Lec 01: Getting Started

On and Off

MATLAB as a calculator

interfaces (Command Window)

arithmetic operations

clearing screen, `clc`

large numbers/small numbers: scientific notation

formatting outputs

```
format short/long
```

```
format loose/compact
```

```
format rat
```

Variables

predefined variables

`pi`, `i`, `j`, `eps`, `realmax`, `realmin`, `Inf`, `NaN`

user-defined variables

rules of naming

- lowercase, uppercase, numbers, and underscore
- no spaces: `my var` -> `my_var`, `myVar`
- no number at the beginning: `2x` -> `twoX`, `x2`

clear

’Hello World’

displaying text

displaying text and number

```
disp(['the␣number␣is␣', num2str(rand())])
```

Script m-file (demo)

commenting: `%`, `%%`

suppressing outputs

`input` and `disp` to make it interactive

Example: Quadratic Equation Solver

Lec 02: More On Scripting and If-Statement

Live Script

interface

text vs. code

markup

running scripts

insert - math, images

non-executable codes

exporting to pdf

If-Statement

Quad Eqn Solver

Angle Finder

References

Lec 03: Relational and Logical Operations

Another Way to Display Text

FPRINTF: Alternate Displaying Function

Combine literal text with numeric data.

- Number of digits to display

```
fprintf('There␣are␣%d␣days␣in␣a␣year.\n', 365)
```

- Complex number

```
z = exp(1i*pi/4);
fprintf('%f+%fi\n', real(z), imag(z));
```

FPRINTF: Formatting Operator

```
%[field width][precision][conversion character]
```

e.g. `%12.5f`.

- `%`: marks the beginning of a formatting operator
- `[field width]`: maximum number of characters to print; optional
- `[precision]` number of digits to the right of the decimal point; optional
- `[conversion character]`

`%d`	integer
`%f`	fixed-point notation
`%e`	exponential notation
`%g`	the more compact of `%f` or `%e`
`%s`	string array
`%x`	hexadecimal

Relational and Logical Operators

Relational Operators

How are two numbers X and Y related?

- `[X>Y]` Is X greater than Y?
- `[X<Y]` Is X less than Y?
- `[X>=Y]` Is X greater than or equal to Y?
- `[X<=Y]` Is X less than or equal to Y?
- `[X==Y]` Is X equal to Y?
- `[X~=Y]` Is X not equal to Y?

The symbols used between X and Y are called the **relational operators**.

Logical Variables and Logical Operators

- A relational statement evaluates to either **True(1)** or **False(0)**; these are called **logical variables** or **boolean variables**.
- As arithmetic operators (`+`, `-`, `*`, `/`) put together two numbers and produce other numbers, **logical operators** combine two logical variables to produce other logical variables.
- **Logical Operators**: *and*, *or*, *not*, *xor*

Logical Operator: `&&` (AND)

Let `A` and `B` be two logical variables. The `&&` operation is completely defined by the following truth table:

`A`	`B`	`A && B`
F	F	F
F	T	F
T	F	F
T	T	T

Note that `A && B` is true if and only if both `A` and `B` are true.

Logical Operator: `||` (OR)

Let `A` and `B` be two logical variables. The `||` operation is completely defined by the following truth table:

`A`	`B`	`A || B`
F	F	F
F	T	T
T	F	T
T	T	T

Note that `A || B` is false if and only if both `A` and `B` are false.

Logical Operator: `xor` (exclusive or)

This is a special variant of the `||` operator.

`A`	`B`	`xor(A,B)`
F	F	F
F	T	T
T	F	T
T	T	F

Note that `xorg(A,B)` is true if only one of `A` or `B` is true.

Logical Operator: ~ (NOT)

This is a negation operator.

A	~A
F	T
T	F

Combination of Logical Operations

Let `A` and `B` be logical variables. Then `~(A && B)` and `~A || ~B` are equivalent:

`A`	`B`	`A && B`	`~(A && B)`
F	F		
F	T		
T	F		
T	T		

`A`	`B`	`~A`	`~B`	`~A` || `~B`
F	F			
F	T			
T	F			
T	T			

Examples

Quadratics Revisited

Consider a monic quadratic function $q(x) = x^2 + bx + c$ on a close interval $[L, R]$.

- Critical point: $x_c = -b/2$
- If $x_c \in (L, R)$, $q(x)$ attains the (global) minimum at x_c; otherwise, the minimum occurs at one of the endpoints $x = L$ or $x = R$.

Question

Write a program which determines whether the critical point of $q(x)$ falls on the interval.

Initialization

```
b = input('Enter␣b:␣');
c = input('Enter␣c:␣');
L = input('Enter␣L:␣');
R = input('Enter␣R␣(L<R):␣');
clc
fprintf('Function:␣x^2␣+␣bx␣+␣c,␣b␣=␣%5.2f,␣c␣=␣%5.2f\n', b, c)
fprintf('Interval:␣[L,␣R],␣L␣=␣%5.2f,␣R␣=␣%5.2f\n', L, R)
xc = -b/2;
```

Main Fragment

```
if L < xc && xc < R
    fprintf('Interior␣critical␣point␣at␣x_c␣=␣%5.2f\n', xc)
else
    disp('Either␣xc␣<=␣L␣or␣xc␣>=␣R')
end
```

Main Fragment – another way

```
if xc <= L || xc >= R
    disp('Either␣xc␣<=␣L␣or␣xc␣>=␣R')
else
    fprintf('Interior␣critical␣point␣at␣x_c␣=␣%5.2f\n', xc)
end
```

Main Fragment – yet another way

```
if ~(xc <= L || xc >= R)
    fprintf('Interior␣critical␣point␣at␣x_c␣=␣%5.2f\n', xc)
else
    disp('Either␣xc␣<=␣L␣or␣xc␣>=␣R')
end
```

The simplest `if` statement?

So far, we have seen

- `if-else` statement
- `if-elseif-else` statement

The simplest `if` statement is of the form

```
if [condition]
  [statements to run]
end
```

Input Errors

If a user mistakenly provides L that is larger than R, fix it silently by swapping L and R.

```
if L > R
    tmp = L;
    L = R;
    R = tmp;
end
```

I will show you how to send an error message and halt a program later.

Exercises

Exercise 1: Simple Minimization Problem

Question

Write a program which $x_{\min} \in [L, R]$ at which $q(x)$ is minimized and the minimum value $q(x_{\min})$.

- This can be done with `if-elseif-else`

Exercise 2: Leap Year

Question

Write a script which determines whether a given year is a leap year or not. A year is a leap year if

- it is a multiple of 4;
- it is not a multiple of 100;
- it is a multiple of 400.

Useful: `mod` function.

Pseudocode

```
if [YEAR] is not divisible by 4
    it is a common year
elseif [YEAR] is not divisible by 100
    it is a leap year
elseif [YEAR] is not divisible by 400
    it is a common year
else
    it is a leap year
end
```

Exercise 3: Angle Finder

Question

Let x and y be given, not both zero. Determine the angle $\theta \in (-\pi, \pi]$ between the positive x-axis and the line segment connecting the origin to (x, y).

Four quandrants:

- 1st or 4th ($x >= 0$): $\theta = \tan^{-1}(y/x)$
- 2nd ($x < 0, y >= 0$): $\theta = \tan^{-1}(y/x) + \pi$
- 3rd ($x < 0, y < 0$): $\theta = \tan^{-1}(y/x) - \pi$

Useful: `atan` (inverse tangent function)

Extended Inverse Tangent

```
if x > 0
    theta = atan(y/x)
elseif y >= 0
    theta = atan(y/x) + pi
else
    theta = atan(y/x) - pi
end
```

- MATLAB provides a function that exactly does this: `atan2(x,y)`.
- **Further Exploration**: What would you do if you are asked to find the angle $\theta \in [0, 2\pi)$, with `atan` alone or with `atan2`?

Lec 04: FOR-Loops

Opening Example

Approximating π

Suppose the circle $x^2 + y^2 = n^2$, $n \in \mathbb{N}$, is drawn on graph paper.

- The area of the circle can be approximated by counting the number uncut grids, N_{in}.

  $$\pi n^2 \approx N_{\text{in}},$$

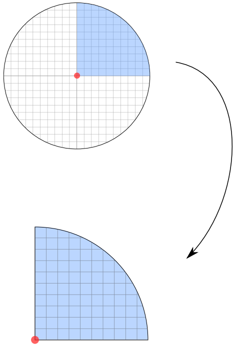
  and so

  $$\pi \approx \frac{N_{\text{in}}}{n^2}.$$

- Using symmetry, may only count the grids in the first quadrant and modify the formula accordingly:

  $$\pi \approx \frac{4N_{\text{in},1}}{n^2},$$

  where $N_{\text{in},1}$ is the number of inscribed grids in the first quadrant.



Approximating π

Problem Statement

Write a script that inputs an integer n and displays the approximation of π by

$$\rho_n = \frac{4N_{\mathrm{in},1}}{n^2},$$

along with the (absolute) error $|\rho_n - \pi|$.

Note. The approximation gets enhanced and approaches the true value of π as $n \to \infty$.

Introduction to =FOR=Loop

Strategy: Iterate

The key to this problem is to count the number of uncut grids in the first quadrant programmatically.

Set $N_{\text{in},1} = 0$.

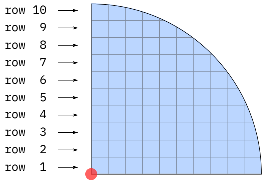
Count the number of uncut grids in `row 1`. Add that to $N_{\text{in},1}$.

Count the number of uncut grids in `row 2`. Add that to $N_{\text{in},1}$.

$\vdots$

Count the number of uncut grids in `row 10`. Add that to $N_{\text{in},1}$.

Set $\rho_{10} = 4N_{\text{in},1}/10^2$.



MATLAB Way

The repeated counting can be delegated to MATLAB using `for`-loop. The procedure outlined above turns into

```
Assume n is initialized and set $N_{\text{in},1}$ to zero.

for k = 1:n
    Count the number of uncut grids
    in row k. Add that to $N_{\text{in},1}$.
end

Set $\rho_{10} = 4N_{\text{in},1}/10^2$.
```

Counting Uncut Tiles

The problem is reduced to counting the number of uncut grids in each row.

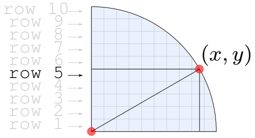
- The x-coordinate of the intersection of the top edge of the kth row and the circle $x^2 + y^2 = n^2$ is

$$x = \sqrt{n^2 - k^2}.$$

- The number of uncut grids in the kth row is the largest integer less than or equal to this value, *i.e.*,

$$\lfloor\sqrt{n^2 - k^2}\rfloor. \qquad \text{(floor function)}$$

- MATLAB provides `floor`.



For $n = 10$ and $k = 5$:

$$\begin{aligned} x &= \sqrt{n^2 - k^2} \\ &= \sqrt{10^2 - 5^2} = 8.6602\ldots \end{aligned}$$

Main Fragment Using FOR-Loop

```
N1 = 0;
for k = 1:n
    m = floor(sqrt(n^2 - k^2));
    N1 = N1 + m;
end
rho_n = 4*N1/n^2;
```

Exercise. Complete the program.

Exercise 1: Overestimation

Question

Note that ρ_n is always less than π. If N_1 denotes the total number of grids, both cut and uncut, within the quarter disk, then $\mu_n = 4N_1/n^2$ is always larger than π. Modify the previous (complete) script so that it prints ρ_n, μ_n, and $\mu_n - \rho_n$.

- `ceil`, an analogue of `floor`, is useful.

Notes on `FOR`-Loop

- The construct is used when a code fragment needs to be repeatedly run. The number of repetition is known in advance.

```
for <loop variable> = 1:<arithmetic expression>
<code fragment>
end
```

- Examples:

```
for k = 1:3
    fprintf('k = %d\n', k)
end
```

```
nIter = 100;
for k = 1:nIter
    fprintf('k = %d\n', k)
end
```

Caveats

Run the following script and observe the displayed result.

```
for k = 1:3
    disp(k)
    k = 17;
    disp(k)
end
```

- The loop header `k = 1:3` guarantees that `k` takes on the values 1, 2, and 3, one at a time even if `k` is modified within the loop body.
- However, it is a recommended practice that the value of the loop variable is *never* modified in the loop body.

Loops and Simulations

Simulation Using `rand`

`rand` is a built-in function which generate a (uniform) “random” number between 0 and 1. Try:

```
for k = 1:10
    x = rand();
    fprintf('%10.6f\n', x);
end
```

Let’s use this function to solve:

Question

A stick with length 1 is split into two parts at a random breakpoint. *On average*, how long is the shorter piece?

Program Development – Single Instance

Consider breaking *one* stick.

- Random breakage can be simulated with `rand`; denote by $x \in (0, 1)$.
- The length of the shorter piece can be determined using `if`-construct; denote by $s \in (0, 1/2)$.

```
x = rand();         % x: the location of breakage
if x <= 0.5         % if x ⩽ 0.5
    s = x;          % shorter part has length x
else                % otherwise
    s = 1-x         % shorter part has length 1 − x
end
```

Program Development – Multiple Instances

- Repeat the previous multiple times using a `for`-loop. Pseudocode: if 1000 breaks are to be simulated:

```
nBreaks = 1000;
for k = 1:nBreaks
<code from previous page>
end
```

- But how are calculating the *average* length of the shorter pieces?

Calculating Average Using Loop

Recall how the total number of uncut grids were calculated using iterations.

```
Assume n is initialized and set $N_{\text{in},1}$ to zero.

for k = 1:n
    Count the number of uncut grids in row k.
    Add that to $N_{\text{in},1}$.
end

The value of $N_{,1}$ is the total numbers of uncut grids.
```

Similarly, we can compute an average by:

```
Assume n is initialized and set $s$ to zero.

for k = 1:n
    Simulate a break and find the length of the shorter piece. Add that to $s$.
end

Set $s_{\text{avg}} = s/\mathtt{n}$.
```

Complete Solution

```
nBreaks = 1000;
s = 0;
for k = 1:nBreaks
    x = rand();
    if x <= 0.5
        s = s + x;
    else
        s = s + (1-x);
    end
end
s_avg = s/nBreaks;
```

Exercise 2: Game of 3-Stick

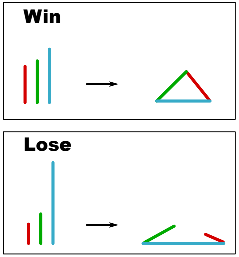
Game: 3-Stick

Pick three sticks each having a random length between 0 and 1. You win if you can form a triangle using the sticks; otherwise, you lose.

Question

Estimate the probability of winning a game of 3-Stick by simulating one million games and counting the number of wins[a].

[a]Of course, divide it by 1,000,000 to calculate the probability



Lec 05: WHILE-Loops

Pop Quiz

Understanding Loops

Question 1

How many lines of output are produced by the following script?

```
for k = 100:200
    disp(k)
end
```

A 99

B 100

C 101

D 200

Understanding Loops

Question 2

How many lines of output are produced by the following script?

```
for k = 100:200
    if mod(k,2) == 0
        disp(k)
    end
end
```

- A 50
- B 51
- C 100
- D 101

FOR-Loop: Tips

- Basic loop header:

```
for <loop var> = 1:<ending value>
```

- To adjust starting value:

```
for <loop var> = <starting value>:<ending value>
```

- To adjust step size:

```
for <loop var> = <starting value>:<step size>:<ending value>
```

Examples

- To iterate over 1, 3, 5, . . . , 9: [*step size = 2*]

```
for k = 1:2:9
```

or

```
for k = 1:2:10
```

- To iterate over 10, 9, 8, . . . , 1: [*negative step size*]

```
for k = 10:-1:1
```

Introduction to `WHILE`-Loop

Need for Another Loop

- `For`-loops are useful when the number of repetitions is known in advance.

  "Simulate the tossing of a fair coin 100 times and print the number of Heads."

- It is not very suitable in other situations such as

  "Simulate the tossing of a fair coin until the gap between the number of Heads and that of Tails reaches 10."

  We need another loop construct that terminates as soon as $|N_{\mathrm{H}} - N_{\mathrm{T}}| = 10$.

WHILE-Loop Basics

WHILE-loop is used when a code fragment needs to be executed repeatedly *while* a certain condition is true.

```
while <continuation criterion>
<code fragment>
end
```

- The number of repetitions is *not* known in advance.
- The continuation criterion is a boolean expression, which is evaluated at the start of the loop.
    - If it is true, the loop body is executed. Then the boolean expression is evaluated again.
    - If it is false, the flow of control is passed to the end of the loop.

Simple WHILE-Loop Examples

```
k = 1; n = 10;
while k <= n
    fprintf('k = %d\n', k)
    k = k+1;
end
```

```
k = 1;
while 2^k < 5000
    k = k+1;
end
fprintf('k = %d\n', k)
```

FOR-Loop to WHILE-Loop

A for-loop can be written as a while-loop. For example,

FOR

```
s = 0;
for k = 1:4
    s = s + k;
    fprintf('%2d %2d\n', k, s)
end
```

WHILE

```
k = 0; s = 0;
while k < 4
    k = k + 1; s = s + k;
    fprintf('%2d %2d\n', k, s)
end
```

- Note that k needed to be initialized before the while-loop.
- The variable k needed to be updated inside the while-loop body.

Examples

Up/Down Sequence

Question

Pick a random integer between 1 and 1,000,000. Call the number n and repeat the following process:

- If n is even, replace n by $n/2$.
- If n is odd, replace n by $3n + 1$.

Does it ever take more than 1000 updates to reach 1?

- To generate a random integer between 1 and k, use `randi`, *e.g.*,

```
randi(k)
```

- To test whether a number n is even or odd, use `mod`, *e.g.*,

```
mod(n, 2) == 0
```

Attempt Using FOR-Loop

```
for step = 1:1000
    if mod(n,2) == 0
        n = n/2;
    else
        n = 3*n + 1;
    end
    fprintf(' %4d %7d\n', step, n)
end
```

- Note that once n becomes 1, the central process yields the following pattern:
$$1, 4, 2, 1, 4, 2, 1, \ldots$$
- This program continues to run even after n becomes 1.

Solution Using `WHILE`-Loop

```
step = 0;
while n > 1
    if mod(n,2) == 0
        n = n/2;
    else
        n = 3*n + 1;
    end
    step = step + 1;
    fprintf(' %4d %7d\n', step, n)
end
```

- This shuts down when n becomes 1!

Exercise: Gap of 10

Question

Simulate the tossing of a fair coin until the gap between the number of Heads and that of Tails reaches 10.

Summary

- `For`-loop is a programming construct to execute statements repeatedly.

```
for <loop index values>
<code fragment>
end
```

- `While`-loop is another construct to repeatedly execute statements. Repetition is controlled by the termination criterion.

```
while <termination criterion is not met>
<repeat these statements>
end
```

Lec 06: Arrays in MATLAB

Basics of Arrays in MATLAB

Introduction to Arrays

Vectors and matrices are often collectively called **arrays**.

Notation

- $\mathbb{R}^m$ (or $\mathbb{C}^m$): the set of all real (or complex) **column vectors** with m elements.
- $\mathbb{R}^{m\times n}$ (or $\mathbb{C}^{m\times n}$): the set of all real (or complex) $m \times n$ matrices.
- If $\mathbf{v} \in \mathbb{R}^m$ with $\mathbf{v} = (v_1, v_2, \ldots, v_m)^{\mathrm{T}}$, then for $1 \leqslant i \leqslant m$, $v_i \in \mathbb{R}$ is called the ith *element* or the ith *index* of $\mathbf{v}$.
- If $A \in \mathbb{R}^{m\times n}$ with $A = (a_{i,j})$, then for $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$, $a_{i,j} \in \mathbb{R}$ is the element in the ith row and jth column of A.

Creating Arrays

- A *row vector* is created by

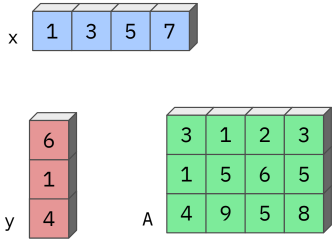
```
x = [1 3 5 7];
x = [1,3,5,7];
```

- A *column vector* is created by

```
y = [6; 1; 4];
y = [6 1 4].';
```

- A matrix is formed by

```
A = [3 1 2 3;
      1 5 6 5;
      4 9 5 8];
```



The MATLAB expression `x.'` means $\mathbf{x}^{\mathrm{T}}$ while `x'` means $\mathbf{x}^{\mathrm{H}} = (\mathbf{x}^{*})^{\mathrm{T}}$.

Shape of Arrays

- To find the number of elements of a vector:

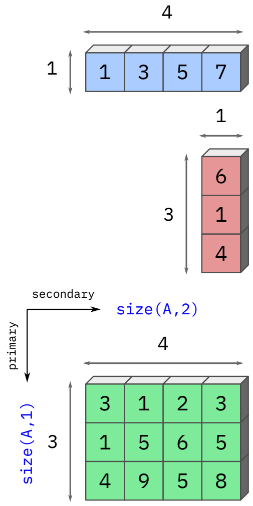
```
length(x)
length(y)
```

- To find the number of rows/columns of an array:

```
size(A,1) % # of rows
size(A,2) % # of cols
size(A)   % both
```

- To find the total number of elements of an array:

```
numel(A)
```



Shape of Arrays (Notes)

- For a matrix `A`, `length(A)` yields the larger of the two dimensions.
- The result of `size(A)` can be stored in two different ways:

```
szA = size(A)
[m, n] = size(A)
```

Q. How are they different?

- All of the following generate *empty arrays*.

```
[]
[1:0]
[1:0].'
```

Q. What are their *sizes*? What are their `numel` values?

Getting/Setting Elements of Arrays

- To access the ith element of a vector:

```
x(2)
y(3)
```

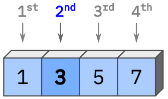
- To access the (i,j)-element of a matrix:

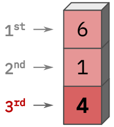
```
A(2,4)
```

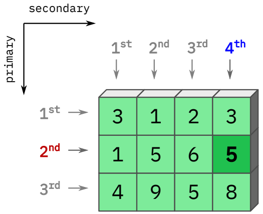
- To assign values to a specific element:

```
x(2) = 2
A(2,4) = 0
```

- Indices start at **1** in MATLAB, not at 0!







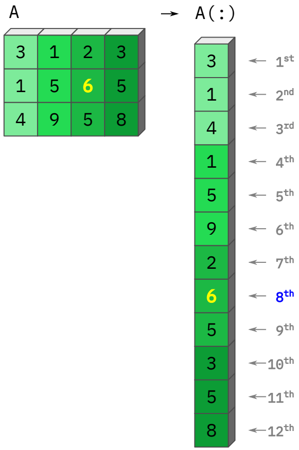
Linear Indexation and Straightening of Matrix

- MATLAB uses *column-major* layout by default, meaning that the elements of the columns are contiguous in memory.
- Consequently, one can get/set an element of a matrix using a single index.

```
A(8)
```

- An array can be put into a column vector using

```
A(:)
```



Array Operations

Two Kinds of Transpose

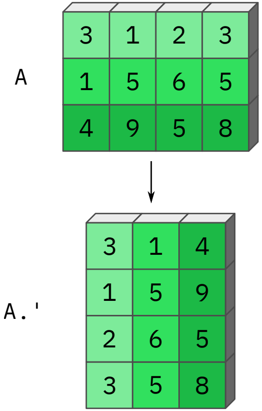
- The transpose of an array: A^{T}

```
A.'
```

- The conjugate transpose of an array: $A^{\mathrm{H}} = A^* = \overline{A}^{\mathrm{T}}$

```
A'
```

- If $A \in \mathbb{R}^{m\times n}$, $A^{\mathrm{H}} = A^{\mathrm{T}}$. So, if `A` is a real array, `A.'` and `A'` are equivalent.



Standard Arithmetic Operation

Standard arithmetic operations seen in linear algebra are executed using the familiar symbols.

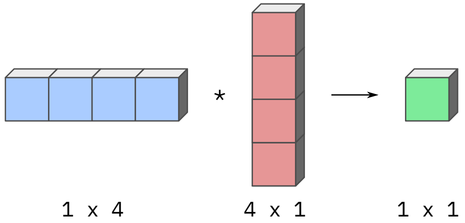
- Let $\texttt{A}, \texttt{B} \in \mathbb{R}^{m\times n}$ and $\texttt{c} \in \mathbb{R}$.
  - $\texttt{A} \pm \texttt{B}$: elementwise addition/subtraction $(A \pm B)$
  - $\texttt{A} \pm \texttt{c}$: *shifting* all elements of $\texttt{A}$ by $\pm$ $\texttt{c}$ $(A \pm c)$
- Let $\texttt{A} \in \mathbb{R}^{m\times p}$, $\texttt{B} \in \mathbb{R}^{p\times n}$, and $\texttt{c} \in \mathbb{R}$.
  - $\texttt{A}{\star}\texttt{B}$: the $m \times n$ matrix obtained by the *linear algebraic* multiplication (AB)
  - $\texttt{c}{\star}\texttt{A}$: scalar multiple of $\texttt{A}$ (cA)
- Let $\texttt{A} \in \mathbb{R}^{m\times m}$ and $\texttt{n} \in \mathbb{N}$.
  - $\texttt{A\^{}n}$: the $\texttt{n}$-th power of $\texttt{A}$; the same as $\texttt{A}{\star}\texttt{A}{\star} \cdots {\star}\texttt{A}$ ($\texttt{n}$ times) (A^n)

Standard Arithmetic Operation – Inner Products

Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ be column vectors. The *inner product* of $\mathbf{x}$ and $\mathbf{y}$ is calculated by

$$\mathbf{x}^{\mathrm{T}}\mathbf{y} = x_1y_1 + x_2y_2 + \cdots + x_my_m = \sum_{j=1}^{m} x_jy_j \in \mathbb{R}.$$

In MATLAB, simply type `x'*y`.

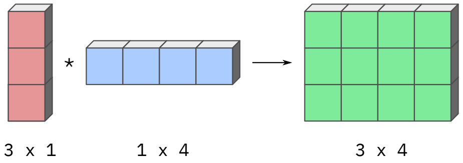


Standard Arithmetic Operation – Outer Products

Let $\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n$ be column vectors. The *outer product* of $\mathbf{x}$ and $\mathbf{y}$ is calculated by

$$\mathbf{x}\mathbf{y}^{\mathrm{T}} = \begin{bmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & x_2y_2 & \cdots & x_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_my_1 & x_my_2 & \cdots & x_my_n \end{bmatrix} \in \mathbb{R}^{m\times n}.$$

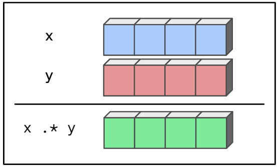
In MATLAB, simply type `x*y'`.



Elementwise Multiplication ($.*$)

- To multiply entries of two arrays of same size, element by element:

```
x .* y
```



Elementwise Division (./)

- To divide entries of an array by corresponding entries of another same-sized array:

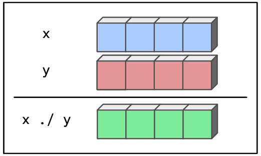
```
x ./ y
```

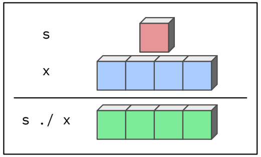
- To divide a number by multiple numbers (specified by entries of an array):

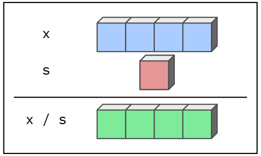
```
s ./ y
```

- To divide all entries of an array by a common number:

```
x / s
```







Elementwise Exponentiation (.^)

- To raise all entries of an array to (different) powers:

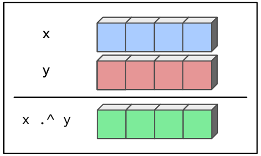
```
x .^ y
```

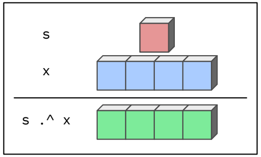
- To raise a number to multiple powers (specified by entries of an array):

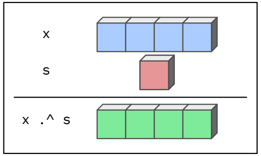
```
s .^ x
```

- To raise all entries of an array to a common power:

```
x .^ s
```



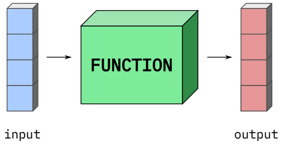




Mathematical Functions

- Built-in mathematical functions accept array inputs and return arrays of function evaluation, *e.g.*,

```
sqrt(A)
sin(A)
mod(A)
...
```



Array Constructors

Colon Operator

Suppose `a < b`.

- To create an arithmetic progression from `a` to `b` (increment by 1):

```
a:b
```

  The result is a row vector `[a, a+1, a+2, ..., a+m]`, where

$$\texttt{m} = \lfloor \texttt{b-a} \rfloor.$$

- To create an arithmetic progression from `a` to `b` with steps of size `d > 0`:

```
a:d:b
```

  The result is a row vector `[a, a+d, a+2*d, ..., a+m*d]`, where

$$\texttt{m} = \lfloor \texttt{(b-a)/d} \rfloor.$$

LINSPACE and LOGSPACE

- To create a row vector of `n` numbers evenly spaced between `a` and `b`:

```
linspace(a, b, n)
```

  The result is `[a, a+d, a+2*d, ..., b]`, where

```
d = (b-a)/(n-1).
```

- To create a row vector of `n` numbers that are logarithmically evenly spaced between 10^a and 10^b:

```
logspace(a, b, n)
```

  The result is $[10^a, 10^{a+d}, 10^{a+2d}, \ldots, 10^b]$, where

```
d = (b-a)/(n-1).
```

ZEROS, ONES, and EYE

- To create an $(m \times n)$ zero matrix:

```
zeros(m, n)
```

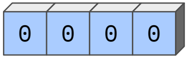
- To create an $(m \times n)$ matrix all whose entries are one:

```
ones(m, n)
```

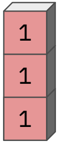
- To create the $(m \times m)$ identity matrix:

```
eye(m)
```

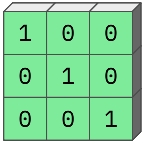
`zeros(1,4)`



`ones(3,1)`



`eye(3)`



Random Arrays

Each of the following generates an $(m \times n)$ array of random numbers:

- `rand(m,n)`: uniform random numbers in $(0,1)$
- `randi(k,m,n)`: uniform random integers in $[1,k]$
- `randn(m,n)`: Gaussian random numbers with mean 0 and standard deviation 1

Random Arrays (Application)

To generate an $(m \times n)$ array of

- uniform random numbers in (a, b):

```
a + (b - a)*rand(m, n)
```

- uniform random integers in $[k_1, k_2]$:

```
randi([k1, k2], m, n)
```

- Gaussian random numbers with mean μ and standard deviation σ:

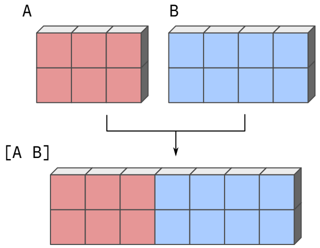
```
mu + sig*randn(m, n)
```

Building Arrays Out Of Arrays

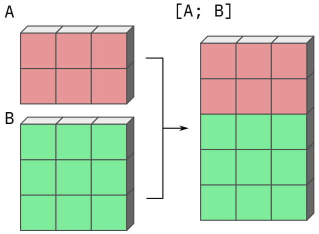
Concatenation

If two arrays `A` and `B` have *comparable* sizes, we can concatenate them.

