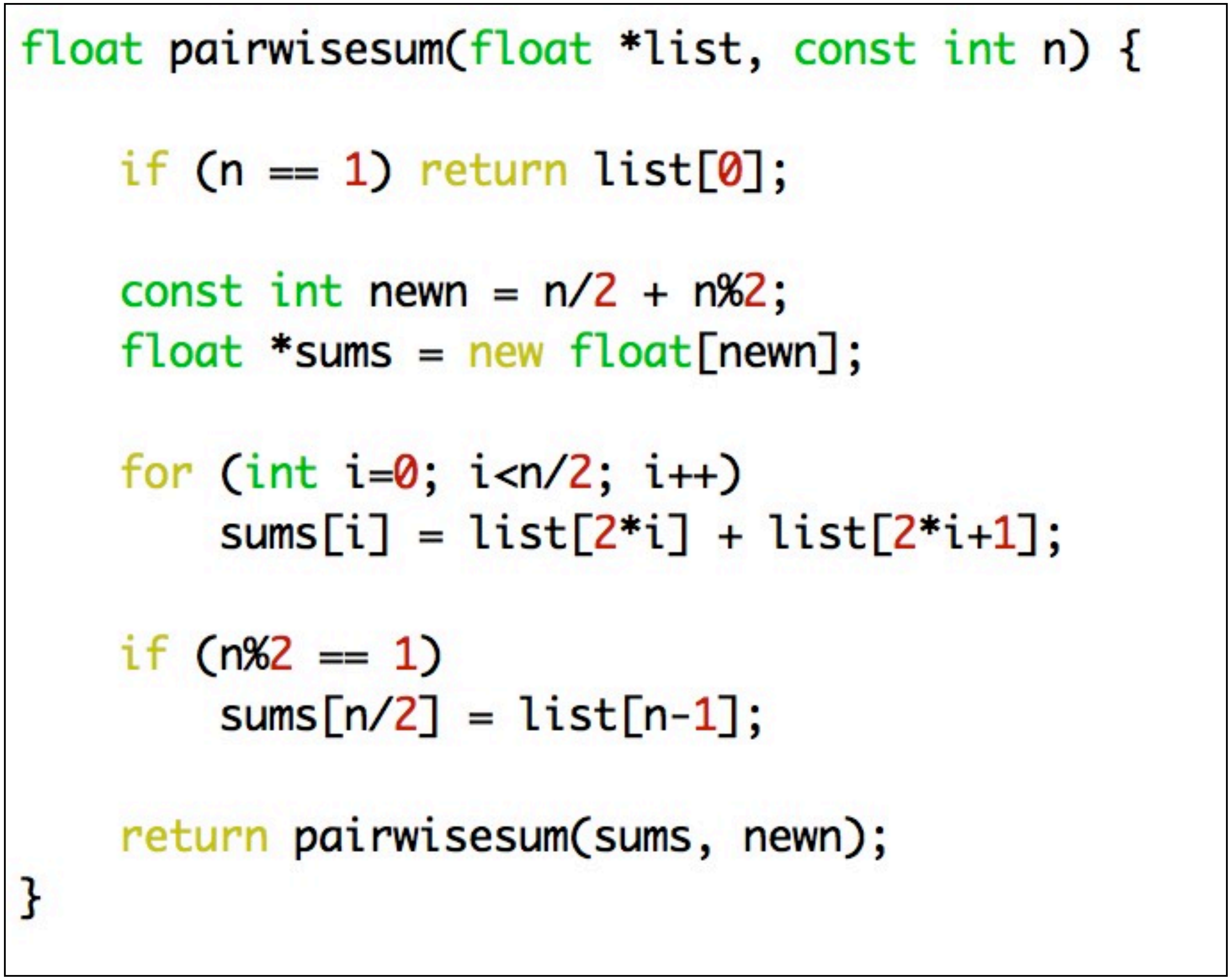
HW1 - Summing

- 1e-8 smaller than machine epsilon (float)
- Forward sum fails utterly
- Backward sum does better (why not correctly?)

```
$ ./part1
Left  sum:    1
Right sum:    1.25
```

HW1 - Summing

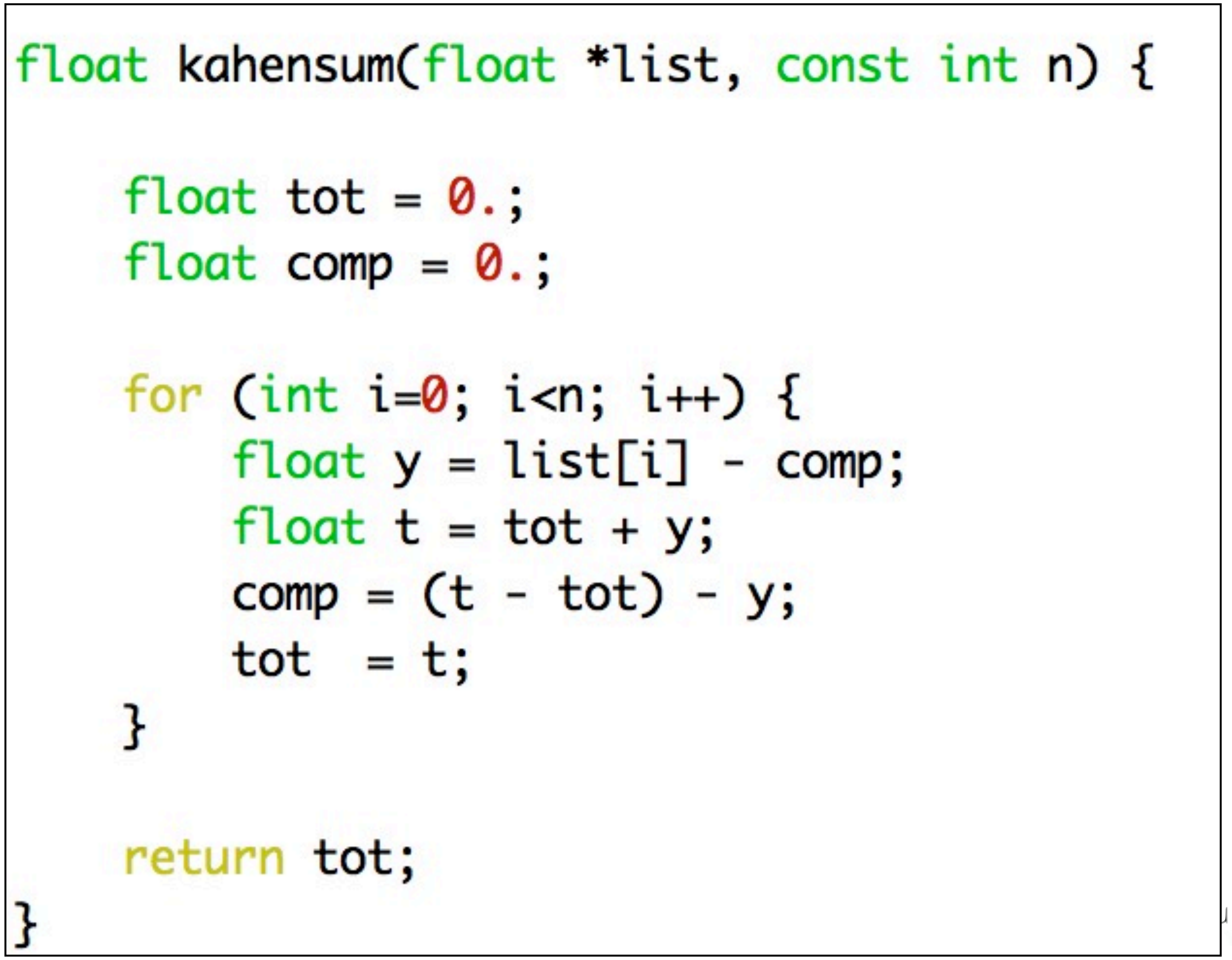
- Lots of ways around this:



```
float pairwisesum(float *list, const int n) {

    if (n == 1) return list[0];

    const int newn = n/2 + n%2;
    float *sums = new float[newn];

    for (int i=0; i<n/2; i++)
        sums[i] = list[2*i] + list[2*i+1];

    if (n%2 == 1)
        sums[n/2] = list[n-1];

    return pairwisesum(sums, newn);
}
```

