It is contained in $\Re^n$, and may be equal to $\Re^n$ (if the columns contain a linearly independent set of n vectors) or be a proper subspace of $\Re^n$.
- **If** $span(A)$ is a proper subspace of $\Re^n$, $span(A) \subset \Re^n$, then there are right hand sides $\mathbf{b} \in \Re^n$ which are not in the $span(A)$ and for those right hand sides $A\mathbf{x} = \mathbf{b}$ has no solution. (There are also right hand sides in the span for which the equation has a solution.) This scenario (span is proper subset) is illustrated geometrically in figure 9. For illustration, we depict the span as a 2-dimensional subspace (a plane) of the bigger space $\Re^3$.

Definition: If $\mathbf{b} \notin span(A)$, the **least squares solution of** $A\mathbf{x} = \mathbf{b}$ is defined to be the vector $\widehat{\mathbf{x}} \in \Re^m$ such that $||A\widehat{\mathbf{x}} - \mathbf{b}||_2$ is minimal. Here $||\mathbf{y}||_2 = \sqrt{\sum_{i=1}^{m} y_i^2}$. The value of $A\widehat{\mathbf{x}}$ is illustrated in figure 10.

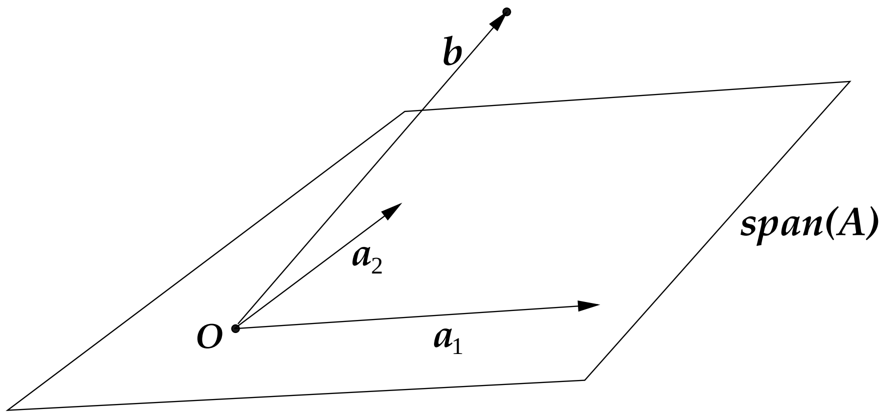


Figure 9: Columns $\mathbf{a}_1, \mathbf{a}_2$ spanning a 2-dimensional subspace in $\Re^3$ and right hand side $\mathbf{b}$ not contained in $span(A)$.

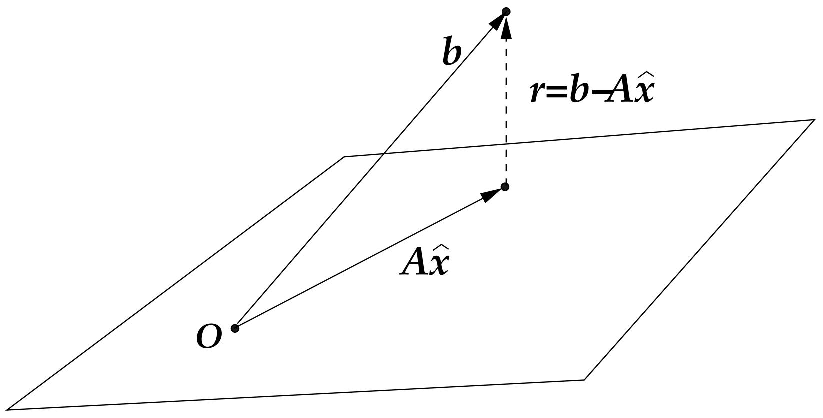


Figure 10: Figure describing least squares solution $\widehat{\mathbf{x}}$ to $A\mathbf{x} = \mathbf{b}$

Definition: The **residual** $\mathbf{r} = \mathbf{b} - A\widehat{\mathbf{x}}$ is the least squares error in the solution $\widehat{\mathbf{x}}$. This error can be measured by its

2-norm $||\mathbf{r}||_2$,

or by the

squared error $||\mathbf{r}||_2^2 = \sum_{i=1}^m r_i^2$,

or by the

root mean squared error (RMSE) $\sqrt{\sum_{i=1}^m r_i^2/m}$.

