Matrices and Numerical Linear Algebra

Matrices – Dense and Sparse

- Dense: No zero elements (or almost none)

$$\begin{bmatrix} 3.24 & 5.76 & -12.3 & 6.11 \\ -7.55 & 4.32 & -5.02 & 4.48 \\ 3.20 & 7.19 & 12.12 & 0.24 \\ -3.14 & 8.27 & -9.81 & -2.11 \end{bmatrix}$$

- Sparse: Many zero elements

$$\begin{bmatrix} 3.24 & 0 & 0 & 6.11 \\ 0 & 0 & -5.02 & 0 \\ 3.20 & 0 & 12.12 & 0 \\ -3.14 & 0 & 0 & 0 \end{bmatrix}$$

Why is there a distinction?
Storage and performance!

Dense matrices: Image processing

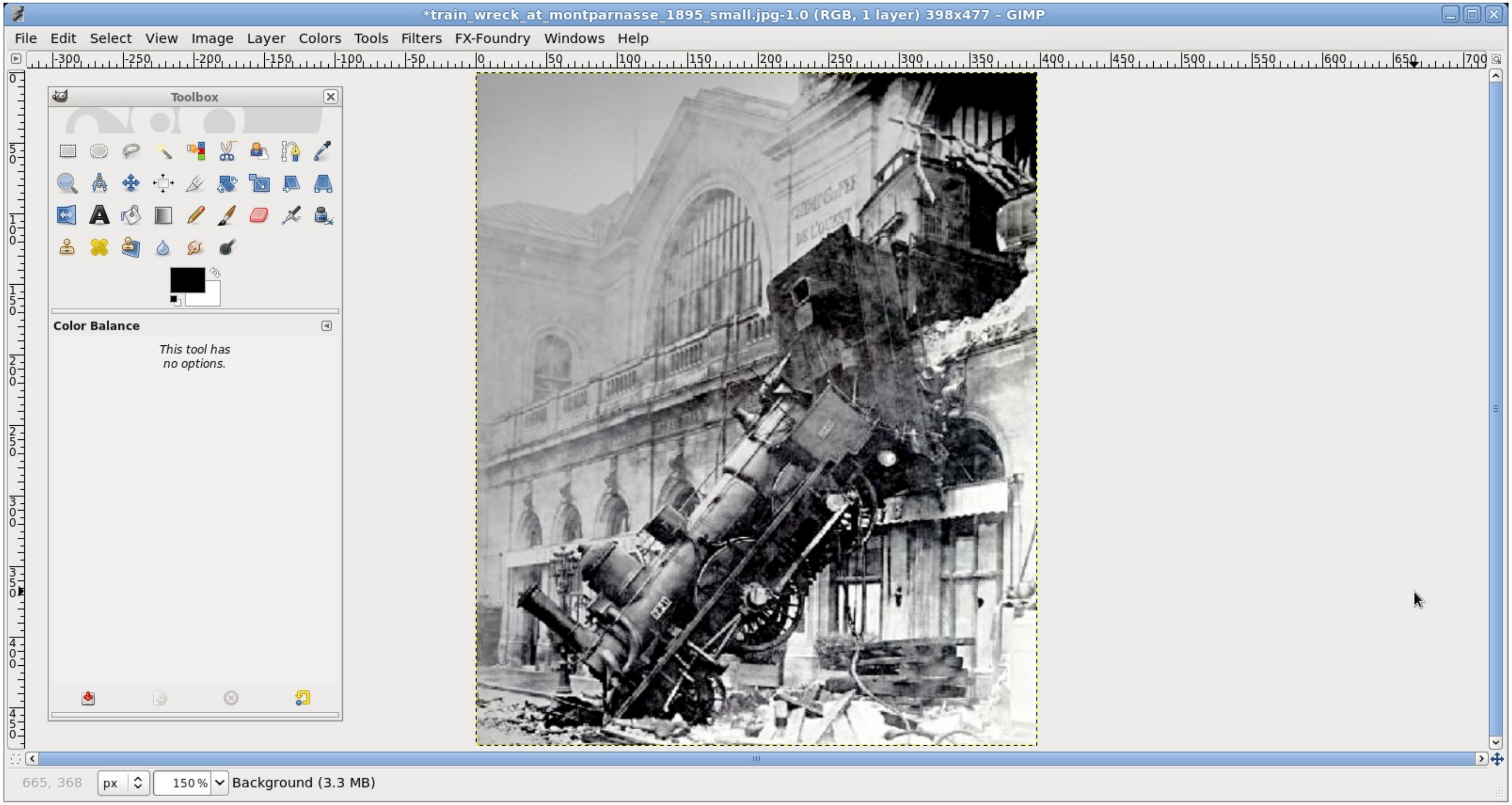
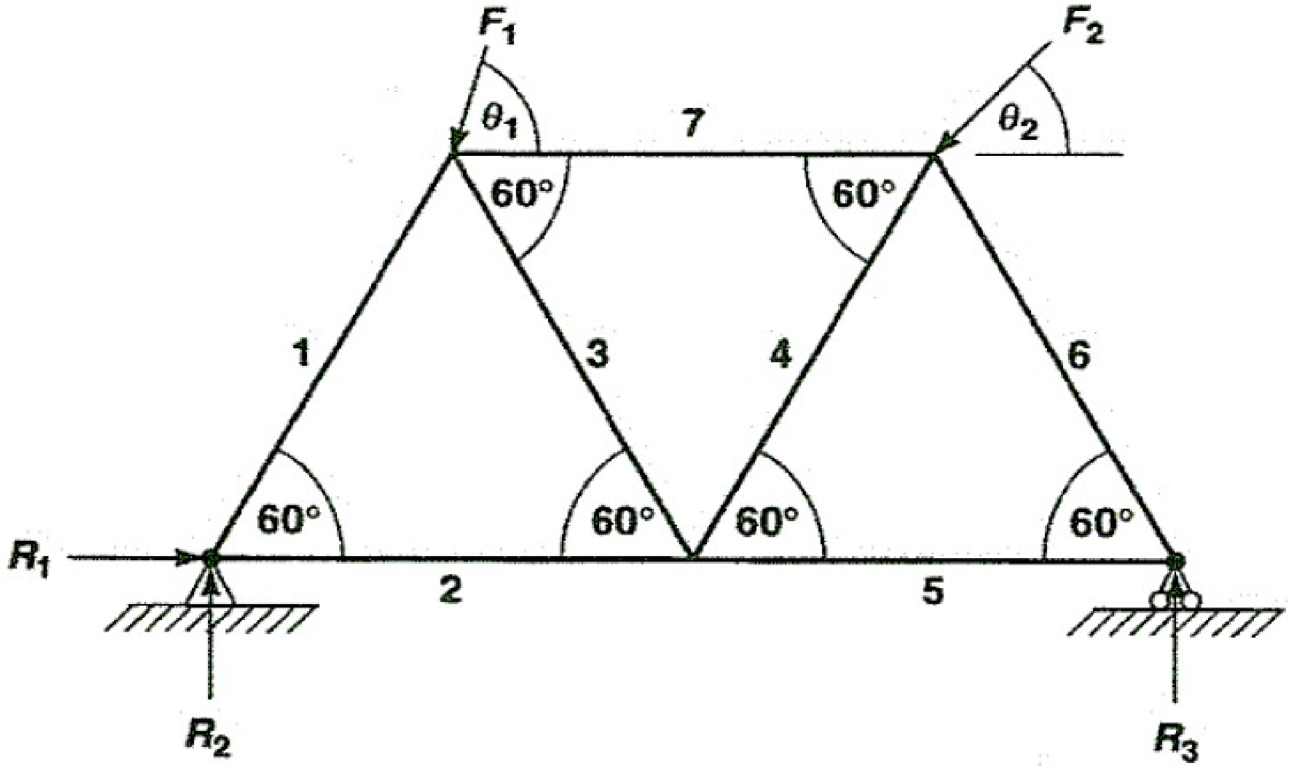