HW1 - Summing

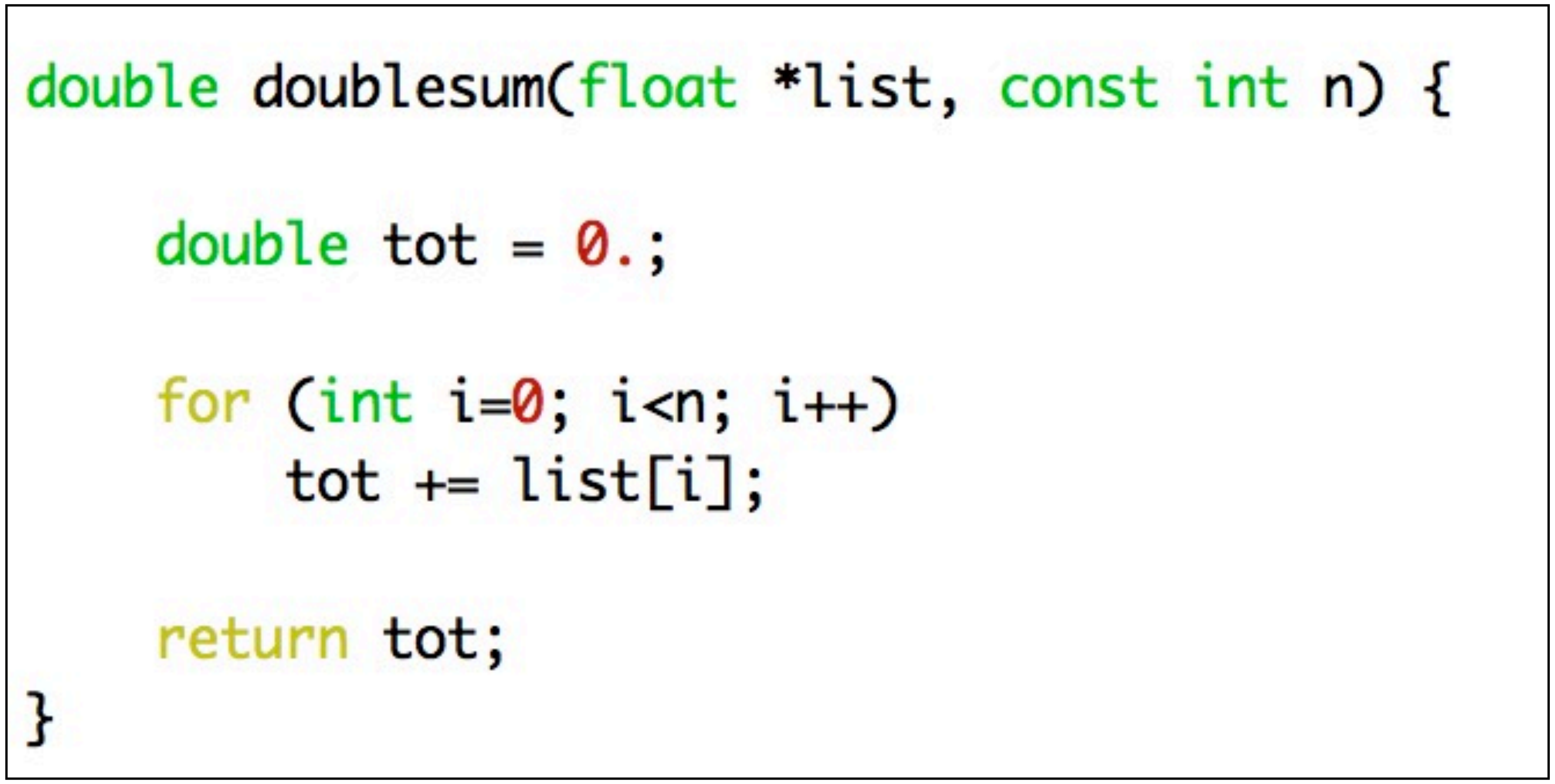
- Lots of ways around this:



```
float kahensum(float *list, const int n) {

    float tot = 0.;
    float comp = 0.;

    for (int i=0; i<n; i++) {
        float y = list[i] - comp;
        float t = tot + y;
        comp = (t - tot) - y;
        tot  = t;
    }

    return tot;
}
```

HW1 - Summing

- Lots of ways around this:



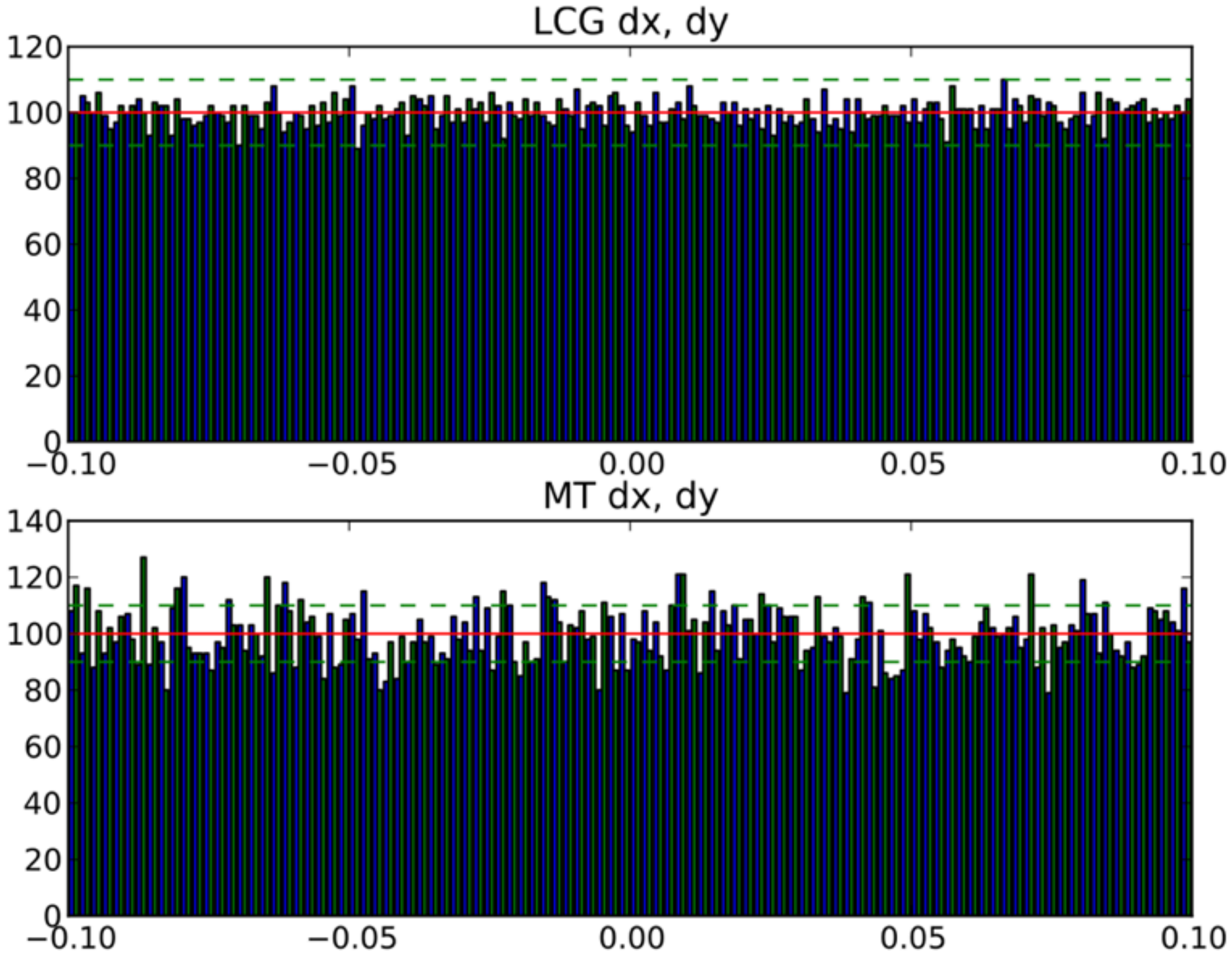
```
double doublesum(float *list, const int n) {

    double tot = 0.;

    for (int i=0; i<n; i++)
        tot += list[i];

    return tot;
}
```

HW1 - Summing

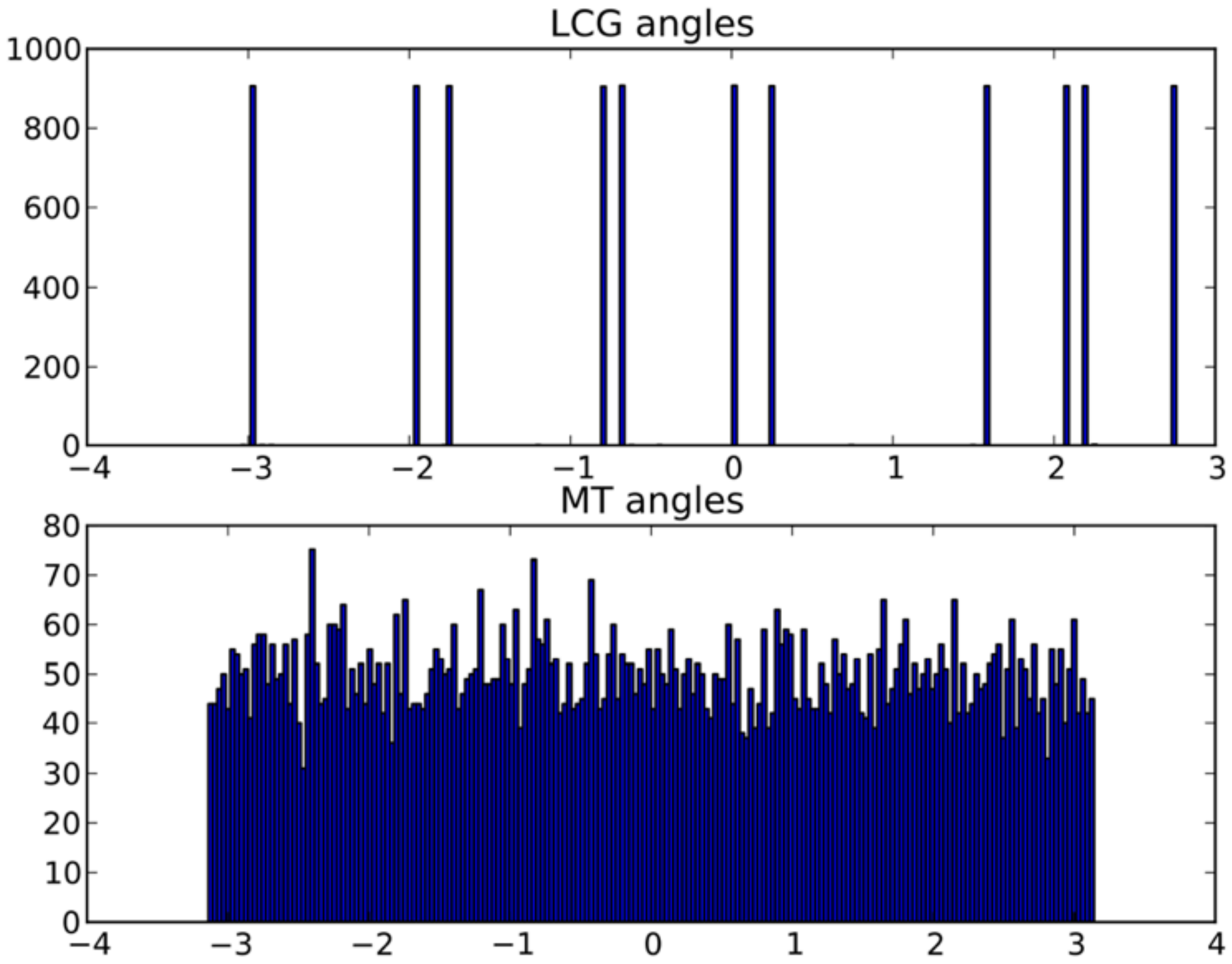
- Lots of ways around this:

```
$ ./part1
Left  sum:    1
Right sum:    1.25
Pairwise sum:2
Kahen sum:    2
Double precision sum:    2
```