From figure 10 it is apparent that $\widehat{\mathbf{x}}$ has to be such that $A\widehat{\mathbf{x}}$ is orthogonal to the residual $\mathbf{r}$. This statement can be proven to be true for general dimension m.

Definition: Two vectors $\mathbf{x}, \mathbf{y} \in \Re^m$ are **orthogonal** if the **inner product**, generally denoted by

$$\langle \mathbf{x}, \mathbf{y} \rangle = 0 \ .$$

The inner product of two vectors is a rule that has to satisfy certain properties, and is corresponds to an associated **vector norm**. We have already seen that there are many norms. Since we are trying to minimize the 2-norm of the residual, this is the norm of interest here, and the corresponding inner product is the one you already know well

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y} = \sum_{i=1}^{m} x_i y_i$$

If two vectors $\mathbf{x}, \mathbf{y}$ are orthogonal this is often denoted by $\mathbf{x} \perp \mathbf{y}$.

How do we find the least squares solution $\widehat{\mathbf{x}}$? The solution $\widehat{\mathbf{x}}$ for which $||\mathbf{b} - A\widehat{\mathbf{x}}||_2$ is minimal satifies that the residual is normal to *any* vector in *span*(A). That is, for any $\mathbf{x} \in \Re^n$

$$\begin{aligned} & A\mathbf{x} \perp (\mathbf{b} - A\widehat{\mathbf{x}}) \\ \Rightarrow \quad & (A\mathbf{x})^T(\mathbf{b} - A\widehat{\mathbf{x}}) = 0 \\ \Rightarrow \quad & \mathbf{x}^T A^T(\mathbf{b} - A\widehat{\mathbf{x}}) = 0 \end{aligned}$$

(Here we used the property that for any two commensurate matrices A, B, $(AB)^T = B^T A^T$.) Since this must hold for all $\mathbf{x} \in \Re^n$, it follows that the $n \times 1$ vector

$$A^T(\mathbf{b} - A\widehat{\mathbf{x}}) = 0$$

This yields the **normal equations**

$$A^T A\widehat{\mathbf{x}} = A^T \mathbf{b}$$

Thus, the least squares solution $\widehat{\mathbf{x}}$ solves the normal equations.

Do the normal equations always have a solution? Yes, by construction. However, they can have multiple solutions. (In the latter case, we define the least squares solution to be the one with smallest 2-norm $||\widehat{\mathbf{x}}||_2$.)

Examples: Find the least squares solutions to the systems

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \quad , \qquad \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \quad , \tag{15}$$

by setting up and solving the normal equations. (Solve them using Gauss Elimination and back substitution.) What is the 2-norm of the residual? What is the RSME? Can we now answer the questions we posed at the beginning of this chapter?

Problem with the normal equations: $A^T A$ usually has much larger condition number than A. Therefore solving the normal equations is often ill-posed. This problem is remedied if we use the QR factorization of a matrix, discussed in Section 4.xx.

6.2 Approximating data by model functions

6.2.1 Linear least squares approximation

Problem statement: Given data $\{x_i, y_i\}_{i=1}^n$, find the linear function $f(x) = a + bx$ that minimizes the least squares error

$$\sum_{i=1}^{n} (y_i - f(x_i))^2$$

This function is called the **linear least squares approximation** to the data.

- Draw sketch showing points and least squares line (graph of $y = f(x)$). (See Figure 4.2 on page 198 of text.)
- Set up equations we'd like to solve $a + bx_i = y_i$, $i = 1, \ldots, n$. Write them in matrix form. Note: the unknowns are $\mathbf{c} = (a, b)$.
- Find the least squares solution to the resulting system $A\mathbf{c} = \mathbf{y}$. Note that this is the solution that we want since it minimizes $||\mathbf{y} - A\mathbf{c}||_2$, and $A\mathbf{c}$ is the vector with entries $f(x_i)$.

Example: Find the linear least squares approximation to $(-1, 1), (0, 0), (1, 0), (2, -2)$. Graph in matlab. Find 2-norm of error.

