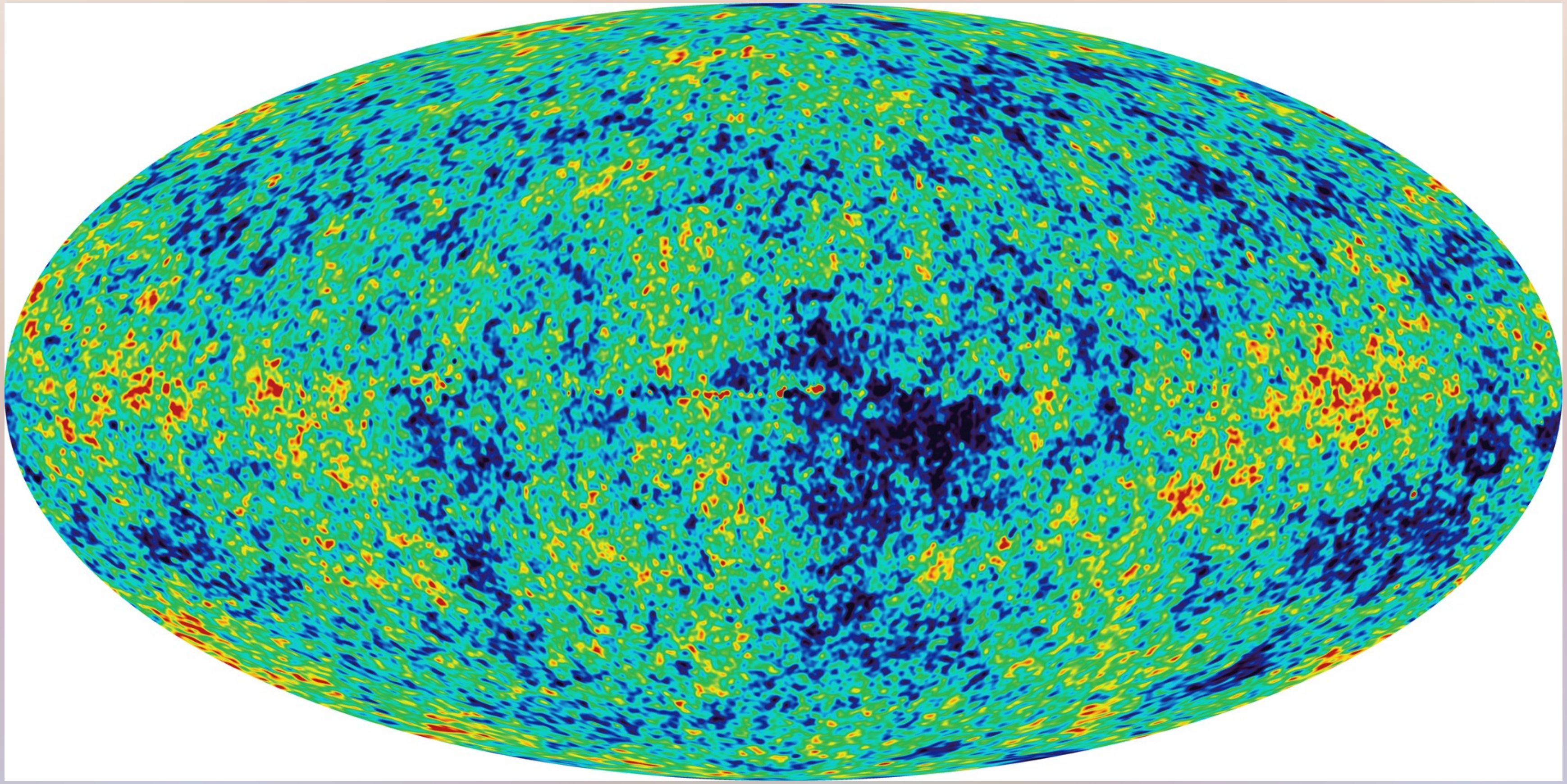
Numerical Linear Algebra

- Useful in a wide range of applications, wherever physical systems can be described with linear equations – linear response systems, e.g.:
  - Reference frame transformations in both classical physics and special relativity.
  - Electric circuits, DC/AC (Kirchoff's laws).
  - Quantum mechanical states are vectors and linear combinations of vectors.
- Furthermore, linear algebra is often useful also in other numerical calculations. We will have various examples of this throughout this course (e.g. interpolation, least squares fitting, ODEs, ...)

WMAP sky maps: linear algebra in action



Map making requires several (very large) matrix multiplications and inversions.

Linear algebra

Computational linear algebra is a very wide field, we can cover only a part of it. Examples include:

solving systems of linear equations $Ax=b$ for x

solve the above for single A and many b

compute inverse of matrix and determinant

different decompositions of matrices (e.g. LU, SVD)

dealing with over- and under-determined systems of equations, linear least-squares problem