- horizontally by `[A B]`

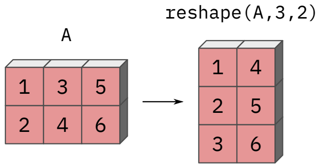


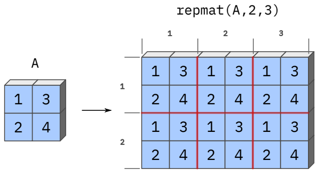
- vertically by `[A; B]`



RESHAPE and REPMAT

- `reshape(A, m, n)` reshapes the array `A` into an $m \times n$ matrix whose elements are taken *columnwise* from `A`.
- `repmat(A, m, n)` replicates the array `A`, m times vertically and n times horizontally.





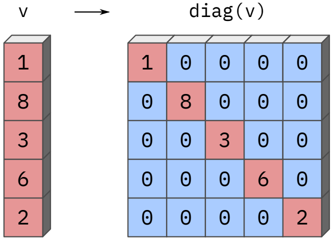
`FLIP`

- Type `help flip` on the Command Window and learn about `flip` function.
- Do the same with its two variants, `flipud` and `fliplr`

Creating Diagonal Matrices

- To create a diagonal matrix

$$\begin{bmatrix} v_1 & 0 & 0 & \cdots & 0 \\ 0 & v_2 & 0 & \cdots & 0 \\ 0 & 0 & v_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & v_n \end{bmatrix}:$$



```
diag(v)
```

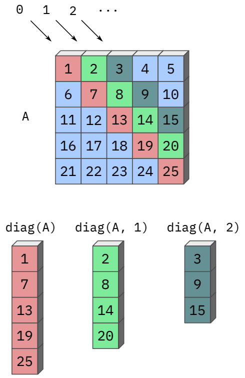
Note.

- `diag(v,k)` puts the elements of `v` on the `k`-th super-diagonal.
- `diag(v,-k)` puts the elements of `v` on the `k`-th sub-diagonal.

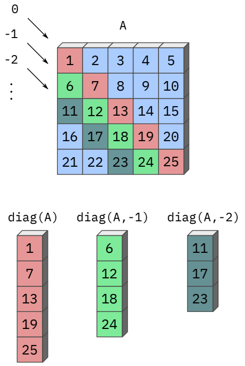
Extracting Diagonal Elements

Use `diag(A,k)` to extract the `k`-th diagonal of `A`. [1]

- `k > 0` for super-diagonals:



- `k < 0` for sub-diagonals:



[1] diag(A) short for diag(A,0) .

Slicing Arrays

Using Vectors as Indices

To get/set multiple elements of an array at once, use vector indices.

- To grab 3rd, 4th, and 5th elements of `x`:

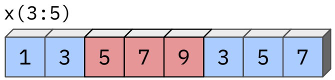
```
x(3:5)   % or x([3 4 5])
```

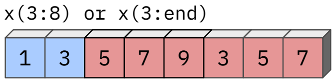
- To grab 3rd to 8th elements of `x`:

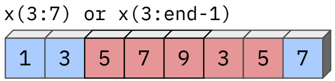
```
x(3:8)
x(3:end)
```

- To grab 3rd to 7th elements of `x`:

```
x(3:7)
x(3:end-1)
```







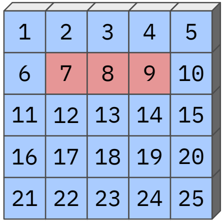
Using Vectors as Indices – Example

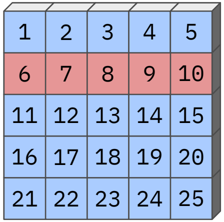
- To extract 2nd, 3rd, and 4th columns of the 2nd row of A:

```
A(2,2:4)   % or A(2,[2 3 4])
```

- To extract the entire 2nd row of A:

```
A(2,1:5)
A(2,1:end)
A(2,:)
```





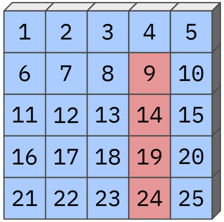
Using Vectors as Indices – Example

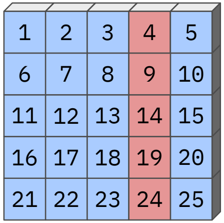
- To extract 2nd through 5th elements of the 4th column of `A`:

```
A([2 3 4 5],4)
A(2:5,4)
A(2:end,4)
```

- To extract the entire 4th column of `A`:

```
A(1:5,4)
A(1:end,4)
A(:,4)
```





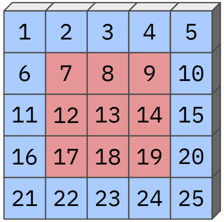
Using Vectors as Indices – Example

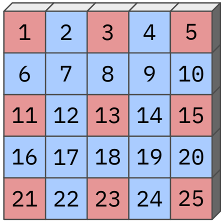
- To grab the *interior block* of `A`:

```
A(2:4,2:4)
A(2:end-1,2:end-1)
```

- To extract every other elements on every other rows as shown:

```
A(1:2:5,1:2:5)
A(1:2:end,1:2:end)
```





Lec 07: More on Arrays

Recap: Creating Arrays Examples

Arithmetic Progressions

Question

Create the following *periodic* arithmetic progressions using ONE MATLAB statement.

$$(1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0).$$

```
m = 5;
n = 15;
mod([1:n], m)
```

Exercise: Arithmetic Progressions

Question

Create each of the following *row* vectors using ONE MATLAB statement.

- $\mathbf{v} = (1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0)$
- $\mathbf{w} = (1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)$

Geometric and Other Progressions

Question

Create each of the following *column* vectors using ONE MATLAB statement.

- $\mathbf{v} = (1, 2, 4, 8, \ldots, 1024)^{\mathrm{T}}$
- $\mathbf{w} = (1, 4, 9, 16, \ldots, 100)^{\mathrm{T}}$

Using the colon operator:

```
v = ( 2.^[0:10] )'
w = ( [1:10].^2 )'
```

Using the `linspace` function:

```
v = ( 2.^linspace(0, 10, 11) )'
w = ( linspace(1, 10, 10).^2 )'
```

Function Evaluation

Recall that mathematical functions such as `sin, sind, log, exp` accept array inputs and return arrays of function evaluation.

Question

Create each of the the following *row* vectors using ONE MATLAB statement.

- $\mathbf{u} = (1!, 2!, 3!, \ldots, n!)$
- $\mathbf{v} = (\sin 0^\circ, \sin 30^\circ, \sin 60^\circ, \ldots, \sin 180^\circ)$
- $\mathbf{w} = (e^1, e^4, e^9, \ldots, e^{64})$

```
v = sind(0:30:180)
w = exp([1:8].^2)
```

Matrices with Patterns

Question

Generate each of the following matrices using ONE MATLAB statement.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1^2 & 1^3 & \cdots & 1^{10} \\ 2 & 2^2 & 2^3 & \cdots & 2^{10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 10 & 10^2 & 10^3 & \cdots & 10^{10} \end{bmatrix}.$$

```
A = reshape(1:16, 4, 4)'
B = ((1:10)').^(1:10)
```

Matrices with Patterns

Question

Suppose n is already stored in MATLAB. Generate each of the following matrices using ONE MATLAB statement. All the elements not shown are 0's.

$$C = \begin{bmatrix} 2 & & & & \\ & 4 & & & \\ & & 6 & & \\ & & & \ddots & \\ & & & & 2n \end{bmatrix}, D = \begin{bmatrix} \cos 1 & -3 & & & & \\ & \cos 2 & -3 & & & \\ & & \cos 3 & -3 & & \\ & & & \ddots & \ddots & \\ & & & & \cos(n-1) & -3 \\ & & & & & \cos n \end{bmatrix}.$$

```
C = diag(2:2:2*n)
D = diag(cos(1:n)) - 3*diag(ones(n-1,1), 1)
```

Data Manipulation Functions

Data Manipulation Functions

There are a number of MATLAB functions with *spreadsheet functionalities* that are suitable for data manipulation.

- `max` and `min`
- `sum` and `prod`
- `cumsum` and `cumprod` (cumulative sum and product)
- `diff`
- `mean`, `std`, and `var` (simple statistics)
- `sort`

Example 1: Finding the Maximum Value of a Vector

Question

Write a program to find the maximum value of a vector.

- **With loops:**

```
% input: x
% output: m      % DON'T USE max FOR THE VARIABLE NAME
m = x(1);        % CODE ABORTS IF THE VECTOR IS EMPTY
for r = 2:length(x)
    if m < x(r)
        m = x(r);
    end
end
```

- **Vectorized code:**

```
m = max(x)
```

Example 1 (cont')

Question

Now modify the previous program to find both the maximum value of a vector and the corresponding index.

- **With loops:**

```
% input: x
% output: m, index_m
m = x(1);
index_m = 1;
for r = 2:length(x)
    if m < x(r)
        m = x(r);
        index_m = r;
    end
end
```

- **Vectorized code:**

```
[m, index_m] = max(x)
```

Example 2: Summing Elements in a Vector

Question

Sum all elements in a vector.

- **With loops:**

```
% input: x
% output: s      % DON'T USE sum FOR THE VARIABLE NAME
s = 0;           % s begins before the first iteration
for el = 1:length(x)
    s = s + x(el);
end
```

- **Vectorized code:**

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

`FIND` Function

Basic Usage of `FIND`

Let `v` be an array of numbers (can be a vector or a matrix). Then

```
find(<condition>)
```

returns the (linear) indices of `v` satisfying `<condition>`.

- **Some examples of `<condition>`:**
  - `v > k` or `v >= k`
  - `v < k` or `v <= k`
  - `v == k` or `v ~= k`

- **To combine more than two conditions:**
  - `&` (and)
  - `|` (or)

Example 3: Comparing Elements in Vectors

Question

Compare two real vectors of the same length, say `x` and `y`, elementwise and determine how many elements of the former are larger than the latter.

- **With loops:**

```
% input: x, y
% output: nr_gt
nr_gt = 0;
for k = 1:length(x)
    if x(k) > y(k)
        nr_gt = nr_gt + 1;
    end
end
```

- **Vectorized code:**

```
nr_gt = length(find(x > y))
```

Timing in MATLAB

CPU Time

`cputime` reads total CPU time used by MATLAB from the time it was started.

- Single measurement:

```
ct = cputime;    % total cputime as of now
        <statements>
t = cputime - ct;
```

- Average CPU time:

```
ct = cputime;    % total cputime as of now
for i = 1:nr_reps
        <statements>
end
t_avg = (cputime - ct)/nr_reps;
```

Elapsed Time

At the execution of `tic`, MATLAB records the internal time (in seconds); at `toc` command, MATLAB displays the elapsed time.

- Single measurement:

```
tic     % starts a stopwatch timer
    <statements>
toc     % reads the elapsed time from tic
```

- Average elapse time:

```
tic    % starts a stopwatch timer
for i = 1:nr_reps
    <statements>
end
t_avg = toc/nr_reps;
```

What Do You Think It Does?

Below is a modified version of an example code from MATLAB's Help documentation for `tic`. What do you think it's doing?

```
REPS = 1000; minTime = Inf; nSum = 10;
tic;
for i  = 1:REPS
    tStart = tic;
    s = 0;
    for j = 1:nsum
        s = s + besselj(j,REPS);
    end
    tElpased = toc(tStart);
    minTime = min(tElapsed, minTime);
end
t_avg = toc/REPS;
```

Example 4: Timing Elementwise Operations

Question

Generate a $10^7 \times 1$ random vector and measure the internal time and CPU time when computing elementwise squares.

```
n = 1e7;
x = rand(n, 1);
t = cputime;
x1 = x.^2;
time1 = cputime - t;

tic
x2 = x.^2;
time2 = toc();
disp([time1, time2])
```

Exercises

Pythagorean Triples

Question

Given $n \in \mathbb{N}$, find all triples $(a, b, c)\mathbb{N}^3$, with $a, b \leqslant n$, satisfying

$$a^2 + b^2 = c^2.$$

Notation.

- $\mathbb{N}$: the set of all natural numbers, $1, 2, 3, \ldots$.
- $\mathbb{N}[1, n] = \{1, 2, \ldots, n\}$.
- $\mathbb{N}^3 = \{(a, b, c) \mid a, b, c \in \mathbb{N}\}$.

Pythagorean Triples – Solution Using Loops

```
% input: n
% output: M
iM = 0;
M = [];
for a = 1:n
    for b = 1:n
        c = sqrt(a^2 + b^2);
        if mod(c, 1) == 0
            iM = iM + 1;
            M(iM, :) = [a, b, c];
        end
    end
end
```

Pythagorean Triples – Solution Without Loops

```
% input: x
% output: M
A = repmat([1:n], n, 1);
B = repmat([1:n]', 1, n);
C = sqrt(A.^2 + B.^2);
M = [A(:), B(:), C(:)];
lM = ( mod(M(:, 3), 1) ~= 0 );
M(lM, :) = [];
```

Birthday Problem

Question

In a group of n randomly chosen people, what is the probability that everyone has a different birthday?

1. Find this probability by hand.
2. Let $n = 30$. Write a script that generates a group of n people randomly and determines if there are any matches.
3. Modify the script above to run a number of simulations and numerically calculate the sought-after probability. Try 1000, 10000, and 100000 simulations. Compare the result with the analytical calculation done in 1.

Birthday Problem (Hints)

- For simplicity, ignore leap years.
- Create a random (column) vector whose elements represent birthdays of individuals (denoted by integers between 1 and 365).
- Line up the birthdays in order and take the difference of successive pairs. What does the resulting vector tell you?
- For 3, to run simulation multiple times, consider creating a random matrix whose rows represent birthdays of individuals and the columns correspond to different simulations.

Lec 08: Graphics in MATLAB

Anonymous Functions

Anonymous Functions

Mathematical functions such as

$$\begin{aligned} f_1(x) &= \cos x \sin\big(\cos(\tan x)\big), \\ f_2(\theta) &= \left(\cos 3\theta + 2\cos 2\theta\right)^2, \\ f_3(x,y) &= \frac{\sin(x+y)}{1+x^2+y^2}, \end{aligned}$$

can be defined in MATLAB using *anonymous functions*:

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
f2 = @(th) ( cos(3*th) + 2*cos(2*th) ).^2;
f3 = @(x, y) sin(x + y)./(1 + x.^2 + y.^2);
```

Anonymous Functions – Syntax

Take a closer look at one of them.

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
```

- `f1` : the function name or the *function handle*
- `@` : marks the beginning of an anonymous function
- `(x)` : denotes the function (input) argument
- `cos(x).*sin(cos(tan(x)))` : MATLAB expression defining $f_1(x)$

Examples

Expressions in function definitions can get very complicated. For example,

```
h1 = @(x) [2*x, sin(x)];
h2 = @(x) [2*x, sin(x); 5*x, cos(x); 10*x, tan(x)];
r = @(a,b,m,n) a + (b-a)*rand(m,n);
```

Exercise: Different Ways of Defining a Function

The function

$$f_4(\theta; c_1, c_2, k_1, k_2) = (c_1 \cos k_1\theta + c_2 \cos k_2\theta)^2$$

can be defined in two different ways:

```
f4s = @(th,c1,c2,k1,k2) c1*cos(k1*th) + c2*cos(k2*th)
f4v = @(th,c,k) c(1)*cos(k(1)*th) + c(2)*cos(k(2)*th)
```

Question

Use `f4s` and `f4v` to define yet another anonymous functions for

- $g(\theta) = 3\cos(2\theta) - 2\cos(3\theta)$
- $h(\theta) = 3\cos(\theta/7) + \cos(\theta)$

Exercise: Understanding Anonymous Functions

Type in the following statements in MATLAB:

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
f2 = @(th) ( cos(3*th) + 2*cos(2*th) ).^2;
x1 = 5; y1 = f1(x1)
x2 = [5:-2:1]; y2 = f1(x2)
TH = diag(0:pi/2:2*pi); R = f2(TH)
```

Question

1. What are the types of the input and output variables?
   - `x1` and `y1`
   - `x2` and `y2`
   - `TH` and `R`
2. Which of the three outputs will be affected if elementwise operations were not used in the definition of `f1` and `f2`?

2-D Graphics

The `PLOT` Function

To draw a curve in MATLAB:

- Construct a pair of n-vectors `x` and `y` corresponding to the set of data points $\{(x_i, y_i) \mid i = 1, 2, \dots, n\}$ which are to appear on the curve in that order.
- Then type `plot(x, y)`.

For example:

```
x = linspace(0, 2*pi, 101);
y = sin(x);
plot(x, y)                              % or simply plot(x, sin(x))
```

or

```
f = @(x) 1 + sin(2*x) + cos(4*x);       % anonymous function
x = linspace(0, 2*pi, 101);
plot(x, f(x))
```

Example: Wiggly Curve

First, run the following script.

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
x = 2*pi*[0:.0001:1];   % or  x = linspace(0, 2*pi, 10001);
plot(x, f1(x))
shg
```

Play Around!

Observe what happens after applying the following modifications one by one.

- Change line 3 into `plot(x, f1(x), 'r')`.

Example: Wiggly Curve

First, run the following script.

```
1  f1 = @(x) cos(x).*sin(cos(tan(x)));
2  x = 2*pi*[0:.0001:1];    % or  x = linspace(0, 2*pi, 10001);
3  plot(x, f1(x))
4  shg
```

Play Around!

Observe what happens after applying the following modifications one by one.

- Change line 3 into `plot(x, f1(x), ’r’)`.
- Change line 3 into `plot(x, f1(x), ’r--’)`.

Example: Wiggly Curve

First, run the following script.

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
x = 2*pi*[0:.0001:1];    % or  x = linspace(0, 2*pi, 10001);
plot(x, f1(x))
shg
```

Play Around!

Observe what happens after applying the following modifications one by one.

- Change line 3 into `plot(x, f1(x), 'r')`.
- Change line 3 into `plot(x, f1(x), 'r--')`.
- After line 3, add `axis equal, axis tight`.

Example: Wiggly Curve

First, run the following script.

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
x = 2*pi*[0:.0001:1];    % or  x = linspace(0, 2*pi, 10001);
plot(x, f1(x))
shg
```

Play Around!

Observe what happens after applying the following modifications one by one.

- Change line 3 into `plot(x, f1(x), 'r')`.
- Change line 3 into `plot(x, f1(x), 'r--')`.
- After line 3, add `axis equal, axis tight`.
- Then add `text(4.6, -0.3, 'very wiggly')`.

Example: Wiggly Curve

First, run the following script.

```
1  f1 = @(x) cos(x).*sin(cos(tan(x)));
2  x = 2*pi*[0:.0001:1];   % or  x = linspace(0, 2*pi, 10001);
3  plot(x, f1(x))
4  shg
```

Play Around!

Observe what happens after applying the following modifications one by one.

- Change line 3 into `plot(x, f1(x), ’r’)`.
- Change line 3 into `plot(x, f1(x), ’r--’)`.
- After line 3, add `axis equal, axis tight`.
- Then add `text(4.6, -0.3, ’very wiggly’)`.
- Then add
  `xlabel(’x axis’), ylabel(’y axis’), title(’A wiggly curve’)`.

Note: Line Properties

- To specify line properties such as colors, markers, and styles:

```
plot(x, y, '--')                          % dashed line
plot(x, y, 'g:')                          % dotted line in green
plot(x, y, '.', 'MarkerSize', 3)          % adjust marker size
plot(x, y, 'b-', 'LineWidth', 5)          % adjust line width
```

Colors

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Markers

.	point
o	circle
x	x-mark
+	plus
*	star
s	square
d	diamond

Line Styles

-	solid
:	dotted
-.	dashdot
--	dashed

Note: Labels and Saving

- To label the axes and the entire plot, add the following after `plot` statement:

```
xlabel('x axis')
ylabel('y axis')
title('my awesome graph')
```

- Save figures using `print` function. Multiple formats are supported.

```
print -dpdf 'wiggly'
% or print('-dpdf', 'wiggly')                                [pdf]

print -djpeg 'wiggly'
% or print('-djpeg', 'wiggly')                              [jpeg]

print -deps 'wiggly'
% or print('-deps', 'wiggly')                                [eps]
```

Note: Drawing Multiple Figures

- To plot multiple curves:

```
plot(x1, y1, x2, y2, x3, y3, ...)
```

- To create a legend, add

```
legend('first graph', 'second graph', 'third graph', ...)
```

Note: Miscellaneous Commands

- `shg`: (show graph) to bring Figure Window to the front
- `figure`: to open a new blank figure window
- `clf`: (clear figure) to clear previously drawn figures
- `axis equal`: to put axes in equal scaling
- `axis tight`: to remove margins around graphs
- `axis image`: same as `axis equal` and `axis tight`
- `grid on`: to put light gray grid lines

Exercise

Question

Do the following:

- Define $f(x) = x^3 + x$ as an anonymous function.
- Find f' and f'' and define them as anonymous functions.
- Plot all three functions in one figure in the interval $[-1, 1]$.
- Include labels and title in your plot.
- Add legend to the graph.
- Save the graph as a pdf file.

Multiple Figures – Stacking

To draw multiple curves in one plot window as in Figure 1:

- One liner:

```
plot(x1, y1, x2, y2, x3, y3)
```

- Or, add curves one at a time using `hold` command.

```
plot(x1, y1)
hold on
plot(x2, y2)
plot(x3, y3)
```

  - `hold on`: holds the current plot for further overlaying
  - `hold off`: turns the *hold* off

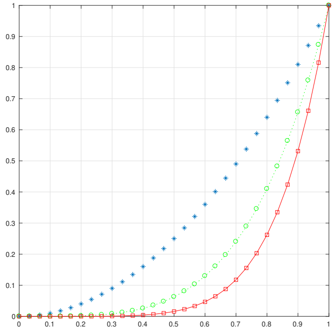


Figure 1: Multiple curves in one plot window

Multiple Figures – Subplots

To plot multiple curves separately and arrange them as in Figure 2:

```
subplot(1,3,1)
plot(x1, y1)
subplot(1,3,2)
plot(x2, y2)
subplot(1,3,3)
plot(x3, y3)
```

`subplot(m,n,p)`:

- `m`, `n`: determine grid dimensions
- `p`: determines grid is to be used

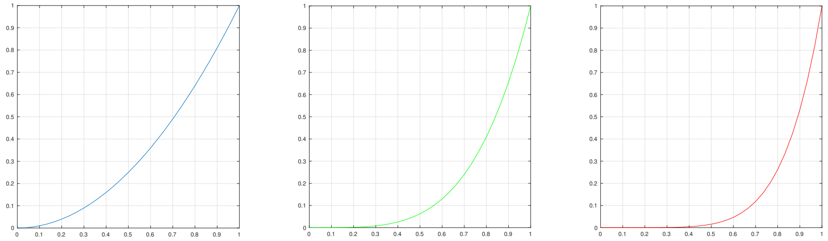


Figure 2: Multiple plots in 1×3 grids

Exercise: Multiple Figures

Do It Yourself

Generate Figures 1 and 2.

- Common: Generating sample points

```
x = linspace(0, 1, 101);
y1 = x.^2; y2 = x.^4; y3 = x.^6;
```

- Figure 1:

```
hold off
plot(x, y1, '*')
hold on
plot(x, y2, 'g:o')
plot(x, y3, 'r-s')
```

- Figure 2:

```
subplot(1, 3, 1)
plot(x, y1)
subplot(1, 3, 2)
plot(x, y2, 'g')
subplot(1, 3, 3)
plot(x, y3, 'r')
```

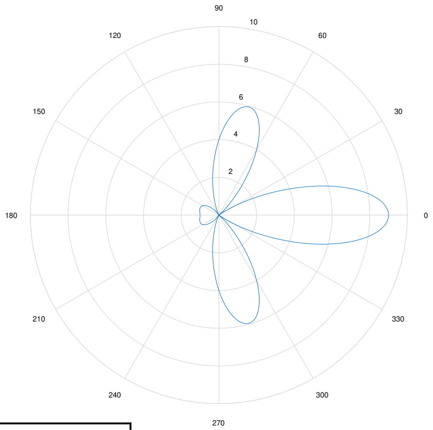
The `POLAR` Function

To draw the polar curve $r = f(\theta)$, for $\theta \in [a, b]$:

- Grab n sample points $\{(\theta_i, r_i) \mid r_i = f(\theta_i),\ 1 \leqslant i \leqslant n\}$ on the curve and form vectors `th` and `r`.
- Then type `polar(th, r)`.
- For example, to plot

  $$r = f_2(\theta) = (\cos 3\theta + 2\cos 2\theta)^2,$$

  for $\theta \in [0, 2\pi]$:



```
th = linspace(0, 2*pi, 361);
f2 = @(th) (cos(3*th) + 2*cos(2*th)).^2;
polar(th, f2(th));
```

Exercise: Drawing Polar Curves

Question

1. Draw the graph of two-petal leaf given by

$$r = f(\theta) = 1 + \sin(2\theta)\,, \quad \theta \in [0, 2\pi]\,.$$

2. Draw the graphs of

$$r = f(\theta - \pi/4)\,, \quad r = f(\theta - \pi/2)\,, \quad r = f(\theta - 3\pi/4)$$

   on the same plotting window.

3. Does your figure make sense?

3-D Graphics

Curves and the `PLOT3` Function

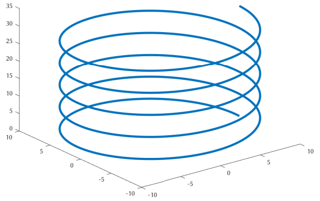
Curves in $\mathbb{R}^3$ are plotted in an analogous fashion.

- Grab n sample points $\{(x_i, y_i, z_i) \mid i = 1, 2, \ldots, n\}$ on the curve and form vectors `x`, `y`, and `z`.
- Then type `plot3(x, y, z)`.
- For example, to plot the helix given by the parametrized equation

$$\mathbf{r}(t) = \langle 10\cos(t), 10\sin(t), t\rangle,$$

for $t \in [0, 10\pi]$:

```
t = linspace(0, 10*pi, 1000);
plot3(10*cos(t), 10*sin(t), t);
```



Exercise: Corkscrew

Question

Modify the code to generate a corkscrew by putting the helix outside of an upside down cone.

Hint: Use $\mathbf{r}(t) = \langle t\cos(t), t\sin(t), t\rangle$.

Surfaces and the `SURF` Function

To plot the surface of $z = f(x, y)$ on $R = [a, b] \times [c, d]$:

- Collect samples points on the intervals $[a, b]$ and $[c, d]$ and form vectors `x` and `y`.
- Based on `x` and `y`, generate grid points $\{(x_i, y_j) \mid i = 1, 2, \ldots, m, j = 1, 2, \ldots, n\}$ on the domain R and separate coordinates into matrices `X` and `Y` using `meshgrid`.
- Type `surf(X, Y, f(X,Y))`

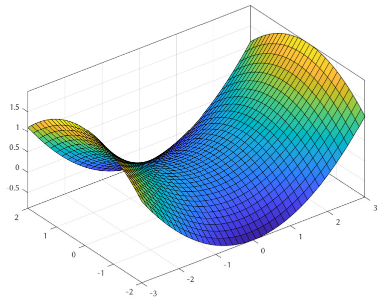
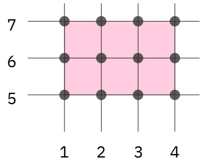
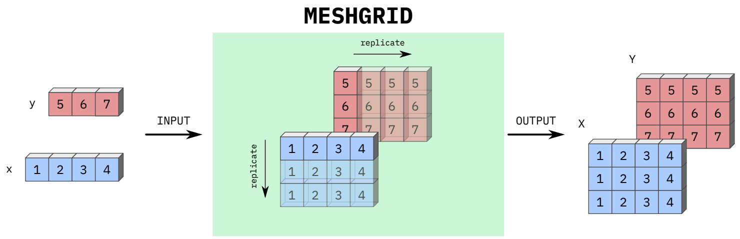


Figure 3: Graph of $z = \frac{2}{9}(x^2 - y^2)$ on $[-3, 3] \times [-2, 2]$

Note: How MESHGRID Work

```
>> x = [1 2 3 4]; y = [5 6 7];
>> [X, Y] = meshgrid(x,y)
X =
     1     2     3     4
     1     2     3     4
     1     2     3     4
Y =
     5     5     5     5
     6     6     6     6
     7     7     7     7
```





Example: Saddle

Question

Plot the saddle parametrized by

$$\frac{z}{c} = \frac{x^2}{a^2} - \frac{y^2}{b^2}$$

for your choice of a, b, and c.

```
x = linspace(-3, 3, 13);
y = linspace(-2, 2, 9);
[X, Y] = meshgrid(x, y);
a = 1.5; b = 1.5; c = .5;
g2 = @(x,y) c*( x.^2 /a^2 - y.^2 /b^2);
surf(X, Y, g2(X,Y))
axis equal, box on
```

Figure 3 was generated using this code.

Example: Oblate Spheroid

The figure for Problem 5 of Homework 1 was generated by the following code.[2]

```
a = 1; b = 1.35; c = 1;
nr_th = 41; nr_ph = 31;
x = @(th, ph) a*cos(th).*sin(ph);
y = @(th, ph) b*sin(th).*sin(ph);
z = @(th, ph) c*cos(ph);
th = linspace(0, 2*pi, nr_th);
ph = linspace(0, pi, nr_ph);
[T, P] = meshgrid(th, ph);
surf(x(T,P), y(T,P), z(T,P))
colormap(winter)
axis equal, axis off, box off
```

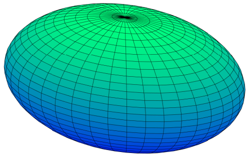


Figure 4: Oblate spheroid.