Image is a matrix, every element corresponds to a pixel (black/white level).

Sparse: Structural Analysis



- For bridge in equilibrium (i.e. not collapsing):
  - Sum of forces at each joint = 0
  - Sum of moments at each joint = 0

Force balance equation is sparse

$$
\begin{pmatrix}
1 & 1/2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \sqrt{3}/2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & -1/2 & 1/2 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \sqrt{3}/2 & \sqrt{3}/2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & -1/2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \sqrt{3}/2 & 1 & 0 \\
0 & -1/2 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 1 \\
0 & -\sqrt{3}/2 & 0 & 0 & -\sqrt{3}/20 & 0 & 0 & 0 & 0 & \\
0 & 0 & 0 & 0 & 0 & -1/2 & 0 & 1/2 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & -\sqrt{3}/2 & 0 & -\sqrt{3}/2 & 0 & 0
\end{pmatrix}
\begin{pmatrix} R_1 \\ T_1 \\ T_2 \\ R_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ R_3 \\ T_7 \end{pmatrix}
=
\begin{pmatrix} F_{1x} \\ F_{1y} \\ F_{2x} \\ F_{2y} \\ F_{3x} \\ F_{3y} \\ F_{4x} \\ F_{4y} \\ F_{5x} \\ F_{5y} \end{pmatrix}
$$

Matrix is sparse because each link pin interacts with its neighbors only.

What matrices are dense or sparse?

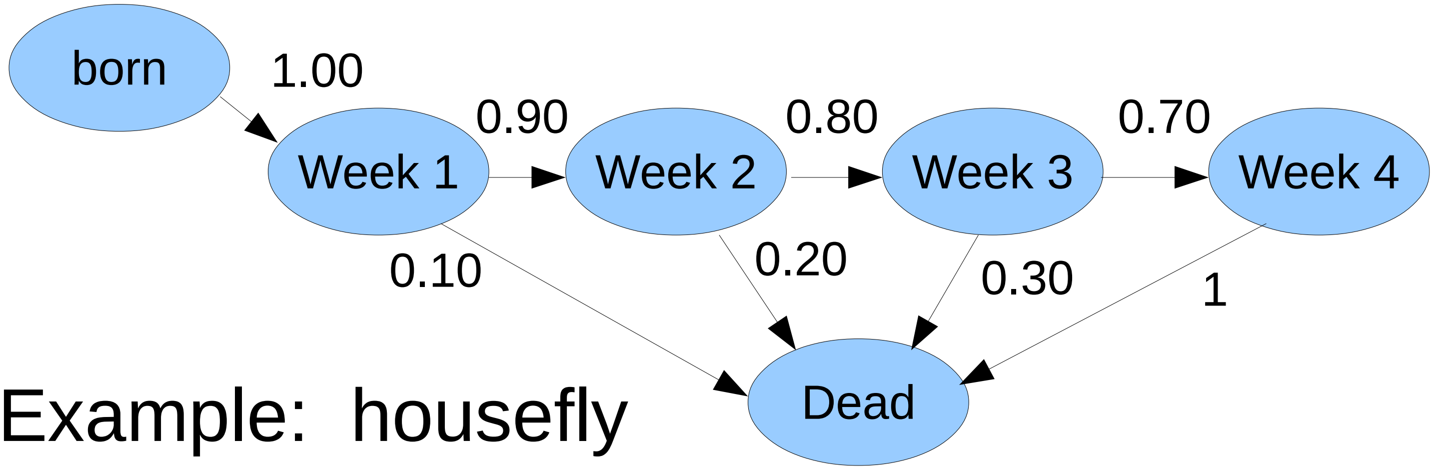
- Dense: Image processing, video.
- Sparse: Statics.
- Dense: Time series – audio processing, stock price analysis.
- Sparse: Electronic circuit analysis.
- Sparse: Solving PDEs.
- Sparse: Graph theory.

Example: population dynamics

- Consider the domestic house fly.
- Lifespan is around 15 – 30 days.
- Model of fly's life: Consider weekly probability of death.

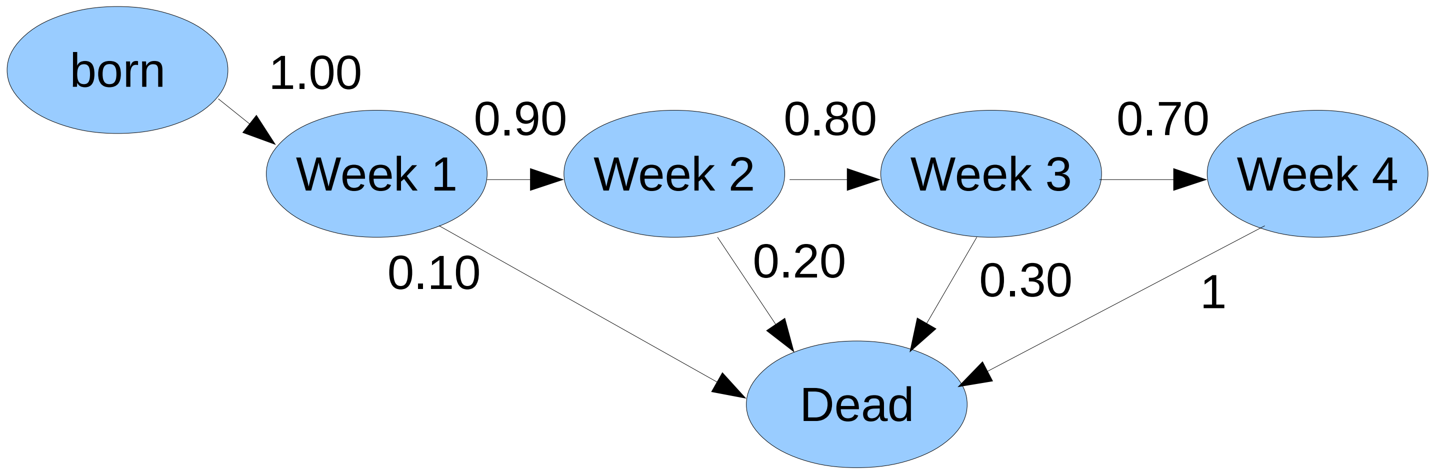
This population model is a Markov process – a **graph** of transitions