HW1 - Random Walks



HW1 - Random Walks



HW1 - Seed

- Some issues with seeding
- General workflow; seed **once**, then generate all the random numbers you need.
- Showing how LCG worked may have confused things; seed was just last (integer) random deviate chosen

HW1 - Seed

- In general, current state of a PRNG can be quite large.
- Generally explicit functions to query state, set state (so can continue exactly where left off)
- Most PRNGs also have a convenience funciton to set state from small (~1 int) seed; bootstrap state from seed + smaller RNG
- Use once; *don't* keep seeding - don't know how it interacts with the PRNG

Numerical Linear Algebra

Scientific Computing Course
Mar 2013

Conclusion

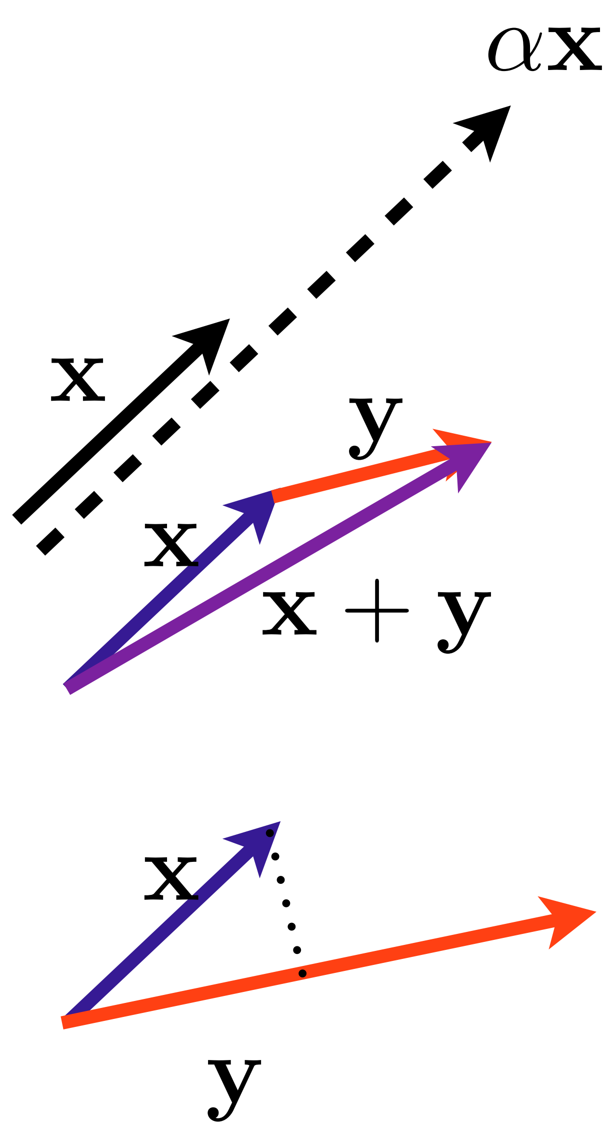
- Linear algebra pops up everywhere, even if you don't notice
- Statistics, data fitting, graph problems, PDE/ coupled ODE solves...
- There exist very highly tuned packages for any sort of problem that can be cast into matricies and vectors - use them
- LAPACK, BLAS
- Exploit structure in your matricies
- Don't ever invert a matrix

Outline

- Reminder of Linear Algebra
- Gaussian Elimination
- BLAS
- Solving Ax = b
- Sparse matricies
- Iterative solvers
- Eigenproblems

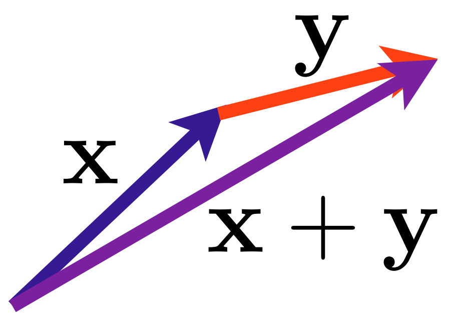
Vector operations

- Geometric Interpretation
- Scaling a vector, adding two vectors together...
- Dot product (or any inner product)



Vector spaces

- A set of vectors x spans a space S iff every vector in S can be expressed as a linear combination of x_i



Vector orthogonality - no overlap

- A set of vectors is said to be orthogonal if

$$x_i \cdot x_j \iff i \neq j$$

and orthonormal if

$$x_i \cdot x_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

- A set of n orthogonal vectors necessarily span a subspace of dimension n

Matrix · Vector: Change of Basis

$$A\mathbf{x} = \mathbf{b}$$

$$\left[\begin{array}{c|c|c|c} & & & \\ a_1 & a_2 & \ldots & a_n \\ & & & \end{array}\right] \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = x_1 \begin{pmatrix} \\ a_1 \\ \\ \end{pmatrix} + x_2 \begin{pmatrix} \\ a_2 \\ \\ \end{pmatrix} + \cdots + x_n \begin{pmatrix} \\ a_n \\ \\ \end{pmatrix}$$

Matrix · Vector: Change of Basis

- $Ax = b$: x is the (unique) vector of coefficients that represents b in the basis of columns of A
- Basis for b: $\{e_1, e_2, \ldots, e_n\}$
- Basis for x: $\{a_1, a_2, \ldots, a_n\}$

Column spaces

- Column space of A - the space spanned by the column vectors a_i
- eg, column space is all vectors that can be formed by linear combinations of the a_i

Matrix Vector: Linear mapping

- $Ax = b$: Linear transformation of x.
  - $Ax_1 = b_1$; $Ax_2 = b_2$
  - $A(x_1 + x_2) = (b_1 + b_2)$
  - $A(\alpha x_1) = \alpha b_1$

Range of A - all possible b

- The range of a matrix A is the space of all possible vectors it can map to:

$$b \in \text{Range}(A) \implies \exists x \mid Ax = b$$

eg, column space.

Nullspace of A: vectors that map to zero

- The nullspace of a matrix A is the space of all vectors it maps to zero:

$$\mathbf{x} \in \mathrm{Null}(\mathbf{A}) \implies \mathbf{A}\mathbf{x} = \mathbf{0}, \mathbf{x} \neq \mathbf{0}$$

- For matricies A with a non-empty nullspace, there may be no solution to Ax=b, or infinitely many solutions.

Column Rank: Dimension of Range

- The Rank of a matrix A is the dimension (eg, minimum number of basis vectors) of it's column space.
- For square (n×n) matrix, a Full-Rank matrix has rank n.
- Column rank = Row Rank (not obvious, but true.)   So generally just say "Rank"

Rank + Nullity

- Rank of Matrix
- + Nullity (rank of nullspace) of matrix
- = # of columns ofmatrix

Invertability

- Square, full-rank nxn matrix A has an inverse, A^{-1}, such that $A A^{-1} = A^{-1}A = I$
- For nxn matrix, following statements are equivalent:
  - Has an inverse
  - rank(A) = n
  - range(A) = R^n
  - null(A) = {}
  - No eigenvalues are 0
  - No singular values are 0
  - determinant is non-zero

Solving Linear Systems

Ax=b, solve for x

Sets of linear equations: don't invert

- $Ax = b$ implies $x = A^{-1}b$
- Mathematically true, but numerically, inversion:
  - is slower than other solution methods
  - is numerically much less stable
  - ruins sparcity (**huge** memory disadvantage for, eg, PDEs on meshes)
  - loses any special structure of matrix A

Easy systems to solve

- We'll talk about methods to solve linear systems of equations
- Will assume nonsingular matricies (so there exists a unique solution)
- But some systems much easier to solve than others. Be aware of "nice" properties of your matricies!