Lec 09: Function M-File

Function M-File

Basics

- A *function* is a piece of code which
  - performs a specific task;
  - has specific input and output arguments;
  - is encapsulated to be independent of the rest of the program.
- In MATLAB, functions are defined in .m files just as scripts.
- The name of the file and that of the function must coincide.

Why Functions?

Two Important Principles

- A piece of code should only occur once in a program. If it tries to appear more times, put it into its own function.
- Each function should do precisely **one** task well. If it has to carry out a number of tasks, separate them into their own functions.

Benefits of writing functions:

- elevated reasoning by hiding details
- facilitates top-down design
- ease of software management

Writing a Function

A function m-file must be written in a specific manner.

- When there is no input/output argument

```
function myfun()
  ....
end     % <-- optional
```

- Where there are multiple input and output arguments

```
function [out1, out2, out3] = myfun(in1, in2, in3, in4)
  ....
end     % <-- optional
```

Calling a Function

- If the function m-file `myfun.m` is saved in your current working directory[3], you can use it as you would use any other built-in functions:

```
% when no input/output argument is required
myfun
```

or

```
% multiple inputs/outputs
[out1, out2, out3] = myfun(in1, in2, in3, in4)
```

- When not all output arguments are needed:

```
out1 = myfun(in1, in2, in3, in4)         % only 1st output
[~, ~, out3] = myfun(in1, in2, in3, in4) % only 3rd output
```

Note that tilde (~) is used as a placeholder.

[3]The `path` function gives more flexibility in this regard.

Practical Matters: Specification

- The comments written below `function` header statement

```
function ... = myfun( ... )
% MYFUN: this awesome function calculates ...
  ....
end
```

  can be easily accessed from the command line using `help myfun`.

- Use this feature to write the function specification such as its purpose, usage, and syntax.
- Write a clear, complete, yet concise specification.

Properties of Function M-Files

- The variable names in a function are completely isolated from the calling code. Variables, other than outputs, used inside a function are unknown outside of the function; they are *local variables*.
- The input arguments can be modified within the body of the function; no change is made on any variables in the calling code. (“ *pass by value* ” as opposed to “ *pass by reference* ”)
- Unlike script m-files, you **CANNOT** execute a function which has input arguments using the RUN icon.

Example: Understanding Local Variables

- The function on the right finds the maximum value of a vector and an index of the maximal element.
- Run the following on the Command Window. Pay attention to `m`.

```
>> m = 33;
>> x = [1 9 2 8 3 7];
>> [M, iM] = mymax(x)
>> disp(m)
```

```
function [m, el] = mymax(x)
    if isempty(x)
        el = [];
        m = [];
        return
    end
    el = 1;
    m = x(el);
    for i = 2:length(x)
        if m < x(i)
            el = i;
            m = x(el);
        end
    end
end
```

Example: Understanding Pass-By-Value

Consider the function

$$f(x) = \sin|x| \frac{e^{-|x|}}{1+|x|^2} + \frac{\ln(1+|x|)}{1+2|x|}$$

written as a MATLAB function:

```
function y = funky(x)
    x = abs(x);  % x is redefined to be abs(x)
    y = sin(x).*exp(-x)./(1 + x.^2)...
        + log(1 + x)./(1 + 2*x);
end
```

Confirm that the function does not affect `x` in the calling routine:

```
>> x = [-3:3]';
>> y = funky(x);
>> disp([x, y])
```

Quiz

Question

Script.

```
x = 2;
x = myfun(x);
y = 2*x
```

Function.

```
function y = myfun(x)
    x = 2*x;
    y = 2*x;
end
```

What is the output when the script on the left is run?

1. `y = 2`
2. `y = 4`
3. `y = 8`
4. `y = 16`

Example: Spiral Triangle

Description of Problem

Problem

Given m and θ (in degrees), draw m equilateral triangles so that for any two successive triangles,

- the vertices of the smaller triangle are lying on the sides of the larger;
- the angle between the two triangles is θ (degrees)

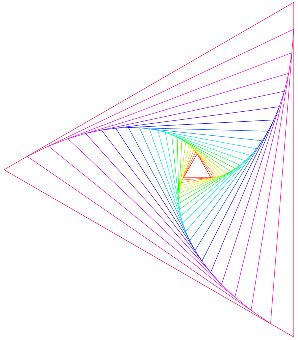


Figure 5: A spiral triangle with $m = 21$ and $\theta = 4.5^\circ$.

Generation and Transformation of Polygons

Let (x_j, y_j), $j = 1, 2, \ldots, n$ be the coordinates of vertices of an n-gon.

- To plot the polygon:

```
x = [x1 x2 ... xn x1]; % note: x1 and y1 are repeated at
y = [y1 y2 ... yn y1]; % end to enclose the polygon.
plot(x, y)
```

- To plot the polygon obtained by scaling the original by a factor of `s`:

```
plot(s*x, s*y)
```

- To plot the polygon obtained by rotating the original by an angle `theta` (in degrees) about the origin:

```
V = [x; y];                          % all vertices
R = [cosd(theta) -sind(theta);       % 2-D rotation matrix
     sind(theta)  cosd(theta)];
Vnew = R*V;                          % rotated vertices
plot(Vnew(1,:), Vnew(2,:))
```

Special Case: Inscribed Regular Polygons

- The vertices of the *regular* n-gon inscribed in the unit circle can be found easily using trigonometry, *e.g.*,

$$(x_j, y_j) = \left(\cos\frac{2\pi(j-1)}{n}, \sin\frac{2\pi(j-1)}{n}\right), \quad j = 1, \ldots, n.$$

- Thus we can plot it by

```
theta = linspace(0, 360, n+1);
V = [cosd(theta);      % 2-by-(n+1) matrix whose cols are
     sind(theta)];     % coordinates of the vertices
plot(V(1,:), V(2,:)), axis equal
```

- To stretch and rotate the polygon:

```
Vnew = s*R*V;
plot(Vnew(1,:), Vnew(2,:)), axis equal
```

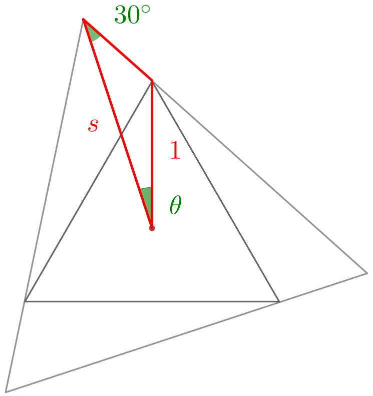
Scale to Spiral

To create the desired spiraling effect, the scaling factor must be calculated carefully.

- Useful:

$$\boxed{\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c}}$$

- Compute the scaling factor s:



Put All Together

```
m = 21; d_angle = 4.5;
th = linspace(0, 360, 4) + 90;
V = [cosd(th);
     sind(th)];
C = colormap(hsv(m));
s = sind(150 - abs(d_angle))/sind(30);
R = [cosd(d_angle) -sind(d_angle);
     sind(d_angle) cosd(d_angle)];
hold off
for i = 1:m
    if i > 1
        V = s*R*V;
    end
    plot(V(1,:), V(2,:), 'Color', C(i,:))
    hold on
end
set(gcf, 'Color', 'w')
axis equal, axis off
```

Exercise: Script to Function

Question

Modify the script so that it can create a spiral n-gon consisting of m regular n-gons successively rotated by θ degrees. Then turn the script into a function m-file `spiralgon.m`.

```
function V = spiralgon(n, m, d_angle)
% SPRIALGON plots spiraling regular n-gons
% input:    n = the number of vertices
%           m = the number of regular n-gons
%           d_angle = the degree angle between successive n-gons
%            (can be positive or negative)
% output:   V = the vertices of the outermost n-gon

% Fill in the rest.
....
```

Appendix: Supplementary Notes

Local Functions

There are “general purpose” functions which may be used by a number of other functions. In this case, we put it into a separate file. However, many functions are quite specific and will only be used in one program. In such a case, we keep them in whatever file calls them.

- The first function is called the *primary* function and any function which follows the primary function is called a *local function* or a *subfunction*.

```
function <primary function>
  ....
end
function <local function #1>
  ....
end
<any number of following local functions>
```

- The primary function interact with local functions through the input and output arguments.
- Only the primary function is accessible from outside the file.

Defining/Evaluating Mathematical Functions in MATLAB

- using an anonymous function
- using a local function
- using a primary funciton
- using a nested function
- putting it *in situ*, i.e., write it out completely in place – not recommended

See `time_anon.m`.

Passing Function to Another Function

- A function can be used as an input argument to another function.
- The function must be stored as a function-handle variable in order to be passed as an argument.
- This is done by prepending " `@` " to the function names. For instance, try

```
>> class(mymax)    % WRONG
>> class(@mymax)   % CORRECT
```

When a Function Is Called

Below is what happens internally when a function is called:

- MATLAB creates a new, local workspace for the function.
- MATLAB evaluates each of the input arguments in the calling statements (if there are any). These values are then assigned to the input arguments of the function. These are now local variables in the new workspace.
- MATLAB executes the body of the function. If there are output arguments, they are assigned values before the execution is complete.
- MATLAB deletes the local workspace, saving only the values in the output arguments. These values are then assigned to the output arguments in the calling statement.
- The code continues following the calling statement.

Lec 10: Review of Topic 1

Tips

Loops

- To kick start a while-loop even when part of loop header is not valid:

```
n = 0;
while n == 0 || err > tol
    n = n + 1;
    q_approx = ...;
    err = abs(q - q_approx);
end
```

- "short-circuiting" `&&` and `||`

Forming Sums

To calculate $\sum_{j=1}^{n} a_j b_j$:

- using a loop
- using `sum`
- **inner product**

Sequence of Partial Sums

To study the convergence of an infinite series $\sum_{j=0}^{\infty} a_j$, form the sequence of partial sums $\{s_n\}$ where

$$s_n = \sum_{j=0}^{n} a_j = a_0 + a_1 + \cdots + a_n.$$

- using a loop
- using `cumsum`

Simple Examples

Biased Coin

Question

Simulate the tossing of a biased coin with

$$P(\mathsf{T}) = p, \quad P(\mathsf{H}) = 1 - p.$$

Biased Coin – Notes

Ideas.

- random number generators
- traditional tools: loops and conditional statements
- the *powerful* `find` function
- one-liner using `ceil` or `floor`

Explore.

- How would you handle similar situations with multiple states with non-uniform probability profile, *e.g.*, a biased dice?

Dice Rolls

Question

Write a script simulating $n = 10,000$ throws of two 6-sided fair dice. What is the probability of obtaining two same numbers? Provide both analytical and numerical answers.

Finding Factors

Question

Given a positive integer n, finds all factors. Do it using a single MATLAB statement.

Finding Factors – Notes

Ideas.

- the `mod` function: detecting a factor
- the `find` function: do it in one scoop

Explore.

- The built-in function `factor` finds all prime factors. Use it to write a prime factorization of an integer.

Data Manipulation

Download `grades.dat` into your current directory and load it using

```
>> grades = load('grades.dat');
```

To read about how the data are organized, use `type grades.dat`.

Question

1. Determine the number of students.
2. Compute the total grade according to the weights specified in the header. Do this without using a loop.
3. The letter grades are determined by

   - A: $[90, 100]$
   - B: $[80, 90)$
   - C: $[70, 80)$
   - D: $[60, 70)$
   - E: $[0, 60)$

   Find the number of students earning each of the letter grades.

Spiral Triangle: Tying Up Loose Ends

Recall: Entire Code

```matlab
m = 21; d_angle = 4.5;
th = linspace(0, 360, 4) + 90;
V = [cosd(th);
     sind(th)];
C = colormap(hsv(m));
s = sind(150 - abs(d_angle))/sind(30);
R = [cosd(d_angle) -sind(d_angle);
     sind(d_angle) cosd(d_angle)];
hold off
for i = 1:m
    if i > 1
        V = s*R*V;
    end
    plot(V(1,:), V(2,:), 'Color', C(i,:))
    hold on
end
set(gcf, 'Color', 'w')
axis equal, axis off
```

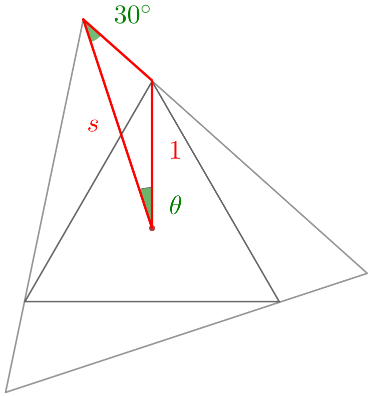
Understanding Line 6

To create the desired spiraling effect, the scaling factor must be calculated carefully.

- Useful:

$$\boxed{\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c}}$$

- Compute the scaling factor s:



Understanding Line 12

```
th = linspace(0, 360, 4) + 90;
V = [cosd(th);
     sind(th)];
s = sind(150 - abs(d_angle))/sind(30);
R = [cosd(d_angle) -sind(d_angle);
     sind(d_angle) cosd(d_angle)];
V = s*R*V;     % <----
```

Understanding Line 5 (More on Coloring)

- Using RGB colors in plots
- `colormap`

Lec 11: Floating-Point Numbers

Floating-Point Numbers

Absolute and Relative Errors

In numerical analysis, we use an **algorithm** to *approximate* some quantity of interest.

Example: Stirling's Formula

Stirling's formula provides a "good" approximation to $n!$ for large n:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n . \qquad (\star)$$

Try in MATLAB:

```
n = ...;
err_abs = sqrt(2*pi*n)*(n/exp(1))^n - factorial(n);
err_rel = err_abs/factorial(n);
disp(err_abs)
disp(err_rel)
```

Limitations of Digital Representations

A digital computer uses a finite number of bits to represent a real number and so it cannot represent all real numbers.

- The represented numbers cannot be arbitrarily large or small;
- There must be gaps between them.

So for all operations involving real numbers, it uses a subset of $\mathbb{R}$ called the **floating-point numbers**, $\mathbb{F}$.

Floating-Point Numbers

A *floating-point number* is written in the form $\pm(1+F)2^E$ where

- E, the *exponent*, is an integer;
- F, the *mantissa*, is a number $F = \sum_{i=1}^{d} b_i 2^{-i}$, with $b_i = 0$ or $b_i = 1$.

Note that F can be rewritten as

$$F = 2^{-d} \underbrace{\sum_{k=0}^{d-1} b_{d-k} 2^k}_{=:M},$$

where M is an integer in $\mathbb{N}[0, 2^d - 1]$.

Consequently, there are 2^d evenly-spaced numbers between 2^E and 2^{E+1} in the floating-point number system.

Floating-Point Numbers – IEEE 754 Standard

- MATLAB, by default, uses *double precision* floating-point numbers, stored in memory in 64 bits (or 8 bytes):

$$\pm \underbrace{1.\mathtt{xxxxxxxx}\cdots\mathtt{xxxxxxxx}_{(2)}}_{\text{mantissa (base 2): 52+1 bits}} \times 2^{\underbrace{\mathtt{xxxx}\cdots\mathtt{xxxx}_{(2)}-1023}_{\text{exponent: 11 bits}}}.$$

- Predefined variables:
  - `eps` = the distance from 1.0 to the next largest double-precision number:

$$\texttt{eps} = 2^{-52} \approx 2.2204 \times 10^{-16}.$$

  - `realmin` = the smallest positive floating-point number that is stroed to full accuracy; the actual smallest is `realmin/2^52`.
  - `realmax` = the largest positive floating-point number

Machine Epsilon and Relative Errors

The IEEE standard guarantees that the *relative representation error* and the *relative computational error* have sizes smaller than $\boxed{\text{eps}}$, the *machine epsilon*:

- **Representation**: The floating-point representation, $\hat{x} \in \mathbb{F}$, of $x \in \mathbb{R}$ satisfies

$$\hat{x} = x(1 + \epsilon_1), \qquad \text{for some } |\epsilon_1| \leqslant \frac{1}{2}\boxed{\text{eps}}.$$

- **Arithmetic**: The floating-point representation, $\hat{x} \oplus \hat{y}$, of the result of $\hat{x} + \hat{y}$ with $\hat{x}, \hat{y} \in \mathbb{F}$ satisfies

$$\hat{x} \oplus \hat{y} = (\hat{x} + \hat{y})(1 + \epsilon_2), \qquad \text{for some } |\epsilon_2| \leqslant \frac{1}{2}\boxed{\text{eps}}.$$

  Similarly with $\ominus, \otimes, \oplus$ corresponding to $-, \times, \div$, respectively.

Round-Off Errors

Computers CANNOT usually

- represent a number correctly;
- add, subtract, multiply, or divide correctly!!

Run the following and examine the answers:

```
format long
1.234567890123456789O
1234567890123456789O
(1 + eps) - 1
(1 + .5*eps) - 1
(1 + .51*eps) - 1
n = input(' n = '); ( n^(1/3) )^3 - n
```

Catastrophic Cancellation

In finite precision storage, two numbers that are close to each other are indistinguishable. So subtraction of two nearly equal numbers on a computer can result in loss of many significant digits.

Example 1: Cancellation for Large Values of x

Question

Compute $f(x) = e^x(\cosh x - \sinh x)$ at $x = 1, 10, 100$, and 1000.

Numerically:

```
format long
x = input(' x = ');
y = exp(x) * ( cosh(x) - sinh(x) );
disp([x, y])
```

Example 2: Cancellation for Small Values of x

Question

Compute $f(x) = \dfrac{\sqrt{1+x} - 1}{x}$ at $x = 10^{-12}$.

Numerically:

```
x = 1e-12;
fx = (sqrt(1+x) - 1)/x;
disp( fx )
```

To Avoid Such Cancellations …

- Unfortunately, there is no universal way to avoid loss of precision.
- One way to avoid catastrophic cancellation is to remove the source of cancellation by simplifying the given expression before computing numerically.
- For Example 1, rewrite the given expression recalling that

$$\cosh x = (e^x + e^{-x})/2 \qquad \sinh x = (e^x - e^{-x})/2.$$

- For Example 2, try again after rewriting $f(x)$ as

$$f(x) = \frac{\sqrt{1+x}-1}{x} \cdot \frac{\sqrt{1+x}+1}{\sqrt{1+x}+1} = \frac{1}{\sqrt{1+x}+1}\,.$$

- Do you now have an improved accuracy?

Lec 12: Conditioning and Stability

Problems and Conditioning

Problems and Conditioning

- A mathematical *problem* can be viewed as a function $f : X \to Y$ from a data/input space X to a solution/output space Y.
- We are interested in changes in $f(x)$ caused by small perturbations of x.
- A *well-conditioned* problem is one with the property that all small perturbations of x lead to only small changes in $f(x)$

Condition Number

Let $f : \mathbb{R} \to \mathbb{R}$ and $\hat{x} = x(1+\epsilon)$ be the representation of $x \in \mathbb{R}$.

- The ratio of the relative error in f due to the change in x to the relative error in x simplifies to
- In the limit of small error (ideal computer), we obtain

$$\kappa_f(x) := \lim_{\epsilon \to 0} \frac{\left| f(x) - f(x(1+\epsilon)) \right|}{\left| \epsilon f(x) \right|} = \left| \lim_{\epsilon \to 0} \frac{f(x+\epsilon x) - f(x)}{\epsilon x} \cdot \frac{x}{f(x)} \right| = \left| \frac{x f'(x)}{f(x)} \right|, \quad (\star)$$

which is called the **(relative) condition number**.

Example: Conditioning of Subtraction

Consider $f(x) = x - c$ where c is some constant. Using the formula ($\star$), we find that the associated condition number is

- It is large when $x \approx c$.

Example: Conditioning of Multiplication

The condition number of $f(x) = cx$ is

- No magnification of error.

Example: Conditioning of Function Evaluation

The condition number of $f(x) = \cos(x)$ is

$$\kappa(x) = \left|\frac{x f'(x)}{f(x)}\right| = \left|\frac{-x\sin x}{\cos x}\right| = \left|x \tan x\right|.$$

- The condition number is large when $x = (n + 1/2)\pi$, where $n \in \mathbb{Z}$.

Example: Conditioning of Root-Finding

Let $r = f(a; b, c)$ be a root of $ax^2 + bx + c = 0$. Instead of direct differentiation, use implicit differentiation

$$r^2 + 2ar\frac{dr}{da} + b\frac{dr}{da} = 0.$$

Solve for the derivative,

$$f'(a) = \frac{dr}{da} = -\frac{r^2}{2ar + b} = -\frac{r^2}{\pm\sqrt{b^2 - 4ac}},$$

then compute the condition number using the formula $(\star)$ to get

$$\kappa(a) = \left|\frac{af'(a)}{f(a)}\right| = \left|\frac{ar^2}{\pm r\sqrt{b^2 - 4ac}}\right| = \left|\frac{ar}{\sqrt{b^2 - 4ac}}\right|.$$

- Conditioning is poor for small discriminant, *i.e.*, near repeated roots.

Stability of Algorithms

Algorithms

- Recall that we defined a *problem* as a function $f : X \to Y$.
- An *algorithm* can be viewed as another map $\tilde{f} : X \to Y$ between the same two spaces, which involves errors arising in
    - representing the actual input x as $\hat{x}$;
    - implementing the function f numerically on a computer.

Analysis – General Framework

The relative error of our interest is

$$\left|\frac{\tilde f(\hat x) - f(x)}{f(x)}\right| \leqslant \left|\frac{\tilde f(\hat x) - f(\hat x)}{f(x)}\right| + \left|\frac{f(\hat x) - f(x)}{f(x)}\right|$$

$$\lessapprox \underbrace{\left|\frac{\tilde f(\hat x) - f(\hat x)}{f(\hat x)}\right|}_{\text{numerical error}} + \underbrace{\left|\frac{f(\hat x) - f(x)}{f(x)}\right|}_{\text{perturbation error}} \leqslant (\hat\kappa_{\text{num}} + \kappa_f)\,\boxed{\text{eps}}.$$

where $\kappa = \kappa_f$ be the (relative) condition number of the exact problem f and

$$\hat\kappa_{\text{num}} = \max \left|\frac{\tilde f(\hat x) - f(\hat x)}{f(\hat x)}\right| \Bigg/ \left|\frac{\hat x - x}{x}\right|.$$

Example: Root-Finding Revisited

Consider again solving the quadratic problem $ar^2 + br + c = 0$.

- Taking $a = c = 1$ and $b = -(10^6 + 10^{-6})$, the roots can be computed exactly by hand: $r_1 = 10^6$ and $r_2 = 10^{-6}$.
- If numerically computed in MATLAB using the quadratic equation formula, r_1 is correct but r_2 has only 5 correct digits.
- Fix it using $r_2 = (c/a)/r_1$.

Lec 13: Square Linear Systems – Introduction

Opening Example: Polynomial Interpolation

Polynomial Interpolation

Formal Statement

Given a set of n data points $\{(x_j, y_j) \mid j \in \mathbb{N}[1,n]\}$ with distinct x_j's, not necessarily sorted, find a polynomial of degree $n-1$,

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

which interpolates the given points, *i.e.*,

$$p(x_j) = y_j, \quad \text{for } j = 1, 2, \ldots, n\,.$$

- The goal is to determine the coefficients $c_1, c_2, \ldots, c_n$.
- Note that the total number of data point is 1 larger than the degree of the interpolating polynomial.

Why Do We Care?

- to find the values between the discrete data points;
- to approximate a (complicated) function by a polynomial, which makes such computations as differentiation or integration easier.

Interpolation to Linear System

Writing out the n *interpolating conditions* $p(x_j) = y_j$:

Equations **Matrix equation**

$$\left\{\begin{array}{lllll} c_1 & + c_2 x_1 & + \cdots & + c_n x_1^{n-1} & = y_1 \\ c_1 & + c_2 x_2 & + \cdots & + c_n x_2^{n-1} & = y_2 \\ \vdots & \vdots & & \vdots & \vdots \\ c_1 & + c_2 x_n & + \cdots & + c_n x_n^{n-1} & = y_n \end{array}\right\} \to \underbrace{\begin{bmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \cdots & x_n^{n-1} \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}}_{\mathbf{c}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y}}$$

- This is a linear system of n equations with n unknowns.
- The matrix V is called a **Vandermonde matrix**.

Example: Fitting Population Data

U.S. Census data are collected every 10 years.

Year	Population (millions)
1980	226.546
1990	248.710
2000	281.422
2010	308.746
2020	332.639

Question. How do we estimate population in other years?

- Interpolate available data to compute population in intervening years.

Example: Fitting Population Data

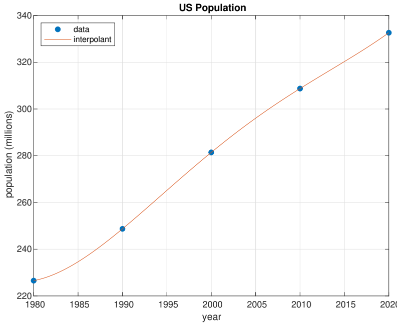
- Input data.
- Match up notation (optional).
- Note the shift in Line 7.
- Construct the Vandermonde matrix V by *broadcasting*.
- Solve the system using the backslash (\) operator.

```
 1  year = (1980:10:2020)';
 2  pop = [226.546;
 3         248.710;
 4         281.422;
 5         308.746;
 6         332.639];
 7  x = year - 1980;
 8  y = pop;
 9  n = length(x);
10  V = x.^(0:n-1);
11  c = V \ y;
```

Post-Processing

```
xx = linspace(0, 40, 100)';
yy = polyval(flip(c), xx);
clf
plot(1980+x, y, '.', 1980+xx, yy)
title('US Population'),
xlabel('year'), ylabel('population (millions)')
legend('data', 'interpolant', 'location', 'northwest')
```

- Use the `polyval` function to evaluate the polynomial.
- MATLAB expects coefficients to be in descending order. (`flip`)



Square Linear Systems

Overview

Let $A \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$. Then the equation $A\mathbf{x} = \mathbf{b}$ has the following possibilities:

- If A is invertible (or nonsingular), then $A\mathbf{x} = \mathbf{b}$ has a unique solution $\mathbf{x} = A^{-1}\mathbf{b}$, or
- If A is not invertible (or singular), then $A\mathbf{x} = \mathbf{b}$ has either no solution or infinitely many solutions.

The Backslash Operator " \ "

To solve for $\mathbf{x}$ in MATLAB, we use the backslash symbol " \ ":

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

This produces the solution without explicitly forming the inverse of A.

Warning: Even though $\mathbf{x} = A^{-1}\mathbf{b}$ analytically, don't use `x = inv(A)*b`!

Triangular Systems

Systems involving triangular matrices are easy to solve.

- A matrix $U \in \mathbb{R}^{n\times n}$ is **upper triangular** if all entries below main diagonal are zero:

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix}.$$

- A matrix $L \in \mathbb{R}^{n\times n}$ is **lower triangular** if all entries above main diagonal are zero:

$$L = \begin{bmatrix} \ell_{11} & 0 & 0 & \cdots & 0 \\ \ell_{21} & \ell_{22} & 0 & \cdots & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \cdots & \ell_{nn} \end{bmatrix}.$$

Example: Upper Triangular Systems

Solve the following 4×4 system

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}.$$

General Results

- **Backward Substitution.** To solve a general $n \times n$ upper triangular system $U\mathbf{x} = \mathbf{y}$:

$$\begin{cases} x_n = \dfrac{b_n}{u_{nn}} \quad \text{and} \\ x_i = \dfrac{1}{u_{ii}} \left(b_i - \displaystyle\sum_{j=i+1}^{n} u_{ij} x_j \right) \end{cases}$$

  for $i = n-1, n-2, \ldots, 1$.

- **Forward Elimination.** To solve a general $n \times n$ lower triangular system $L\mathbf{x} = \mathbf{y}$:

$$\begin{cases} x_1 = \dfrac{b_1}{\ell_{11}} \quad \text{and} \\ x_i = \dfrac{1}{\ell_{ii}} \left(b_i - \displaystyle\sum_{j=1}^{i-1} \ell_{ij} x_j \right) \end{cases}$$

  for $i = 2, 3, \ldots, n$.

Implementation: Backward Substitution

```
function x = backsub(U,b)
% BACKSUB x = backsub(U,b)
% Solve an upper triangular linear system.
% Input:
%   U    upper triangular square matrix (n by n)
%   b    right-hand side vector (n by 1)
% Output:
%   x    solution of Ux=b (n by 1 vector)
    n = length(U);
    x = zeros(n,1); % preallocate
    for i = n:-1:1
        x(i) = ( b(i) - U(i,i+1:n)*x(i+1:n) ) / U(i,i);
    end
end
```

Implementation: Forward Elimination

Exercise. Complete the code below.

```
function x = forelim(U,b)
% FORELIM x = forelim(L,b)
% Solve a lower triangular linear system.
% Input:
%   L    lower triangular square matrix (n by n)
%   b    right-hand side vector (n by 1)
% Output:
%   x    solution of Lx=b (n by 1 vector)





end
```

Does It Always Work?

Theorem 1 (Singularity of Triangular Matrix)

A triangular matrix is singular if and only if at least one of its diagonal elements is zero.