- Transition matrix
  - Notes are states
  - Edges are probabilities to jump
- Markov process



Example: housefly
life expectancy

Markov Matrix



$$
\begin{array}{cc}
 & \text{Current state (col)} \\
\begin{array}{l}\text{Next}\\ \text{state}\\ \text{(row)}\end{array} &
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & .9 & 0 & 0 & 0 & 0 \\
0 & 0 & .8 & 0 & 0 & 0 \\
0 & 0 & 0 & .7 & 0 & 0 \\
0 & .1 & .2 & .3 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
b \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ d
\end{bmatrix}
\end{array}
$$

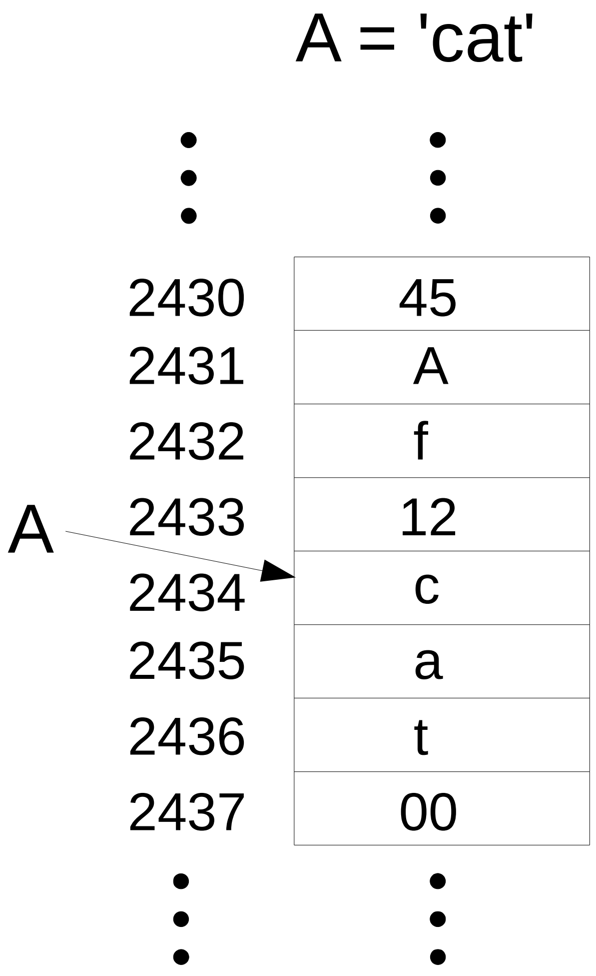
This matrix is sparse!

Storage of matrices in memory

- Why do I care about sparse vs. dense?
  - Because they are stored in totally different ways in memory.
  - When you start to scale up t[o] large problems ("big data"), thinking about dealing with your memory becomes important.
- We'll also find later that certain algorithms are more performant with sparse matrices.

Memory as linear address space

- Memory is 1D array of storage locations
- Locations are each 8 bytes wide
- Each byte has an address (32 or 64 bit)
- Longer words occupy successive bytes
  - Endianness



Storage of floating point scalars

- Consider 32 bit float (single) representation B = -5.625
- Hex representation: C0B40000
- Word starts at address 2432
- Intel is “little endian”.

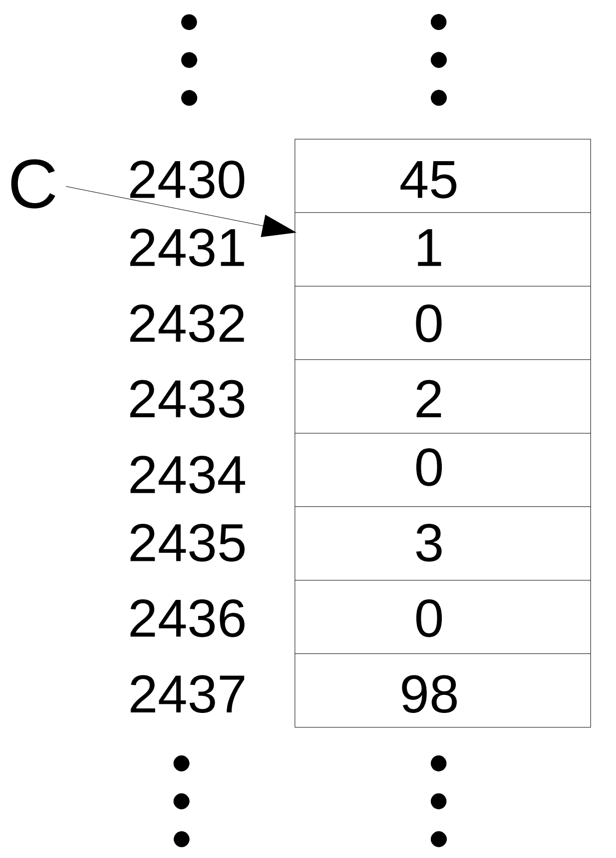
B = float(-5.625)

	⋮	⋮
	2430	45
	2431	A
B	2432	00
	2433	00
	2434	B4
	2435	C0
	2436	t
	2437	00
	⋮	⋮

Storage of vectors

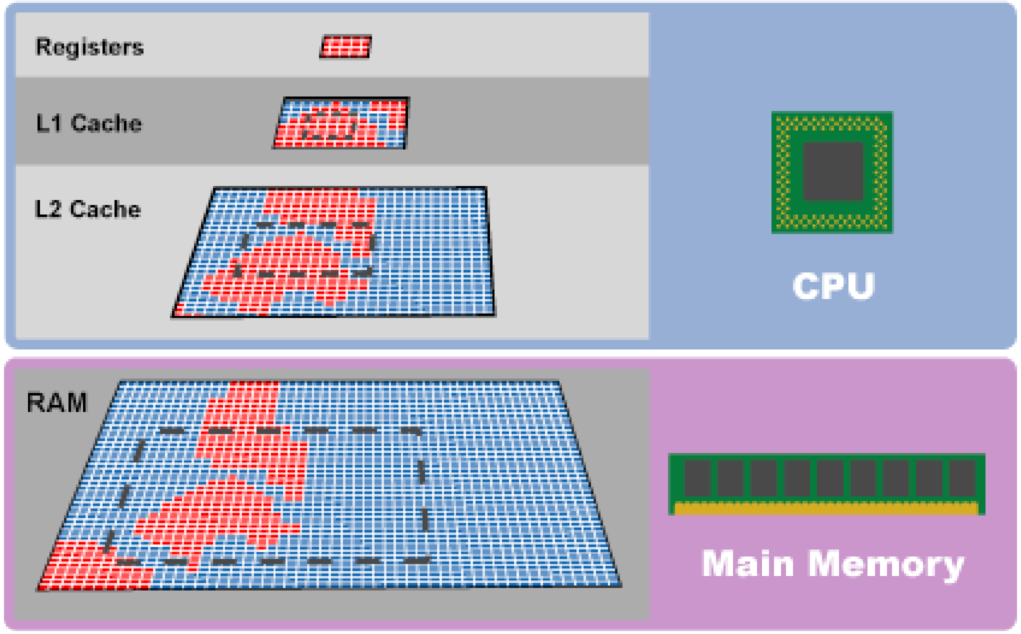
- Vector elements are stored sequentially as words in memory
- In this case, the words are 16 bits long
- Note storage is little endian

C = int16[1, 2, 3]



Locality of Reference

- Computations frequently use values in memory which lie close to each other.
- Smart compilers exploit this fact by loading memory blocks up to 1KB into cache for faster access.