Note: the approach described in this section works because the resulting system is linear in the unknowns a, b. We can use the same approach for slightly different problems, three examples of which will be given next. For another example, see bottom of page 214 - 215 in Sauer.

6.2.2 Quadratic least squares approximation

Problem statement: Given data $\{x_i, y_i\}_{i=1}^n$, find the quadratic function $f(x) = a+bx+cx^2$ that minimizes the least squares error $\sum_{i=1}^n (y_i - f(x_i))^2$ This function is called the **quadratic least squares approximation** to the data.

- Draw sketch showing points and least squares parabola (graph of $y = f(x)$). (See Figure 4.4 on page 201 of text.)
- Set up equations we'd like to solve $a + bx_i + cx_i^2 = y_i$, $i = 1, \ldots, n$. Write them in matrix form. Note: the unknowns are $\mathbf{c} = (a, b, c)$.

- Find the least squares solution to the resulting system $A\mathbf{c} = \mathbf{y}$. Note that this is the solution that we want since it minimizes $||\mathbf{y} - A\mathbf{c}||_2$, and $A\mathbf{c}$ is the vector with entries $f(x_i)$.

Example: Find the quadratic least squares approximation to $(-1, 1), (0, 0), (1, 0), (2, -2)$. Graph in matlab. Find 2-norm of error.

6.2.3 Approximating data by an exponential function

Problem statement: Given data $\{x_i, y_i\}_{i=1}^n$, find an exponential function $f(x) = ae^{bx}$ that best approximates the data

- Problem: the equations we'd like to solve $ae^{bx_i} = y_i$, $i = 1, \ldots, n$. are NOT linear in the unknowns $\mathbf{c} = (a, b)$.
- Fix: instead, solve $\ln(a) + b\ln(x_i) = \ln(y_i)$, $i = 1, \ldots, n$. This system is linear in the unknowns $\ln(a), b$. The least squares solution of this linear system minimizes the least squares error $\sum_{i=1}^n (\ln(y_i) - \ln(f(x_i)))^2$.

Example: The world automobile supply data between 1950 and 1980 is given in the following table (Sauer, Ex 4.8, page 211). Find the exponential function that best approximates the data. What is the error?

year	cars($\times 10^6$)
1950	53.05
1955	73.04
1960	98.31
1965	139.78
1970	193.48
1975	260.20
1980	320.39

6.2.4 Approximating data by an algebraic function

Problem statement: Given data $\{x_i, y_i\}_{i=1}^n$, find an algebraic function $f(x) = ax^p$ that best approximates the data

- Problem: the equations we'd like to solve $ax_i^p = y_i$, $i = 1, \ldots, n$. are NOT linear in the unknowns $\mathbf{c} = (a, p)$.

- Fix: instead, solve $\ln(a) + p\ln(x_i) = y_i$, $i = 1, \ldots, n$. This system is linear in the unknowns $\ln(a), p$.

Example: The mean height and weight of boys ages 2-11 collected in 2002 is given in the following table. Find an exponential function that relates weight and height as $W = aH^p$. What is the error?

age(yrs)	height(m)	weight(kg)
2	0.9120	13.7
3	0.9860	15.9
4	1.0600	18.5
5	1.1300	21.3
6	1.1900	23.5
7	1.2600	27.2
8	1.3200	32.7
9	1.3800	36.0
10	1.4100	38.6
11	1.4900	43.7

6.2.5 Periodic approximations

Problem statement: Given data $\{x_i, y_i\}_{i=1}^{n}$, find a trigonometric polynomials of degree d that best approximates the data.

Solution: Find the trigonometric interpolant of degree $n-1$ that we found in section 4.4, and truncate it, using only the lowest order d terms. (This turns out to be the best approximation in the least squares sense. This fact follows from the orthogonality of the basis functions e^{ikx}, which is discussed in §10.3 of the book.)

Example: Apply to solve example 4.6 (Sauer, page 207), and compare with their solution.

6.3 QR Factorization

7 Numerical Integration (Quadrature)

7.1 Newton-Cotes Rules

This section: Given $f(x)$, integrate interpolating polynomial using as many points as needed.

n=2 points: integrate linear interpolant using 2 points on an interval of size h. We derived the error $\int_{x_0}^{x_0+h} f(x)\,dx - \int_{x_0}^{x_0+h} p_1(x)\,dx = O(h^3)$.

n=3 points: integrate quadrating interpolant using 3 points on an interval of size $2h$. We outlined the derivation of the error $\int_{x_0}^{x_0+2h} f(x)\,dx - \int_{x_0}^{x_0+2h} p_2(x)\,dx = O(h^5)$.