Lec 14: Square Linear Systems – LU Factorization

Gaussian Elimination

General Method: Gaussian Elimination

- *Gaussian elimination* is an algorithm for solving a general system of linear equations that involves a sequence of row operations performed on the associated matrix of coefficients.
- This is also known as the method of row reduction.
- There are three variations to this method:
    - G.E. without pivoting
    - G.E. with partial pivoting (that is, row pivoting)
    - G.E. with full pivoting (that is, row and column pivoting)

G.E. Without Pivoting: Example

Key Example

Solve the following system of equations.

$$\begin{cases} 2x_1 + 2x_2 + x_3 = 6 \\ -4x_1 + 6x_2 + x_3 = -8 \\ 5x_1 - 5x_2 + 3x_3 = 4 \end{cases} \xrightarrow{\text{matrix equation}} \underbrace{\begin{bmatrix} 2 & 2 & 1 \\ -4 & 6 & 1 \\ 5 & -5 & 3 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 6 \\ -8 \\ 4 \end{bmatrix}}_{\mathbf{b}}$$

Step 1: Write the corresponding *augmented matrix* and row-reduce to an echelon form.

$$\left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ -4 & 6 & 1 & -8 \\ 5 & -5 & 3 & 4 \end{array}\right] \to \left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ 0 & 10 & 3 & 4 \\ 0 & 10 & -0.5 & 11 \end{array}\right] \to \left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ 0 & 10 & 3 & 4 \\ 0 & 0 & 3.5 & -7 \end{array}\right].$$

Step 2: Solve for x_3, then x_2, and then x_1 via *backward substitution*.

$$\mathbf{x} = (3, 1, -2)^{\mathrm{T}}.$$

G.E. without Pivoting: General Procedure

As shown in the example, G.E. without pivoting involves two steps:

1. **Row reduction:** Transform $A\mathbf{x} = \mathbf{b}$ to $U\mathbf{x} = \boldsymbol{\beta}$ where

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \cdots & u_{2n} \\ & & \ddots & \vdots \\ \mathbf{0} & & & u_{nn} \end{bmatrix} \quad \text{and} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}.$$

2. **Backward substitution:** Solve $U\mathbf{x} = \boldsymbol{\beta}$ for $\mathbf{x}$ by

$$\begin{cases} x_n = \dfrac{\beta_n}{u_{nn}} \quad \text{and} \\ x_i = \dfrac{1}{u_{ii}} \left(\beta_i - \displaystyle\sum_{j=i+1}^{n} u_{ij} x_j \right), \quad \text{for } i = n-1, n-2, \ldots, 1\,. \end{cases}$$

G.E. without Pivoting: MATLAB Implementation

```
 1  function x = GEnp(A, b)
 2    % Step 1: Row reduction to upper tri. system
 3    S = [A, b];           % augmented matrix
 4    n = size(A, 1);
 5    for j = 1:n-1
 6        for i = j+1:n
 7            mult = -S(i,j)/S(j,j);
 8            S(i,:) = S(i,:) + mult*S(j,:);
 9        end
10    end
11    % Step 2: Backward substitution
12    U = S(:,1:end-1);
13    beta = S(:,end);
14    x = backsub(U, beta);
15  end
```

Exercise. Rewrite Lines 6–9 without using a loop. (Think *vectorized*!)

G.E. with Partial Pivoting: Procedure

In this variation of G.E., reduction to echelon form is done slightly differently.

- On the augmented matrix $[A \mid \mathbf{b}]$,

Key Process (partial pivoting)

1. Find the entry in the first column with the largest absolute value. This entry is called the *pivot*.
2. Perform a row interchange, if necessary, so that the pivot is on the first diagonal position.
3. Use elementary row operations to reduce the remaining entries in the first column to zero.

- Once done, ignore the first row and first column and repeat the **Key Process** on the remaining submatrix.
- Continue this until the matrix is in a row-echelon form.

G.E. with Partial Pivoting: Example

Let's solve the example on p. 267 again, now using G.E. with partial pivoting.

1st column:

$$\left[\begin{array}{ccc|c} 2 & 2 & 1 & 6 \\ -4 & 6 & 1 & -8 \\ \boxed{5} & -5 & 3 & 4 \end{array}\right] \xrightarrow{\text{pivot}} \left[\begin{array}{ccc|c} 5 & -5 & 3 & 4 \\ -4 & 6 & 1 & -8 \\ 2 & 2 & 1 & 6 \end{array}\right] \xrightarrow{\text{zero}} \left[\begin{array}{ccc|c} 5 & -5 & 3 & 4 \\ 0 & 2 & 3.4 & -4.8 \\ 0 & 4 & -0.2 & 4.4 \end{array}\right]$$

2nd column:

$$\left[\begin{array}{ccc|c} 5 & -5 & 3 & 4 \\ 0 & 2 & 3.4 & -4.8 \\ 0 & \boxed{4} & -0.2 & 4.4 \end{array}\right] \xrightarrow{\text{pivot}} \left[\begin{array}{ccc|c} 5 & -5 & 3 & 4 \\ 0 & 4 & -0.2 & 4.4 \\ 0 & 2 & 3.4 & -4.8 \end{array}\right] \xrightarrow{\text{zero}} \left[\begin{array}{ccc|c} 5 & -5 & 3 & 4 \\ 0 & 4 & -0.2 & 4.4 \\ 0 & 0 & 3.5 & -7 \end{array}\right]$$

Now that the last matrix is upper triangular, we work up from the third equation to the second to the first and obtain the same solution as before.

G.E. with Partial Pivoting: MATLAB Implementation

Exercise

Write a MATLAB function `GEpp.m` which carries out G.E. with partial pivoting.

- Modify `GEnp.m` on p. 269 to incorporate partial pivoting.
- The only part that needs to be changed is the for-loop starting at Line 5.
    - Right after `for j = 1:n-1`, find the index of the pivot element of the jth column of A below the diagonal.

```
[~, iM] = max(abs(A(j:end,j)));
iM = iM + j - 1;
```

    - If the pivot element is not on the diagonal, swap rows so that it is on the diagonal.

```
if j ~= iM
    S([j iM], :) = S([iM j], :)
end
```

Why Is Pivoting Necessary?

Example

Given $\epsilon \ll 1$, solve the system

$$\begin{bmatrix} -\epsilon & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1-\epsilon \\ 0 \end{bmatrix}$$

using Gaussian elimination with and without partial pivoting.

Without pivoting: By $R_2 \to R_2 + (1/\epsilon)R_1$, we have

$$\begin{bmatrix} -\epsilon & 1 \\ 0 & -1+1/\epsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1-\epsilon \\ 1/\epsilon - 1 \end{bmatrix} \implies \begin{cases} x_2 = 1, \\ x_1 = \dfrac{(1-\epsilon)-1}{-\epsilon}. \end{cases}$$

- In exact arithmetic, this yields the correct solution.
- In floating-point arithmetic, calculation of x_1 suffers from catastrophic cancellation.

Why Is Pivoting Necessary? (Cont')

Example

Given $\epsilon \ll 1$, solve the system

$$\begin{bmatrix} -\epsilon & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1-\epsilon \\ 0 \end{bmatrix}$$

using Gaussian elimination with and without partial pivoting.

With partial pivoting: First, swap the rows $R_1 \leftrightarrow R_2$, and then do $R_2 \to R_2 + \epsilon R_1$ to obtain

$$\begin{bmatrix} 1 & -1 \\ 0 & 1-\epsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1-\epsilon \end{bmatrix} \quad \Longrightarrow \quad \begin{cases} x_2 = 1, \\ x_1 = \dfrac{0-(-1)}{1}. \end{cases}$$

- Each of the arithmetic steps (to compute x_1, x_2) is well-conditioned.
- The solution is computed stably.

LU Factorization

Emulation of Gaussian Elimination

In this section, we emulate row operations steps required in Gaussian elimination by matrix multiplications. **Two major operations.**

- Row interchange $R_i \leftrightarrow R_j$:

  $$P(i,j)A, \quad \text{where } P(i,j) \text{ is an elementary permutation matrix.}$$

- Row replacement $R_i \to R_i + cR_j$:

  $$(I + c\mathbf{e}_i\mathbf{e}_j^{\mathrm{T}})A$$

See Appendix for more details.

Key Example Revisited

Let's work out the key example from last time once again, now in matrix form $A\mathbf{x} = \mathbf{b}$.

$$\begin{bmatrix} 2 & 2 & 1 \\ -4 & 6 & 1 \\ 5 & -5 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -8 \\ 4 \end{bmatrix}.$$

[Pivot] Switch R_1 and R_3 using $P(1,3)$:

$$\underbrace{\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}}_{P(1,3)} \left[\begin{bmatrix} 2 & 2 & 1 \\ -4 & 6 & 1 \\ 5 & -5 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -8 \\ 4 \end{bmatrix} \right] \longrightarrow \begin{bmatrix} 5 & -5 & 3 \\ -4 & 6 & 1 \\ 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -8 \\ 6 \end{bmatrix}$$

[Zero] Do row operations $R_2 \to R_2 + (4/5)R_1$ and $R_3 \to R_3 - (2/5)R_1$:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 4/5 & 1 & 0 \\ -2/5 & 0 & 1 \end{bmatrix}}_{G_1} \left[\begin{bmatrix} 5 & -5 & 3 \\ -4 & 6 & 1 \\ 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -8 \\ 6 \end{bmatrix} \right]$$

$$\longrightarrow \begin{bmatrix} 5 & -5 & 3 \\ 0 & 2 & 3.4 \\ 0 & 4 & -0.2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -4.8 \\ 4.4 \end{bmatrix}$$

Key Example Revisited (cont')

[Pivot] Switch R_2 and R_3 using $P(2,3)$:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}}_{P(2,3)} \left[\begin{bmatrix} 5 & -5 & 3 \\ 0 & 2 & 3.4 \\ 0 & 4 & -0.2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -4.8 \\ 4.4 \end{bmatrix} \right]$$

$$\longrightarrow \begin{bmatrix} 5 & -5 & 3 \\ 0 & 4 & -0.2 \\ 0 & 2 & 3.4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 4.4 \\ -4.8 \end{bmatrix}$$

[Zero] Do a row operation $R_3 \to R_3 - (1/2)R_2$:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/2 & 1 \end{bmatrix}}_{G_2} \left[\begin{bmatrix} 5 & -5 & 3 \\ 0 & 4 & -0.2 \\ 0 & 2 & 3.4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 4.4 \\ -4.8 \end{bmatrix} \right]$$

$$\longrightarrow \underbrace{\begin{bmatrix} 5 & -5 & 3 \\ 0 & 4 & -0.2 \\ 0 & 0 & 3.5 \end{bmatrix}}_{U} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 4.4 \\ -7 \end{bmatrix}$$

Analysis of Example

- The previous calculations can be summarized as

$$G_2P(2,3)G_1P(1,3)A = U\,. \tag{$\star$}$$

- Using the noted properties of permutation matrices and GTMs, ($\star$) can be written as

$$G_2P(2,3)G_1\underbrace{P(2,3)P(2,3)}_{=I}P(1,3)A = U$$
$$\longrightarrow G_2\underbrace{P(2,3)G_1P(2,3)}_{=:\tilde{G}_1}\underbrace{P(2,3)P(1,3)}_{=:P}A = U\,.$$

- The above can be summarized as $PA = LU$ where $L = (G_2\widetilde{G}_1)^{-1}$ is a lower triangular matrix.

Generalization – PLU Factorization

For an arbitrary matrix $A \in \mathbb{R}^{n\times n}$, the partial pivoting and row operations are intermixed as

$$G_{n-1}P(n-1,r_{n-1})\cdots G_2P(2,r_2)G_1P(1,r_1)A = U\,.$$

Going through the same calculations as above, it can always be written as

$$\left(\widetilde{G}_{n-1}\cdots\widetilde{G}_2\widetilde{G}_1\right)P(n-1,r_{n-1})\cdots P(2,r_2)P(1,r_1)A = U\,,$$

which again leads to $PA = LU$:

$$\underbrace{P(n-1,r_{n-1})\cdots P(2,r_2)P(1,r_1)}_{=:P}A = \underbrace{\left(\widetilde{G}_{n-1}\cdots\widetilde{G}_2\widetilde{G}_1\right)^{-1}}_{=:L}U\,.$$

This is called the **PLU factorization** of matrix A.

LU and PLU Factorization

If no pivoting is required, the previous procedure simplifies to

$$G_{n-1} \cdots G_2 G_1 A = U \,.$$

which leads to $A = LU$:

$$A = \underbrace{\left(G_{n-1} \cdots G_2 G_1\right)^{-1}}_{=:L} U \,.$$

This is called the **LU factorization** of matrix A.

Implementation of LU Factorization

```
function [L,U] = mylu(A)
% MYLU   LU factorization (demo only--not stable!).
% Input:
%   A    square matrix
% Output:
%   L,U  unit lower triangular and upper triangular such that
   LU=A
  n = length(A);
  L = eye(n);    % ones on diagonal
  % Gaussian elimination
  for j = 1:n-1
    for i = j+1:n
      L(i,j) = A(i,j) / A(j,j);    % row multiplier
      A(i,j:n) = A(i,j:n) - L(i,j)*A(j,j:n);
    end
  end
  U = triu(A);
end
```

Implementation of LU Factorization

Exercise. Write a MATLAB function `myplu` for PLU factorization by modifying the previous function `mylu.m`.

```
function [L,U,P] = myplu(A)
% MYPLU   PLU factorization (demo only--not stable!).
% Input:
%   A     square matrix
% Output:
%   P,L,U  permutation, unit lower triangular, and upper
    triangular such that LU=PA

% Your code here.




end
```

Solving a Square System Using PLU Factorization

Multiplying $A\mathbf{x} = \mathbf{b}$ on the left by P we obtain

$$\underbrace{PA}_{=LU}\mathbf{x} = \underbrace{P\mathbf{b}}_{=:\boldsymbol{\beta}} \quad \longrightarrow \quad LU\mathbf{x} = \boldsymbol{\beta}\,,$$

which can be solved in two steps:

- Define $U\mathbf{x} = \mathbf{y}$ and solve for $\mathbf{y}$ in the equation

$$L\mathbf{y} = \boldsymbol{\beta}\,. \qquad \text{(forward elimination)}$$

- Having calculated $\mathbf{y}$, solve for $\mathbf{x}$ in the equation

$$U\mathbf{x} = \mathbf{y}\,. \qquad \text{(backward substitution)}$$

Solving a Square System Using PLU Factorization

- Using the instructional codes (`backsub`, `forelim`, `myplu`):

```
[L,U,P] = myplu(A);
x = backsub( U, forelim(L, P*b) );
```

- Using MATLAB's built-in functions:

```
[L,U,P] = lu(A);
x = U \ (L \ (P*b));
```

  - The backslash is designed so that triangular systems are solved with the appropriate substitution.

- The most compact way:

```
x = A \ b;
```

  - The backslash does partial pivoting and triangular substitutions silently and automatically.

Tutorial: Row and Column Operations

Notation and Terminology

Notation: Unit Basis Vectors

Throughout this tutorial, suppose $n \in \mathbb{N}$ is fixed. Let I be the $n \times n$ identity matrix and denote by $\mathbf{e}_j$ its jth column, *i.e.*,

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \left[\begin{array}{c|c|c|c} & & & \\ \mathbf{e}_1 & \mathbf{e}_2 & \cdots & \mathbf{e}_n \\ & & & \end{array}\right].$$

That is,

$$\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \cdots, \quad \mathbf{e}_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

Notation: Concatenation

Let $A \in \mathbb{R}^{n\times n}$. We can view it as a concatenation of its rows or columns as visualized below.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \left[\begin{array}{c|c|c|c} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{array}\right] = \left[\begin{array}{c} \boldsymbol{\alpha}_1^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_2^{\mathrm{T}} \\ \hline \vdots \\ \hline \boldsymbol{\alpha}_n^{\mathrm{T}} \end{array}\right].$$

Basic Row and Column Operations

Row or Column Extraction

A row or a column of A can be extracted using columns of I.

Operation	Mathematics	MATLAB
extract the ith row of A	$\mathbf{e}_i^{\mathrm{T}} A$	`A(i,:)`
extract the jth column of A	$A\mathbf{e}_j$	`A(:,j)`
extract the (i, j) entry of A	$\mathbf{e}_i^{\mathrm{T}} A\mathbf{e}_j$	`A(i,j)`

Elementary Permutation Matrices

Definition 2 (Elementary Permutation Matrix)

For $i, j \in \mathbb{N}[1, n]$ distinct, denote by $P(i, j)$ the $n \times n$ matrix obtained by interchanging the ith and jth rows of the $n \times n$ identity matrix. Such matrices are called *elementary permutation matrices*.

Example. ($n = 4$)

$$P(1,2) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad P(1,3) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \cdots$$

Notable Properties.

- $P(i, j) = P(j, i)$
- $P(i, j)^2 = I$

Row or Column Interchange

Elementary permutation matrices are useful in interchanging rows or columns.

Operation	Mathematics	MATLAB
$\boldsymbol{\alpha}_i^{\mathrm{T}} \leftrightarrow \boldsymbol{\alpha}_j^{\mathrm{T}}$	$P(i,j)A$	`A([i,j],:)=A([j,i],:)`
$\mathbf{a}_i \leftrightarrow \mathbf{a}_j$	$AP(i,j)$	`A(:,[i,j])=A(:,[j,i])`

Permutation Matrices

Definition 3 (Permutation Matrix)

A *permutation matrix* $P \in \mathbb{R}^{n\times n}$ is a square matrix obtained from the same-sized identity matrix by re-ordering of rows.

Notable Properties.

- $P^{\mathrm{T}} = P^{-1}$
- A product of *elementary permutation matrices* is a permutation matrix.

Row and Column Operations. For any $A \in \mathbb{R}^{n\times n}$,

- PA permutes the rows of A.
- AP permutes the columns of A.

Row or Column Rearrangement

Question

Let $A \in \mathbb{R}^{6\times 6}$, and suppose that it is stored in MATLAB. Rearrange rows of A by moving 1st to 2nd, 2nd to 3rd, 3rd to 5th, 4th to 6th, 5th to 4th, and 6th to 1st, that is,

$$\begin{bmatrix} \boldsymbol{\alpha}_1^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_2^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_3^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_4^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_5^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_6^{\mathrm{T}} \end{bmatrix} \longrightarrow \begin{bmatrix} \boldsymbol{\alpha}_6^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_1^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_2^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_5^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_3^{\mathrm{T}} \\ \hline \boldsymbol{\alpha}_4^{\mathrm{T}} \end{bmatrix}$$

Row or Column Rearrangement

Solution.

- Mathematically: PA where

$$P = \begin{bmatrix} \mathbf{e}_6^{\mathrm{T}} \\ \hline \mathbf{e}_1^{\mathrm{T}} \\ \hline \mathbf{e}_2^{\mathrm{T}} \\ \hline \mathbf{e}_5^{\mathrm{T}} \\ \hline \mathbf{e}_3^{\mathrm{T}} \\ \hline \mathbf{e}_4^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

- MATLAB:

```
A = A([6 1 2 5 3 4], :)
% short for  A([1 2 3 4 5 6], :) = A([6 1 2 5 3 4], :)
```

Gaussian Transformation Matrices

Elementary Row Operation and GTM

Let $1 \leqslant j < i \leqslant n$.

- The row operation $R_i \to R_i + cR_j$ on $A \in \mathbb{R}^{n\times n}$, for some $c \in \mathbb{R}$, can be emulated by a matrix multiplication[4]

$$(I + c\,\mathbf{e}_i\mathbf{e}_j^{\mathrm{T}})A.$$

- In the context of Gaussian elimination, the operation of introducing zeros below the jth diagonal entry can be done via

$$\underbrace{\Big(I + \sum_{i=j+1}^{n} c_{i,j}\,\mathbf{e}_i\mathbf{e}_j^{\mathrm{T}}\Big)}_{=G_j} A, \quad 1 \leqslant j < n.$$

  The matrix G_j is called a *Gaussian transformation matrix* (GTM).

[4] Many linear algebra texts refer to the matrix in parentheses as an *elementary matrix*.

Elementary Row Operation and GTM (cont')

- To emulate $(I + c\mathbf{e}_i\mathbf{e}_j^{\mathrm{T}})A$ in MATLAB:

```
A(i,:) = A(i,:) + c*A(j,:);
```

- To emulate

$$G_jA = (I + \sum_{i=j+1}^{n} c_{i,j}\mathbf{e}_i\mathbf{e}_j^{\mathrm{T}})A$$

  in MATLAB:

```
for i = j+1:n
    c = ....
    A(i,:) = A(i,:) + c*A(j,:);
end
```

  This can be done without using a loop.

Analytical Properties of GTM

- GTMs are *unit* lower triangular matrices.
- The product of GTMs is another unit lower triangular matrix.
- The inverse of a GTM is also a unit lower triangular matrix.

Lec 15: Square Linear Systems – Analysis

Efficiency

Notation: Big-O and Asymptotic

Let f, g be positive functions defined on $\mathbb{N}$.

- $f(n) = O\left(g(n)\right)$ ("f is *big-O* of g") as $n \to \infty$ if

$$\frac{f(n)}{g(n)} \leqslant C, \quad \text{for all sufficiently large } n.$$

- $f(n) \sim g(n)$ ("f is *asymptotic* to g") as $n \to \infty$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

Timing Vector/Matrix Operations – FLOPS

- One way to measure the “efficiency” of a numerical algorithm is to count the number of floating-point arithmetic operations (FLOPS) necessary for its execution.
- The number is usually represented by $\sim cn^p$ where c and p are given explicitly.
- We are interested in this formula when n is large.

FLOPS for Major Operations

Vector/Matrix Operations

Let $x, y \in \mathbb{R}^n$ and $A, B \in \mathbb{R}^{n \times n}$. Then

- (vector-vector) $x^{\mathrm{T}} y$ requires $\sim 2n$ *flops*.
- (matrix-vector) Ax requires $\sim 2n^2$ *flops*.
- (matrix-matrix) AB requires $\sim 2n^3$ *flops*.

Cost of PLU Factorization

Note that we only need to count the number of *flops* required to zero out elements below the diagonal of each column.

- For each $i > j$, we replace R_i by $R_i + cR_j$ where $c = -a_{i,j}/a_{j,j}$. This requires approximately $2(n-j+1)$ *flops*:
  - 1 division to form c
  - $n-j+1$ multiplications to form cR_j
  - $n-j+1$ additions to form $R_i + cR_j$
- Since $i \in \mathbb{N}[j+1, n]$, the total number of *flops* needed to zero out all elements below the diagonal in the jth column is approximately $2(n-j+1)(n-j)$.
- Summing up over $j \in \mathbb{N}[1, n-1]$, we need about $(2/3)n^3$ *flops*:

$$\sum_{j=1}^{n-1} 2(n-j+1)(n-j) \sim 2\sum_{j=1}^{n-1}(n-j)^2 = 2\sum_{j=1}^{n-1} j^2 \sim \frac{2}{3}n^3$$

Cost of Forward Elimination and Backward Substitution

Forward Elimination

- The calculation of $y_i = \beta_i - \sum_{j=1}^{i-1} \ell_{ij} y_j$ for $i > 1$ requires approximately $2i$ *flops*:
  - 1 subtraction
  - $i - 1$ multiplications
  - $i - 2$ additions
- Summing over all $i \in \mathbb{N}[2, n]$, we need about n^2 *flops*:

$$\sum_{i=2}^{n} 2i \sim 2\frac{n^2}{2} = n^2 \,.$$

Backward Substitution

- The cost of backward substitution is also approximately n^2 *flops*, which can be shown in the same manner.

Cost of G.E. with Partial Pivoting

Gaussian elimination with partial pivoting involves three steps:

- PLU factorization: $\sim (2/3)n^3$ *flops*
- Forward elimination: $\sim n^2$ *flops*
- Backward substitution: $\sim n^2$ *flops*

Summary

The total cost of Gaussian elimination with partial pivoting is approximately

$$\frac{2}{3}n^3 + n^2 + n^2 \sim \frac{2}{3}n^3$$

flops for large n.

Application: Solving Multiple Square Systems Simultaneously

To solve two systems $A\mathbf{x}_1 = \mathbf{b}_1$ and $A\mathbf{x}_2 = \mathbf{b}_2$.

Method 1.

- Use G.E. for both.
- It takes $\sim (4/3)n^3$ *flops*.

```
%% method 1
 x1 = A \ b1;
 x2 = A \ b2;
```

Method 2.

- Do it in two steps:
  1. Do PLU factorization $PA = LU$.
  2. Then solve $LU\mathbf{x}_1 = P\mathbf{b}_1$ and $LU\mathbf{x}_2 = P\mathbf{b}_2$.
- It takes $\sim (2/3)n^3$ *flops*.

```
%% method 2
 [L, U, P] = lu(A);
 x1 = U \ (L \ (P*b1));
 x2 = U \ (L \ (P*b2));
```

```
%% compact implementation
 X = A \ [b1, b2];
 x1 = X(:, 1);
 x2 = X(:, 2);
```

Lec 16: Square Linear Systems – Further Analysis

Vector and Matrix Norms

Vector Norms

The "length" of a vector $\mathbf{v}$ can be measured by its **norm**.

Definition 4 (p-Norm of a Vector)

Let $p \in [1, \infty)$. The p-norm of $\mathbf{v} \in \mathbb{R}^m$ is denoted by $\|\mathbf{v}\|_p$ and is defined by

$$\|\mathbf{v}\|_p = \left(\sum_{i=1}^{m} |v_i|^p\right)^{1/p}.$$

When $p = \infty$,

$$\|\mathbf{v}\|_\infty = \max_{1\leqslant i\leqslant m} |v_i| .$$

The most commonly used p values are 1, 2, and ∞:

$$\|\mathbf{v}\|_1 = \sum_{i=1}^{m} |v_i| , \quad \|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{m} |v_i|^2}.$$

Vector Norms

In general, any function $\|\cdot\| : \mathbb{R}^m \to \mathbb{R}^+ \cup \{0\}$ is called a **vector norm** if it satisfies the following three properties:

1. $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = 0$.
2. $\|\alpha\mathbf{x}\| = |\alpha| \, \|\mathbf{x}\|$ for any constant α and any $\mathbf{x} \in \mathbb{R}^m$.
3. $\|\mathbf{x}+\mathbf{y}\| \leqslant \|\mathbf{x}\| + \|\mathbf{y}\|$ for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$. This is called the *triangle inequality*.

Unit Vectors

- A vector $\mathbf{u}$ is called a **unit vector** if $\|\mathbf{u}\| = 1$.
- Depending on the norm used, unit vectors will be different.
- For instance:

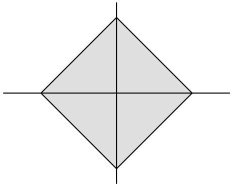


Figure 6: 1-norm

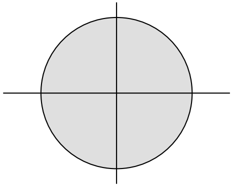


Figure 7: 2-norm

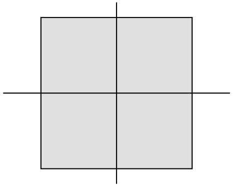


Figure 8: ∞-norm

Matrix Norms

The “size” of a matrix $A \in \mathbb{R}^{m\times n}$ can be measured by its **norm** as well. As above, we say that a function $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}^{+} \cup \{0\}$ is a **matrix norm** if it satisfies the following three properties:

1. $\|A\| = 0$ if and only if $A = 0$.
2. $\|\alpha A\| = |\alpha| \, \|A\|$ for any constant α and any $A \in \mathbb{R}^{m\times n}$.
3. $\|A + B\| \leqslant \|A\| + \|B\|$ for any $A, B \in \mathbb{R}^{m\times n}$. This is called the *triangle inequality*.

Matrix Norms (Cont')

- If, in addition to satisfying the three conditions, it satisfies

$$\|AB\| \leqslant \|A\| \, \|B\| \quad \text{for all } A \in \mathbb{R}^{m\times n} \text{ and all } B \in \mathbb{R}^{n\times p},$$

  it is said to be **consistent**.

- If, in addition to satisfying the three conditions, it satisfies

$$\|A\mathbf{x}\| \leqslant \|A\| \, \|\mathbf{x}\| \quad \text{for all } A \in \mathbb{R}^{m\times n} \text{ and all } \mathbf{x} \in \mathbb{R}^{n},$$

  then we say that it is **compatible** with a vector norm.

Induced Matrix Norms

Definition 5 (p-Norm of a Matrix)

Let $p \in [1, \infty]$. The p-norm of $A \in \mathbb{R}^{m \times n}$ is given by

$$\|A\|_p = \max_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p} = \max_{\|\mathbf{x}\|_p = 1} \|A\mathbf{x}\|_p \ .$$

- The definition of this particular matrix norm is **induced** from the vector p-norm.
- By construction, matrix p-norm is a compatible norm.
- Induced norms describe how the matrix stretches unit vectors with respect to the vector norm.

Induced Matrix Norms

The commonly used p-norms (for $p = 1, 2, \infty$) can also be calculated by

$$\begin{aligned}
\|A\|_1 &= \max_{1 \leqslant j \leqslant n} \sum_{i=1}^{m} \left|a_{ij}\right|, \\
\|A\|_2 &= \sqrt{\lambda_{\max}(A^{\mathrm{T}}A)} = \sigma_{\max}(A), \\
\|A\|_\infty &= \max_{1 \leqslant i \leqslant m} \sum_{j=1}^{n} \left|a_{ij}\right|.
\end{aligned}$$

In words,

- The 1-norm of A is the maximum of the 1-norms of all column vectors.
- The 2-norm of A is the square root of the largest eigenvalue of $A^{\mathrm{T}}A$.
- The ∞-norm of A is the maximum of the 1-norms of all row vectors.

Non-Induced Matrix Norm – Frobenius Norm

Definition 6 (Frobenius Norm of a Matrix)

The Frobenius norm of $A \in \mathbb{R}^{m \times n}$ is given by

$$\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}.$$

- This is not induced from a vector p-norm.
- However, both p-norm and the Frobenius norm are consistent and compatible.
- For compatibility of the Frobenius norm, the vector norm must be the 2-norm, that is, $\|A\mathbf{x}\|_2 \leqslant \|A\|_F \|\mathbf{x}\|_2$.