Storage of Matrices?

- Row major order

$$\begin{bmatrix} 3.24 & 5.76 & -12.3 & 6.11 \\ -7.55 & 4.32 & -5.02 & 4.48 \\ 3.20 & 7.19 & 12.12 & 0.24 \\ -3.14 & 8.27 & -9.81 & -2.11 \end{bmatrix}$$

- Column major order

$$\begin{bmatrix} 3.24 & 5.76 & -12.3 & 6.11 \\ -7.55 & 4.32 & -5.02 & 4.48 \\ 3.20 & 7.19 & 12.12 & 0.24 \\ -3.14 & 8.27 & -9.81 & -2.11 \end{bmatrix}$$

Column major

$$\begin{bmatrix} 3.24 & 5.76 & -12.3 & 6.11 \\ -7.55 & 4.32 & -5.02 & 4.48 \\ 3.20 & 7.19 & 12.12 & 0.24 \\ -3.14 & 8.27 & -9.81 & -2.11 \end{bmatrix}$$

- Fortran
- Matlab
  - Linear indexing

```
octave:37> A = [1 2 3; 4 5 6]
A =

   1   2   3
   4   5   6

octave:38> A(1)
ans =  1
octave:39> A(2)
ans =  4
octave:40> A(3)
ans =  2
octave:41> A(4)
ans =  5
octave:42> A(5)
ans =  3
octave:43> A(6)
ans =  6
```

Row major

$$\begin{bmatrix} 3.24 & 5.76 & -12.3 & 6.11 \\ -7.55 & 4.32 & -5.02 & 4.48 \\ 3.20 & 7.19 & 12.12 & 0.24 \\ -3.14 & 8.27 & -9.81 & -2.11 \end{bmatrix}$$

- C, C++
  - Must be mindful when passing pointers to arrays to subroutines
  - Also important for “locality of reference” when iterating over large arrays.
- Python/NumPy

```
In [26]: A
Out[26]:
array([[1, 2, 3],
       [4, 5, 6]])

In [27]: A[numpy.unravel_index(0, A.shape)]
Out[27]: 1

In [28]: A[numpy.unravel_index(1, A.shape)]
Out[28]: 2

In [29]: A[numpy.unravel_index(2, A.shape)]
Out[29]: 3

In [30]: A[numpy.unravel_index(3, A.shape)]
Out[30]: 4

In [31]: A[numpy.unravel_index(4, A.shape)]
Out[31]: 5
```

Sparse storage – the naive way

- Matrix stored as list of (i, j, value) triplets

$$\begin{bmatrix} 3.24 & 0 & 0 & 6.11 \\ 0 & 0 & -5.02 & 0 \\ 3.20 & 0 & 12.12 & 0 \\ -3.14 & 0 & 0 & 0 \end{bmatrix}$$

i	1	1	2	3	3	4
j	1	4	3	1	3	1
x	3.24	6.11	-5.02	3.20	12.12	-3.14

- This representation grows as O(nnz)
- Dense grows as $O(N^2)$

Also in Matlab/Octave....

```
octave:3> sprandn(3, 4, .3)
ans =

Compressed Column Sparse (rows = 3, cols = 4, nnz = 3 [25%])

  (2, 1) ->  0.97173
  (3, 1) ->  0.76803
  (3, 3) -> -0.64265
```

- Draw this on blackboard.
- This is how it's displayed, not how it's actually stored...

The basic difference between sparse and dense

- Think of dense matrices stored as one big rectangle laid out in linear memory.
- Think of sparse matrices as a list of (i, j, value) 3-tuples (or as three separate lists).
- Algorithms processing them will be completely different, and will try to take advantage of the difference between the two storage formats.

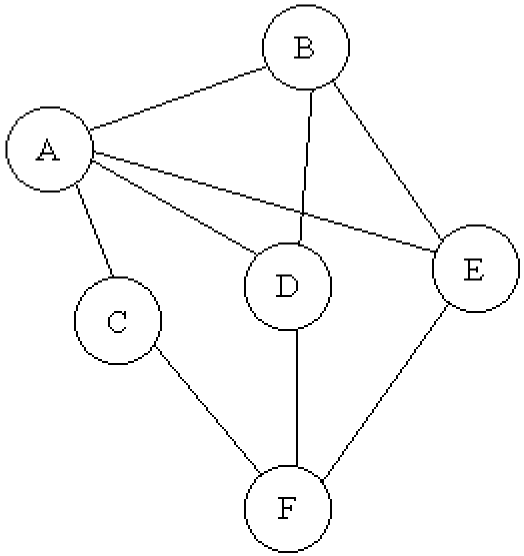
A big sparse matrix

- Consider Facebook.

1.5 Billion users at end of 2015.

- The connections of “friends” can be represented as a big adjacency matrix.

	A	B	C	D	E	F
A	-	1	1	1	1	
B	1	-		1	1	
C	1		-			1
D	1	1		-		1
E	1	1			-	1
F			1	1	1	-



- What is size of adjacency matrix (dense)?
- What is size of adjacency matrix (sparse)?

New Topic: Numerical Linear Algebra

$$\boxed{Ax = b}$$

- The classic linear algebra problem
- First we talk about matrix multiplication (today).
- Then we talk about solvers and matrix decompositions (today and next several classes).