General results: integrate a polynomial interpolant of degree n using $n+1$ points on interval of size nh. The error is

$$\int_{x_0}^{x_0+nh} f(x)\,dx - \int_{x_0}^{x_0+nh} p_2(x)\,dx = \begin{cases} O(h^{n+2}) \text{ if } n \text{ odd} \\ O(h^{n+3}) \text{ if } n \text{ even} \end{cases}$$

These formulas are exact for polynomials of degree n if n is odd, and $n+1$ if n is even. This is called the **degree of precision** of the quadrature rule.

7.2 Composite Newton-Cotes Rules

Here we combine the Newton-Cotes Rules to integrate over a larger interval $[a,b]$.

Using piecewise linear interpolants, linear on intervals of size h: get **trapezoid rule** with error $\int_a^b f(x)\,dx - \int_a^b p(x)\,dx = O(h^2)$.

Using piecewise quadratic interpolants, quadratic on consecutive intervals of size $2h$: get **Simpson's rule** with error $\int_a^b f(x)\,dx - \int_a^b p(x)\,dx = O(h^4)$.

Using piecewise polynomial interpolant, of degree n on consecutive intervals of size nh: get **composite Newton Cotes rule** with error is

$$\int_a^b f(x)\,dx - \int_a^b p_2(x)\,dx = \begin{cases} O(h^{n+1}) \text{ if } n \text{ odd} \\ O(h^{n+2}) \text{ if } n \text{ even} \end{cases}$$

We implemented the trapezoid rule in MATLAB and tested it using

```
    function z = trapez(f,a,b,n)
    % Computes trapezoid rule approximation of integral of f from a to b using n interva
    % sum_{k=0}^n w_k f(x_k) where w_k are the trapezoid weights and x_k are equally spa
    % Input: f (function name), a,b (interval of integration), n (number of subintervals
    % Output: z=sum_{k=0}^n w_k f(x_k) where w_k are the trapezoid weights
    %                                   and x_k are equally spaced points spanning [a,b

    h=(b-a)/n;
    x=a+(0:n)*h; %x has n+1 elements
    y=f(x);

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

(Note: can replace loop by `z=z+sum(y(2:n))`.) The calling script we used to apply and test this function is

```
     clear
     f=inline('1./(1+x.^2)');  a=0; b=4;    ex=atan(4); fpa=0; fpb=-8/17^2;
     for i=1:7
       n(i)=2^i;
       approx=trapez(f,a,b,n(i));
       err(i) = abs(approx-ex);
     end

     m=length(err);
     ratio=err(1:m-1)./err(2:m); ratio'
     h=(b-a)./n;