Diagonal Matrices

- (generally called D, or Λ)
- Ridiculously easy
- Matrix multiplication - just $d_i\ x_i$

$$\begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \cdots & \\ & & & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

$$x_i = \frac{b_i}{d_i}$$

Upper Triangular Matrices

- Generally called U
- "Back Substition": solve (easy) last one first
- Use that to solve previous one, etc.
- Lower triangular (L): "Forward substitution", same deal.

$$\begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ & u_{2,2} & \cdots & u_{2,n} \\ & & \cdots & \vdots \\ & & & u_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

$$x_n = \frac{b_n}{u_{n,n}}$$

$$x_{n-1} = \frac{b_n - u_{n-1,n} x_n}{u_{n-1,n-1}}$$

$$\vdots$$

Orthogonal matrices

- Generally called Q
- Columns (rows) are orthogonal unit vectors
- Transpose is inverse!
- *That* inverse I'll let you compute.
- Orthogonal matrices are numerically very nice - all row, col vectors are same "length".

$$\begin{aligned} Q^T Q &= I \\ Q\mathbf{x} &= \mathbf{b} \\ Q^T Q\mathbf{x} &= Q^T\mathbf{b} \\ \mathbf{x} &= Q^T\mathbf{b} \end{aligned}$$

Symmetric Matrices

- No special nomenclature
- Half the work; only have to deal with half the matrix
- (I'm assuming real matrices, here; complex: Hermetian)

$$A^T = A$$

$$a_{i,j} = a_{j,i}$$

Symmetric Positive Definite

- Very special but common (covariance matricies, some PDEs)
- Always non-singular
- All eigenvalues positive
- Numerically very nice to work with

$$A^T = A$$

$$\mathbf{x}^T A \mathbf{x} > 0$$

$$A = LL^T$$

Structure matters

- Find structure in your problems
- If writing equations in slightly different way gives you nice structure, do it
- Preserve structure when possible

Gaussian Elimination

- For general square matrices (can't exploit above properties)
- We all learned this in high school:
    - Subtract off multiples of previous rows to zero out below-diagonals
    - Back-subsitute when done

$$\begin{pmatrix} 10 & -7 & 0 \\ 5 & -1 & 5 \\ -2 & 2 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 6 \\ 4 \end{pmatrix}$$

$$\begin{pmatrix} 10 & -7 & 0 \\ & 2.5 & 5 \\ & 3.4 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ -0.5 \\ 2.6 \end{pmatrix}$$

$$\begin{pmatrix} 10 & -7 & 0 \\ & 2.5 & 5 \\ & & -0.8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ -0.5 \\ 3.28 \end{pmatrix}$$

Basic Linear Algebra Subroutines

- Linear algebra fairly simple: matricies and vectors
- Row vector operations, column vector operations, matrix-matrix operations
- BLAS: Basic Linear Algebra Subroutines.
    - Level 1: vector-vector operations
    - Level 2: matrix-vector operations
    - Level 3: matrix-matrix operations

Basic Linear Algebra Subroutines

- A well defined standard interface for these routines
- Many highly-tuned implementations exist for various platforms. (Atlas, Flame, Goto, PLASMA, cuBLAS...)
- (Interface vs. Implementation! Trick is designing a sufficiently general interface.)
- Higher-order operations (matrix factorizations, like as we'll see, gaussian elimiation) defined in LAPACK, on top of BLAS.

Typical BLAS routines

- Level 1: sdot (dot product, single), zaxpy (a**x** + **y,** dbl complex)
- Level 2: dgemv (dbl matrix*vec), dsymv (dbl symmetric matrix*vec)
- Level 3: sgemm (general matrix-matrix), ctrmm (triangular matrix-matrix)
- Incredibly cryptic names, interfaces.

Prefixes:

S: Single	C: Complex
D: Double	Z: Double Complex

Matrix Types:

GE: General	SY: Symmetric
GB: General Banded	SB: Symmetric Banded
HY: Hermetian	HB: Hermetian Banded
TR: Triangular	TB: Triangular Banded
TP: Triangular Packed	

Why bother?

- Finding, downloading library
- Figuring out how to link
- C/Fortran issues
- Just write it - it's not rocket science.