Matrix Multiplication

$$\overset{A}{\begin{pmatrix} -3 & -1 & -1 & -5 & 1 \\ -3 & -3 & -4 & -5 & 3 \\ -1 & -5 & 3 & -1 & -3 \\ 3 & 2 & -1 & -4 & -4 \\ -5 & 3 & -2 & -1 & -1 \end{pmatrix}} \overset{B}{\begin{pmatrix} 0 & 5 & 3 & -3 & 0 \\ 5 & 5 & 2 & 0 & -1 \\ 3 & 0 & -4 & -1 & -4 \\ 4 & 0 & -3 & 2 & 4 \\ 4 & -2 & 0 & -1 & 3 \end{pmatrix}}$$

$$AB = \begin{pmatrix} -24 & -22 & 8 & -1 & -12 \\ -35 & -36 & 16 & 0 & 8 \\ -32 & -24 & -22 & 1 & -20 \\ -25 & 33 & 29 & -12 & -26 \\ 1 & -8 & 2 & 16 & -2 \end{pmatrix}$$

Naive algorithm uses 3 nested loops:

1. Rows of A
2. Cols of B
3. Loop over elements doing multiplication and sum

Matrix Multiplication -- Naive

```
function z = matmul(x, y)
  % Matrix multiplication the naive way, using loops.
  % This algorithm is O(n^3)
  % z = x*y
  % size(x) = [n, m]
  % size(y) = [m, p]
  % size(z) = [n, p]

  [n, m] = size(x);
  [m, p] = size(y);

  z = zeros(n, p);    % Preallocate z for performance
  for row = 1:n
    for col = 1:p
      for idx = 1:m
        z(row, col) = z(row, col) + x(row, idx)*y(idx, col);
      end
    end
  end

return
```

- Three nested loops.

An aside.....

$$\begin{array}{cc} A & B \\ \begin{pmatrix} -3 & -1 & -1 & -5 & 1 \\ -3 & -3 & -4 & -5 & 3 \\ -1 & -5 & 3 & -1 & -3 \\ 3 & 2 & -1 & -4 & -4 \\ -5 & 3 & -2 & -1 & -1 \end{pmatrix} & \begin{pmatrix} 0 & 5 & 3 & -3 & 0 \\ 5 & 5 & 2 & 0 & -1 \\ 3 & 0 & -4 & -1 & -4 \\ 4 & 0 & -3 & 2 & 4 \\ 4 & -2 & 0 & -1 & 3 \end{pmatrix} \end{array}$$

$$AB = \begin{pmatrix} -24 & -22 & 8 & -1 & -12 \\ -35 & -36 & 16 & 0 & 8 \\ -32 & -24 & -22 & 1 & -20 \\ -25 & 33 & 29 & -12 & -26 \\ 1 & -8 & 2 & 16 & -2 \end{pmatrix}$$

- Think about how dense matrix multiplication must be different from sparse matrix multiplication.
  - If both matrices have lots of zeros, it's a waste to loop over each and every element.

How does run time scale with matrix size?

- Three nested loops suggests run time should grow as N^3.
- We call this an $O(N^3)$ algorithm
- Timing using time_matmul()

```
>> [x, y] = time_matmul;
N =     10, avg multiplication time =        0.00010812 sec
N =     30, avg multiplication time =        0.00107062 sec
N =    100, avg multiplication time =        0.02607275 sec
N =    300, avg multiplication time =        0.63386387 sec
N =   1000, avg multiplication time =       32.70939337 sec
Scaled time vec (seconds per multiplication) =     1.0e-06 *

    0.1081    0.0397    0.0261    0.0235    0.0327
```

Timing matmul

- This uses my naive implementation

```
>> [x, y] = time_matmul;
N =     10, avg multiplication time =       0.00010812 sec
N =     30, avg multiplication time =       0.00107062 sec
N =    100, avg multiplication time =       0.02607275 sec
N =    300, avg multiplication time =       0.63386387 sec
N =   1000, avg multiplication time =      32.70939337 sec
Scaled time vec (seconds per multiplication) =    1.0e-06 *

    0.1081    0.0397    0.0261    0.0235    0.0327

Multiplication is O(2.941)
```

Improving matmul: Strassen's algorithm

- Consider the usual algorithm:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

$$c_{11} = a_{11} b_{11} + a_{12} b_{21} \qquad c_{12} = a_{11} b_{12} + a_{12} b_{22}$$

$$c_{21} = a_{21} b_{11} + a_{22} b_{21} \qquad c_{22} = a_{21} b_{12} + a_{22} b_{22}$$

- 8 multiplies, 4 additions.

Floating point multiplication performed in hardware. Takes more time than addition.

- Note that the elements can themselves be matrices (recursive computation).