     figure(1), set(gca,'FontSize',20)
     loglog(h,err,'*-'), xlabel('h'), ylabel('error')
     axis([10^(-2),10,10^(-12),1])
```

7.3 More on Trapezoid Rule

Euler-Mc Laurin Formula for error.

Consequence for periodic integrands. (Actually, trapezoid exact for trig polys of degree n.)

Corrected trapezoid rules.

Richardson extrapolation to obtain a 4th order scheme (which happens to be Simpsons).

7.4 Gauss Quadrature

The Newton-Cotes formulas we have looked at so far are of the form

$$\int_{x_0}^{x_0+nh} f(x)\,dx \approx \sum_{k=0}^{n} w_k f(x_k)$$

where the points $x_k, k = 0, \ldots n$ are prescribed (they are equally spaced). We find $n+1$ weights to integrate polynomials with $n+1$ coefficients (of degree n) exactly.

Gauss quadrature refers to finding quadrature rules of the form

$$\int_{x_0}^{x_n} f(x)\,dx \approx \sum_{k=0}^{n} w_k f(x_k)$$

where we choose the $n+1$ weights w_k **and** the $n+1$ points x_k to find formulas as accurate as possible. These turn out to be exact for polynomials with $2(n+1)$ coefficients (of degree $2n+1$). For details see section §5.5 in book.

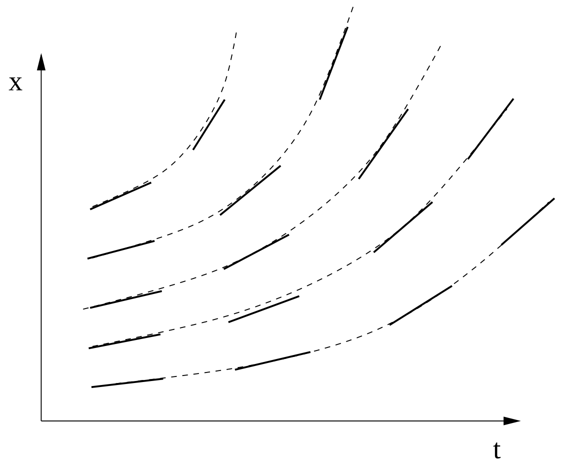


Figure 11: Slopefield $x' = f(t, x)$ (short line segments) and some integral curves (dashed)

8 Numerical Methods for ODEs

8.1 Problem statement

Find a numerical approximation for the solution to

$$\frac{dx}{dt} = f(x, t) \ , \quad x(t_0) = x_0 \tag{16}$$

for $t_0 \leq t \leq T$, where $x(t)$ is a scalar function or a vector valued function. Notice that this differential equation states the slope at any point t, x in the x-t plane. We can thus draw a **slopefield** or **direction field** as shown in the figure. The solution $x(t)$ is everywhere tangent to the slopefield (it has the given slope) and the solution curves are also called **integral curves**.

Two different solution curves dont touch, assuming f is sufficiently nice. This is the result of a **uniqueness theorem**:

> **Theorem:** *If f is continuous and has continuous first derivatives $\partial f/\partial t$, $\partial f/\partial x$, then the initial value problem (16) has a unique solution.*

In class we drew direction fields for three examples: $x' = t$, $x' = x$, $x' = x^2$, all of which can be solved exactly. The difference between the third and the other two is that in that case f is nonlinear. From exact solutions we see that slope grows so fast as x increases that solutions blow up in finite time. This is an example why in general we cannot guarantee solutions for more than just a little neighbourhood of t_0. This is the statement of the following **existence theorem**:

> **Theorem:** *If f is continuous and has continuous first derivatives, then a solution to the initial value problem (16) exists in some neighbourhood of t_0.*

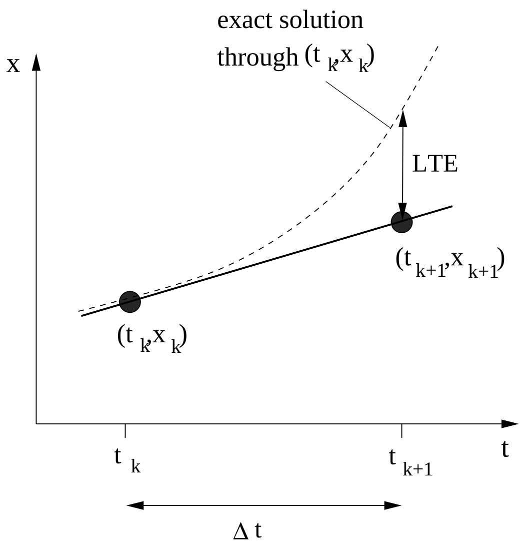


Figure 12: Advancing from t_k to t_{k+1} with Euler's method

Note however that finite time blowup can only occur if f is nonlinear. For linear f solutions are guaranteed to exist for all times.

8.2 Euler's Method

All the methods we'll consider are **step methods**. They consist of discretizing time by $t_k = t_0 + k\Delta t, k = 0, \ldots, m$ where $\Delta t = (T - t)/m$, and finding some rule for approximating the solution at t_k:

$$x_k \approx x(t_k)$$

The left hand side is the numerical approximation, the right hand side is the exact solution at t_k.

Figure 12 indicates the basic idea of Euler's Method. The slope at (t_k, x_k) is approximated by the one sided difference $(x_{k+1} - x_k)/\Delta t$, yielding the method

$$\frac{x_{k+1} - x_k}{\Delta t} = \textit{slope at } (t_k, x_k) = f(t_k, x_k) \ , \quad k = 0, \ldots, m-1$$

that is

$$x_{k+1} = x_k + \Delta t f(t_k, x_k) \ , \quad k = 0, \ldots, m-1$$

Together with an initial condition x_0 given, this rule determines a sequence of values $x_k, k = 0, \ldots, m$ which hopefully approximate $x(t_k)$. Here is the MATLAB function that implements Euler's method for scalar equations. In class we rewrote it for systems of equations.