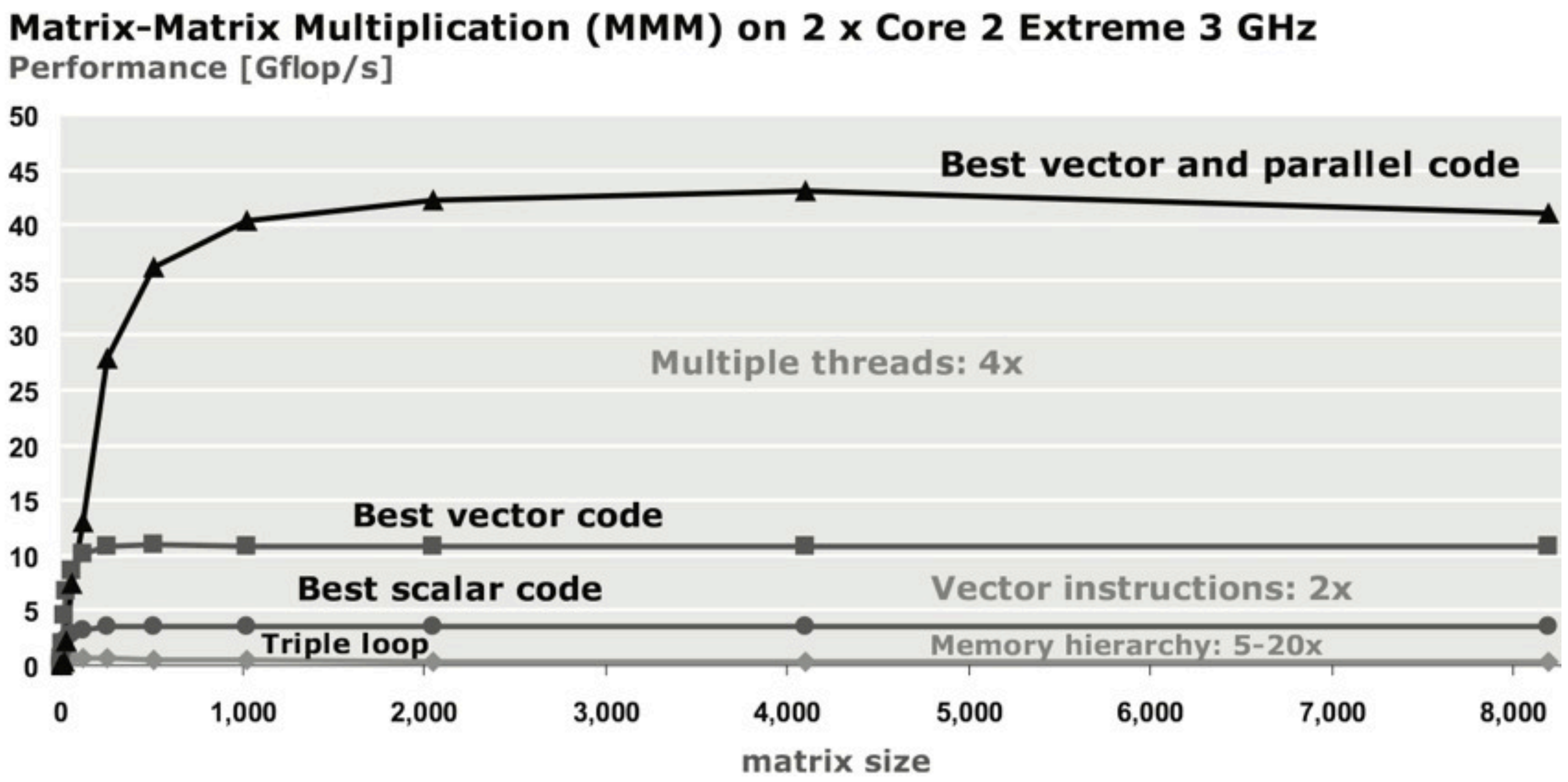
$$C = AB$$

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}$$

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            c[i][j] = a[i][k]*b[k][j];
```

SciNet compute • calcul CANADA

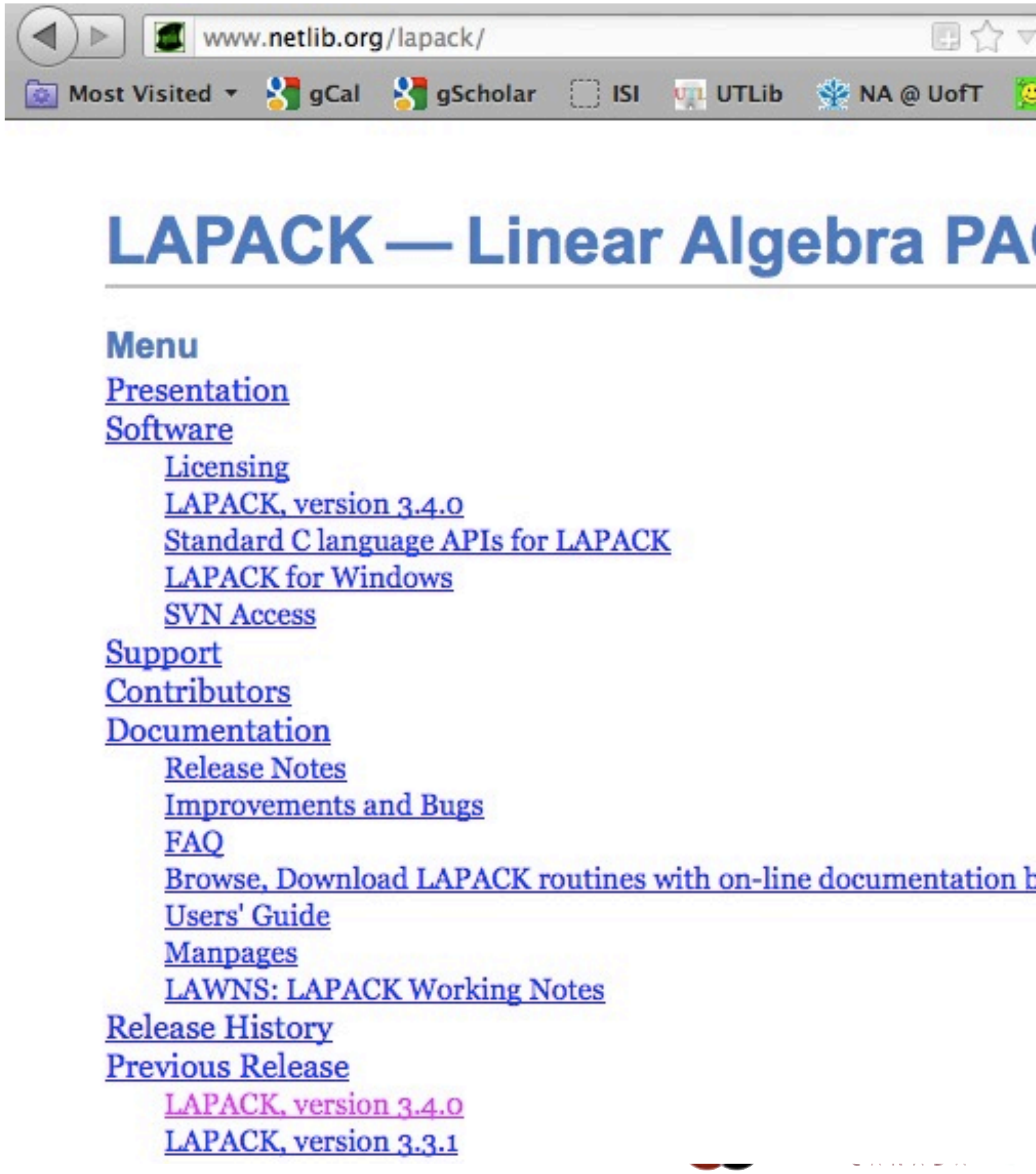
Never, ever, write your own



"How to Write Fast Numerical Code: A Small Introduction", Chellappa *et al*
<u>www.ece.cmu.edu/~franzf/papers/gttse07.pdf</u>

Division of Labour

- Focus on the science you need to do
- Write code for your problem - stuff that you know best
- Let people who enjoy making fast linear algebra software for a living do that.



LAPACK — Linear Algebra PA

Menu

Presentation

Software

- Licensing
- LAPACK, version 3.4.0
- Standard C language APIs for LAPACK
- LAPACK for Windows
- SVN Access

Support

Contributors

Documentation

- Release Notes
- Improvements and Bugs
- FAQ
- Browse, Download LAPACK routines with on-line documentation b
- Users' Guide
- Manpages
- LAWNS: LAPACK Working Notes

Release History

Previous Release

- LAPACK, version 3.4.0
- LAPACK, version 3.3.1

Gaussian Elimiation = LU Decomposition

- With each stage of the elimination, we were subtracting off some multiple of a previous row

- That means the factored U can have the same multiple of the row added to it to get back to A

- Decomposing to give us A = L U

$$\begin{pmatrix} 10 & -7 & 0 \\ 5 & -1 & 5 \\ -2 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix} \begin{pmatrix} 10 & -7 & 0 \\ 5 & -1 & 5 \\ -2 & 2 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 10 & -7 & 0 \\ 5 & -1 & 5 \\ -2 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 1 & & \\ +\frac{1}{2} & 1 & \\ -\frac{1}{5} & & 1 \end{pmatrix} \begin{pmatrix} 10 & -7 & 0 \\ & 2.5 & 5 \\ & 0.6 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 10 & -7 & 0 \\ 5 & -1 & 5 \\ -2 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 1 & & \\ -\frac{1}{2} & 1 & \\ +\frac{1}{5} & +\frac{6}{25} & 1 \end{pmatrix} \begin{pmatrix} 10 & -7 & 0 \\ & 2.5 & 5 \\ & & 4.8 \end{pmatrix}$$

$$A = LU$$

Solving is fast with LU

- Once have A = LU ($O(n^3)$ steps) can solve for x quickly ($O(n^2)$ steps)
- Can solve for same A with different b very cheaply
- Backsubstitute, then forward substitute

$$\begin{aligned}
A\mathbf{x} &= \mathbf{b} \\
LU\mathbf{x} &= \mathbf{b} \\
L(\mathbf{y}) &= \mathbf{b} \\
y &= \text{Backsubst}(L, \mathbf{b}) \\
U\mathbf{x} &= \mathbf{y} \\
x &= \text{Forwardsubst}(U, \mathbf{y})
\end{aligned}$$

Conditioning

- A problem is said to be inherently ill-conditioned if any small perturbation in the initial conditions generates huge changes in the results

- Say, calculating $f(x)$: if

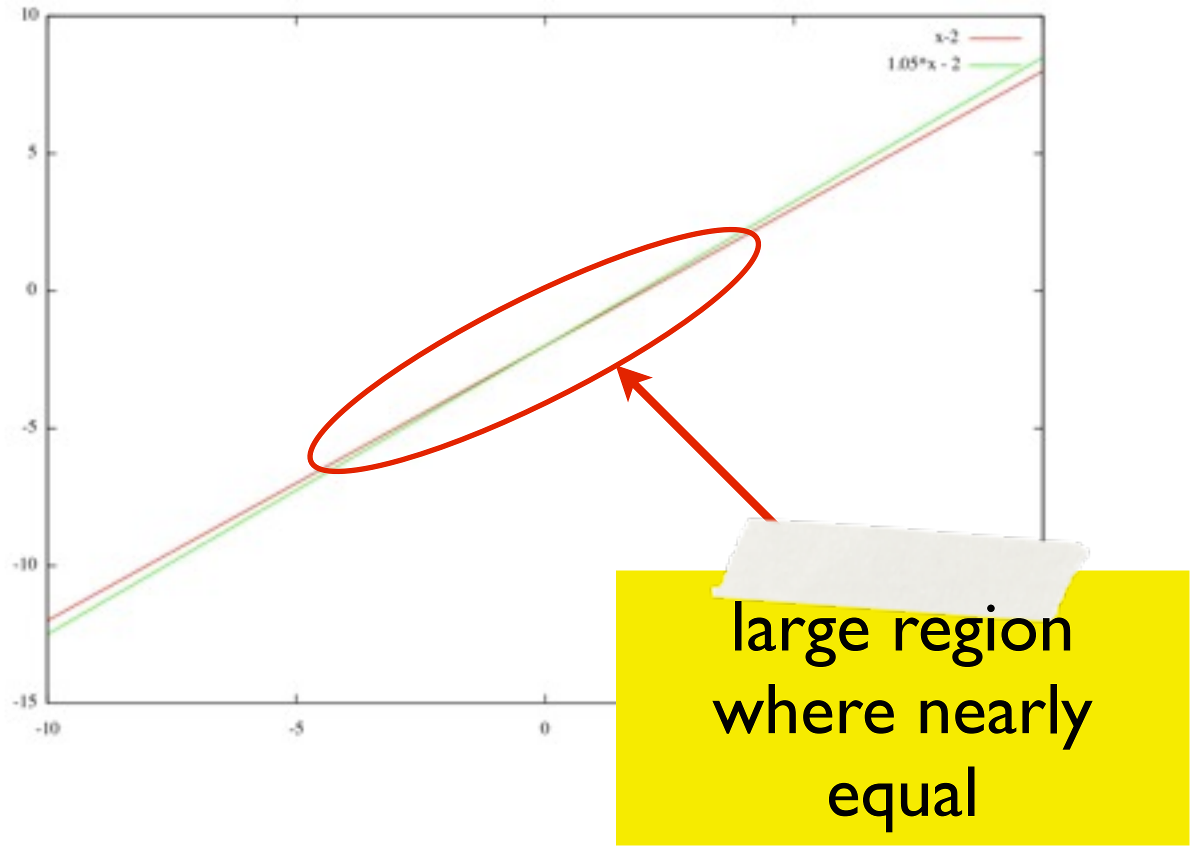
$$\frac{||f(x+\delta x)||}{||f(x)||} \gg \frac{||\delta x||}{||x||}$$

  then the problem is inherently hard to do numerically (or with any sort of approximate method)

Conditioning

- In matrix problems, this can happen in nearly singular matricies - nearly linearly dependant columns.
- Carve out strongly overlapping subspaces
- Very small changes in b (say) can result in hugely different change in x

$$\begin{pmatrix} 1 & 1 \\ 1 & 1.05 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$



Try it

- Order unity change in answer with 1/2 part in 10^{-4} change in input.
- Would be true with infinite-precision arithmetic.
- Inherently a tough problem.