Strassen's algorithm

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Take

$q_1 = (a_{11} + a_{22}) * (b_{11} + b_{22})$

$q_2 = (a_{21} + a_{22}) * b_{11}$

$q_3 = a_{11} * (b_{12} - b_{22})$

$q_4 = a_{22} * (b_{21} - b_{11})$

$q_5 = (a_{11} + a_{12}) * b_{22}$

$q_6 = (a_{21} - a_{11}) * (b_{11} + b_{12})$

$q_7 = (a_{12} - a_{22}) * (b_{21} + b_{22})$

Then

$c_{11} = q_1 + q_4 - q_5 + q_7$

$c_{12} = q_3 + q_5$

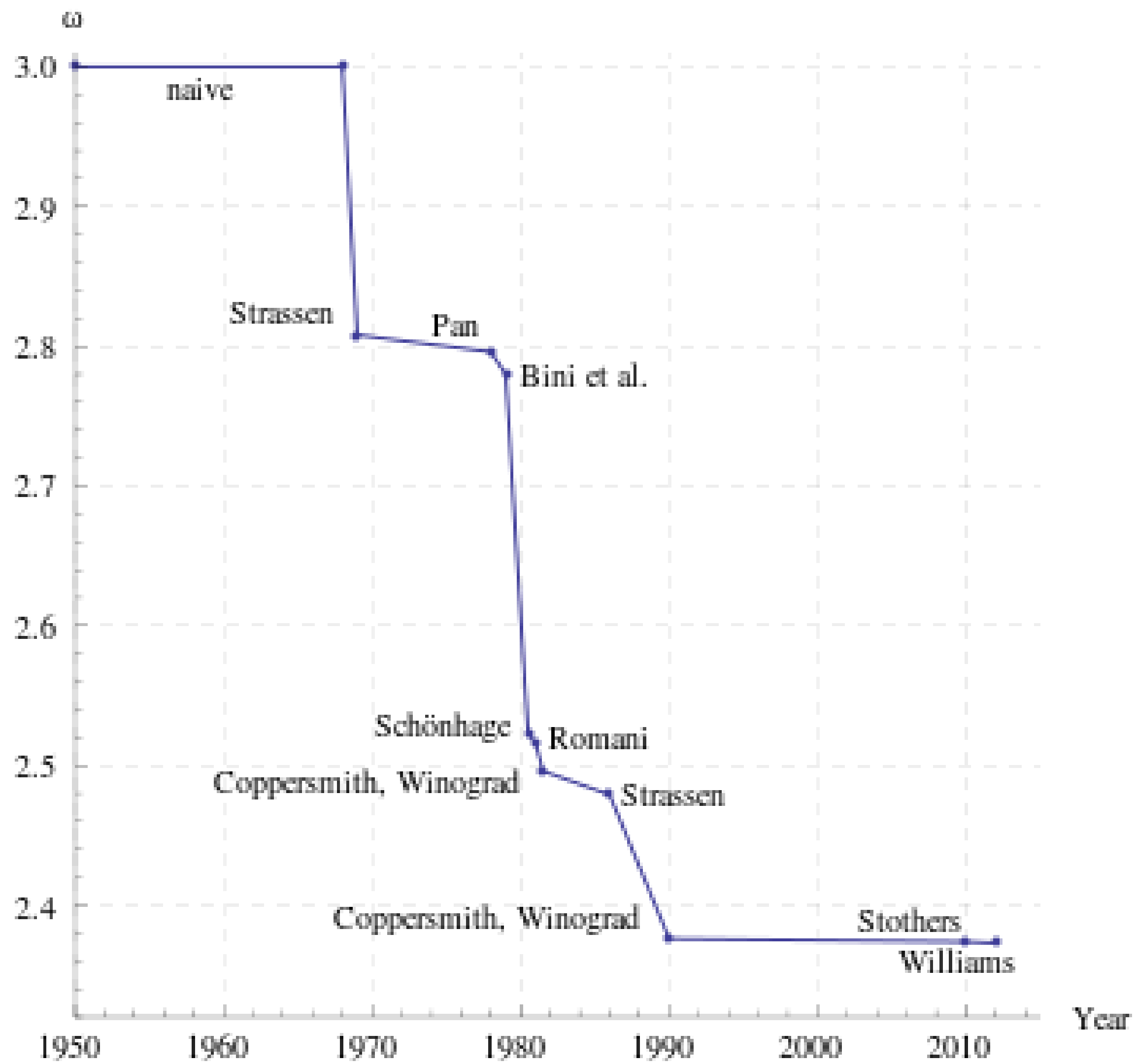
$c_{21} = q_2 + q_4$

$c_{22} = q_1 + q_3 - q_2 + q_6$

Fewer multiplications but more additions.

- 7 multiplies, 13 additions.
- Elements can also be matrices – recursive algorithm.

Progress in matmul



$O(\omega)$

Year

Source: Wikipedia

Timing matmul

- This uses MATLAB built-in multiplication (BLAS)

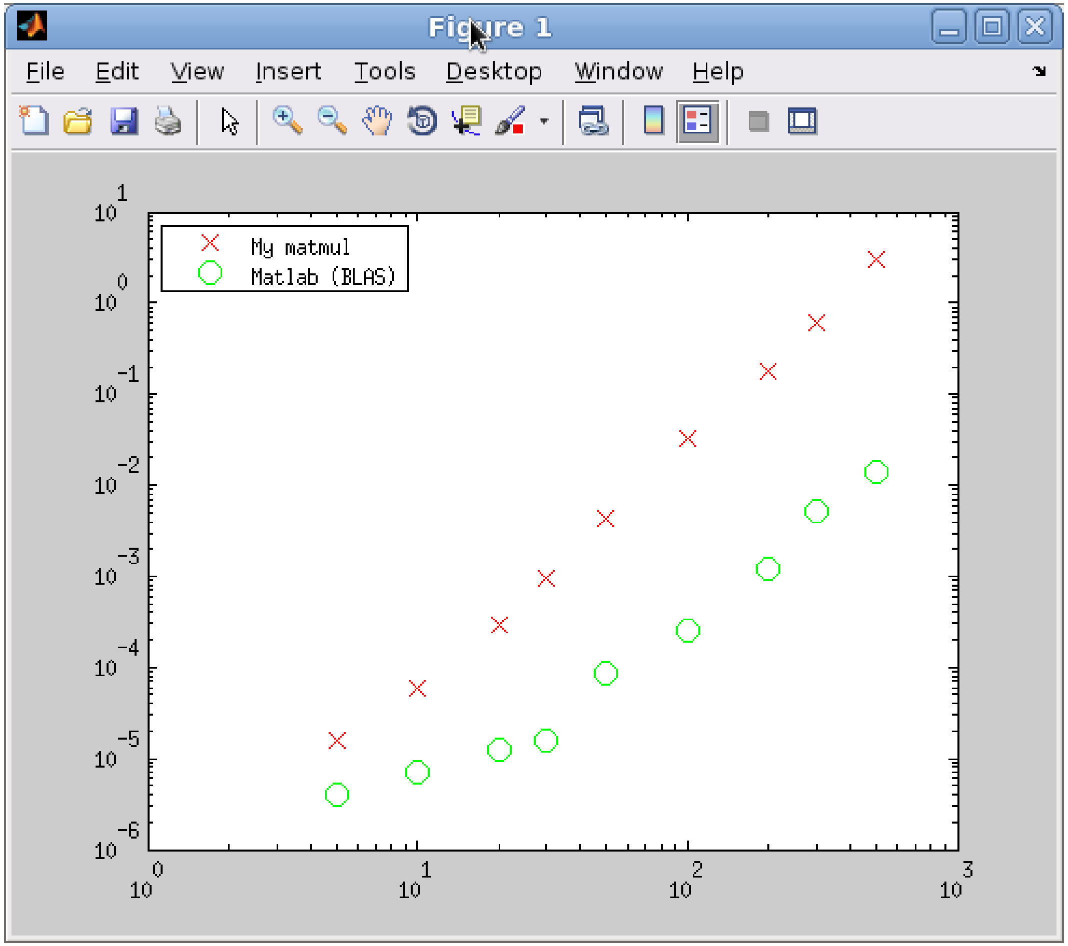
```
>> [x, y] = time_matmul;
N =     10, avg multiplication time =       0.00001400 sec
N =     30, avg multiplication time =       0.00002912 sec
N =    100, avg multiplication time =       0.00025200 sec
N =    300, avg multiplication time =       0.00333462 sec
N =   1000, avg multiplication time =       0.08979813 sec
Scaled time vec (seconds per multiplication) =    1.0e-07 *

    0.1400    0.0108    0.0025    0.0012    0.0009

Multiplication is O(2.296)
```

Much better than $O(N^3)$

Comparison: mymatmul vs. Matlab

```
>> time_matmul
---------------  mymatmul  ---------------------
N =      3, avg multiplication time =       0.00005175 sec
N =      5, avg multiplication time =       0.00001300 sec
N =     10, avg multiplication time =       0.00003575 sec
N =     20, avg multiplication time =       0.00024750 sec
N =     30, avg multiplication time =       0.00093600 sec
N =     50, avg multiplication time =       0.00428825 sec
N =    100, avg multiplication time =       0.03385675 sec
N =    200, avg multiplication time =       0.20763750 sec
N =    300, avg multiplication time =       0.62138575 sec
N =    500, avg multiplication time =       3.58971225 sec
N =   1000, avg multiplication time =      36.35315700 sec
---------------  BLAS  ---------------------
N =      3, avg multiplication time =       0.00000800 sec
N =      5, avg multiplication time =       0.00000525 sec
N =     10, avg multiplication time =       0.00000850 sec
N =     20, avg multiplication time =       0.00001350 sec
N =     30, avg multiplication time =       0.00002050 sec
N =     50, avg multiplication time =       0.00008450 sec
N =    100, avg multiplication time =       0.00023975 sec
N =    200, avg multiplication time =       0.00169225 sec
N =    300, avg multiplication time =       0.00588450 sec
N =    500, avg multiplication time =       0.02320725 sec
N =   1000, avg multiplication time =       0.08831325 sec
My multiplication is O(N^2.962)
Matlabs multiplication is O(N^2.428)
```