Norms in MATLAB

- Vector p-norms can be easily computed:

```
norm(v, 1)       % = sum(abs(v))
norm(v, 2)       % = sqrt(v'*v)  if v is a column
norm(v, 'inf')   % = max(abs(v))
```

- The same function `norm` is used to calculate matrix p-norms:

```
norm(A, 1)        % = max(sum(abs(A), 1))
norm(A, 2)        % = max(sqrt(eig(A'*A)))
norm(A, Inf)      % = max(sum(abs(A), 2))
```

- To calculate the Frobenius norm:

```
norm(A, 'fro')    % = sqrt(A(:)'*A(:))
                  % = norm(A(:), 2)
```

Conditioning

Conditioning of Solving Linear Systems: Overview

- Analyze how robust (or sensitive) the solutions of $A\mathbf{x} = \mathbf{b}$ are to perturbations of A and $\mathbf{b}$.
- For simplicity, consider separately the cases where
  1. $\mathbf{b}$ changes to $\mathbf{b} + \delta\mathbf{b}$, while A remains unchanged, that is
  $$A\mathbf{x} = \mathbf{b} \quad \longrightarrow \quad A(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}.$$
  2. A changes to $A + \delta A$, while $\mathbf{b}$ remains unchanged, that is
  $$A\mathbf{x} = \mathbf{b} \quad \longrightarrow \quad (A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b}.$$

Sensitivity to Perturbation of RHS

Case 1. $A\mathbf{x} = \mathbf{b} \ \rightarrow \ A(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$

- Bound $\|\delta\mathbf{x}\|$ in terms of $\|\delta\mathbf{b}\|$:

$$\begin{aligned} A\mathbf{x} + A\delta\mathbf{x} &= \mathbf{b} + \delta\mathbf{b} \\ A\delta\mathbf{x} &= \delta\mathbf{b} \\ \delta\mathbf{x} &= A^{-1}\delta\mathbf{b} \end{aligned} \qquad \Longrightarrow \qquad \|\delta\mathbf{x}\| \leqslant \|A^{-1}\|\|\delta\mathbf{b}\|.$$

- Sensitivity in terms of relative errors:

$$\frac{\dfrac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|}}{\dfrac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}} = \frac{\|\delta\mathbf{x}\|\,\|\mathbf{b}\|}{\|\delta\mathbf{b}\|\,\|\mathbf{x}\|} \leqslant \frac{\|A^{-1}\|\,\|\delta\mathbf{b}\| \cdot \|A\|\,\|\mathbf{x}\|}{\|\delta\mathbf{b}\|\,\|\mathbf{x}\|} = \|A^{-1}\|\|A\|.$$

Sensitivity to Perturbation of Matrix

Case 2. $A\mathbf{x} = \mathbf{b} \;\rightarrow\; (A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b}$

- Bound $\|\delta\mathbf{x}\|$ now in terms of $\|\delta A\|$:

$$\begin{aligned} A\mathbf{x} + A\delta\mathbf{x} + (\delta A)\mathbf{x} + (\delta A)\delta\mathbf{x} &= \mathbf{b} \\ A\delta\mathbf{x} &= -(\delta A)\mathbf{x} - (\delta A)\delta\mathbf{x} \\ \delta\mathbf{x} &= -A^{-1}(\delta A)\mathbf{x} - A^{-1}(\delta A)\delta\mathbf{x} \end{aligned} \implies \|\delta\mathbf{x}\| \lesssim \|A^{-1}\|\|\delta A\|\,\|\mathbf{x}\|\,.$$

(first-order truncation)

- Sensitivity in terms of relative errors:

$$\frac{\dfrac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|}}{\dfrac{\|\delta A\|}{\|A\|}} = \frac{\|\delta\mathbf{x}\|\,\|A\|}{\|\delta A\|\,\|\mathbf{x}\|} \lesssim \frac{\|A^{-1}\|\|\delta A\|\,\|\mathbf{x}\| \cdot \|A\|}{\|\delta A\|\,\|\mathbf{x}\|} = \|A^{-1}\|\|A\|.$$

Matrix Condition Number

- Motivated by the previous estimations, we define the **matrix condition number** by
$$\kappa(A) = \|A^{-1}\|\|A\|,$$
where the norms can be any p-norm or the Frobenius norm.
- A subscript on κ such as 1, 2, ∞, or F(robenius) is used if clarification is needed.

Matrix Condition Number (Cont')

- We can write

$$\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leqslant \kappa(A)\frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|}, \qquad \frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leqslant \kappa(A)\frac{\|\delta A\|}{\|A\|},$$

  where the second inequality is true only in the limit of infinitesimal perturbations δA.

- The matrix condition number $\kappa(A)$ is equal to the condition number of solving a linear system of equation $A\mathbf{x} = \mathbf{b}$.
- The exponent of $\kappa(A)$ in scientific notation determines the approximate number of digits of accuracy that will be lost in calculation of $\mathbf{x}$.
- Since $1 = \|I\| = \|A^{-1}A\| \leqslant \|A^{-1}\|\|A\| = \kappa(A)$, a condition number of 1 is the best we can hope for.
- If $\kappa(A) > \boxed{\text{eps}}^{-1}$, then for computational purposes the matrix is singular.

Condition Numbers in MATLAB

- Use `cond` to calculate various condition numbers:

```
cond(A)             % the 2-norm; or  cond(A, 2)
cond(A, 1)          % the 1-norm
cond(A, Inf)        % the infinity-norm
cond(A, 'fro')      % the Frobenius norm
```

- A condition number estimator (in 1-norm)

```
condest(A)          % faster than cond
```

- The fastest method to estimate the condition number is to use `linsolve` function as below:

```
[x, inv_condest] = linsolve(A, b);
fast_condest = 1/inv_condest;
```

Lec 17: Overdetermined Linear Systems – Introduction

Opening Example: Polynomial Approximation

Introduction

Problem: Fitting Functions to Data

Given data points $\{(x_i, y_i) \mid i \in \mathbb{N}[1, m]\}$, pick a form for the “fitting” function $f(x)$ and minimize its total error in representing the data.

- With real-world data, interpolation is often not the best method.
- Instead of finding functions lying exactly on given data points, we look for ones which are “close” to them.
- In the most general terms, the fitting function takes the form

$$f(x) = c_1 f_1(x) + \cdots + c_n f_n(x),$$

where $f_1, \dots, f_n$ are known functions while $c_1, \dots, c_n$ are to be determined.

Linear Least Squares Approximation

In this discussion:

- use a polynomial fitting function $p(x) = c_1 + c_2 x + \cdots + c_n x^{n-1}$ with $n < m$;
- minimize the 2-norm of the error $r_i = y_i - p(x_i)$:

$$\|\mathbf{r}\|_2 = \sqrt{\sum_{i=1}^{m} r_i^2} = \sqrt{\sum_{i=1}^{m} \big(y_i - p(x_i)\big)^2}.$$

Since the fitting function is linear in unknown coefficients and the 2-norm is minimized, this method of approximation is called the **linear least squares (LLS) approximation**.

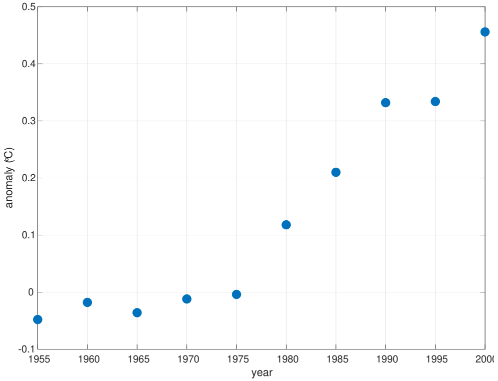
Example: Temperature Anomaly

Below are 5-year averages of the worldwide temperature anomaly as compared to the 1951-1980 average (source: NASA).

Year	Anomaly ($^\circ C$)
1955	-0.0480
1960	-0.0180
1965	-0.0360
1970	-0.0120
1975	-0.0040
1980	0.1180
1985	0.2100
1990	0.3320
1995	0.3340
2000	0.4560

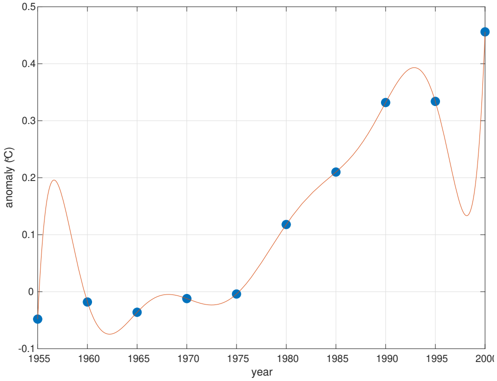
Example: Import and Plot Data

```
t = (1955:5:2000)';
y = [-0.0480; -0.0180;
     -0.0360; -0.0120;
     -0.0040;  0.1180;
      0.2100;  0.3320;
      0.3340;  0.4560];
plot(t, y, '.')
```



Example: Interpolation

```
t = (t-1950)/10;
n = length(t);
V = t.^(0:n-1);
c = V\y;
p = @(x) polyval(flip(c),
      (x-1950)/10);
hold on
fplot(p, [1955 2000])
```



Fitting by a Straight Line

Suppose that we are fitting data to a linear polynomial: $p(x) = c_1 + c_2 x$.

- If it were to pass through all data points:

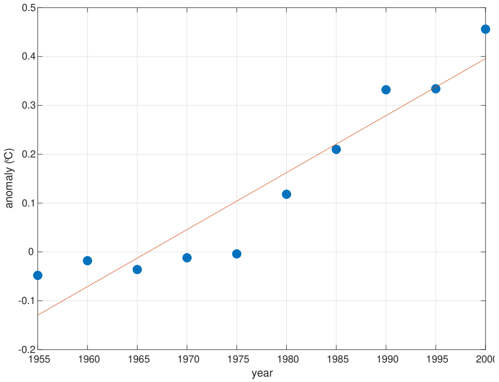
$$\left\{\begin{aligned} y_1 &= p(x_1) = c_1 + c_2 x_1 \\ y_2 &= p(x_2) = c_1 + c_2 x_2 \\ &\vdots \qquad\quad \vdots \qquad \vdots \\ y_m &= p(x_m) = c_1 + c_2 x_m \end{aligned}\right. \xrightarrow{\text{matrix equation}} \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}}_{\mathbf{y}} = \underbrace{\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} c_1 \\ c_2 \end{bmatrix}}_{\mathbf{c}}$$

- The above is unsolvable; instead, find $\mathbf{c}$ which makes the *residual* $\mathbf{r} = \mathbf{y} - V\mathbf{c}$ "as small as possible" in the sense of vector 2-norm.
- **Notation:** $\mathbf{y}$ "=" $V\mathbf{c}$

MATLAB Implementation

Revisiting the temperature anomaly example again:

```
year = (1955:5:2000)';
t = year - 1955;
V = t.^(0:1);
c = V\y;
p = @(x) polyval(flip(c),
      x-1955);
plot(year, y, '.')
hold on
fplot(p, [1955, 2000])
```



Fitting by a General Polynomial

In general, when fitting data to a polynomial

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

we need to solve

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}}_{\mathbf{y}} \text{“="} \underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^{n-1} \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}}_{\mathbf{c}} .$$

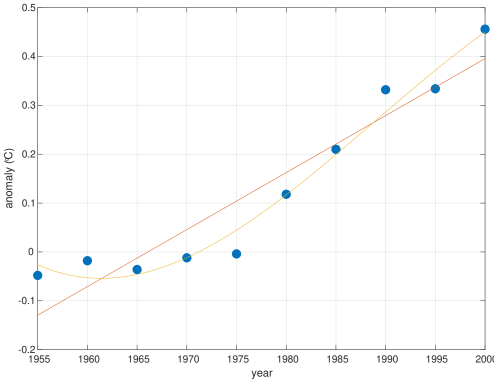
- The solution $\mathbf{c}$ of $\mathbf{y}$ “=” $V\mathbf{c}$ turns out to be the solution of the *normal equation*

$$V^{\mathrm{T}} V \mathbf{c} = V^{\mathrm{T}} \mathbf{y} .$$

MATLAB Implementation

Revisiting the temperature anomaly example again:

```
V = t.^(0:3);
c = V\y;
q = @(x) polyval(flip(c),
    x-1955);
hold on
fplot(q, [1955 2000])
```



Backslash Again

The Versatile Backslash

In MATLAB, the generic linear equation $A\mathbf{x} = \mathbf{b}$ is solved by `x = A\b`.

- When A is a square matrix, Gaussian elimination is used.
- When A is NOT a square matrix, the normal equation $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$ is solved instead.

- As long as $A \in \mathbb{R}^{m\times n}$ where $m \geqslant n$ has rank n, the square matrix $A^{\mathrm{T}}A$ is nonsingular. (unique solution)
- Though $A^{\mathrm{T}}A$ is a square matrix, MATLAB does not use Gaussian elimination to solve the normal equation.
- Rather, a faster and more accurate algorithm is used.

The Normal Equations

LLS and Normal Equation

Big Question: How is the least square solution $\mathbf{x}$ to $A\mathbf{x}$ “=” $\mathbf{b}$ equivalent to the solution of the normal equation $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$?

Theorem (Normal Equation)

Let $A \in \mathbb{R}^{m\times n}$ with $m \geqslant n$. If $\mathbf{x} \in \mathbb{R}^n$ satisfies $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$, then $\mathbf{x}$ solves the LLS problems, *i.e.*, $\mathbf{x}$ minimizes $\|\mathbf{b} - A\mathbf{x}\|_2$.

Proof of the Theorem

Idea of Proof[5]**.** Enough show to that $\|\mathbf{b} - A(\mathbf{x} + \mathbf{y})\|_2 \geqslant \|\mathbf{b} - A\mathbf{x}\|_2$ for any $\mathbf{y} \in \mathbb{R}^n$.

- Useful algebra:

$$(\mathbf{u} + \mathbf{v})^{\mathrm{T}}(\mathbf{u} + \mathbf{v}) = \mathbf{u}^{\mathrm{T}}\mathbf{u} + \mathbf{u}^{\mathrm{T}}\mathbf{v} + \mathbf{v}^{\mathrm{T}}\mathbf{v} + \mathbf{v}^{\mathrm{T}}\mathbf{v} = \mathbf{u}^{\mathrm{T}}\mathbf{u} + 2\mathbf{v}^{\mathrm{T}}\mathbf{u} + \mathbf{v}^{\mathrm{T}}\mathbf{v}.$$

- **Exercise:** Prove it.

[5] Alternately, one can derive the normal equation using calculus. See Appendix.

Appendix: Derivation of Normal Equation

Derivation of Normal Equation

Consider $A\mathbf{x}$ "=" $\mathbf{b}$ where $A \in \mathbb{R}^{m\times n}$ where $m \geqslant n$.

- **Requirement:** minimize the 2-norm of the residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}$:

$$g(x_1, x_2, \ldots, x_n) := \|\mathbf{r}\|_2^2 = \sum_{i=1}^{m}\left(b_i - \sum_{j=1}^{n} a_{ij}x_j\right)^2.$$

- **Strategy:** using calculus, find the minimum by setting

$$\mathbf{0} = \nabla g(x_1, x_2, \ldots, x_n)$$

which yields n equations in n unknowns $x_1, x_2, \ldots, x_n$.

Derivation of Normal Equation (cont')

Noting that $\partial x_j / \partial x_k = \delta_{j,k}$, the n equations $\partial g / \partial x_k = 0$ are written out as

$$0 = \sum_{i=1}^{m} 2\Big(b_i - \sum_{j=1}^{n} a_{ij} x_j\Big)(-a_{ik}), \qquad \text{for } k \in \mathbb{N}[1,n]\,,$$

which can be rearranged into

$$\sum_{i=1}^{m} a_{ik} b_i = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} a_{ik} x_j, \qquad \text{for } k \in \mathbb{N}[1,n]\,.$$

One can see that the two sides correspond to the k^{th} elements of $A^{\mathrm{T}}\mathbf{b}$ and $A^{\mathrm{T}}A\mathbf{x}$ respectively:

$$A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}\,,$$

showing the desired equivalence.

Lec 18: Overdetermined Linear Systems

– QR Factorization

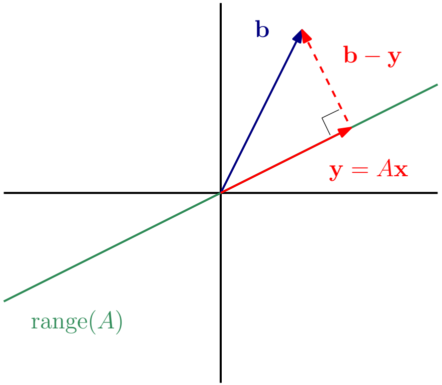
Preliminary: Orthogonality

Normal Equation Revisited

Alternate perspective on the “normal equation”:

$$A^{\mathrm{T}}(\mathbf{b} - A\mathbf{x}) = \mathbf{0} \quad \Longleftrightarrow \quad \mathbf{z}^{\mathrm{T}}(\underbrace{\mathbf{b} - A\mathbf{x}}_{\text{residual } = \mathbf{r}}) = \mathbf{0} \quad \text{for all } \mathbf{z} \in \mathcal{R}(A)\,,$$

i.e., $\mathbf{x}$ solves the normal equation if and only if the residual is orthogonal to the range of A.



Orthogonal Vectors

Recall that the angle θ between two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ satisfies

$$\cos(\theta) = \frac{\mathbf{u}^{\mathrm{T}}\mathbf{v}}{\|\mathbf{u}\|_2 \|\mathbf{v}\|_2}.$$

Definition 7

- Two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ are **orthogonal** if $\mathbf{u}^{\mathrm{T}}\mathbf{v} = 0$.
- Vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k \in \mathbb{R}^n$ are **orthogonal** if $\mathbf{q}_i^{\mathrm{T}}\mathbf{q}_j = 0$ for all $i \neq j$.
- Vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k \in \mathbb{R}^n$ are **orthonormal** if $\mathbf{q}_i^{\mathrm{T}}\mathbf{q}_j = \delta_{i,j}$.

Notation. (Kronecker delta function)

$$\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

Matrices with Orthogonal Columns

Let $Q = \left[\, \mathbf{q}_1 \mid \mathbf{q}_2 \mid \cdots \mid \mathbf{q}_k \,\right] \in \mathbb{R}^{n\times k}$. Note that

$$Q^{\mathrm{T}}Q = \left[\begin{array}{c} \mathbf{q}_1^{\mathrm{T}} \\ \hline \mathbf{q}_2^{\mathrm{T}} \\ \hline \vdots \\ \hline \mathbf{q}_k^{\mathrm{T}} \end{array}\right] \left[\begin{array}{c|c|c|c} \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_k \end{array}\right] = \begin{bmatrix} \mathbf{q}_1^{\mathrm{T}}\mathbf{q}_1 & \mathbf{q}_1^{\mathrm{T}}\mathbf{q}_2 & \cdots & \mathbf{q}_1^{\mathrm{T}}\mathbf{q}_k \\ \mathbf{q}_2^{\mathrm{T}}\mathbf{q}_1 & \mathbf{q}_2^{\mathrm{T}}\mathbf{q}_2 & \cdots & \mathbf{q}_2^{\mathrm{T}}\mathbf{q}_k \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{q}_k^{\mathrm{T}}\mathbf{q}_1 & \mathbf{q}_k^{\mathrm{T}}\mathbf{q}_2 & \cdots & \mathbf{q}_k^{\mathrm{T}}\mathbf{q}_k \end{bmatrix}.$$

Therefore,

- $\mathbf{q}_1, \ldots, \mathbf{q}_k$ are orthogonal. $\iff$ $Q^{\mathrm{T}}Q$ is a $k \times k$ diagonal matrix.
- $\mathbf{q}_1, \ldots, \mathbf{q}_k$ are orthonormal. $\iff$ $Q^{\mathrm{T}}Q$ is the $k \times k$ identity matrix.

Matrices with Orthonormal Columns

Theorem 8

Let $Q = \left[\, \mathbf{q}_1 \mid \mathbf{q}_2 \mid \cdots \mid \mathbf{q}_k \,\right] \in \mathbb{R}^{n\times k}$ *and suppose that* $\mathbf{q}_1, \ldots, \mathbf{q}_k$ *are orthonormal. Then*

1. $Q^{\mathrm{T}} Q = I \in \mathbb{R}^{k\times k}$;
2. $\|Q\mathbf{x}\|_2 = \|\mathbf{x}\|_2$ *for all* $\mathbf{x} \in \mathbb{R}^k$;
3. $\|Q\|_2 = 1$.

Orthogonal Matrices

Definition 9

We say that $Q \in \mathbb{R}^{n\times n}$ is an **orthogonal matrix** if $Q^{\mathrm{T}}Q = I \in \mathbb{R}^{n\times n}$.

- A square matrix with orthogonal columns is not, in general, an orthogonal matrix!

Properties of Orthogonal Matrices

Theorem 10

Let $Q \in \mathbb{R}^{n\times n}$ be orthogonal. Then

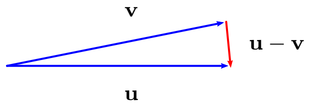
1. $Q^{-1} = Q^{\mathrm{T}}$;
2. Q^{T} *is also an orthogonal matrix;*
3. $\kappa_2(Q) = 1$;
4. *For any* $A \in \mathbb{R}^{n\times n}$, $\|AQ\|_2 = \|A\|_2$;
5. *if* $P \in \mathbb{R}^{n\times n}$ *is another orthogonal matrix, then* PQ *is also orthogonal.*

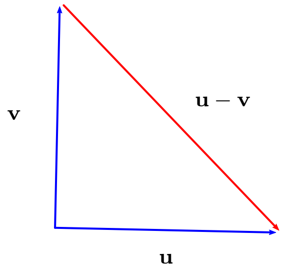
Why Do We Like Orthogonal Vectors?

- If $\mathbf{u}$ and $\mathbf{v}$ are orthogonal, then

$$\|\mathbf{u} \pm \mathbf{v}\|_2^2 =$$

- Without orthogonality, it is possible that $\|\mathbf{u} - \mathbf{v}\|_2$ is much smaller than $\|\mathbf{u}\|_2$ and $\|\mathbf{v}\|_2$.
- The addition and subtraction of orthogonal vectors are guaranteed to be well-conditioned.





QR Factorization

The QR Factorization

The following matrix factorization plays a role in solving linear least squares problems similar to that of LU factorization in solving linear systems.

Theorem 11

Let $A \in \mathbb{R}^{m\times n}$ where $m \geqslant n$. Then $A = QR$ where $Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R \in \mathbb{R}^{m\times n}$ is upper triangular.

$$\underbrace{\left[\begin{array}{c|c|c|c} & & & \\ \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \\ & & & \end{array}\right]}_{A} = \underbrace{\left[\begin{array}{c|c|c|c} & & & \\ \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_m \\ & & & \end{array}\right]}_{Q} \underbrace{\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}}_{R}$$

Thick VS Thin QR Factorization

- Here is the QR factorization again.

$$A = \underbrace{\begin{bmatrix} \Big| & \Big| & & \Big| \\ \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_m \\ \Big| & \Big| & & \Big| \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}}_{R} \qquad \text{(thick)}$$

- When m is much larger than n, it is much more efficient to use the *thin* or *compressed* QR factorization.

$$A = \underbrace{\begin{bmatrix} \Big| & \Big| & & \Big| \\ \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_n \\ \Big| & \Big| & & \Big| \end{bmatrix}}_{\hat{Q}} \underbrace{\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{bmatrix}}_{\hat{R}} \qquad \text{(thin)}$$

QR Factorization in MATLAB

Either type of QR factorization is computed by `qr` command.

- Thick/Full QR factorization

```
[Q, R] = qr(A)
```

- Thin/Compressed QR factorization

```
[Q, R] = qr(A, 0)
```

Test the orthogonality of `Q` by calculating the norm of $Q^{\mathrm{T}}Q - I$ where I is the identity matrix with *suitable* dimensions.

```
norm(Q'*Q - eye(m))        % full QR
norm(Q'*Q - eye(n))        % thin QR
```

Least Squares and QR Factorization

Substitute the thin factorization $A = \widehat{Q}\widehat{R}$ into the normal equation $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$ and simplify.

Least Squares and QR Factorization (cont')

Summary: Algorithm for LLS Approximation

If A has rank n, the normal equation $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$ is consistent and is equivalent to $\widehat{R}\mathbf{x} = \widehat{Q}^{\mathrm{T}}\mathbf{b}$.

1. Factor $A = \widehat{Q}\widehat{R}$.
2. Let $\mathbf{z} = \widehat{Q}^{\mathrm{T}}\mathbf{b}$.
3. Solve $\widehat{R}\mathbf{x} = \mathbf{z}$ for $\mathbf{x}$ using backward substitution.

Least Squares and QR Factorization (cont')

```
function x = lsqrfact(A,b)
% LSQRFACT x = lsqrfact(A,b)
% Sove linear least squares by QR factorization
% Input:
%   A    coefficient matrix (m-by-n, m>n)
%   b    right-hand side (m-by-1)
% Output:
%   x    minimizer of || b - Ax || (2-norm)
    [Q,R] = qr(A,0);              % thin QR fact.
    z = Q'*b;
    x = backsub(R,c);
end
```

Appendix: Gram-Schmidt Orthogonalization

The Gram–Schmidt Procedure

Problem: Orthogonalization

Given $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{R}^m$, construct orthonormal vectors $\mathbf{q}_1, \ldots, \mathbf{q}_n \in \mathbb{R}^m$ such that

$$\operatorname{span}\{\mathbf{a}_1, \ldots, \mathbf{a}_k\} = \operatorname{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_k\}, \quad \text{for any } k \in \mathbb{N}[1, n].$$

- **Strategy.** At the jth step, find a unit vector $\mathbf{q}_j \in \operatorname{span}\{\mathbf{a}_1, \ldots, \mathbf{a}_j\}$ that is orthogonal to $\mathbf{q}_1, \ldots, \mathbf{q}_{j-1}$.
- **Key Observation.** The vector $\mathbf{v}_j$ defined by

$$\mathbf{v}_j = \mathbf{a}_j - \left(\mathbf{q}_1^{\mathrm{T}}\mathbf{a}_j\right)\mathbf{q}_1 - \left(\mathbf{q}_2^{\mathrm{T}}\mathbf{a}_j\right)\mathbf{q}_2 - \cdots - \left(\mathbf{q}_{j-1}^{\mathrm{T}}\mathbf{a}_j\right)\mathbf{q}_{j-1}$$

  satisfies the required properties.

GS Algorithm

The Gram–Schmidt iteration is outlined below:

$$
\begin{aligned}
\mathbf{q}_1 &= \frac{\mathbf{a}_1}{r_{11}}, \\
\mathbf{q}_2 &= \frac{\mathbf{a}_2 - r_{12}\mathbf{q}_1}{r_{22}}, \\
\mathbf{q}_3 &= \frac{\mathbf{a}_3 - r_{13}\mathbf{q}_1 - r_{23}\mathbf{q}_2}{r_{33}}, \\
&\ \vdots \\
\mathbf{q}_n &= \frac{\mathbf{a}_n - \sum_{i=1}^{n-1} r_{in}\mathbf{q}_i}{r_{nn}},
\end{aligned}
$$

where

$$
r_{ij} = \begin{cases} \mathbf{q}_i^{\mathrm{T}}\mathbf{a}_j, & \text{if } i \neq j \\ \pm\left\| \mathbf{a}_j - \sum_{k=1}^{j-1} r_{kj}\mathbf{q}_k \right\|_2, & \text{if } i = j \end{cases}.
$$

GS Procedure and Thin QR Factorization

- The GS algorithm, written as a matrix equation, yields a **thin QR factorization**:

$$A = \underbrace{\left[\begin{array}{c|c|c} \\ \mathbf{a}_1 & \cdots & \mathbf{a}_n \\ \\ \end{array}\right]}_{A} = \underbrace{\left[\begin{array}{c|c|c} \\ \mathbf{q}_1 & \cdots & \mathbf{q}_n \\ \\ \end{array}\right]}_{\hat{Q}} \underbrace{\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{bmatrix}}_{\hat{R}} = \widehat{Q}\widehat{R}$$

- While it is an important tool for theoretical work, the GS algorithm is numerically unstable.

Lec 19: Overdetermined Linear Systems

– QR Algorithm

Revisiting Least Squares

Moore-Penrose Pseudoinverse

Let $A \in \mathbb{R}^{m\times n}$ with $m \geqslant n$ and suppose that columns of A are linearly independent.

- The least square problem $A\mathbf{x}$ "=" $\mathbf{b}$ is equivalent to the normal equation $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$, which is a square matrix equation.
- The solution can be written as

$$\mathbf{x} = \left(A^{\mathrm{T}}A\right)^{-1} A^{\mathrm{T}}\mathbf{b}.$$

- The matrix

$$A^{+} = \left(A^{\mathrm{T}}A\right)^{-1} A^{\mathrm{T}} \in \mathbb{R}^{n\times m},$$

  is called the **(Moore-Penrose) pseudoinverse**.
- MATLAB's backslash is mathematically equivalent to left-multiplication by the inverse or pseudoinverse of a matrix.
- MATLAB's `pinv` calculates the pseudoinverse, but it is rarely used in practice, just as `inv`.

Moore-Penrose Pseudoinverse (cont')

- A^+ can be calculated by using the thin QR factorization[6] $A = \widehat{Q}\widehat{R}$.

$$A^+ = \widehat{R}^{-1}\widehat{Q}^{\mathrm{T}}.$$

[6]It can be done using the thick QR factorization as seen on p.1624 of the text.

Least Squares and QR Factorization

Substitute the thin factorization $A = \widehat{Q}\widehat{R}$ into the normal equation $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$ and simplify.

Least Squares and QR Factorization (cont')

Summary: Algorithm for LLS Approximation

If A has rank n, the normal equation $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$ is consistent and is equivalent to $\widehat{R}\mathbf{x} = \widehat{Q}^{\mathrm{T}}\mathbf{b}$.

1. Factor $A = \widehat{Q}\widehat{R}$.
2. Let $\mathbf{z} = \widehat{Q}^{\mathrm{T}}\mathbf{b}$.
3. Solve $\widehat{R}\mathbf{x} = \mathbf{z}$ for $\mathbf{x}$ using backward substitution.

Least Squares and QR Factorization (cont')

```
function x = lsqrfact(A,b)
% LSQRFACT x = lsqrfact(A,b)
% Sove linear least squares by QR factorization
% Input:
%   A    coefficient matrix (m-by-n, m>n)
%   b    right-hand side (m-by-1)
% Output:
%   x    minimizer of || b - Ax || (2-norm)
    [Q,R] = qr(A,0);             % thin QR fact.
    z = Q'*b;
    x = backsub(R,c);
end
```

Householder Transformation and QR Algorithm

Motivation

Problem

Given $\mathbf{z} \in \mathbb{R}^m$, find an orthogonal matrix $H \in \mathbb{R}^{m\times m}$ such that $H\mathbf{z}$ is nonzero only in the first element.

- Since orthogonal matrices preserve the 2-norm, H must satisfy

$$H\mathbf{z} = \begin{bmatrix} \pm\|\mathbf{z}\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \pm\|\mathbf{z}\|_2\,\mathbf{e}_1.$$

- The **Householder transformation matrix** H defined by

$$H = I - 2\frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}, \quad \text{where } \mathbf{v} = \pm\|\mathbf{z}\|_2\,\mathbf{e}_1 - \mathbf{z},$$

solves the problem. See Theorem 1.

Properties of Householder Transformation

Theorem 12

Let $\mathbf{v} = \|\mathbf{z}\|_2 \, \mathbf{e}_1 - \mathbf{z}$ *and let* H *be the Householder transformation defined by*

$$H = I - 2\frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}.$$

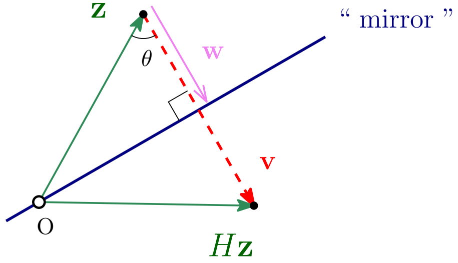
Then

1. H *is symmetric;*
2. H *is orthogonal;*
3. $H\mathbf{z} = \|\mathbf{z}\|_2 \, \mathbf{e}_1$.

- H is invariant under scaling of $\mathbf{v}$.
- If $\|\mathbf{v}\|_2 = 1$, then $H = I - \mathbf{v}\mathbf{v}^{\mathrm{T}}$.