```
$ ipython --pylab

In [1]: a = numpy.array([[1,1],
                [1,1.0001]])

In [2]: b = numpy.array([2,2])

In [3]: scipy.linalg.solve(a,b)
Out[3]: array([ 2.,   0.])

In [4]: scipy.linalg.solve(a,
           b+numpy.array([0,0.0001]))
Out[4]: ??
```

Condition number

- Condition number can be estimated using "sizes" (matrix norms) of A, inverse of A.
- Lapack routines exist: ___CON
- Relative error in x can't be less than condition number * machine epsilon.

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

$$\frac{||\delta x||}{||x||} < \kappa(A)\frac{||\delta b||}{||b||}$$

Residuals

- Computational scientists have over 20 words for "numerical error"
- Absolute, relative error - error in x.
- **Residual**: answer in result provided by erroneous x - error in b.
- Which is more important is entirely problem dependant

Pivoting

- The diagonal elements we use to "zero out" lower elements are called pivots.

- May need to change pivots, if for instance zeros appear in wrong place

- Matrix might be singular, or fixed by reordering

- PLU factorization

$$A = \begin{pmatrix} 0 & a & b \\ 0 & 0 & c \\ d & e & f \end{pmatrix}$$

Pivoting

- Important numerically, too - avoid catastrophic loss of precision.
- Consider 3 digits of decimal precision. Problem nowhere near singular
- What does scipy say?

$$\begin{pmatrix} 10^{-4} & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 10^{-4} & 1 \\ & 1 + 10^4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 + 10^4 \end{pmatrix}$$

$$\begin{pmatrix} 10^{-4} & 1 \\ & 10^4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 10^4 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Residuals

- Good linear algebra algorithms (and implementations) should give residuals no more than (some function of size of matrix) x (machine epsilon)

- And errors in x no more than condition number times that.

- An exact solution to a nearby problem

- Bad algorithms/implementations will depend on sqrt(machine epsilon) or worse, and/or will be matrix dependant (eg, LU without pivoting).

Cholesky Factorization

- For symmetric, positive definite matrices (surpisingly common), use Cholesky factorization instead.
- $A = LL^T$
- No pivoting; more numerically stable; faster.

```
In [10]: a =
numpy.array([[25,15,-5],
[15,18,0],[-5,0,11]])

In [11]:
scipy.linalg.cholesky(a)
Out[11]:
array([[ 5.,  3., -1.],
       [ 0.,  3.,  1.],
       [ 0.,  0.,  3.]])
```

A x ~ b : QR factorizations

- Not all Ax=b s can be solved; consider an overdetermined system (data fitting).
- LU won't even work on non-square systems.
- What to do?

$$\begin{pmatrix} x_0^3 & x_0^2 & x_0 & 1 \\ x_1^3 & x_1^2 & x_1 & 1 \\ \ldots & & & \\ x_n^3 & x_n^2 & x_n & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \ldots \\ y_n \end{pmatrix}$$

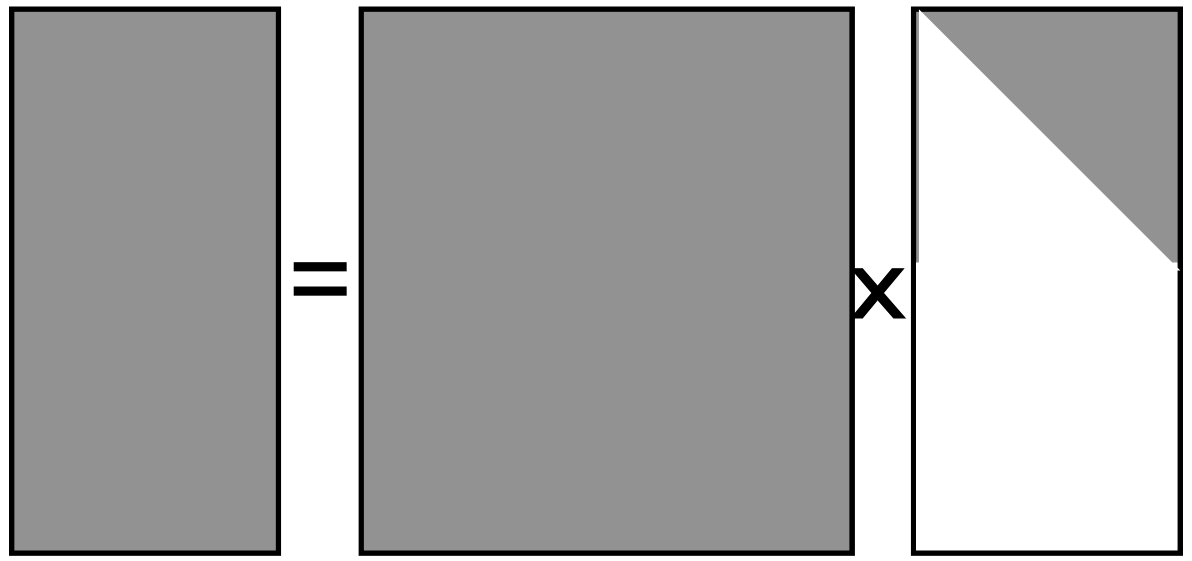
Minimize residual: Residual not in Range(A)

- Want to project out residual somehow
- Normal equations
- Much of linear algebra is decompositions into useful forms

$$\begin{aligned}
\mathbf{r}^2 &= ||\mathbf{b} - A\mathbf{x}||_2^2 \\
&= (\mathbf{b} - A\mathbf{x})^T(\mathbf{b} - A\mathbf{x}) \\
&= \mathbf{b}\cdot\mathbf{b} - 2\mathbf{b}^T A\mathbf{x} + \mathbf{x}^T A^T A\mathbf{x} \\
0 &= -2\mathbf{b}^T A + 2\mathbf{x}^T A^T A \\
(A^T A)\mathbf{x} &= A^T\mathbf{b}
\end{aligned}$$

QR decomposition

- All matrices can be decomposed into QR, even mxn, m>n
- Bottom half of R is necessarily empty (below diagonal)
- All columns in Q are orthogonal bases of m-d space, and R is the combination of them that makes up A



Orthogonalizing columns of A

- Let's take these n column vectors of length m and make an orthonormal basis.
- Divide a1 by its norm; 1 done. What about rest?

$$\left[\begin{array}{c|c|c|c} a_1 & a_2 & \ldots & a_n \end{array}\right]$$

$$\left[\begin{array}{c} q_1 \end{array}\right] = \frac{\mathbf{a_1}}{||\mathbf{a_1}||}$$

Gram-Schmidt (don't use this)

- Easiest to follow at first isn't numerically stablest (should use Householder transforms).
- Subtract off q_1 component from a2, take unit vector of that - q_2.
- And so on.
- Bit like LU, but instead of making zeros, you're making orthogonality

$$\left[\begin{array}{c|c|c|c} a_1 & a_2 & \ldots & a_n \end{array}\right]$$

$$\left[\begin{array}{c} q_2 \end{array}\right] = \frac{\mathbf{a_2} - (\mathbf{a_2} \cdot \mathbf{q_1})\,\mathbf{q_1}}{||\mathbf{a_2} - \mathbf{a_2} \cdot \mathbf{q_1}||}$$

Gram-Schmidt (don't use this)

- Gram-Schmidt handy for generating orthgonal series of basis functions from (say) polynomials, as well.
- Same procedure, just different definition of inner product, norm.

$$\left[\begin{array}{c|c|c|c} a_1 & a_2 & \ldots & a_n \end{array}\right]$$

$$\left[\begin{array}{c} q_2 \end{array}\right] = \frac{\mathbf{a_2} - (\mathbf{a_2} \cdot \mathbf{q_1})\,\mathbf{q_1}}{||\mathbf{a_2} - \mathbf{a_2} \cdot \mathbf{q_1}||}$$

SciNet compute • calcul CANADA

QR Factor a random matrix

```
In [13]: r = numpy.random.random((50,50))

In [14]: for i in xrange(50):
   ....:     for j in xrange(i):
   ....:         r[i,j] = 0.
   ....:

In [15]: print r[0:3,0:3]
[[ 0.4147775   0.64843642  0.41133882]
 [ 0.          0.88592831  0.54711704]
 [ 0.          0.          0.23438925]]

In [16]: q,x = scipy.linalg.qr(numpy.random.random((50,50)))

In [17]: a = numpy.dot(q,r)

In [18]: q2,r2 = scipy.linalg.qr(a)

In [19]: a2 = numpy.dot(q2,r2)

In [20]: print scipy.linalg.norm(a2-a)/scipy.linalg.norm(a)
6.60894445883e-16
```

Errors and residuals

- Generate random matrices Q,R; calculate A
- QR factorization of A
- Errors in Q2, R2 ~ sqrt(machine epsilon)
- (Random matrix tends to be ill-conditioned)
- Residual in A: (machine epsilon). Would be sqrt with classical G-S