Scaling of matmul



- O(N^p) is asymptotic behavior.
  - Naive implementation: $O(N^3)$
  - Matlab/BLAS: $\approx O(N^{2.4})$

Detour: Libraries for Linear Algebra

- Accessible as function calls from C, Fortran, Java, etc.
- BLAS – Basic operations on vectors and matrices
  - Level 1: vector-vector operations.
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations
- LAPACK – higher-level algorithms
  - Linear solvers
  - Matrix decompositions

BLAS/LAPACK naming convention

X	Y	Y	Z	Z	Z
d	g	e	m	m	

- X = S, D, C, Z for single, double, single complex, double complex.
- YY = Code for matrix type
  - ge = general
  - dl = diagonal
  - he = Hermitian
- ZZZ = Name of algorithm

Examples:

- dgemm – general matrix-matrix multiply for double: $A * B + \beta C$
- daxpy – vector vector addition for double: $\alpha X + Y$

This naming convention has carried over to other math packages as well.

BLAS

- BLAS cheat sheet

  http://www.netlib.org/lapack/lug/node145.html

- Exploits locality of reference

$$\overset{A}{\begin{pmatrix} -3 & -1 & -1 & -5 & 1 \\ -3 & -3 & -4 & -5 & 3 \\ -1 & -5 & 3 & -1 & -3 \\ 3 & 2 & -1 & -4 & -4 \\ -5 & 3 & -2 & -1 & -1 \end{pmatrix}} \overset{B}{\begin{pmatrix} 0 & 5 & 3 & -3 & 0 \\ 5 & 5 & 2 & 0 & -1 \\ 3 & 0 & -4 & -1 & -4 \\ 4 & 0 & -3 & 2 & 4 \\ 4 & -2 & 0 & -1 & 3 \end{pmatrix}}$$

$$AB = \begin{pmatrix} -24 & -22 & 8 & -1 & -12 \\ -35 & -36 & 16 & 0 & 8 \\ -32 & -24 & -22 & 1 & -20 \\ -25 & 33 & 29 & -12 & -26 \\ 1 & -8 & 2 & 16 & -2 \end{pmatrix}$$

  - Try to perform all operation on elements close to each other in memory before moving to next set of elements.
  - Block strategies for matrix multiply.
  - Rely on optimizing cache hits for best performance.

Dgemm example

- Demo under:

/home/sbrorson/Northeastern1_Spring2016/Class3/dgemm

LAPACK – Linear algebra

- Overview

  http://www.netlib.org/lapack/lug/node19.html

- Naming scheme

  http://www.netlib.org/lapack/lug/node24.html

- Available routines

  http://www.netlib.org/lapack/lug/node27.html

Users of BLAS and LAPACK

- Matlab – built in
  - http://www.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html
- Octave, SciLab, Julia.... -- built in.
- C++/Boost – must #include
- You should too. These programs have been optimized by scores of incredibly smart people for many years. Don't re-invent the wheel.

New topic: Linear Solvers

- Linear system written out:

$$\begin{aligned} a_{11}x_1 + \; & a_{12}x_2 + \cdots + \; a_{1n}x_n = b_1 \\ a_{21}x_1 + \; & a_{22}x_2 + \cdots + \; a_{2n}x_n = b_2 \\ \vdots \qquad & \;\vdots \qquad\qquad\quad \vdots \qquad\;\; \vdots \\ a_{m1}x_1 + \; & a_{m2}x_2 + \cdots + a_{mn}x_n = b_m . \end{aligned}$$

- Matrix notation:

$$Ax = b$$

- In general, we know A and b, want to find x.
- Never do this: $x = A^{-1}b$!!!

http://www.johndcook.com/blog/2010/01/19/dont-invert-that-matrix/

Solving a linear system in Matlab

- Linear system written out:

$$\begin{aligned} a_{11}x_1 + \ & a_{12}x_2 + \cdots + \ a_{1n}x_n = b_1 \\ a_{21}x_1 + \ & a_{22}x_2 + \cdots + \ a_{2n}x_n = b_2 \\ \vdots \qquad & \ \ \vdots \qquad\qquad\quad \vdots \qquad \vdots \\ a_{m1}x_1 + & a_{m2}x_2 + \cdots + a_{mn}x_n = b_m . \end{aligned}$$

- In matrix notation

$$Ax = b$$

A, b known, x is vector of unknowns

- Matlab solution:

$$x = A \backslash b$$

Solve Ax = b using \ operator

```
octave:55> A = eye(4) + 0.006*randn(4);
octave:56> x = A\b;
octave:57> r1 = b - A*x
r1 =

  -2.7756e-17
  -2.2204e-16
   0.0000e+00
  -2.2204e-16
```

Residual: This should be zero

- Residual is (related to) the error of your solution x.
- Residual is generally non-zero, but the goal is to minimize it.

Four different division operators in Matlab

Solve

- mldivide "\\": $Ax=b \rightarrow x=A^{-1}b \rightarrow x=b\backslash A$
- mrdivide "/": $xA=b \rightarrow x=b\,A^{-1} \rightarrow x=b/A$
- ldivide ".\\":

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \backslash \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} b_{11}/a_{11} & b_{12}/a_{12} & b_{13}/a_{13} \\ b_{21}/a_{21} & b_{22}/a_{22} & b_{23}/a_{23} \end{pmatrix}$$

- rdivide "./":

Elementwise divide

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} / \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} a_{11}/b_{11} & a_{12}/b_{12} & a_{13}/b_{13} \\ a_{21}/b_{21} & a_{22}/b_{22} & a_{23}/b_{23} \end{pmatrix}$$

Difference between residual and error

- Consider: $x = A \backslash b$

A and b are both known exactly. The computed x will have some small error.

- Error: $x_{true} = x_{comp} + e$
- Residual: $r = b - Ax_{comp}$
- By definition: $Ax_{true} = b$
- Therefore: $A(x_{comp} + e) = b$

$$Ae = b - Ax_{comp}$$

$$\boxed{Ae = r}$$

- Note that we can know *r* and *A*, but the value of *e* is not directly observable.

Find residual of simple computation

```
b = randn(4, 1);
A = eye(4) + 0.006*randn(4);
x = A\b;
r1 = b - A*x

 r1 =

    1.0e-15 *

   -0.0555
         0
    0.4441
    0.0069
```

Residual: This should be zero

- Residual is small, but not zero

Try a different matrix

```
A = ones(4) + 0.006*randn(4);
x = A\b;
r2 = b - A*x

r2 =

   1.0e-13 *

   -0.0466
   -0.2043
   -0.1577
   -0.1016
```

ones(4) is singular

Residual: This should be zero

- Larger error!
- This A is closer to singular matrix.
- In general, the closer A is to singular, the larger the residual.