Geometry Behind Householder Transformation (cont')

The Householder transformation matrix H can be thought of as a *reflector*[7].



[7] See Supplementary 1 on for review on projection and reflection operators

Factorization Algorithm

- The Gram-Schmidt orthogonalization (thin QR factorization) is unstable in floating-point calculations.
- **Stable alternative:** Find orthogonal matrices $H_1, H_2, \dots, H_n$ so that

$$\underbrace{H_n H_{n-1} \cdots H_2 H_1}_{=:Q^{\mathrm{T}}} A = R\,.$$

  introducing zeros one column at a time below diagonal terms.
- As a product of orthogonal matrices, Q^{T} is also orthogonal and so $(Q^{\mathrm{T}})^{-1} = Q$. Therefore,

$$A = QR\,.$$

MATLAB Demonstration Code MYQR

```
function [Q, R] = myqr(A)
  [m, n] = size(A);
  A0 = A;
  Q = eye(m);
  for j = 1:min(m,n)
      Aj = A(j:m, j:n);
      z = Aj(:, 1);
      v = z + sign0(z(1))*norm(z)*eye(length(z), 1);
      Hj = eye(length(v)) - 2/(v'*v) * v*v';
      Aj = Hj*Aj;
      H = eye(m);
      H(j:m, j:m) = Hj;
      Q = Q*H;
      A(j:m, j:n) = Aj;
  end
  R = A;
end
```

MATLAB Demonstration Code MYQR (cont')

(continued from the previous page)

```
% local function
function sign0(x)
  y = ones(size(x));
  y(x < 0) = -1;
end
```

- The MATLAB command `qr` works similar to, but more efficiently than, this.
- The function finds the factorization in $\sim (2mn^2 - n^3/3)$ flops asymptotically.

Supplementary 1: Projection and Reflection

Projection and Reflection Operators

Let $\mathbf{u}, \mathbf{v} \in \mathbb{R}^m$ be nonzero vectors.

- Projection of $\mathbf{u}$ onto $\langle \mathbf{v} \rangle = \text{span}(\mathbf{v})$:

$$\frac{\mathbf{v}^{\mathrm{T}}\mathbf{u}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\mathbf{v} = \underbrace{\left(\frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\right)}_{=:P}\mathbf{u} =: P\mathbf{u}.$$

- Projection of $\mathbf{u}$ onto $\langle \mathbf{v} \rangle^{\perp}$, the orthogonal complement of $\langle \mathbf{v} \rangle$:

$$\mathbf{u} - \frac{\mathbf{v}^{\mathrm{T}}\mathbf{u}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\mathbf{v} = \left(I - \frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\right)\mathbf{u} =: (I - P)\mathbf{u}.$$

- Reflection of $\mathbf{u}$ across $\langle \mathbf{v} \rangle^{\perp}$:

$$\mathbf{u} - 2\frac{\mathbf{v}^{\mathrm{T}}\mathbf{u}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\mathbf{v} = \left(I - 2\frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\right)\mathbf{u} =: (I - 2P)\mathbf{u}.$$

Projection and Reflection Operators (cont')

Summary: for given $\mathbf{v} \in \mathbb{R}^m$, a nonzero vector, let

$$P = \frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}} \in \mathbb{R}^{m \times m}.$$

Then the following matrices carry out geometric transformations

- Projection onto $\langle \mathbf{v} \rangle$: P
- Projection onto $\langle \mathbf{v} \rangle$: $I - P$
- Reflection across $\langle \mathbf{v} \rangle^{\perp}$: $I - 2P$

Note. If $\mathbf{v}$ were a unit vector, the definition of P simplifies to $P = \mathbf{v}\mathbf{v}^{\mathrm{T}}$.

Supplementary 2: Conditioning and Stability

Analytical Properties of Pseudoinverse

The matrix $A^{\mathrm{T}}A$ appearing in the definition of A^{+} satisfies the following properties.

Theorem 13

For any $A \in \mathbb{R}^{m\times n}$ with $m \geqslant n$, the following are true:

1. *$A^{\mathrm{T}}A$ is symmetric.*
2. *$A^{\mathrm{T}}A$ is singular if and only if rank$(A) < n$.*
3. *If $A^{\mathrm{T}}A$ is nonsingular, then it is positive definite.*

A symmetric positive definite (SPD) matrix S such as $A^{\mathrm{T}}A$ permits so-called the **Cholesky factorization**

$$S = R^{\mathrm{T}}R$$

where R is an upper triangular matrix.

Least Squares Using Normal Equation

One can solve the LLS problem $A\mathbf{x}$ “=” $\mathbf{b}$ by solving the normal equation $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$ directly as below.

1. Compute $N = A^{\mathrm{T}}A$.
2. Compute $\mathbf{z} = A^{\mathrm{T}}\mathbf{b}$.
3. Solve the square linear system $N\mathbf{x} = \mathbf{z}$ for $\mathbf{x}$.

Step 3 is done using `chol` which implements the Cholesky factorization.

MATLAB Implementarion.

```
N = A'*A;
z = A'*b;
R = chol(N);
w = forelim(R',z);     % solve R'w = z
x = backsub(R,w);      % solve R x = w
```

Conditioning of Normal Equations

- Recall that the condition number of solving a square linear system $A\mathbf{x} = \mathbf{b}$ is $\kappa(A) = \|A\| \left\|A^{-1}\right\|$.
- Provided that the residual norm at the least square solution is relatively small, the conditioning of LLS problem is similar:

$$\kappa(A) = \|A\| \left\|A^{+}\right\| .$$

- If A is rank-deficient (columns are linearly dependent), then $\kappa(A) = \infty$.
- If an LLS problem is solved solving the normal equation, it can be shown that the condition number is

$$\kappa(A^{\mathrm{T}}A) = \kappa(A)^2.$$

Which Reflector Is Better?

- Recall:
$$H = I - 2\frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}, \quad \text{where } \mathbf{v} = \pm \|\mathbf{z}\|_2 \, \mathbf{e}_1 - \mathbf{z},$$
- In `myqr.m`, the statement

```
v = z + sign0(z(1))*norm(z)*eye(length(z), 1);
```

defines $\mathbf{v}$ slightly differently[8], namely,

$$\mathbf{v} = \mathbf{z} \pm \|\mathbf{z}\|_2 \, \mathbf{e}_1.$$

[8]This does not cause any difference since H is invariant under scaling of $\mathbf{v}$; see p.379

Which Reflector Is Better? (cont')

The sign of $\pm\|\mathbf{z}\|_2$ is chosen so as to avoid possible catastrophic cancellation in forming $\mathbf{v}$:

$$\mathbf{v} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix} + \begin{bmatrix} \pm\|\mathbf{z}\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} z_1 \pm \|\mathbf{z}\|_2 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}$$

Subtractive cancellation may arise when $z_1 \approx \pm\|\mathbf{z}\|_2$.

- if $z_1 > 0$, use $z_1 + \|\mathbf{z}\|_2$;
- if $z_1 < 0$, use $z_1 - \|\mathbf{z}\|_2$;
- if $z_1 = 0$, either works.

For numerical stability, it is desirable to reflect $\mathbf{z}$ to the vector $s\|\mathbf{z}\|_2\,\mathbf{e}_1$ that is not too close to $\mathbf{z}$ itself. (Trefethen & Bau)

Lec 20: Review of Topic 2

Preliminaries

Square Linear Systems

Overdetermined Linear Systems

Key Problems

Lec 21: Rootfinding Problem – Introduction

The Rootfinding Problem

Problem Statement

Rootfinding Problem

Given a continuous scalar function of a scalar variable, find a real number r such that $f(r) = 0$.

- r is a **root** of the function f.
- The formulation $f(x) = 0$ is general enough; *e.g.*, to solve $g(x) = h(x)$, set $f = g - h$ and find a root of f.

Iterative Methods

- Unlike the earlier linear problems, the root cannot be produced in a finite number of operations.
- Rather, a sequence of approximations that formally converge to the root is pursued.

Iteration Strategy for Rootfinding. To find the root of f:

1. Start with an initial iterate, say x_0.
2. Generate a sequence of iterates $x_1, x_2, \ldots$ using an *iteration algorithm* of the form
$$x_{k+1} = g(x_k), \quad k = 0, 1, \ldots$$
3. Continue the iteration process until you find an x_i such that $f(x_i) = 0$. (In practice, continue until some member of the sequence seems to be “good enough”.)

MATLAB's FZERO

`fzero` is MATLAB's general purpose rootfinding tool.

Syntax:

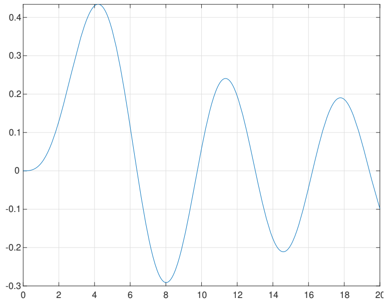
```
x_zero = fzero( <function>, <initial iterate> )
x_zero = fzero( <function>, <initial interval> )
[x_zero, fx_zero] = ....
```

Example

The roots of J_m, a Bessel function of the first kind, is found by

- Plot the function.
- Find approximate locations of roots.

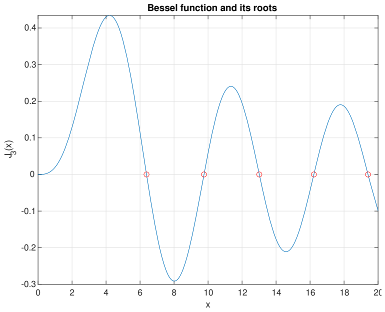
```
J3 = @(x) besselj(3,x);
fplot(J3,[0 20])
grid on
guess = [6,10,13,16,19];
```



Example (cont')

- Then use `fzero` to locate the roots:

```
omega = zeros(size(guess));
for j = 1:length(guess)
  omega(j) = fzero(J3,guess(j));
end
hold on
plot(omega,J3(omega),'ro')
```



Conditioning

- Sensitivity of the rootfinding problem can be measured in terms of the condition number:

$$\text{(absolute condition number)} = \frac{|\text{abs. error in output}|}{|\text{abs. error in input}|},$$

  where, in the context of finding roots of f,

  - input: f (function)
  - output: r (root)

- Denote the changes by:
  - error/change in input: ϵg, where $\epsilon > 0$ is small $(f \mapsto f + \epsilon g)$
  - error/change in output: Δr $(r \mapsto r + \Delta r)$

Conditioning (cont')

- The *perturbed equation*

$$f(r+\Delta r) + \epsilon g(r+\Delta r) = 0$$

  is linearized to (Taylor expansion)

$$f(r) + f'(r)\Delta r + g(r)\epsilon + g'(r)\epsilon\Delta r \approx 0,$$

  ignoring $O((\Delta r)^2)$ terms[9].

- Since $f(r) = 0$, we solve for Δr to get

$$\Delta r \approx -\epsilon \frac{g(r)}{f'(r) + \epsilon g'(r)} \approx -\epsilon \frac{g(r)}{f'(r)},$$

  for small ϵ compared with $f'(r)$.

[9] That is, terms involving $(\Delta r)^2$ and higher powers of Δr

Conditioning (cont')

- Therefore, the absolute condition number of the rootfinding problem is

$$\kappa_{f \mapsto r} = \frac{1}{\left|f'(r)\right|},$$

  which implies that the problem is highly sensitive whenever $f'(r) \approx 0$.

- In other words, if $|f'|$ is small at the root, a computed *root estimate* may involve large errors.

Residual and Backward Error

- Without knowing the exact root, we cannot compute the error.
- But the **residual** of a root estimate $\tilde{r}$ can be computed:

$$\text{(residual)} = f(\tilde{r}).$$

- Small residual *might* be associated with a small error.
- The residual $f(\tilde{r})$ is the *backward error* of the estimate.

Multiple Roots

Definition 14 (Multiplicity of Roots)

Assume that r is a root of the differentiable function f. Then if

$$0 = f(r) = f'(r) = \cdots = f^{(m-1)}(r) \quad \text{but} \quad f^{(m)}(r) \neq 0,$$

we say that f has a root of **multiplicity** m at r.

- We say that f has a **multiple root** at r if the multiplicity is greater than 1.
- A root is called **simple** if its multiplicity is 1.
- If r is a multiple root, the condition number is infinite.
- Even if r is a simple root, we expect difficulty in numerical computation if $f'(r) \approx 0$.

Lec 22: Rootfinding Problem – One Dimension

Fixed Point Iteration

Fixed Point

Definition 15 (Fixed Point)

The real number r is a **fixed point** of the function g if $g(r) = r$.

- The rootfinding problem $f(x) = 0$ can always be written as a fixed point problem $g(x) = x$ by, *e.g.*, setting[10]

$$g(x) = x - f(x).$$

- The fixed point problem is true at, and only at, a root of f.

[10] This is not the only way to transform the rootfinding problem. More on this later.

Fixed Point Iteration

A fixed point problem $g(x) = x$ naturally provides an iteration scheme:

$$\begin{cases} x_0 = \text{initial guess} \\ x_{k+1} = g(x_k), \quad k = 0, 1, 2, \ldots . \end{cases} \qquad \text{(fixed point iteration)}$$

- The sequence $\{x_k\}$ may or may not converge as $k \to \infty$.
- If g is continuous and $\{x_k\}$ converges to a number r, then r is a fixed point of g.

$$g(r) = g\left(\lim_{k\to\infty} x_k\right) = \lim_{k\to\infty} g(x_k) = \lim_{k\to\infty} x_{k+1} = r.$$

Fixed Point Iteration Algorithm

```
function x = fpi(g, x0, n)
% FPI x = fpi(g, x0, n)
% Computes approximate solution of g(x)=x
% Input:
%   g    function handle
%   x0   initial guess
%   n    number of iteration steps
    x = x0;
    for k = 1:n
        x = g(x);
    end
end
```

Examples

- To find a fixed point of $g(x) = 0.3\cos(2x)$ near 0.5 using `fpi`:

```
g = @(x) 0.3*cos(2*x);
xc = fpi(g,0.5,20)
```

```
xc = 0.260266319627758
```

Not All Fixed Point Problems Are The Same

The rootfinding problem $f(x) = x^3 + x - 1 = 0$ can be transformed to various fixed point problems:

- $g_1(x) = x - f(x) = 1 - x^3$
- $g_2(x) = \sqrt[3]{1-x}$
- $g_3(x) = \dfrac{1+2x^3}{1+3x^2}$

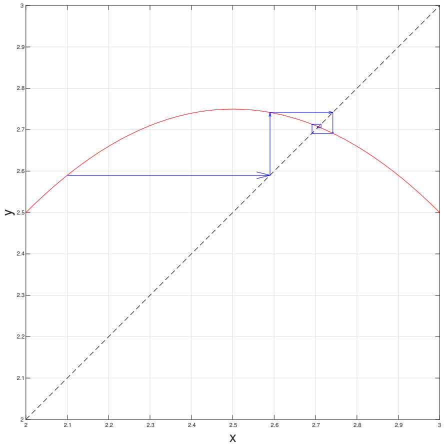
Note that all $g_j(x) = x$ are equivalent to $f(x) = 0$. However, not all these find a fixed point of g, that is, a root of f on the computer.

Exercise. Run `fpi` with g_j and $x_0 = 0.5$. Which fixed point iterations converge?

Geometry of Fixed Point Iteration

The following script[11] finds a root of $f(x) = x^2 - 4x + 3.5$ via FPI.

```
f = @(x) x.^2 - 4*x + 3.5;
g = @(x) x - f(x);
fplot(g, [2 3], 'r');
hold on
plot([2 3], [2 3], 'k--')
x = 2.1;
y = g(x);
for k = 1:5
    arrow([x y], [y y], 'b');
    x = y; y = g(x);
    arrow([x x], [x y], 'b');
end
```



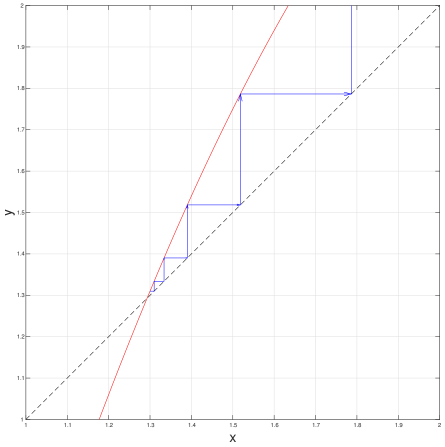
Note the line segments spiral in towards the fixed point.

[11] Modified from FNC.

Geometry of Fixed Point Iteration (cont')

However, with a different starting point, the process does not converge.

```
clf
fplot(g, [1 2], 'r');
hold on
plot([1 2], [1 2], 'k--'),
ylim([1 2])
x = 1.3; y = g(x);
for k = 1:5
    arrow([x y], [y y], 'b');
    x = y; y = g(x);
    arrow([x x], [x y], 'b');
end
```



Custom function: `arrow = @(p1, p2, varargin) quiver(p1(1), p1(2), p2(1)-p1(1), p2(2)-p1(2), 0, varargin{:})`

Series Analysis

Let $\epsilon_k = x_k - r$ be the sequence of errors.

- The iteration formula $x_{k+1} = g(x_k)$ can be written as

$$\begin{aligned} \epsilon_{k+1} + r &= g(\epsilon_k + r) \\ &= g(r) + g'(r)\epsilon_k + \frac{1}{2}g''(r)\epsilon_k^2 + \cdots, \qquad \text{(Taylor series)} \end{aligned}$$

  implying

$$\epsilon_{k+1} = g'(r)\epsilon_k + O(\epsilon_k^2)$$

  assuming sufficient regularity of g.

- Neglecting the second-order term, we have $\epsilon_{k+1} \approx g'(r)\epsilon_k$, which is satisfied if $\epsilon_k \approx C\left[g'(r)\right]^k$ for sufficiently large k.
- Therefore, the iteration converges if $\left|g'(r)\right| < 1$ and diverges if $\left|g'(r)\right| > 1$.

Note: Rate of Convergence

Definition 16 (Linear Convergnece)

Suppose $\lim_{k\to\infty} x_k = r$ and let $\epsilon_k = x_k - r$, the error at step k of an iteration method. If

$$\lim_{k\to\infty} \frac{|\epsilon_{k+1}|}{|\epsilon_k|} = \sigma < 1,$$

the method is said to obey **linear convergence** with rate σ.

Note. In general, say

$$\lim_{k\to\infty} \frac{|\epsilon_{k+1}|}{|\epsilon_k|^p} = \sigma$$

for some $p \geqslant 1$ and $\sigma > 0$.

- If $p = 1$ and
  - $\sigma = 1$, the convergence is *sublinear*;
  - $0 < \sigma < 1$, the convergence is *linear*;
  - $\sigma = 0$, the convergence is *superlinear*.
- If $p = 2$, the convergence is *quadratic*;
- If $p = 3$, the convergence is *cubic*, …

Convergence of Fixed Point Iteration

Theorem 17 (Convergence of FPI)

Assume that g is continuously differentiable, that $g(r) = r$, and that $\sigma = \left|g'(r)\right| < 1$. Then the fixed point iterates x_k generated by

$$x_{k+1} = g(x_k), \quad k = 1, 2, \dots,$$

converge linearly with rate σ to the fixed point r for x_0 sufficiently close to r.

In the previous example with $g(x) = x - f(x) = -x^2 + 5x - 3.5$:

- For the first fixed point, near 2.71, we get $g'(r) \approx -0.42$ (convergence);
- For the second fixed point, near 1.29, we get $g'(r) \approx 2.42$ (divergence).

Note. An iterative method is called **locally convergent** to r if the method converges to r for initial guess sufficiently close to r.

Contraction Maps

Lipschitz Condition

A function g is said to satisfy a **Lipschitz condition** with constant L on the interval $S \subset \mathbb{R}$ if

$$\left|g(s) - g(t)\right| \leqslant L\,|s - t| \quad \text{for all } s, t \in S.$$

- A function satisfying the Lipschitz condition is continuous on S.
- If $L < 1$, g is called a **contraction map**.

When Does FPI Succeed?

Contraction Mapping Theorem

Suppose that g satisfies Lipschitz condition on S with $L < 1$, *i.e.*, g is a contraction map on S. Then S contains exactly one fixed point r of g. If $x_1, x_2, \ldots$ are generated by the fixed point iteration $x_{k+1} = g(x_k)$, and $x_1, x_2, \ldots$ all lie in S, then

$$\left|x_k - r\right| \leqslant L^{k-1} \left|x_1 - r\right|, \quad k > 1.$$

Newton’s Method

Newton's Method

To find the root of f:

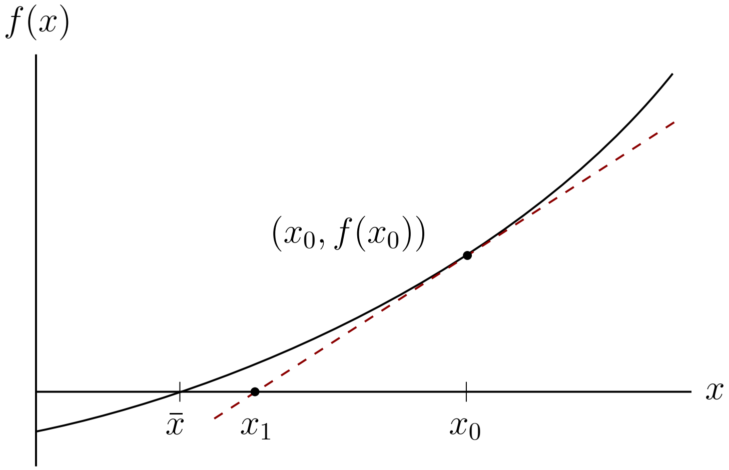
Newton's Method (Algorithm)

- Begin at the point $(x_0, f(x_0))$ on the curve and draw the tangent line at the point using the slope $f'(x_0)$:
$$y = f(x_0) + f'(x_0)(x - x_0).$$
- Find the x-intercept of the line and call it x_1:
$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$
- Continue this procedure to find $x_2, x_3, \ldots$ until the sequence converges to the root.

General iterative formula:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad \text{for } k = 0, 1, 2, \ldots \tag{$\star$}$$

Newton's Method: Illustration



Series Analysis

Let $\epsilon_k = x_k - r$, $k = 1, 2, \ldots$, where r is the limit of the sequence and $f(r) = 0$.

Substituting $x_k = r + \epsilon_k$ into the iterative formula ($\star$):

$$\epsilon_{k+1} = \epsilon_k - \frac{f(r+\epsilon_k)}{f'(r+\epsilon_k)}.$$

Taylor-expand f about $x = r$ and simplify (assuming $f'(r) \neq 0$):

$$\begin{aligned}
\epsilon_{k+1} &= \epsilon_k - \frac{f(r) + \epsilon_k f'(r) + \frac{1}{2}\epsilon_k^2 f''(r) + O(\epsilon_k^3)}{f'(r) + \epsilon_k f''(r) + O(\epsilon_k^2)} \\
&= \epsilon_k - \epsilon_k \left[1 + \frac{1}{2}\frac{f''(r)}{f'(r)}\epsilon_k + O(\epsilon_k^2)\right]\left[1 + \frac{f''(r)}{f'(r)}\epsilon_k + O(\epsilon_k^2)\right]^{-1} \\
&= \frac{1}{2}\frac{f''(r)}{f'(r)}\epsilon_k^2 + O(\epsilon_k^3).
\end{aligned}$$

Series Analysis (cont')

- Previous calculation shows that $\epsilon_{k+1} \approx C\epsilon_k^2$, eventually. Written differently,

$$\left|\epsilon_{k+1}\right| / \left|\epsilon_k\right|^2 \to \text{(some positive number)}, \text{ as } k \to \infty.$$

  that is, each Newton iteration roughly squares the previous error. This is **quadratic convergence**[12].

- Alternately, note that

$$\log\left|\epsilon_{k+1}\right| \approx 2\log\left|\epsilon_k\right| + \text{(constant)},$$

  ignoring high-order terms. This means that the number of accurate digits[13] approximately doubles at each iteration.

[12] Recall the formal definition given in p. 416.
[13] We say that an iterate is **correct within** p **decimal places** if the error is less than 0.5×10^{-p}.

Convergence of Newton's Method

Theorem 18 (Quadratic Convergence of Newton's Method)

Let f be twice continuously differentiable and $f(r) = 0$. If $f'(r) \neq 0$, then Newton's method is locally and quadratically convergent to r. The error $\epsilon_k = x_k - r$ at step k satisfies

$$\lim_{k\to\infty} \frac{|\epsilon_{k+1}|}{|\epsilon_k|^2} = \left|\frac{f''(r)}{2f'(r)}\right|.$$

Implementation

```
function x = newton(f,dfdx,x1)
% NEWTON   Newton's method for a scalar equation.
% Input:
%   f         objective function
%   dfdx      derivative function
%   x1        initial root approximation
% Output
%   x         vector of root approximations (last one is best)

% Operating parameters.
    funtol = 100*eps;  xtol = 100*eps;  maxiter = 40;

    x = x1;
    y = f(x1);
    dx = Inf;   % for initial pass below
    k = 1;

    while (abs(dx) > xtol) && (abs(y) > funtol) && (k < maxiter)
        dydx = dfdx(x(k));
        dx = -y/dydx;           % Newton step
        x(k+1) = x(k) + dx;

        k = k+1;
        y = f(x(k));
    end

    if k==maxiter, warning('Maximum number of iterations reached.'), end
end
```

Note: Stopping Criteria

For a set tolerance, `TOL`, some example stopping criteria are:

- Absolute error:
$$\left|x_{k+1} - x_k\right| < \mathtt{TOL}.$$
- Relative error: (useful when the solution is not too close to zero)
$$\frac{\left|x_{k+1} - x_k\right|}{\left|x_{k+1}\right|} < \mathtt{TOL}.$$
- Hybrid:
$$\frac{\left|x_{k+1} - x_k\right|}{\max(\left|x_{k+1}\right|, \theta)} < \mathtt{TOL},$$
  for some $\theta > 0$.
- Residual:
$$\left|f(x_k)\right| < \mathtt{TOL}.$$

Also useful to set a limit on the maximum number of iterations in case convergence fails.

Secant Method

Secant Method

- Newton’s method requires calculation and evaluation of $f'(x)$, which may be challenging at times.
- The most common alternative to such situations is the **secant method**.
- The secant method replaces the instantaneous slope in Newton’s method by the average slope using the last two iterates.

Secant Method (cont')

Secant Method (Algorithm)

- Begin with two initial iterates x_{-1} and x_0; draw the secant line connecting $(x_{-1}, f(x_{-1}))$ and $(x_0, f(x_0))$:

$$y = f(x_0) + \frac{f(x_0) - f(x_{-1})}{x_0 - x_{-1}}(x - x_0).$$

- Find the x-intercept of the line and call it x_1:

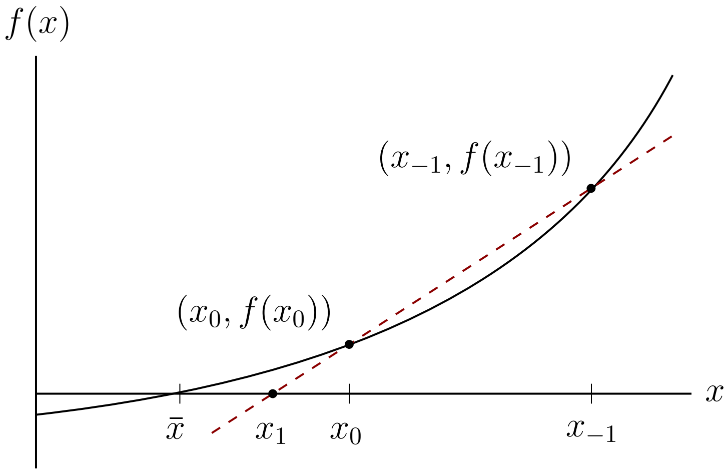
$$x_1 = x_0 - f(x_0)\frac{x_0 - x_{-1}}{f(x_0) - f(x_{-1})}.$$

- Continue this procedure to find $x_2, x_3, \ldots$ until convergence is obtained.

General iterative formula:

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \quad \text{for } k = 0, 1, 2, \ldots$$

Secant Method: Illustration



Series Analysis

Assume that the secant method converges to r and $f'(r) \neq 0$. Let $\epsilon_k = x_k - r$ as before.

It can be shown that

$$\left|\epsilon_{k+1}\right| \approx \left|\frac{f''(r)}{2f'(r)}\right| \left|\epsilon_k\right| \left|\epsilon_{k-1}\right|,$$

which implies that

$$\left|\epsilon_{k+1}\right| \approx \left|\frac{f''(r)}{2f'(r)}\right|^{\alpha-1} \left|\epsilon_k\right|^{\alpha},$$

where

$$\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618,$$

the *golden ratio*.

Therefore, the convergence of the secant method is **superlinear**; it lies between linearly and quadratically convergent methods.

Series Analysis (cont')

Exercise. Confirm the statements in the previous page. Namely, show that

1. The error ϵ_k satisfies the approximate equation

$$|\epsilon_{k+1}| \approx \left|\frac{f''(r)}{2f'(r)}\right| |\epsilon_k| \, |\epsilon_{k-1}| \, .$$

2. If in addition $\lim_{k\to\infty} |\epsilon_{k+1}| / |\epsilon_k|^\alpha$ exists and is nonzero for some $\alpha > 0$, then

$$|\epsilon_{k+1}| \approx \left|\frac{f''(r)}{2f'(r)}\right|^{\alpha-1} |\epsilon_k|^\alpha \, , \quad \text{where } \alpha = \frac{1+\sqrt{5}}{2}.$$

Implementation

```
function x = secant(f,x1,x2)
% SECANT   Secant method for a scalar equation.
% Input:
%   f        objective function
%   x1,x2    initial root approximations
% Output
%   x        vector of root approximations (last is best)

% Operating parameters.
    funtol = 100*eps;  xtol = 100*eps;  maxiter = 40;

    x = [x1 x2];
    dx = Inf;  y1 = f(x1);
    k = 2;  y2 = 100;

    while (abs(dx) > xtol) && (abs(y2) > funtol) && (k < maxiter)
        y2 = f(x(k));
        dx = -y2 * (x(k)-x(k-1)) / (y2-y1);   % secant step
        x(k+1) = x(k) + dx;

        k = k+1;
        y1 = y2;    % current f-value becomes the old one next time
    end

    if k==maxiter, warning('Maximum number of iterations reached.'), end
end
```

Other Methods

Inverse Interpolation

The **inverse quadratic interpolation** (IQI) is a generalization of the secant method to parabolas.