```
In [18]: q2,r2 = scipy.linalg.qr(a)

In [19]: a2 = numpy.dot(q2,r2)

In [20]: print scipy.linalg.norm(a2-a)/
                    scipy.linalg.norm(a)
6.60894445883e-16

In [21]: print scipy.linalg.norm(q2-q)/
                    scipy.linalg.norm(q)
3.67030163525e-07

In [22]: print scipy.linalg.norm(r2-r)/
                    scipy.linalg.norm(r)
6.36755093518e-08
```

Normal equations with QR are easy

- Now this is fairly straightforward
- End up with (Rx) -- forward solve -- equal to matrix-vector product.
- Done!

$$
\begin{aligned}
(A^T A)\mathbf{x} &= A^T \mathbf{b} \\
R^T Q^T Q R \mathbf{x} &= R^T Q^T \mathbf{b} \\
R^T R \mathbf{x} &= R^T Q^T \mathbf{b} \\
R \mathbf{x} &= Q^T \mathbf{b}
\end{aligned}
$$

Eigenproblems

$$A\mathbf{x} = \lambda\mathbf{x}$$

- Tells a great deal about the structure of a matrix
- How it will act on a vector: project onto its eigenvectors, mutiply by eigenvalues.
- Goal is a complete decomposition:

$$A \left[\begin{array}{c|c|c|c} x_1 & x_2 & \ldots & x_n \end{array}\right] = \left[\begin{array}{c|c|c|c} x_1 & x_2 & \ldots & x_n \end{array}\right] \begin{bmatrix} \lambda & & & \\ & \lambda & & \\ & & \ddots & \\ & & & \lambda \end{bmatrix}$$

Eigenvalue Decomposition

- For square matrix
- "Similarity Transform"
- No restrictions on the structure of X
- Can only happen if there are a full set of eigenvectors.
- Diagonalizability: N non-null eigenvectors;
- Invertability: N non-zero eigenvalues

$$A \left[\begin{array}{c|c|c|c} x_1 & x_2 & \dots & x_n \end{array} \right] = \left[\begin{array}{c|c|c|c} x_1 & x_2 & \dots & x_n \end{array} \right] \begin{bmatrix} \lambda & & & \\ & \lambda & & \\ & & \ddots & \\ & & & \lambda \end{bmatrix}$$

$$AX = X\Lambda$$

$$A = X\Lambda X^{-1}$$

Defective Matrices

- Both these matrices have eigenvalue 2, with multiplicity 3
- But A has full set of eigenvectors (e_1,e_2,e_3)
- B has only one eigevector; e_1
- Not diagonalizable

$$A = \begin{pmatrix} 2 & & \\ & 2 & \\ & & 2 \end{pmatrix}$$

$$B = \begin{pmatrix} 2 & 1 & \\ & 2 & 1 \\ & & 2 \end{pmatrix}$$

Iterative Methods

- So far, have dealt solely with direct methods.
- Solution takes one (long) step, then answer is complete, as exact as matrix/method allows.
- Other approach; take successive approximations, get closer.
- Typically converge to machine accuracy in much less time than direct, esp for large matricies

Krylov Subspaces

- Krylov subspace: repeated action on b by A.
- For sufficiently large n, final term should converge to eigenvector with largest eigenvalue
- But slow, and only one eigenvalue?

$$A\mathbf{x} = \mathbf{b}$$

$$\mathcal{K} = [\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \cdots, A^{n-1}\mathbf{b}]$$

Krylov Subspaces

- Can orthogonalize (Gram Schmidt, Householder) to project out other components
- Should give approximations to eigenvectors (random b)
- But not numerically stable

$$A\mathbf{x} = \mathbf{b}$$

$$\mathcal{K} = [\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \cdots, A^{n-1}\mathbf{b}]$$

Arnoldi Iteration

- Stabilized orthogonalization
- Becomes Lanczos iteration for symmetric A
- Orthogonal projection of A onto the Krylov subspace, H
- H is of modest size, can have eigenvalues calculated
- Note: Only requires matrix-vector, vector-vector products
- GMRES: Arnoldi iteration for solving Ax=b