```
function [t,x]=euler(f,t0,x0,T,delt)
```

```
% Applies the Euler Method to solve x'=f(t,x), x(0)=x0 on (0,T)
% Input: f     (name of function f(x)
%        T     (last time)
%        delt  (stepsize)
%        x0    (initial condition)
% Output: t    (vector, time)
%         x    (vector, solution x(t))
t=t0:delt:T;
n=length(t);
x=zeros(1,n);

x(1) = x0;
for k=1:n-1
  x(k+1) = x(k) + delt*f(t(k),x(k));
end
```

To use this function, type

```
f=inline('t^2+y','t','y')
[t,y]=euler(f,0,1,2,0.5);
```

where f is a function defined using inline, as example shown, or a user defined function.

Does the sequence x_k approximate the exact solution $x(t_k)$? If so, how well? This is the question of whether the method converges, and if so, at what rate. To answer it we need to investigate the **discretization error**. For simplicity, we consider the case when $x(t)$ is a scalar.

Local Truncation Error (LTE): We define the Local Truncation Error (LTE) as the error made in one step with exact input data. It is indicated in figure 12. At the kth step it is defined by

$$LTE_k = x^{loc}(t_{k+1}) - x_{k+1} \tag{17}$$

where $x^{loc}(t)$ is the solution curve going through (t_k, x_k), ie it satisfies

$$dx^{loc}/dt = f(t, x^{loc})\ , \quad x^{loc}(t_k) = x_k \tag{18}$$

and $\quad x_{k+1} = x_k + \Delta t\ f(t_k, x_k)$. If the LTE vanishes as $\Delta t \to 0$, the approximation is called **consistent**.

To compute the LTE for Euler's method we need Taylor series expansions of $x^{loc}(t_{k+1})$ about the base point t_k. Using the Taylor Remainder Theorem it follows (let me drop the *loc*

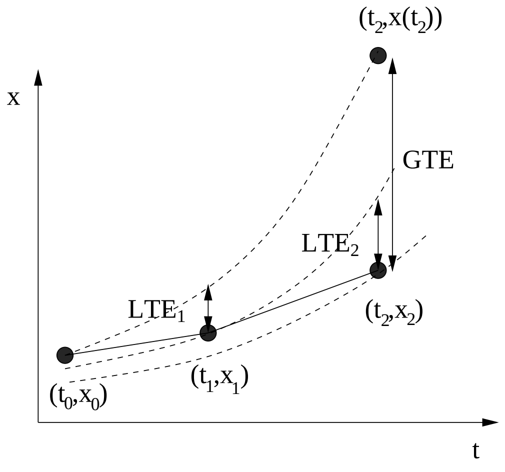


Figure 13: Local and global truncation errors

superscript for now) that:

$$\begin{aligned} x^{loc}(t_{k+1}) &= x(t_k + \Delta t) \\ &= x(t_k) + \Delta t x'(t_k) + \Delta t^2 x''(\eta)/2 \\ &= x_k + \Delta t f(t_k, x_k) + \Delta t^2 x''(\eta)/2 \\ &= x_{k+1} + \Delta t^2 t x''(\eta)/2 \end{aligned}$$

where $\eta \in [t_k, t_{k+1}]$ and we used the differential equation 18 for the third equality. Thus the local truncation error at the kth step is

$$LTE_k = x^{loc}(t_{k+1}) - x_{k+1} = \Delta t^2 x''(\eta)/2$$

The value of η is not known, but IF $x''(t)$ is known to be bounded above, ie there exists some M such that $|x''| \leq M$ for all $a \leq t \leq b$, then

$$LTE_k \leq \Delta t^2 M/2 = O(\Delta t^2) \tag{19}$$

Global Truncation Error (GTE): What we are really interested in is the Global truncation error

$$GTE = |x(t_m) - x_m|$$

that is the (absolute) difference between the exact and the numerical solution at the last time. One may think that the global error is roughly the sum of the local truncation errors. One has to beware however. It may be that the local truncation errors, ie the difference between the solution curves through (t_k, x_k) and (t_{k+1}, x_{k+1}) increases in time (see sketch 13), so that

$$GTE > \sum_k LTE_k$$

As can be seen from the figure, this happens when the solution curves "feather out", or equivalently, when the slope of f increases as x increases, ie when $\partial f/\partial x > 0$. Indeed, it is fairly easy to show that