- Instead of using two most recent points (to determine a straight line), use three and obtain an quadratic interpolant.
- The parabola of the form $y = p(x)$ may have zero, one, or two x-intercept(s). So use the form $x = p(y)$, a parabola open sideways.

Algorithm.

- Begin with three initial iterates x_{-2}, x_{-1}, x_0; find the parabola of the form $x = p(y)$ passing through the three points $(x_{-2}, f(x_{-2}))$, $(x_{-1}, f(x_{-1}))$, and $(x_0, f(x_0))$.
- Find the x-intercept of the parabola and call it x_1.
- Continue the procedure to find $x_2, x_3, \ldots$ until convergence is obtained.

Inverse Interpolation (cont')

General iterative formula:

$$x_{k+1} = x_k - \frac{r(r-q)(x_k - x_{k-1}) + (1-r)s(x_k - x_{k-2})}{(q-1)(r-1)(s-1)}, \quad \text{for } k = 0, 1, 2, \ldots,$$

where

$$q = \frac{f(x_{k-2})}{f(x_{k-1})}, \quad r = \frac{f(x_k)}{f(x_{k-1})}, \quad s = \frac{f(x_k)}{f(x_{k-2})}.$$

Rather than deriving and implementing the formula, try using `polyfit` to perform the interpolation step.

Bisection Method: Bracketing a Root

The following is a corollary to the intermediate value theorem.

Theorem 19 (Existence of a Root)

Let f be a continuous function on $[a, b]$, satisfying $f(a)f(b) < 0$. Then f has a root between a and b, that is, there exists a number $r \in (a, b)$ such that $f(r) = 0$.

Bisection Method (cont')

Algorithm.

- Start with an interval $[a, b]$ where $f(a)f(b) \leqslant 0$.
- Bisect the interval into $[a, m] \cup [m, b]$ where $m = (a + b)/2$ is the midpoint.
- Select the subinterval in which $f(x)$ changes signs, *i.e.*, calculate $f(a)f(m)$ and $f(m)f(b)$, choose the nonpositive one, and update the values of a and b.
- Repeat the process until you get close enough to the solution.

Notes

Let $[a, b]$ be the initial interval and let $[a_k, b_k]$ be the interval after k bisection steps.

- The length of $[a_k, b_k]$ is $(b-a)/2^k$.
- Using the midpoint $x_k = (a_k + b_k)/2$ as an estimate of the root r, note that

$$|\epsilon_k| = |x_k - r| < \frac{b-a}{2^{k+1}}.$$

- This accuracy is obtained by $k+2$ function evaluations.

Bisection Method: Pseudocode

```
while <a NOT CLOSE ENOUGH TO b>
  m = (a + b)/2;
  fm = f(m);
  if sign(fa) ~= sign(fm)
    b = m;
    fb = fm;
  else
    a = m;
    fa = fm;
  end
end
x_zero = .5*(a + b);
```

Lec 23: Rootfinding Problem – Higher Dimensions

Newton’s Method for Nonlinear Systems

Multidimensional Rootfinding Problem

Rootfinding Problem: Vector Version

Given a continuous vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, find a vector $\mathbf{r} \in \mathbb{R}^n$ such that $\mathbf{f}(\mathbf{r}) = \mathbf{0}$.

The rootfinding problem $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ is equivalent to solving the *nonlinear* system of n scalar equations in n unknowns:

$$\begin{aligned} f_1(x_1, \dots, x_n) &= 0, \\ f_2(x_1, \dots, x_n) &= 0, \\ &\vdots \\ f_n(x_1, \dots, x_n) &= 0. \end{aligned}$$

Multidimensional Taylor Series

If $\mathbf{f}$ is differentiable, we can write

$$\mathbf{f}(\mathbf{x}+\mathbf{h}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\mathbf{h} + O(\|\mathbf{h}\|^2),$$

where $\mathbf{J}$ is the **Jacobian matrix** of $\mathbf{f}$

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} = \left[\frac{\partial f_i}{\partial x_j}\right]_{i,j=1,\ldots,n}.$$

- The first two terms $\mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\mathbf{h}$ is the “linear approximation” of $\mathbf{f}$ near $\mathbf{x}$.
- If $\mathbf{f}$ is actually linear, *i.e.*, $\mathbf{f}(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$, then the Jacobian matrix is the coefficient matrix A and the rootfinding problem $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ is simply $A\mathbf{x} = \mathbf{b}$.

Example

Let

$$\begin{aligned} f_1(x_1, x_2, x_3) &= -x_1 \cos(x_2) - 1, \\ f_2(x_1, x_2, x_3) &= x_1 x_2 + x_3, \\ f_3(x_1, x_2, x_3) &= e^{-x_3} \sin(x_1 + x_2) + x_1^2 - x_2^2. \end{aligned}$$

Then

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} -\cos(x_2) & x_1 \sin(x_2) & 0 \\ x_2 & x_1 & 1 \\ e^{-x_3} \cos(x_1 + x_2) + 2x_1 & e^{-x_3} \cos(x_1 + x_2) - 2x_2 & -e^{-x_3} \sin(x_1 + x_2) \end{bmatrix}.$$

Exercise. Write out the linear part of the Taylor expansion of

$$f_1(x_1 + h_1, x_2 + h_2, x_3 + h_3), \quad \text{near } (x_1, x_2, x_3).$$

The Multidimensional Newton's Method

Recall the idea of Newton's method:

> *If finding a zero of a function is difficult, replace the function with a simpler approximation (linear) whose zeros are easier to find.*

Applying the principle:

- Linearize $\mathbf{f}$ at the kth iterate $\mathbf{x}_k$:

$$\mathbf{f}(\mathbf{x}) \approx L(\mathbf{x}) = \mathbf{f}(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k).$$

- Define the next iterate $\mathbf{x}_{k+1}$ by solving $L(\mathbf{x}_{k+1}) = \mathbf{0}$:

$$\mathbf{0} = \mathbf{f}(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) \quad \Longrightarrow \quad \mathbf{x}_{k+1} = \mathbf{x}_k - \left[\mathbf{J}(\mathbf{x}_k)\right]^{-1} \mathbf{f}(\mathbf{x}_k).$$

  Note that $\mathbf{J}^{-1}\mathbf{f}$ plays the same role as f/f' in the scalar Newton.

The Multidimensional Newton's Method (cont')

- In practice, we do not compute $\mathbf{J}^{-1}$. Rather, the kth Newton step $\mathbf{s}_k = x_{k+1} - x_k$ is found by solving the square linear system

$$\mathbf{J}(\mathbf{x}_k)\mathbf{s}_k = -\mathbf{f}(\mathbf{x}_k),$$

  which is solved using the backslash in MATLAB.

- Suppose `f` and `J` are MATLAB functions calculating $\mathbf{f}$ and $\mathbf{J}$, respectively. Then the Newton iteration is done simply by

```
% x is a Newton iterate (a column vector).
% The following is the key fragment
% inside Newton iteration loop.
fx = f(x)
s = -J(x)\fx;
x = x + s;
```

- Since $\mathbf{f}(x_k)$ is the residual and $\mathbf{s}_k$ is the gap between two consecutive iterates at the kth step, monitor their norms to determine when to stop iteration.

Computer Illustration

1. Define $\mathbf{f}$ and $\mathbf{J}$, either as anonymous functions or as function m-files.

```
f = @(x) [exp(x(2)-x(1)) - 2;
          x(1)*x(2) + x(3);
          x(2)*x(3) + x(1)^2 - x(2)];
J = @(x) [-exp(x(2)-x(1)),exp(x(2)-x(1)), 0;
          x(2), x(1), 1;
          2*x(1), x(3)-1, x(2)];
```

1. Define an initial iterate x, say $\mathbf{x}_0 = (0, 0, 0)^{\mathrm{T}}$.

```
x = [0 0 0]';
```

1. Iterate.

```
for k = 1:7
    s = -J(x)\f(x);
    x = x + s;
end
```

Implementation

```
function x = newtonsys(f,x1)
% NEWTONSYS   Newton's method for a system of equations.
% Input:
%   f         function that computes residual and Jacobian matrix
%   x1        initial root approximation (n-vector)
% Output
%   x         array of approximations (one per column, last is best)

% Operating parameters.
    funtol = 1000*eps;  xtol = 1000*eps;  maxiter = 40;

    x = x1(:);
    [y,J] = f(x1);
    dx = Inf;
    k = 1;

    while (norm(dx) > xtol) && (norm(y) > funtol) && (k < maxiter)
        dx = -(J\y);   % Newton step
        x(:,k+1) = x(:,k) + dx;

        k = k+1;
        [y,J] = f(x(:,k));
    end

    if k==maxiter, warning('Maximum number of iterations reached.'), end
end
```

Lec 24: Rootfinding Problem – Closing Notes

Revisiting `FZERO`

How Does `FZERO` Work?

Quasi-Newton Methods

Jacobian by Finite Differences

```
function J = fdjac(f,x0,y0)
% FDJAC   Finite-difference approximation of a Jacobian.
% Input:
%   f         function to be differentiated
%   x0        evaluation point (n-vector)
%   y0        value of f at x0 (m-vector)
% Output
%   J         approximate Jacobian (m-by-n)

    delta = sqrt(eps);    % FD step size
    m = length(y0);   n = length(x0);
    J = zeros(m,n);
    I = eye(n);
    for j = 1:n
        J(:,j) = ( f(x0+delta*I(:,j)) - y0) / delta;
    end

end
```

Broyden’s Update

Levenberg’s Method

Basin of Attraction

Nonlinear Least Squares (NLS)

Gauss-Newton Method

Convergence

Nonlinear Data Fitting

Lec 25: Piecewise Interpolation – Piecewise Linear

Interpolation Problem

Problem Statement

General Interpolation Problem

Given a set of n data points $\{(x_j, y_j) \mid j \in \mathbb{N}[1, n]\}$ with $x_1 < x_2 < \ldots < x_n$, find a function $p(x)$, called the **interpolant**, such that

$$p(x_j) = y_j, \quad \text{for } j = 1, 2, \ldots, n\,.$$

The ordered pair (x_j, y_j) is called the **data point**.

- x_j is called the **abscissa** or the **node**.
- y_j is called the **ordinate**.

Polynomials

One approach is to find an interpolating *polynomial* of degree (at most) $n-1$,

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

- The unknown coefficients $c_1, \ldots, c_n$ are determined by solving the square linear system $V\mathbf{c} = \mathbf{y}$ where

$$V = \begin{bmatrix} 1 & x_1 & \cdots & x_1^{n-2} & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-2} & x_2^{n-1} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & x_n & \cdots & x_n^{n-2} & x_n^{n-1} \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}, \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

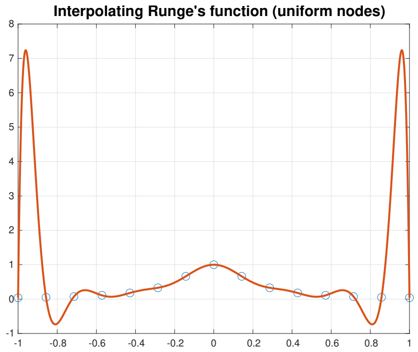
  the matrix V is called the **Vandermonde matrix**; see Lecture 13.

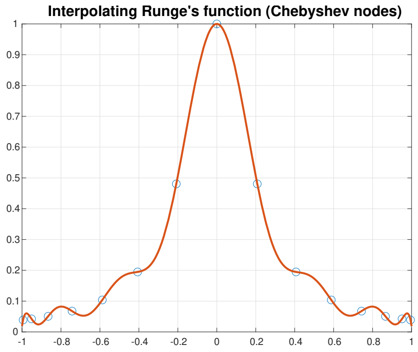
- A polynomial interpolant has severe oscillations as n grows large, unless nodes are special; see illustration in the next slide.

Illustration Runge's Phenomenon

Polynomial Interpolation of 15 data points collected from the same function

$$f(x) = \frac{1}{1+25x^2}.$$





Piecewise Polynomials

To handle real-life data sets with large n and unrestricted node distribution:

- An alternate approach is to use a low-degree polynomial between each pair of data points; it is called the **piecewise polynomial interpolation**.
- The simplest case is **piecewise linear interpolation** (degree 1) in which the interpolant is linear between each pair of consecutive nodes.
- The most commonly used method is **cubic spline interplation** (degree 3).
- The endpoints of the low-degree polynomials are called **breakpoints** or **knots**.
- The breakpoints and the data points almost always coincide.

MATLAB Function: `INTERP1`

In MATLAB, piecewise polynomials are constructed using `interp1` function. Suppose the x and y data are stored in vectors `xdp` and `ydp`. To evaluate the piecewise interpolant at `x` (an array):

- By default, it finds a piecewise linear interpolant.

```
y = interp1(xdp, ydp, x);
```

- To obtain a smoother interpolant that is piecewise cubic, use `'spline'` option.

```
y = interp1(xdp, ydp, x, 'spline');
```

Demonstration: Piecewise Polynomial Interplation

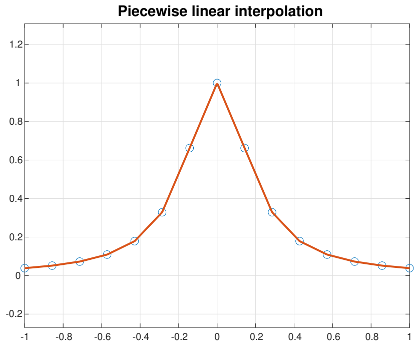
To interpolate data obtained from [14]

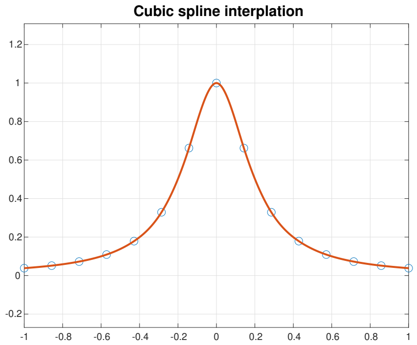
$$f(x) = \frac{1}{1+25x^2}.$$

```
% Generate data and eval pts
n = 15;
xdp = linspace(-1,1,n)';
ydp = 1./(1+25*xdp.^2);
x = linspace(-1,1,400)';

% PL
plot(xdp,ydp,'o'), hold on
plot(x, interp1(xdp,ydp,x))

% Cubic spline
plot(xdp,ydp,'o'), hold on
plot(x, interp1(xdp,ydp,x,'spline'));
```

[14] This function is often called the Runge's function.

Demonstration: Piecewise Polynomial Interplation (cont’)





Conditioning

Set-up for analysis.

- Let (x_j, y_j) for $j = 1, \ldots, n$ denote the data points. Assume that the nodes x_j are fixed and let $a = x_1$, $b = x_n$.
- View the interpolation problem as a mathematical function $\mathcal{I}$ with
  - Input: a vector $\mathbf{y}$ (ordinates, or y-data points)
  - Output: a function $p(x)$ such that $p(x_j) = y_j$ for all j.

  (That is, $\mathcal{I}$ is a *black box* that produces the interpolant from a data vector.)
- For the interpolation methods under consideration (polynomial or piecewise polynomial), $\mathcal{I}$ is *linear*:

$$\mathcal{I}(\alpha\mathbf{y} + \beta\mathbf{z}) = \alpha\mathcal{I}(\mathbf{y}) + \beta\mathcal{I}(\mathbf{z}),$$

  for all vectors $\mathbf{y}, \mathbf{z}$ and scalars α, β.

Conditioning: Main Theorem

Theorem 20 (Conditioning of General Interpolation)

Suppose that $\mathcal{I}$ is a linear interpolation method. Then the absolute condition number of $\mathcal{I}$ satisfies

$$\max_{1 \leqslant j \leqslant n} \left\| \mathcal{I}(\mathbf{e}_j) \right\|_\infty \leqslant \kappa(\mathbf{y}) \leqslant \sum_{j=1}^{n} \left\| \mathcal{I}(\mathbf{e}_j) \right\|_\infty ,$$

where all vectors and functions are measured in the infinity norm.

Conditioning: Notes

- The functional infinity norm is defined by

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

  in a manner similar to vector infinity norm.

- The expression $\mathcal{I}(\mathbf{e}_j)$ represents the interpolant $p(x)$ which is *on* at x_j and *off* elsewhere, *i.e.*,

$$p(x_k) = \delta_{k,j} = \begin{cases} 1, & k = j \\ 0, & k \neq j \end{cases}.$$

  Such interpolants are known as **cardinal functions**.

- The theorem says that the (absolute) condition number is larger than the largest of $\left\|\mathcal{I}(\mathbf{e}_j)\right\|_\infty$, but smaller than the sum of these.

Piecewise Linear Interpolation

Piecewise Linear Interpolation

Assume that $x_1 < x_2 < \cdots < x_n$ are fixed. The function $p(x)$ defined piecewise[15] by

$$p(x) = y_j + \frac{y_{j+1} - y_j}{x_{j+1} - x_j}(x - x_j)\,, \quad \text{for } x \in [x_j, x_{j+1}],\ 1 \leqslant j \leqslant n-1$$

- is linear on each interval $[x_j, x_{j+1}]$;
- connects any two consecutive data points (x_j, y_j) and (x_{j+1}, y_{j+1}) by a straight line.

[15] Note the formula changes depending on which interval x lies in.

Hat Functions

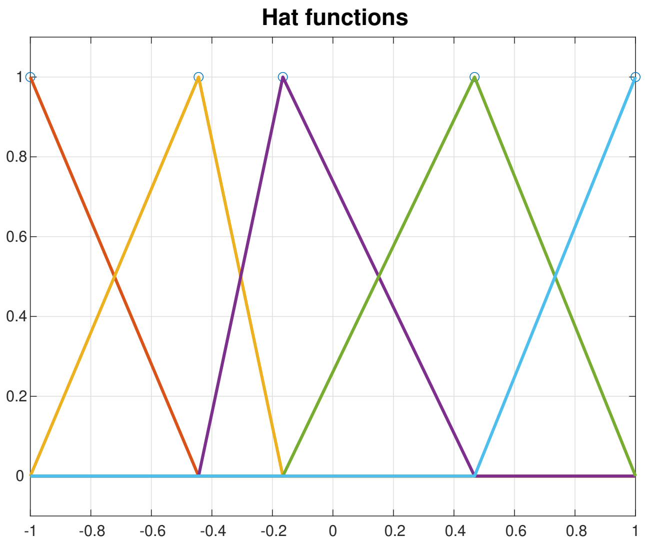
Denote by $H_j(x)$ the jth *piecewise linear* cardinal function:

$$H_j(x) = \begin{cases} \dfrac{x - x_{j-1}}{x_j - x_{j-1}}, & x \in [x_{j-1}, x_j], \\ \dfrac{x_{j+1} - x}{x_{j+1} - x_j}, & x \in [x_j, x_{j+1}], \\ 0, & \text{otherwise}, \end{cases} \qquad j = 1, 2, \ldots, n.$$

- The functions $H_1, \ldots, H_n$ are called **hat functions** or **tent functions.**
- Each H_j is globally continuous and is linear inside each interval $[x_j, x_{j+1}]$

Note: The definitions of $H_1(x)$ and $H_n(x)$ require additional nodes x_0 and x_{n+1} for x outside of $[x_1, x_n]$, which is not relevant in the discussion of interpolation.

Hat Functions (cont')



Hat Functions As Basis

- Any linear combination of hat functions is continuous and is linear inside each interval $[x_j, x_{j+1}]$.
- Conversely, *any* such function is expressible as a unique linear combination of hat functions, *i.e.*,

$$\sum_{j=1}^{n} c_j H_j(x), \quad \text{for some choice of } c_1, \ldots, c_n.$$

- No smaller set of functions has the same properties.

The hat functions form a **basis** of the set of functions that are continuous and piecewise linear relative to $\mathbf{x}$ (the vector of nodes).

Cardinality Conditions

- By construction, the hat functions are cardinal functions for piecewise linear (PL) interpolation, *i.e.*, they satisfy

$$H_j(x_k) = \delta_{j,k}. \qquad \text{(cardinality condition)}$$

- Key consequence of this property is that the piecewise linear interpolant $p(x)$ for the data values in $\mathbf{y}$ is trivially expressed by

$$p(x) = \sum_{j=1}^{n} y_j H_j(x).$$

Recipe for PL Interpolant

Piecewise Linear Interpolant

The piecewise linear polynomial

$$p(x) = \sum_{j=1}^{n} y_j H_j(x)$$

is the unique such function which passes through all the data points.

Proof: It is easy to check the interpolating property:

$$p(x_k) = \sum_{j=1}^{n} y_j H_j(x_k) = \sum_{j=1}^{n} y_j \delta_{j,k} = y_k \quad \text{for every } k \in \mathbb{N}[1,n].$$

To show uniqueness, suppose $\tilde{p}$ is another such function in the form

$$\tilde{p}(x) = \sum_{j=1}^{n} c_j H_j(x)\,.$$

Then $p(x_k) - \tilde{p}(x_k) = 0$ for all $k \in \mathbb{N}[1,n]$. This implies that $c_k = y_k$ for all k $\square$

Conditioning

Lemma

Let $\mathcal{I}$ is the piecewise linear interpolation operator and $\mathbf{z} \in \mathbb{R}^n$. Then

$$\left\|\mathcal{I}(\mathbf{z})\right\|_\infty = \|\mathbf{z}\|_\infty .$$

- It follows from the lemma that <u>the absolute condition number of piecewise linear interpolation in the infinity norm equals one</u>.

Conditioning (cont')

Proof of lemma. Let

$$p(x) = \mathcal{I}(\mathbf{z}) = \sum_{j=1}^{n} z_j H_j(x).$$

Let k be the index corresponding to the element of $\mathbf{z}$ with the largest absolute value, that is, $z_k = \|\mathbf{z}\|_\infty$. Since $z_k = p(x_k)$, it follows that $\left|p(x_k)\right| = \|\mathbf{z}\|_\infty$ and so $\|p\|_\infty \geqslant \|\mathbf{z}\|_\infty$.

To show the other inequality, note that

$$\left|p(x)\right| = \left|\sum_{j=1}^{n} z_j H_j(x)\right| \leqslant \sum_{j=1}^{n} |z_j|\, H_j(x) \leqslant \|\mathbf{z}\|_\infty \sum_{j=1}^{n} H_j(x) = \|\mathbf{z}\|_\infty\,,$$

where the final step uses the fact[16] that $\sum_{j=1}^{n} H_j(x) = 1$. It implies that $\|p\|_\infty \leqslant \|\mathbf{z}\|_\infty$. Therefore, $\|p\|_\infty = \|\mathbf{z}\|_\infty$. □

[16] This property is called the *partition of unity*. Confirm it!

Convergence: Error Analysis

Set-up for analysis.

- Generate a set of data points using a "nice" function f on an interval containing all nodes, *i.e.*, $y_j = f(x_j)$. (The *niceness* of a function is described in precise terms below.)
- Then perform PL interpolation of the data to obtain the interpolant p.
- **Question.** How close is p to f?

Notation (Space of Differentiable Functions)

Let $C^n[a, b]$ denote the set of all functions that are n-times continuously differentiable on $[a, b]$. That is, if $f \in C^n[a, b]$, then $f^{(n)}$ exists and is continuous on $[a, b]$, where derivatives at the end points are taken to be one-sided derivatives.

Convergence: Error Analysis (cont')

Theorem 21 (Error Theorem for PL Interpolation)

Suppose that $f \in C^2[a,b]$*. Let* p_n *be the piecewise linear interpolant of* $(x_j, f(x_j))$ *for* $j = 1, \ldots, n$*, where*

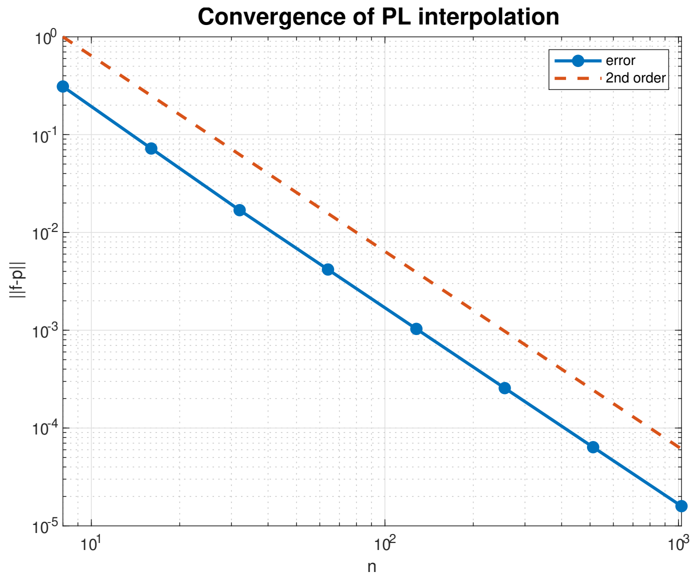
$$x_j = a + (j-1)h \quad \textit{and} \quad h = \frac{b-a}{n-1}.$$

Then

$$\|f - p_n\|_\infty \leqslant \|f''\|_\infty h^2.$$

- The theorem pertains to the interpolation on equispaced nodes.
- The significance of the theorem is that the error in the interpolant is $O(h^2)$ as $h \to 0$. (We say that PL interpolation is *second-order accurate*.)
- **Practical implication:** If n is doubled, the PL interpolant becomes about four times more accurate. A log-log graph (`loglog`) of error against n is a straight line.

Convergence: Error Analysis (cont')



Lec 26: Piecewise Interpolation – Piecewise Cubic

Hermite Cubic Interpolation

Problem Set-Up: General Piecewise Cubic Interpolation

We now seek a piecewise cubic polynomial pp which interpolates the data (x_i, y_i) for $i = 1, \ldots, n$, with $x_1 < x_2 < \cdots < x_n$, defined as

$$pp(x) = \begin{cases} p_1(x), & x \in [x_1, x_2) \\ p_2(x), & x \in [x_2, x_3) \\ \quad\vdots & \quad\vdots \\ p_{n-1}(x), & x \in [x_{n-1}, x_n] \end{cases},$$

where the ith *local* cubic polynomial p_i is written in shifted power form as

$$p_i(x) = c_{i,1} + c_{i,2}(x - x_i) + c_{i,3}(x - x_i)^2 + c_{i,4}(x - x_i)^3.$$

Hermite Cubic Interpolation

If the slopes at the breakpoints are prescribed, *i.e.*, for each $i = 1, \ldots, n-1$,

$$p_i(x_i) = y_i\,, \quad p_i'(x_i) = \sigma_i\,, \quad p_i(x_{i+1}) = y_{i+1}\,, \quad p_i'(x_{i+1}) = \sigma_{i+1}\,,$$

then we can solve for the four unknown coefficients $c_{i,j}$, $j = 1, \ldots, 4$:

$$\begin{aligned} c_{i,1} &= y_i\,, \qquad & c_{i,3} &= \frac{3y[x_i, x_{i+1}] - 2\sigma_i - \sigma_{i+1}}{\Delta x_i}\,, \\ c_{i,2} &= \sigma_i\,, \qquad & c_{i,4} &= \frac{\sigma_i + \sigma_{i+1} - 2y[x_i, x_{i+1}]}{(\Delta x_i)^2}\,. \end{aligned}$$

where $\Delta x_i = x_{i+1} - x_i$ and

$$y[x_i, x_{i+1}] = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}. \qquad \text{(Newton's divided difference)}$$

This is called **Hermite cubic interpolation.**

Implementation

```
function c = hermiteCoeff(x,y,s)
% Input:
%   x,y,s   data points and slopes
% Ouput:
%   c       coefficients in matrix form
    n = length(x);
    c = zeros(n-1, 4);
    dx = diff(x);
    dy = diff(y);
    dydx = dy./dx;
    c(:,1) = y;
    c(:,2) = s;
    c(:,3) = (3*dydx - 2*s(1:n-1) - s(2:n))./dx;
    c(:,4) = (s(1:n-1) + s(2:n-1) - 2*dydx))./(dx.^2);
end
```

Convergence: Error Analysis

Theorem 22 (Error Theorem for Hermite Cubic Interpolation)

Let $f \in C^4[a,b]$ *and let* $p(x)$ *be the Hermite cubic interpolant of*

$$\left(x_i, f(x_i), f'(x_i)\right), \quad \text{for } i = 1, \dots, n,$$

where

$$x_j = a + (j-1)h \quad \text{and} \quad h = \frac{b-a}{n-1}.$$

Then

$$\|f - p\|_\infty \leqslant \frac{1}{384}\|f^{(4)}\|_\infty h^4.$$

Drawbacks of Hermite Cubic Interpolation

- The interpolant $p(x)$ is in C^1 and so its display may bee to crude in graphical applications.
- In other applications, there may be difficulties if $p''(x)$ is discontinuous.
- In experimental settings where y_i are measurements of some sort, we may not have the first derivative information required for the cubic Hermite process.

Cubic Splines

Cubic Splines

Idea: Put together cubic polynomials to make the result as smooth as possible.

- At interior breakpoints: for $j = 2, 3, \cdots, n-1$
  - matching values: $p_{j-1}(x_j) = p_j(x_j)$ [$(n-2)$ eqns]
  - matching first derivatives: $p'_{j-1}(x_j) = p'_j(x_j)$ [$(n-2)$ eqns]
  - matching second derivative: $p''_{j-1}(x_j) = p''_j(x_j)$ [$(n-2)$ eqns]
- So, together with the n interpolating conditions, we have total of $(4n-6)$ equations.
- To match up with the number of unknowns $(4n-4)$, we need to impose two more conditions on the boundary:
  1. slopes at each end (clamped cubic spline)
  2. second derivatives at the endpoints (natural cubic spline)
  3. periodic boundary condition
  4. *not-a-knot* boundary condition: $p_1(x) \equiv p_2(x)$ and $p_{n-2}(x) \equiv p_{n-1}(x)$.

Convergence: Error Analysis

Theorem 23 (Error Theorem for Clamped Cubic Splines)

Let $f \in C^4[a,b]$ and let $\mathfrak{p}(x)$ be the cubic spline interpolant of

$$\left(x_i, f(x_i)\right), \quad \text{for } i = 1, \ldots, n,$$

with the exact boundary conditions

$$\sigma_1 = f'(x_1) \quad \text{and} \quad \sigma_n = f'(x_n),$$

in which

$$x_j = a + (j-1)h \quad \text{and} \quad h = \frac{b-a}{n-1}.$$

Then

$$\|f - \mathfrak{p}\|_\infty \leqslant \frac{5}{384}\|f^{(4)}\|_\infty h^4.$$

Remarks

- Hermite cubic interpolation is about five times as accurate as cubic spline interpolation, yet both have *fourth-order accuracy*.
- Unlike the former, the latter does not require first derivatives.