```
$q_1 \leftarrow e_1$
 for $j \in [1, k-1]$ :
        $h_{j,k-1} \leftarrow q_j^T q_k$
        $q_k \leftarrow q_k - h_{j,k-1} q$
$h_{k,k-1} \leftarrow ||q_k||$
$q_k \leftarrow \frac{q_k}{h_{k,k-1}}$
```

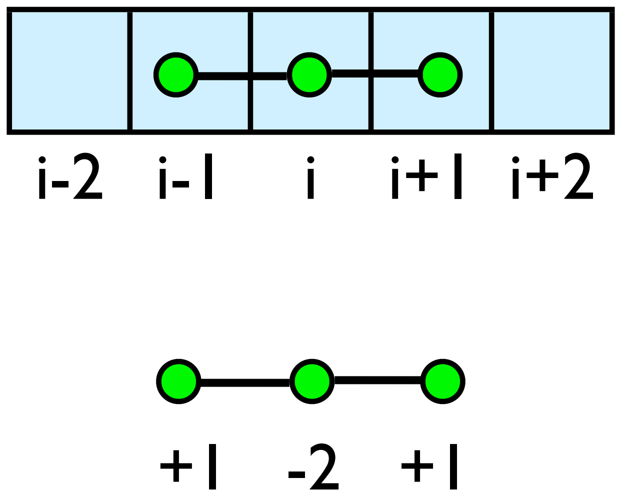
Sparse Matricies

- So far, we've been assuming our matrices are dense; there are numbers stored for every entry in matrix.
- This is indeed often the case, but it's also often that huge numbers of the entries are zero: some roughly constant number of entries per row, much less than n.
- Difference between n^2 and n can be huge if $n \sim 10^6$; difference between doing and not doing the problem.
- Happens particularly often in discretizing PDEs.

Discretizing Derivatives

$$\left.\frac{d^2q}{dx^2}\right|_i \approx \frac{q_{i+1} - 2q_i + q_{i-1}}{\Delta x^2}$$

- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating data and taking derivative of the interpolant



$$\frac{d\mathbf{q}}{dt} = \sigma \begin{pmatrix} -2 & 1 & & & & \\ & 1 & -2 & 1 & & \\ & & & \cdots & & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{pmatrix} \mathbf{q}$$

$$\frac{d\mathbf{q}}{dt} \approx \frac{\mathbf{q^{n+1}} - \mathbf{q^{n}}}{\Delta t}$$

$$\mathbf{q^{n+1}} = \mathbf{q^{n}} + \sigma \mathbf{\Delta t A q^{n}}$$

$$\mathbf{q^{n+1}} = \left(\mathbf{I} + \sigma \mathbf{\Delta t A}\right) \mathbf{q^{n}}$$

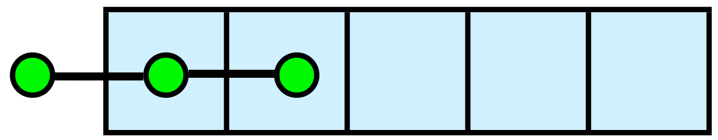
$$\mathbf{q^{n+1}} = \sigma \mathbf{\Delta t} \begin{pmatrix} \frac{1}{\sigma \Delta t} - 2 & 1 & & & & \\ & 1 & \frac{1}{\sigma \Delta t} - 2 & 1 & & \\ & & & \cdots & & \\ & & & 1 & \frac{1}{\sigma \Delta t} - 2 & 1 \\ & & & & 1 & \frac{1}{\sigma \Delta t} - 2 \end{pmatrix} \mathbf{q^{n}}$$

SciNet compute • calcul CANADA

Boundary Conditions

$$\left.\frac{d^2q}{dx^2}\right|_i \approx \frac{q_{i+1} - 2q_i + q_{i-1}}{\Delta x^2}$$

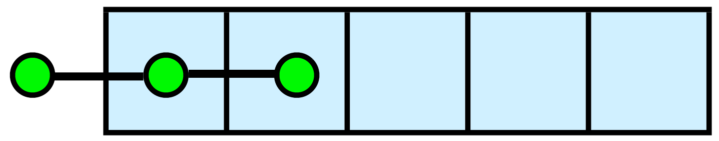
- What happens when stencil goes off of the end of the box?
- Depends on how you want to handle boundary conditions.
- Typically easiest to have extra points on end, set values to enforce desired BCs.



Boundary Conditions

$$\left.\frac{d^2q}{dx^2}\right|_i \approx \frac{q_{i+1} - 2q_i + q_{i-1}}{\Delta x^2}$$

- Dirichlet (fixed value) boundary conditions: just have 1 on diagonal, 0 elsewhere, keeps value there constant.
- Neumann (derivitave) bcs: requires more manipulation of the equations.



Inverses destroy sparsity

- For sparse matrices like above, LU decompositions may maintain much sparsity (particularly if banded, etc)

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

$$\begin{pmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}^{-1} = \begin{pmatrix} 5 & 4 & 3 & 2 & 1. \\ 4 & 4 & 3 & 2 & 1. \\ 3 & 3 & 3 & 2 & 1. \\ 2 & 2 & 2 & 2 & 1. \\ 1 & 1 & 1 & 1 & 1. \end{pmatrix}$$

- Inverses in general are full

- For large n, difference beween cn and n^2 huge.

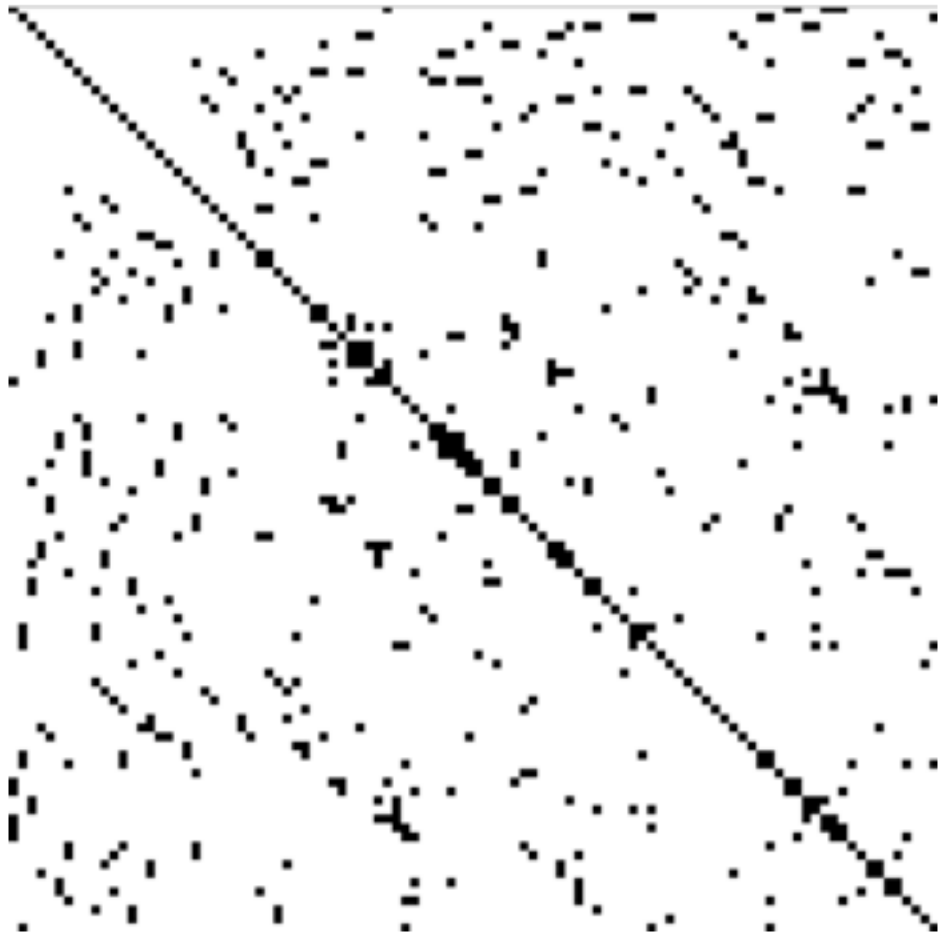
Sparse (banded) LU

- If entries only exist within a narrow band around diagonal, then row, column operations fast.
- May get significant "fill in" depending on exact structure of matrix
- (This is artificially good example)

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

Sparsity patterns

- Sparse matrices can have arbitray sparsity patterns
- Typically need at less than 10% nonzeros to make dealing with sparse matricies worth it.
- Half zeros - typically just store full matrix.



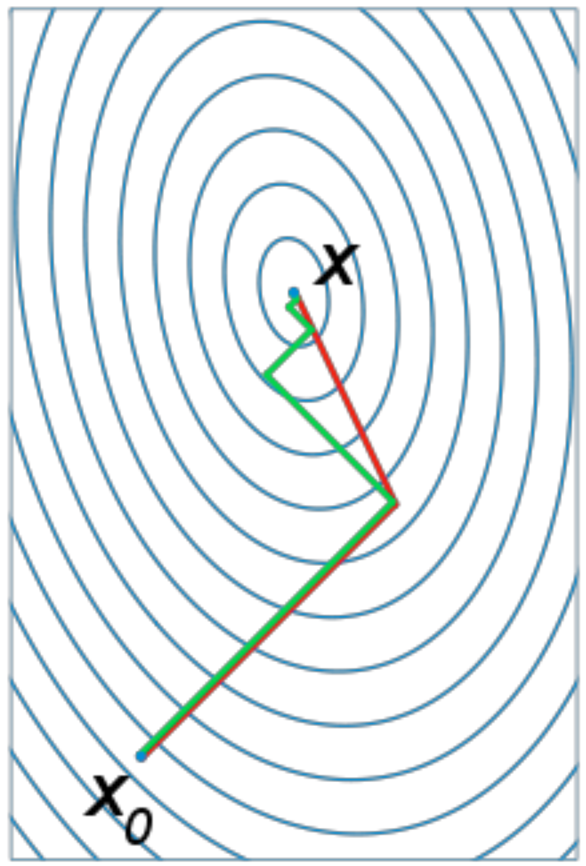
http://en.wikipedia.org/wiki/File:Finite_element_sparse_matrix.png

Common Sparse Matrix Formats:

- CSR (Compressed Sparse Row): Just join all the nonzeros in rows together, with pointers to where each starts, and (similar sized) array of column for each value
- CSC (Compressed Sparse Column): Same, but flip row/column
- Banded: just store diagonals +/- some bandwidth
- Many many more.

Iterative Ax=b solvers: Conjuate Gradient

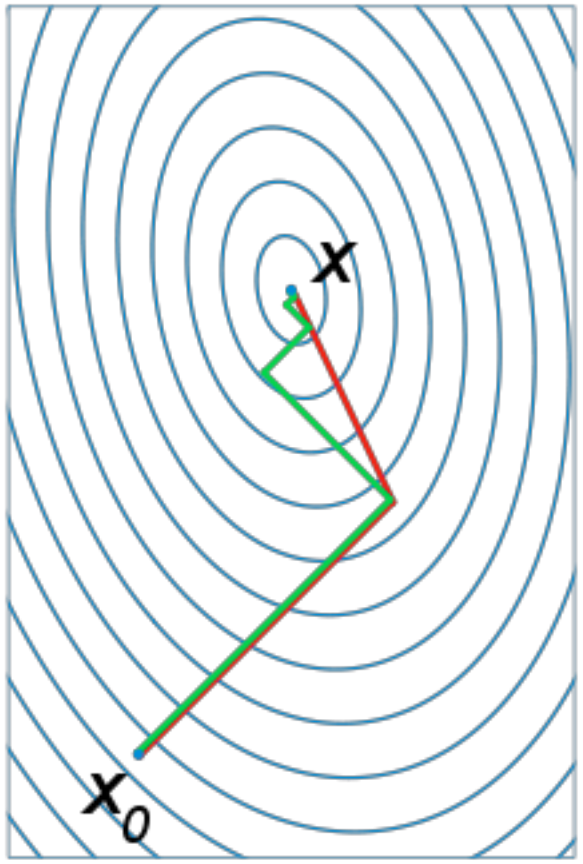
- SPD matrices, works particularly well on sparse systems
- "Steepest Descent", but only on conjugate (w/rt A) directions: no "doubling back"



http://en.wikipedia.org/wiki/Conjugate_gradient_method

Conjugate Gradient Method

```
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
**repeat**
    $\alpha_k := \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$
    **if** $\mathbf{r}_{k+1}$ is sufficiently small **then** exit loop **end if**
    $\beta_k := \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
    $k := k + 1$
**end repeat**
```



http://en.wikipedia.org/wiki/Conjugate_gradient_method

Resources

- Trefethen & Bau, "Numerical Linear Algebra"
  http://people.maths.ox.ac.uk/trefethen/text.html

- Strang on ITunes U: "Mathematical Methods for Engineers" or "Linear Algebra" - excellent lectures by a master.

Homework

Educational *and* fun.

Homework: Part I

- The time-explicit formulation of the 1d heat diffusion equation has a term that looks like this (ignoring boundary conditions)

$$\frac{D\Delta t}{\Delta x^2}\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & & \cdots & & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{pmatrix} x^n$$

Homework: Part 1

- Ignoring the constants, what are the eigenvalues for this problem - what might we expect to get amplified/damped by this operator? (use 100 points; D__EV)
- Plot the eigenmode with the largest and smallest absolute eigenvalues, and explain them.
- Use the largest abs. eigenvalue to put a constraint on dt given dx, D. This is a stability constraint on the numerical method; for larger timesteps, method blows up.

Lapack Hints

- If you are using an nxn array, the "leading dimension" of the array is n. (This argument is so that you could work on sub-matrices if you wanted)
- Have to make sure the 2d array is contiguous block of memory
- C vs FORTRAN array orderings
- C bindings for LAPACK - lapacke

Homework: Part 2

- For a 1d grid of size 100 (eg, a 100x100 matrix A), using lapack, evolve this PDE. Plot and explain results.
- Have an initial condition where x = 1 at the first zone, and zero everywhere else (hot plate “turns on” in a cold domain.
- You’ll want to use driver routines for linear solves (http://www.netlib.org/lapack/lug/node26.html). Do solve in double precision (D__SV). Which solver should you use?
- Using a small enough timestep, timestep the temperature evolution by finding dT. Do solution in double precision (D__SV) .