Theorem: If $\dfrac{\partial f}{\partial x} \leq 0$ then $GTE \leq \displaystyle\sum_{k=1}^{m} LTE_k$.

If we now use Equation 19 and the fact that $m = (T - t_0)/\Delta t$ then

$$\begin{aligned} GTE &\leq \sum_k LTE_k \leq \sum_{k=1}^{m} \Delta t^2 M/2 \\ &= M\Delta t^2 m/2 \\ &= M(T - t_0)\Delta t/2 = O(\Delta t) \end{aligned}$$

If $\partial f/\partial x > 0$ the result that $GTE = O(\Delta t)$ also holds but the proof is more complex (see Gear, Numerical IVPs in ODEs, 1971).

Order of convergence. A method with global error $O(h^p)$ is called **a method of order** p**.** The Euler method is thus a first order method. That is, the method converges (the discretization errors $\to 0$ as $\Delta t \to 0$) but relatively slowly, namely to first order.

In summary, since there are of the order $m = O(1/\Delta t)$ timesteps and the global error is roughly the sum of the local errors, we loose one power of Δt in the decay rate of the global error. This is true in general: for a method of order p the LTE $= O(h^{p+1})$. For example, if we want to find a second order method then the LTE needs to be $O(\Delta t^3)$, etc.

8.3 Second order Method obtained by Richardson Extrapolation

For the Euler method it can be shown specifically that

$$e^{\Delta t} = x(t_k) - x_k^{\Delta t} = \Delta t g(t) + O(\Delta t^2) \tag{20}$$

where $t = k\Delta t$. Richardson extrapolation consists of using the solution obtained with two or more values of Δt to eliminate the highest order term in the error. For example using a time step of $2\Delta t$ one obtains

$$e^{2\Delta t} = x(t_k) - x_k^{2\Delta t} = 2\Delta t g(t) + O(\Delta t^2) \tag{21}$$

and taking the two times Equation (20) minus Equation (21) we obtain

$$2e^{\Delta t} - e^{2\Delta t} = x(t_k) - (2x_k^{\Delta t} - x_k^{2\Delta t}) = O(\Delta t^2)$$

So get second order approximation of solution. This is a way of using a coarse mesh and a fine mesh solution to obtain a higher order approximation at a fixed time.

One can now imagine a method which consists of applying Euler's method and Richardson extrapolation at every timestep. That is, at each time t_k we obtain the solution at t_{k+1} from a linear combination of a coarse mesh and a fine mesh result. The fine mesh result is obtained by doing two steps of Euler of size $\Delta t/2$. The coarse mesh result is obtained by doing one step of Euler of size Δt.

$$\begin{aligned} x_{k+1/2}^{\Delta t/2} &= x_k + \frac{\Delta t}{2} f(t_k, x_k) \\ x_{k+1}^{\Delta t/2} &= x_{k+1/2}^{\Delta t/2} + \frac{\Delta t}{2} f(t_{k+1/2}, x_{k+1/2}^{\Delta t/2}) \\ x_{k+1}^{\Delta t} &= x_k + \Delta t f(t_k, x_k) \end{aligned}$$

We now take the appropriate linear combination

$$x_{k+1} = 2x_{k+1}^{\Delta t/2} - x_{k+1}^{\Delta t} \tag{22}$$

This can be summarized as

$$\begin{aligned} &k_1 = \Delta t f(t_k, x_k) \\ &k_2 = \Delta t f(t_k + \frac{\Delta t}{2}, x_k + \frac{k_1}{2}) \\ &x_{k+1} = x_k + k_2 \end{aligned}$$