How to characterize this effect?

We want a measure of how singular a matrix is......

- Matrix condition number

Matrix norm ||A||

$$\boxed{k = \|A\| \cdot \|A^{-1}\|}$$

- Small condition number (near 1), solvers are stable -> low error.
- Large condition number, solvers are error-prone -> high error.
- Matlab: cond(A)

Condition number requires matrix norm

- Vector norms:
  - 1 norm: $\|x\|_1 = \sum_i |x_i|$
  - Infinity norm: $\|x\|_\infty = max(|x_0|, |x_1|, \cdots, |x_n|)$
  - Euclidian norm: $\|x\| = \sqrt{\sum_i |x_i^2|}$

- General p-norm for vectors:

$$\|x\|_p = \left(\sum_i |x_i^p|\right)^{1/p}$$

Matrix norm is generalization of vector norms

- 1 norm:

$$\|A\|_1 = max_j \left(\sum_i |A_{ij}| \right)$$

$$\begin{bmatrix} 3.24 & 5.76 & -12.3 & 6.11 \\ -7.55 & 4.32 & -5.02 & 4.48 \\ 3.20 & 7.19 & 12.12 & 0.24 \\ -3.14 & 8.27 & -9.81 & -2.11 \end{bmatrix}$$

- Infinity norm:

$$\|A\|_\infty = max_i \left(\sum_j |A_{ij}| \right)$$

$$\begin{bmatrix} 3.24 & 5.76 & -12.3 & 6.11 \\ -7.55 & 4.32 & -5.02 & 4.48 \\ 3.20 & 7.19 & 12.12 & 0.24 \\ -3.14 & 8.27 & -9.81 & -2.11 \end{bmatrix}$$

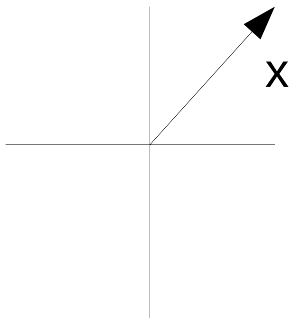
- Induced norm:

$$\|A\| = max_i \left(\sqrt{\lambda(A^T A)} \right)$$

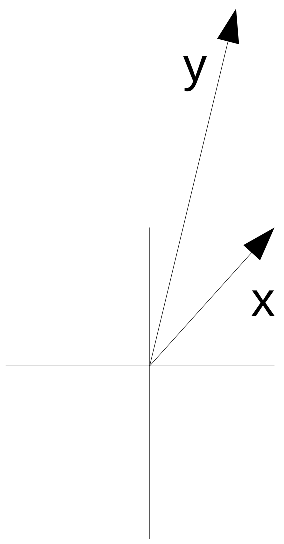
Maximum singular value of A

Matrix as linear transformation

- Consider vector x



- Apply matrix A to vector x via multiplication.



- We get new vector y = Ax

Induced matrix norm

- Ax is a vector. A acts on x, stretching and rotating x.
- Therefore, we can extend the concept of vector norm by focusing on product Ax.
- Consider action of A on all vectors x satisfying $||x|| = 1$.
- This gives the "Induced norm":

  $||A|| = \max(||Ax||)$ when $||x|| = 1$.
- Therefore, the induced norm is (related to) max eigenvalue (singular value) of A

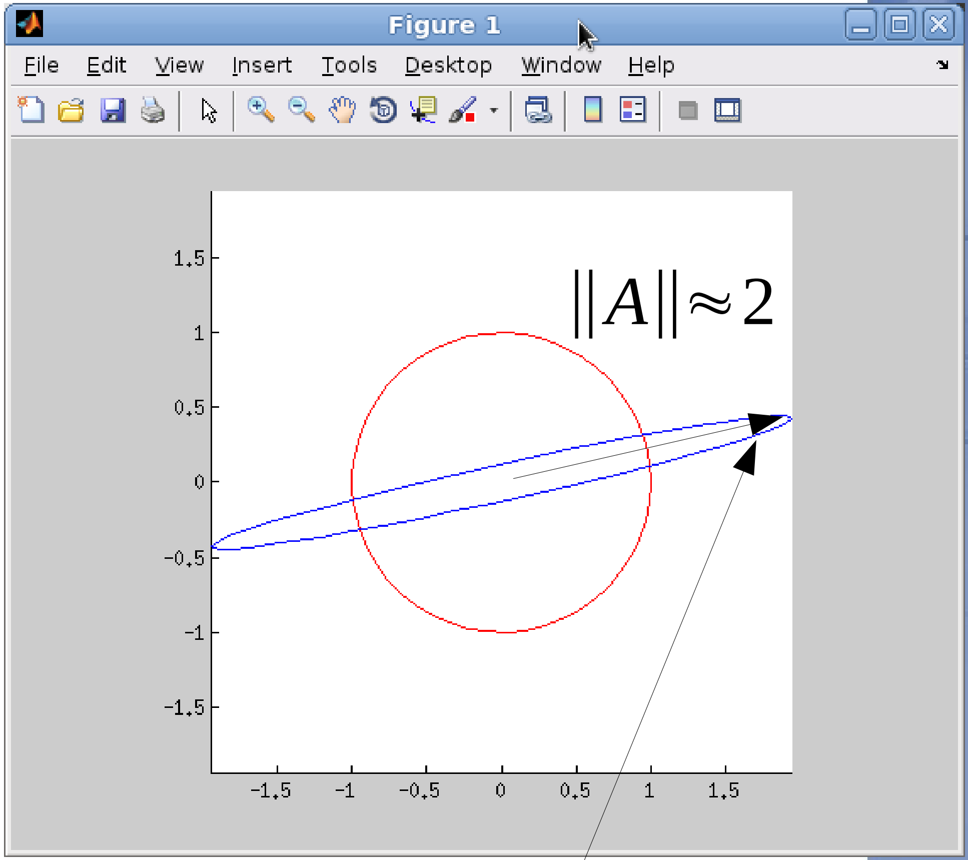
Induced matrix norm

- Start with vector norm:

$$||x||$$

- Define matrix norm by considering action of matrix on all vectors:

$$||A|| = max\left(\frac{||A\,x||}{||x||} : x \in K^n\right)$$



Find largest extension of unit circle induced by matrix.

Condition number

$$k = \| A \| \cdot \| A^{-1} \|$$

```
>> A = eye(4) + 0.006*randn(4);
>> cond(A)

ans =

    1.0107
```

Matrix is far from singular – condition number close to 1.

```
>> A = ones(4) + 0.006*randn(4);
>> cond(A)

ans =

   2.3771e+03
```

Matrix is close to singular – high condition number.

- Which matrix norm is used for *k*? Doesn't matter....

Main ideas in lecture

- Types of matrix: dense and sparse.
  - Difference is in how they are stored in memory.
  - This also implies a difference in processing algorithms.
- Matrix multiplication (dense).
- Solving $Ax = b$, and errors (residual).
- Matrix condition number Important!
  - Characterizes potential for error.