Lec 27: Piecewise Interpolation – Derivation of Cubic Spline

Derivation of Cubic Spline Algorithm

Cubic Spline: Problem Set-Up

Given $\{(x_i, y_i) \mid i = 1, 2, \ldots, n\}$, find a piecewise polynomial $\mathbb{p}(x) = p_i(x)$ on $[x_i, x_{i+1}]$ with

$$p_i(x) = c_{i,1} + c_{i,2}(x - x_i) + c_{i,3}(x - x_i)^2 + c_{i,4}(x - x_i)^3,$$

satisfying[17]

1. $\mathbb{p}(x_i) = y_i$ for $i = 1, \ldots, n$;
2. $p_i(x_{i+1}) = p_{i+1}(x_{i+1})$ for $i = 1, \ldots, n-2$;
3. $p_i'(x_{i+1}) = p_{i+1}'(x_{i+1})$ for $i = 1, \ldots, n-2$;
4. $p_i''(x_{i+1}) = p_{i+1}''(x_{i+1})$ for $i = 1, \ldots, n-2$.

[17] Let us not worry about the boundary conditions yet.

Connection to Hermite Cubic Interpolation

Key Observation. If $c_{i,j}$'s are set to be

$$\begin{aligned} c_{i,1} &= y_i\,, \qquad & c_{i,3} &= \frac{3y[x_i, x_{i+1}] - 2\sigma_i - \sigma_{i+1}}{\Delta x_i}\,, \\ c_{i,2} &= \sigma_i\,, \qquad & c_{i,4} &= \frac{\sigma_i + \sigma_{i+1} - 2y[x_i, x_{i+1}]}{(\Delta x_i)^2}\,, \end{aligned} \qquad (\star)$$

as in the Hermite cubic interpolation for some constants $\sigma_1, \sigma_2, \ldots, \sigma_n$ to be determined, then $\mathbb{p}(x)$ satisfies the first three requirements from the previous slide.

Reduction. Determine $\sigma_1, \sigma_2, \ldots, \sigma_n$ so that the fourth requirement is satisfied.

Derivation of a Linear System for Cubic Splines (1)

Using ($\star$), write out the fourth requirement $p''_{i-1}(x_i) = p''_i(x_i)$ in terms of $\sigma_{i-1}, \sigma_i, \sigma_{i+1}$, where $i \in \mathbb{N}[2, n-1]$.

Derivation of a Linear System for Cubic Splines (2)

Express the system of $n-2$ equations for $\sigma_1, \dots, \sigma_n$ as a matrix equation $X\boldsymbol{\sigma} = \mathbf{r}$, where $\boldsymbol{\sigma} = (\sigma_1, \dots, \sigma_n)^{\mathrm{T}}$ and $X \in \mathbb{R}^{n \times n}$ and $\mathbf{r} \in \mathbb{R}^n$ are to be found[18].

[18] Since two equations are still missing, leave the first and last rows of X and $\mathbf{r}$ empty for now.

Tridiagonal System for Cubic Splines

Notation. $\Delta x_i = x_{i+1} - x_i$ and $\nabla x_i = \Delta x_{i-1} + \Delta x_i = x_{i+1} - x_{i-1}$.

$$X = \begin{bmatrix} * & * & * & \cdots & * & * & * \\ \Delta x_2 & 2\nabla x_2 & \Delta x_1 & & & & \\ & \Delta x_3 & 2\nabla x_3 & \Delta x_2 & & & \mathbf{0} \\ & & \ddots & \ddots & \ddots & & \\ \mathbf{0} & & & \Delta x_{n-2} & 2\nabla x_{n-2} & \Delta x_{n-3} & \\ & & & & \Delta x_{n-1} & 2\nabla x_{n-1} & \Delta x_{n-2} \\ * & * & * & \cdots & * & * & * \end{bmatrix},$$

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \vdots \\ \sigma_{n-2} \\ \sigma_{n-1} \\ \sigma_n \end{bmatrix}, \quad \text{and} \quad \mathbf{r} = \begin{bmatrix} * \\ 3\left(y[x_1,x_2]\Delta x_2 + y[x_2,x_3]\Delta x_1\right) \\ 3\left(y[x_2,x_3]\Delta x_3 + y[x_3,x_4]\Delta x_2\right) \\ \vdots \\ 3\left(y[x_{n-3},x_{n-2}]\Delta x_{n-2} + y[x_{n-2},x_{n-1}]\Delta x_{n-3}\right) \\ 3\left(y[x_{n-2},x_{n-1}]\Delta x_{n-1} + y[x_{n-1},x_n]\Delta x_{n-2}\right) \\ * \end{bmatrix}.$$

Invertibility

A tridiagonal matrix which is strictly diagonally dominant is invertible.

Implementation of Boundary Conditions (1)

- (**clamped cubic spline**) If slopes at each end are known, fill in the first and the last equation of $X\boldsymbol{\sigma} = \mathbf{r}$ with

$$\sigma_1 = y_1', \quad \sigma_n = y_n'.$$

- (**natural cubic spline**) If the second derivatives at the endpoints are known, then use

$$\begin{aligned} 2\sigma_1 + \sigma_2 &= 3y[x_1, x_2] - \frac{1}{2}\Delta x_1 y_1'' \\ \sigma_{n-1} + 2\sigma_n &= 3y[x_{n-1}, x_n] + \frac{1}{2}\Delta x_{n-1} y_n''. \end{aligned}$$

Implementation of Boundary Conditions (2)

- (**periodic boundary condition**) If the data points come from a periodic function with period $P = x_n - x_1$ so that $\sigma_1 = \sigma_n$, then use

$$\begin{aligned} \Delta x_1 \sigma_{n-1} + 2\nabla x_1 \sigma_1 + \Delta x_{n-1}\sigma_2 &= 3\left(y[x_{n-1}, x_n]\Delta x_1 + y[x_1, x_2]\Delta x_{n-1}\right) \\ \sigma_1 - \sigma_n &= 0. \end{aligned}$$

  Here, take $\nabla x_1 = x_2 - x_0 = x_2 - (x_{n-1} - P)$.

Implementation of Boundary Conditions (3)

- (***not-a-knot*** **boundary condition**) If nothing is known about the endpoints, require $p_1(x) \equiv p_2(x)$ and $p_{n-2}(x) = p_{n-1}(x)$:

$$\begin{aligned}(\Delta x_2)^2\,\sigma_1 + \left((\Delta x_2)^2 - (\Delta x_1)^2\right)\sigma_2 - (\Delta x_1)^2\,\sigma_3 \\ = 2\left(y[x_1, x_2]\,(\Delta x_2)^2 - y[x_2, x_3]\,(\Delta x_1)^2\right),\end{aligned}$$

$$\begin{aligned}-(\Delta x_{n-1})^2\,\sigma_{n-2} + \left((\Delta x_{n-2})^2 - (\Delta x_{n-1})^2\right)\sigma_{n-1} + (\Delta x_{n-2})^2\,\sigma_n \\ = 2\left(y[x_{n-1}, x_n]\,(\Delta x_{n-2})^2 - y[x_{n-2}, x_{n-1}]\,(\Delta x_{n-1})^2\right).\end{aligned}$$

Exercise. Derive the equations shown above.

Lec 28: Piecewise Interpolation – Problem Solving Session 1

Rootfinding

FZERO to Solve Complex Problem

- **FNC** 4.1.5 (Kepler's Law)

Lambert W-Function

- **FNC** 4.1.6

More With Lambert W-Function

Question. Show that solutions of the equation $2^x = 5x$

$$r = -\frac{W\left(-\log(2)/5\right)}{\log 2}.$$

(Here, as usual in this class, $\log(\,\cdot\,) = \ln(\,\cdot\,)$ is the natural logarithmic function.)
Then numerically verify the result using `fzero`[19]

[19] Two real-valued solutions, $r_1 \approx 0.2355$ and $r_2 \approx 4.488$.

FPI: When Convergence Is Faster Than Expected

- **FNC** 4.2.6

FPI: Conditions for Convergence

- **FNC** 4.2.7

Stopping Criteria

- **FNC** 4.3.8

Lec 29: Piecewise Interpolation – Problem Solving Session 2

Exercise with Series Analysis

Linear Convergence of Newton's Method

Newton's Method for Multiple Roots

Assume that $f \in C^{m+1}[a,b]$ has a root r of multiplicity m. Then Newton's method is locally convergent to r, and the error ϵ_k at step k satisfies

$$\lim_{k\to\infty} \frac{\epsilon_{k+1}}{\epsilon_k} = \frac{m-1}{m} \qquad \text{(linear convergence)}$$

- See Problem 4 of HW07 (**FNC** 4.3.7)
- Remedy: Modify the iteration formula

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

Calculating nth Roots

Question. Let n be a positive integer. Use Newton's method to produce a quadratically convergent method for calculating the nth root of a positive number a. Prove quadratic convergence.

Predicting Next Error

Question. Let $f(x) = x^3 - 4x$.

- The function $f(x)$ has a root at $r = 2$. If the error $\epsilon_k = x_k - r$ after four steps of Newton's method is $\epsilon_4 = 10^{-6}$, estimate ϵ_5.
- Do the same to the root $r = 0$.

Secant Method

Assume that iterates $x_1, x_2, \dots$ generated by the secant method converge to a root r and $f'(r) \neq 0$. Let $\epsilon_k = x_k - r$.

Exercise.[20] Show that

1. The error ϵ_k satisfies the approximate equation
$$\left|\epsilon_{k+1}\right| \approx \left|\frac{f''(r)}{2f'(r)}\right| \left|\epsilon_k\right| \left|\epsilon_{k-1}\right| .$$

2. If in addition $\lim_{k\to\infty} \left|\epsilon_{k+1}\right| / \left|\epsilon_k\right|^{\alpha}$ exists and is nonzero for some $\alpha > 0$, then
$$\left|\epsilon_{k+1}\right| \approx \left|\frac{f''(r)}{2f'(r)}\right|^{\alpha-1} \left|\epsilon_k\right|^{\alpha}, \quad \text{where } \alpha = \frac{1+\sqrt{5}}{2}.$$

[20]This exercise is from Lecture 22.

Lec 30: Piecewise Interpolation – Problem Solving Session 3

Exercise with Piecewise Interpolation

Derivation of Hermite Cubic Interpolation

Cubic Spline by Hand

- **FNC** 5.3.1

Cubic Spline with Periodic Boundary Conditions

- **FNC** 5.3.7
- Also see the appendix to Lecture 26 slides for various other boundary conditions.

Error Analysis

- **FNC** 5.3.4

Lec 33: Problem Solving Session (1/2)

Key Tools and Techniques

Lagrange Polynomials

Summary of Key Formulas (Differentiation)

Summary of Key Formulas (Integration)

Formula for Second Derivatives

Richardson Extrapolation

Extrapolation Exercise (Difference Formula)

- Done 2nd-order forward difference in lecture.
- Need to derive 3rd-order forward difference for homework.

Extrapolation Exercise (Quadrature Formula)

- Derived Simpson from midpoint and trapezoidal.
- How about composite case?

Optimal h

- Done 2nd-order centered difference.
- How about in general?

Quadrature Exercises

- Euler Spiral
- Area of an ellipse

Lec 34: Problem Solving Session (2/2)

Exercise Problems

Optimal h

Question. Suppose that a function $f(x)$ is numerically calculated by the following procedure.

```
function y = f(x)
    a = 1; b = cos(x);
    for i = 1:5
        c = b;
        b = sqrt(a.*b);
        a = (a + c)/2;
    end
    y = (pi/2)./a;
end
```

Compute $f'(\pi/4)$ as accurately as possible using a method of numerical differentiation.

Logarithmic Integral

The **logarithmic integral** is a special mathematical function defined by the equation

$$\mathrm{li}(x) = \int_2^x \frac{dt}{\ln t}.$$

Find $\mathrm{li}(200)$ by means of the composite trapezoid method.

Quadrature Exercise

Compute

$$\int_0^\infty e^{-x^2}\,dx = \frac{\sqrt{\pi}}{2}$$

by using small and large values for the limits of integration and applying a numerical method. Then compute it by making the change of variable

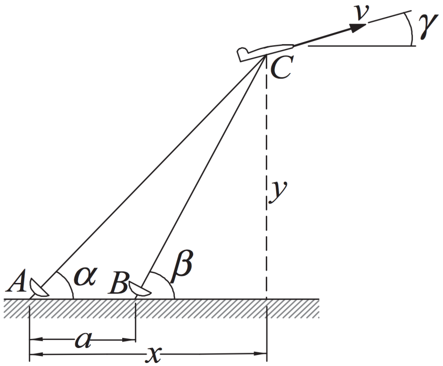
$$x = -\ln t.$$

Quadrature Exercise

Find the area of the ellipse $y^2 + 4x^2 = 1$.

Airplane Velocity

The radar stations A and B, separated by the distance $a = 500$ m, track a plane C by recording the angles α and β at one-second intervals. Your goal, back at air traffic control, is to determine the speed of the plane.



Lec 35: Spectral Theory

Eigenvalue Decomposition

Preliminary: Complex Numbers to Complex Arrays

Complex Numbers

In what follows, we assume all scalars, vectors, and matrices may be complex.

Notation.

- $\mathbb{R}$: the set of all real numbers
- $\mathbb{C}$: the set of all complex numbers, *i.e.*,

$$\{z = x + iy \,|\, x, y \in \mathbb{R}\} \quad \text{where } i = \sqrt{-1}.$$

Complex Numbers in MATLAB

Let $z = x + iy \in \mathbb{C}$.

MATLAB	Name	Notation
`real(z)`	real part of z	$\operatorname{Re} z$
`imag(z)`	imaginary part of z	$\operatorname{Im} z$
`conj(z)`	conjugate of z	$\overline{z}$
`abs(z)`	modulus of z	$\lvert z\rvert$
`angle(z)`	argument of z	$\arg(z)$

Euler's Formula

- Recall that the Maclaurin series for e^t is

$$e^t = 1 + t + \frac{t^2}{2} + \cdots + \frac{t^n}{n!} + \cdots = \sum_{n=0}^{\infty} \frac{t^n}{n!}, \quad -\infty < t < \infty.$$

- Replacing t by it and separating real and imaginary parts (using the cyclic behavior of powers of i), we obtain

$$e^{it} = \underbrace{\sum_{k=0}^{\infty} \frac{(-1)^k t^{2k}}{(2k)!}}_{\cos(t)} + i \underbrace{\sum_{k=0}^{\infty} \frac{(-1)^k t^{2k+1}}{(2k+1)!}}_{\sin(t)}$$

- The result is called the **Euler's formula**.

$$\boxed{e^{it} = \cos(t) + i \sin(t).}$$

Polar Representation and Complex Exponential

- **Polar representation:** A complex number $z = x + iy \in \mathbb{C}$ can be written as $z = re^{i\theta}$ where

$$r = |z|, \quad \tan\theta = \frac{y}{x}.$$

- **Complex exponentiation:**

$$e^{z} = e^{x+iy} = e^{x}e^{iy} = e^{x}\left(\cos y + i\sin y\right).$$

Complex Vectors

Denote by $\mathbb{C}^n = \mathbb{C}^{n\times 1}$ the space of all column vectors of n *complex* elements.

- The **hermitian** or **conjugate transpose** of $\mathbf{u} \in \mathbb{C}^n$ is denoted by $\mathbf{u}^*$:

$$\mathbf{u}^* \in \mathbb{C}^{1\times n}.$$

- The inner product of $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$ is defined by

$$\mathbf{u}^*\mathbf{v} = \sum_{k=1}^{n} \overline{u}_k v_k.$$

  The 2-norm for complex vectors is defined in terms of this inner product:

$$\|\mathbf{u}\|_2^2 = \mathbf{u}^*\mathbf{u}.$$

Complex Matrices

Denote by $\mathbb{C}^{m\times n}$ the space of all complex matrices with m rows and n columns.

- The **hermitian** or conjugate transpose of $A \in \mathbb{C}^{m\times n}$ is denoted by A^*:

$$A^* = \left(\overline{A}\right)^{\mathrm{T}} = \overline{\left(A^{\mathrm{T}}\right)} \in \mathbb{C}^{n\times m}.$$

- A **unitary** matrix is a complex analogue of an orthogonal matrix. If $U \in \mathbb{C}^{n\times n}$ is unitary, then

$$U^*U = UU^* = I$$

  and

$$\|U\mathbf{z}\|_2 = \|\mathbf{z}\|_2\,, \quad \text{for any } \mathbf{z} \in \mathbb{C}^n.$$

Complex Matrices: Some Analogies

	Real	Complex
Norm	$\|\mathbf{v}\|_2 = \sqrt{\mathbf{v}^{\mathrm{T}}\mathbf{v}}$	$\|\mathbf{u}\|_2 = \sqrt{\mathbf{u}^*\mathbf{u}}$
Symmetry	$S^{\mathrm{T}} = S$ (symmetric matrix)	$S^* = S$ (hermitian matrix)
Orthonormality	$Q^{\mathrm{T}}Q = I$ (orthogonal matrix)	$U^*U = I$ (unitary matrix)
Householder	$H = I - \dfrac{2}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\mathbf{v}\mathbf{v}^{\mathrm{T}}$	$H = I - \dfrac{2}{\mathbf{u}^*\mathbf{u}}\mathbf{u}\mathbf{u}^*$

Eigenvalue Decomposition (EVD)

Eigenvalue Decomposition

Eigenvalue Problem

Find a scalar **eigenvalue** λ and an associated nonzero **eigenvector** $\mathbf{v}$ satisfying

$$A\mathbf{v} = \lambda\mathbf{v}.$$

- The **spectrum** of A is the set of all eigenvalues; the **spectral radius** is $\max_j |\lambda_j|$.
- The problem is equivalent to
- An eigenvalue of A is a root of the **characteristic polynomial**

Eigenvalue Decomposition (cont')

Let $A \in \mathbb{C}^{n\times n}$ and suppose that $A\mathbf{v}_k = \lambda_k \mathbf{v}_k$ for $k \in \mathbb{N}[1, n]$.

- Then
- If V is nonsingular, we can further write

  which is called an **eigenvalue decomposition (EVD)** of A. If $\mathbf{v}$ is an eigenvector of A, then so is $c\mathbf{v}$, $c \neq 0$. Thus an EVD is not unique.

Eigenvalue Decomposition (cont')

If A has an EVD, we say that A is **diagonalizable**; otherwise **nondiagonalizable**.

Theorem 24 (Diagonalizability)

If $A \in \mathbb{C}^{n \times n}$ has n distinct eigenvalues, then A is diagonalizable.

Notes.

- Let $A, B \in \mathbb{C}^{n \times n}$. We say that B is **similar** to A if there exists a nonsingular matrix X such that

$$B = XAX^{-1}.$$

- So *diagonalizability* is *similarity to a diagonal matrix*.
- Similar matrices share the same eigenvalues.

Calculating EVD in MATLAB

- `E = eig(A)`
  produces a column vector `E` containing the eigenvalues of `A`.
- `[V, D] = eig(A)`
  produces V and D in an EVD of A, $A = VDV^{-1}$.

Understanding EVD: Change of Basis

Let $X \in \mathbb{C}^{n \times n}$ be a nonsingular matrix.

- The columns $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ of X form a basis of $\mathbb{C}^n$.
- Any $\mathbf{z} \in \mathbb{C}^n$ is uniquely written as

$$\mathbf{z} = X\mathbf{u} = u_1\mathbf{x}_1 + u_2\mathbf{x}_2 + \cdots + u_n\mathbf{x}_n.$$

  - The scalars $u_1, \ldots, u_n$ are called the **coordinates** of $\mathbf{z}$ with respect to the columns of X.
  - The vector $\mathbf{u} = X^{-1}\mathbf{z}$ is the representation of $\mathbf{z}$ with respect to the basis consisting of the columns of X.

Upshot

Left-multiplication by X^{-1} performs a **change of basis** into the coordinates associated with the columns of X.

Understanding EVD: Change of Basis (cont')

Suppose $A \in \mathbb{C}^{n\times n}$ has an EVD $A = VDV^{-1}$. Then, for any $\mathbf{z} \in \mathbb{C}^n$, $\mathbf{y} = A\mathbf{z}$ can be written as

Interpretation

The matrix A is a diagonal transformation in the coordinates with respect to the V-basis.

What Is EVD Good For?

Suppose $A \in \mathbb{C}^{n\times n}$ has an EVD $A = VDV^{-1}$.

- Economical computation of powers A^k:

- Analyzing convergence of iterates $(\mathbf{x}_1, \mathbf{x}_2, \ldots)$ constructed by

$$\mathbf{x}_{j+1} = A\mathbf{x}_j, \quad j = 1, 2, \ldots$$

Conditioning of Eigenvalues

Theorem 25 (Bauer-Fike)

Let $A \in \mathbb{C}^{n\times n}$ be diagonalizable, $A = VDV^{-1}$, with eigenvalues $\lambda_1, \ldots, \lambda_n$. If μ is an eigenvalue of $A + \delta A$ for a complex matrix δA, then

$$\min_{1\leqslant j\leqslant n} \left|\mu - \lambda_j\right| \leqslant \kappa_2(V) \left\|\delta A\right\|_2 .$$

Lec 36: Spectral Theory

Singular Value Decomposition

Singular Value Decomposition: Overview

Singular Value Decomposition

Theorem 26 (SVD)

Let $A \in \mathbb{C}^{m\times n}$. Then A can be written as

$$A = U\Sigma V^*, \tag{SVD}$$

where $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{n\times n}$ are unitary and $\Sigma \in \mathbb{R}^{m\times n}$ is diagonal. If A is real, then so are U and V.

- The columns of U are called the **left singular vectors** of A;
- The columns of V are called the **right singular vectors** of A;
- The diagonal entries of Σ, written as $\sigma_1, \sigma_2, \ldots, \sigma_r$, for $r = \min\{m, n\}$, are called the **singular values** of A and they are nonnegative numbers ordered as

$$\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_r \geqslant 0.$$

Singular Value Decomposition (cont’)

Thick vs Thin SVD

Suppose that $m > n$ and observe that:

SVD in MATLAB

- Thick SVD: `[U,S,V] = svd(A);`
- Thin SVD: `[U,S,V] = svd(A, 0);`

Understanding SVD

Geometric Perspective

Write $A = U\Sigma V^*$ as $AV = U\Sigma$:

$$A\mathbf{v}_k = \sigma_k \mathbf{u}_k, \quad k = 1, \ldots, r = \min\{m, n\}.$$

The image of the unit sphere under any $m \times n$ matrix is a hyperellipse.

Algebraic Perspective

Alternately, note that $\mathbf{y} = A\mathbf{z} \in \mathbb{C}^m$ for any $\mathbf{z} \in \mathbb{C}^n$ can be written as

$$\left(U^*\mathbf{y}\right) = \Sigma\left(V^*\mathbf{z}\right).$$

Any matrix $A \in \mathbb{C}^{m \times n}$ can be viewed as a diagonal transformation from $\mathbb{C}^n$ (source space) to $\mathbb{C}^m$ (target space) with respect to suitably chosen orthonormal bases for both spaces.

SVD vs. EVD

Recall that a diagonalizable $A = VDV^{-1} \in \mathbb{C}^{n\times n}$ satisfies

$$\mathbf{y} = A\mathbf{z} \quad \longrightarrow \quad \left(V^{-1}\mathbf{y}\right) = D\left(V^{-1}\mathbf{z}\right).$$

This allowed us to view any diagonalizable square matrix $A \in \mathbb{C}^{n\times n}$ as a diagonal transformation from $\mathbb{C}^n$ to itself[21] with respect to the basis formed by a set of eigenvector of A.

Differences.

- **Basis:** SVD uses two ONBs (left and right singular vectors); EVD uses one, usually non-orthogonal basis (eigenvectors).
- **Universality:** all matrices have an SVD; not all matrices have an EVD.
- **Utility:** SVD is useful in problems involving the behavior of A or A^+; EVD is relevant to problems involving A^k.

[21] The source and the target spaces of the transformation coincide.

Lec 37: Spectral Theory

Properties of SVD

Properties of SVD

SVD and the 2-Norm

Theorem 27

Let $A \in \mathbb{C}^{m\times n}$ have an SVD $A = U\Sigma V^*$. Then

1. $\|A\|_2 = \sigma_1$ *and* $\|A\|_F = \sqrt{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2}$.
2. *The rank of A is the number of nonzero singular values.*
3. *Let $r = \min\{m, n\}$. Then*

$$\kappa_2(A) = \|A\|_2 \|A^+\|_2 = \frac{\sigma_1}{\sigma_r}.$$

Connection to EVD

Let $A = U\Sigma V^* \in \mathbb{C}^{m\times n}$ and $B = A^*A$. Observe that

- $B \in \mathbb{C}^{n\times n}$ is a *hermitian matrix*[22], *i.e.*, $B^* = B$.
- B has an EVD:

    - The squares of singular values of A are eigenvalues of B.
    - An EVD of $B = A^*A$ reveals the singular values and a set of right singular vectors of A.

[22] This is the $\mathbb{C}$-extension of real symmetric matrices.

Connection to EVD (cont')

Theorem 28

The nonzero singular values of $A \in \mathbb{C}^{m \times n}$ *are the square roots of the nonzero eigenvalues of* A^*A *or* AA^*.

Unitary Diagonalization and SVD

Unitary Diagonalization of Hermitian Matrices

The previous discussion is relevant to hermitian matrices constructed in a specific manner. For a generic hermitian matrix, we have the following result.

Theorem 29 (Spectral Decomposition)

Let $A \in \mathbb{C}^{n\times n}$ be hermitian. Then A has a unitary diagonalization

$$A = VDV^{-1},$$

where $V \in \mathbb{C}^{n\times n}$ is unitary and $D \in \mathbb{R}^{n\times n}$ is diagonal.

In words, a hermitian matrix (or symmetric matrix) has a complete set of orthonormal eigenvectors and all its eigenvalues are real.

Notes on Unitary Diagonalization and Normal Matrices

- A unitarily diagonalizable matrix $A = VDV^{-1}$ with $D \in \mathbb{C}^{n \times n}$, is called a **normal matrix**[23]. All hermitian matrices are normal.
- Let $A = VDV^{-1} \in \mathbb{C}^{n \times n}$ be normal. Since $\kappa_2(V) = 1$ (why?), Bauer-Fike implies that eigenvalues of A can be changed by no more than $\|\delta A\|_2$.

[23]Usual defintion: $A \in \mathbb{C}^{n \times n}$ is normal if $AA^* = A^*A$.

Unitary Diagonalization and SVD

Theorem 30

Let $A \in \mathbb{C}^{n\times n}$ be hermitian. Then the singular values of A are the absolute values of the eigenvalues of A.

Precisely, if $A = VDV^{-1}$ is a unitary diagonalization of A, then

$$A = \big(V\,\mathrm{sign}(D)\big)\,|D|\,V^*$$

is an SVD, where

$$\mathrm{sign}(D) = \begin{bmatrix} \mathrm{sign}(d_1) & & \\ & \ddots & \\ & & \mathrm{sign}(d_n) \end{bmatrix}, \qquad |D| = \begin{bmatrix} |d_1| & & \\ & \ddots & \\ & & |d_n| \end{bmatrix}.$$

When Do Unitary EVD and SVD Coincide?

Theorem 31

If $A = A^*$, then the following statements are equivalent:

1. *Any unitary EVD of A is also an SVD of A.*
2. *The eigenvalues of A are positive numbers.*
3. *$\mathbf{x}^* A\mathbf{x} > 0$ for all nonzero $\mathbf{x} \in \mathbb{C}^n$.* *(HPD)*

- The equivalence of 1 and 2 is immediate from Theorem~30
- The property in 3 is called the **hermitian positive definiteness**, *c.f.*, symmetric positive definiteness.

Rayleigh Quotient

Let $A \in \mathbb{R}^{n\times n}$ be fixed. The **Rayleigh quotient** is the map $R_A : \mathbb{R}^n \to \mathbb{R}$ given by

$$R_A(\mathbf{x}) = \frac{\mathbf{x}^{\mathrm{T}} A\mathbf{x}}{\mathbf{x}^{\mathrm{T}}\mathbf{x}}.$$

- R_A maps an eigenvector of A into its associated eigenvalue, *i.e.*, if $A\mathbf{v} = \lambda\mathbf{v}$, then $R_A(\mathbf{v}) = \lambda$.
- If $A = A^{\mathrm{T}}$, then $\nabla R_A(\mathbf{v}) = \mathbf{0}$ for an eigenvector $\mathbf{v}$, and so

  $$R_A(\mathbf{v} + \epsilon\mathbf{z}) = R_A(\mathbf{v}) + 0 + O(\epsilon^2) = \lambda + O(\epsilon^2), \quad \text{as } \epsilon \to 0.$$

  The Rayleigh quotient is a quadratic approximation of an eigenvalue.

Reduction of Dimensions

Low-Rank Approximations

Let $A \in \mathbb{C}^{m \times n}$ with $m \geqslant n$. Its thin SVD $A = \widehat{U}\widehat{\Sigma}V^*$ can be written as

$$\begin{aligned} A &= \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_n \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^* \\ \vdots \\ \mathbf{v}_n^* \end{bmatrix} \\ &= \begin{bmatrix} \sigma_1 \mathbf{u}_1 & \cdots & \sigma_n \mathbf{u}_n \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^* \\ \vdots \\ \mathbf{v}_n^* \end{bmatrix} = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^*, \end{aligned}$$

where r is the rank of A.

- Each outer product $\mathbf{u}_j \mathbf{v}_j^*$ is a rank-1 matrix.
- Since $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_r > 0$, important contributions to A come from terms with small j.

Low-Rank Approximations (cont')

For $1 \leqslant k \leqslant r$, define

$$A_k = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^* = U_k \Sigma_k V_k^*,$$

where

- U_k is the first k columns of U;
- V_k is the first k columns of V;
- Σ_k is the upper-left $k \times k$ submatrix of Σ.

This is a rank-k approximation of A.

Best Rank-k Approximation

Theorem 32 (Eckart-Young)

Let $A \in \mathbb{C}^{m\times n}$. Suppose A has rank r and let $A = U\Sigma V^*$ be an SVD. Then

- $\|A - A_k\|_2 = \sigma_{k+1}$, *for* $k = 1, \dots, r-1$.
- *For any matrix B with* $\operatorname{rank}(B) \leqslant k$, $\|A - B\|_2 \geqslant \sigma_{k+1}$.