This is called the Midpoint method, since it is an approximation to $\frac{x_{k+1}-x_k}{\Delta t} = x'_{k+1/2}$, which is a second order approximation to the derivative, as opposed to the first order approximation we used to derive the Euler method. The method can be shown to be of second order (this will also follow from next section).

8.4 Second order Method obtained using Taylor Series

In general, can look for scheme of the form

$$\begin{aligned} &k_1 = \Delta t f(t_k, x_k) \\ &k_2 = \Delta t f(t_k + \alpha h, x_k + \beta k_1) \\ &x_{k+1} = x_n + a k_1 + b k_2 \end{aligned}$$

Choose α, β, a, b such that LTE as small as possible. Write Taylor expansion for actual solution $x(t_{k+1})$ of the PDE $x' = f(x, t)$.

$$\begin{aligned} x(t_{k+1}) &= x(t_k) + \Delta t x'(t_k) + \frac{\Delta t^2}{2} x''(t_k) + O(\Delta t^3) \\ &= x(t_k) + \Delta t f(t_k, x_k) + \frac{\Delta t^2}{2} \Big[\frac{\partial f}{\partial t}(t_k, x_k) + \frac{\partial f}{\partial x}(t_k, x_k) f(t_k, x_k) \Big] + O(\Delta t^3) \end{aligned}$$

Write Taylor expansion for a solution x_{k+1} of the difference scheme.

$$\begin{aligned} x_{k+1} &= x_k + a\Delta t f(t_k, x_k) + b\Delta t \Big[f(t_k, x_k) + \frac{\partial f}{\partial t}(t_k, x_k)\alpha\Delta t + \frac{\partial f}{\partial x}(t_k, x_k)\beta\Delta t f(t_k, x_k) + O(\Delta t^2) \Big] \\ &= x_k + (a+b)\Delta t f(t_k, x_k) + \Delta t^2 \Big[b\alpha \frac{\partial f}{\partial t}(t_k, x_k) + b\beta \frac{\partial f}{\partial x}(t_k, x_k) f(t_k, x_k) \Big] + O(\Delta t^3) \end{aligned}$$

Choose the parameters α, β, a, b so that they match to third order. This gives a $LTE = O(\Delta t^3)$, and thereby a second order method.

$$a + b = 1, \quad \alpha = \beta = \frac{1}{2b}$$

For $b = 1$, $a = 0, \alpha = \beta = 1/2$, get midpoint method. For $b = a = 1/2$, $\alpha = \beta = 1$, get the Heun method, or modified trapezoidal method, since it approximates $\frac{x_{k+1} - x_k}{\Delta t} = \frac{x'_k + x'_{k+1}}{2}$ (convince yourself of this by writing the method out).

8.5 4th order Runge Kutta Method

By similar approach, one can derive 4th oder methods. One starts with an Ansatz of the form (3.1), but with 4 intermediate steps k_1, k_2, k_3, k_4. The condition that the LTE be $O(h^5)$ yields a set of equations for the unknowns $\alpha_i, \beta_i, a_i, b_i$ which has infinitely many solutions. The most common 4th order scheme derived this way is the 4th order Runge Kutta scheme (RK4):

$$\begin{aligned}
k_1 &= \Delta t f(t_k, x_k) \\
k_2 &= \Delta t f(t_k + \Delta t/2, x_k + k_1/2) \\
k_3 &= \Delta t f(t_k + \Delta t/2, x_k + k_2/2) \\
k_4 &= \Delta t f(t_k + \Delta t, x_k + k_3) \\
x_{k+1} &= x_k + (k_1 + 2k_2 + 2k_3 + k_4)/6
\end{aligned}$$

The MATLAB code to implement RK4 for scalar equations is

```
function [t,x]=rk4(f,t0,x0,T,h)
% applies RK4 to solve x'=f, x(0)=x0 on (0,T)

t=t0:h:T;
n=length(t);
x=zeros(1,n);

x(1) = x0;
for j=1:n-1
  k1 = h*f(t(j),x(j));
  k2 = h*f(t(j)+h/2,x(j)+k1/2);
  k3 = h*f(t(j)+h/2,x(j)+k2/2);
  k4 = h*f(t(j)+h,x(j)+k3);
  x(j+1) = x(j) + (k1+2*k2+2*k3+k4)/6;
end
```