computing eigenvalues and eigenvectors of matrices

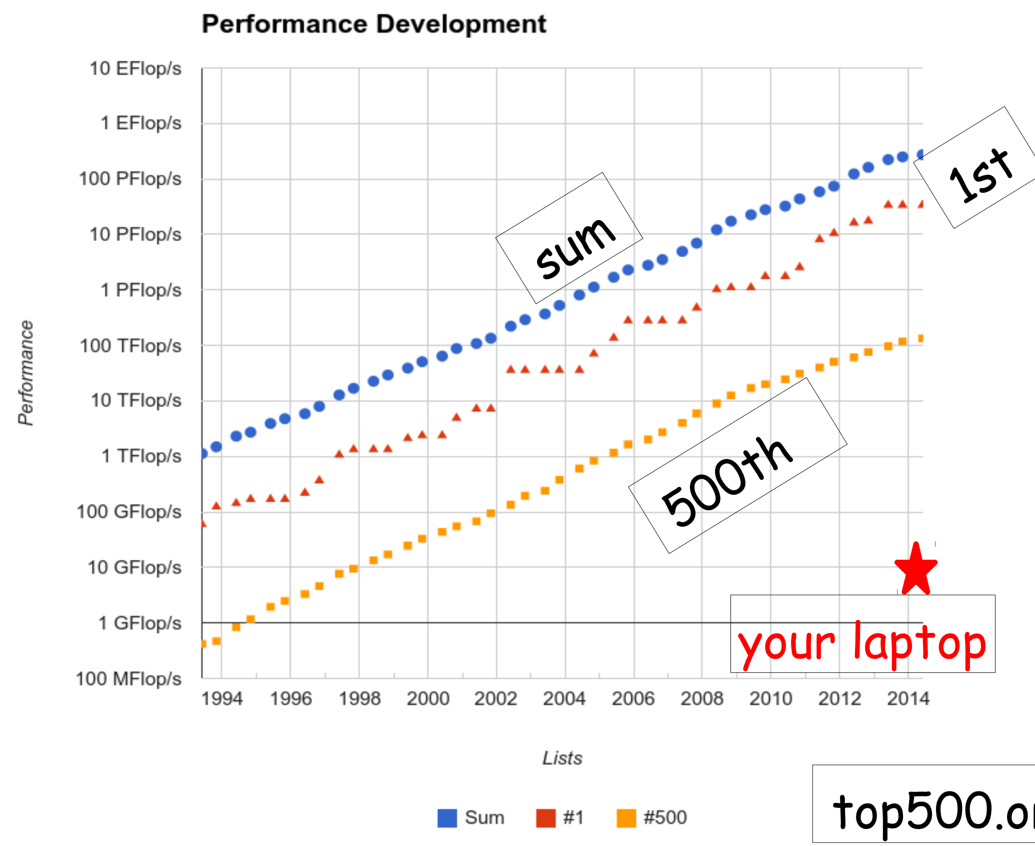
"old" problem, large packages exist, e.g. LINPACK/LAPACK, much work has been done on optimisation and parallelization

Linear algebra as benchmark

Q: How do we know which are the fastest, most powerful computers today?

Top 500 supercomputer benchmark uses LINPACK implementation of Gauss elimination w/partial pivoting. (discussed in our next lecture)

→ Moore's law



top500.org

Linear systems of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad & \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned}$$

or

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

- Can be written in the matrix form Ax=b

  where: b=input, x=system response, A=physical system (independent of input).

- Solution exists if |A| is not 0 (number of independent equations = number of unknowns), i.e. when matrix A is non-singular.

Linear systems of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad &\quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned}$$

or

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

- Can be written in

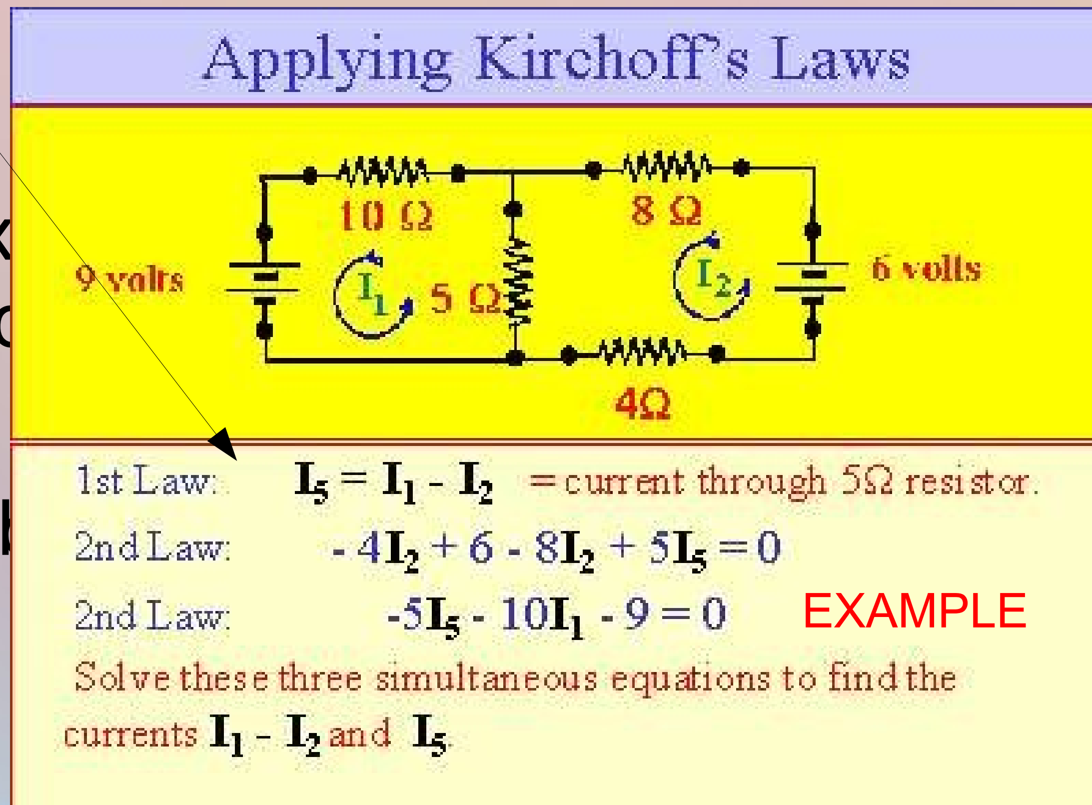
  where: b=input, x

  system (independ

- Solution exists if ... dent

  equations = numb ... rix A

  is non-singular.



EXAMPLE

Solving linear systems

- Direct methods – transform the system into an equivalent one which is easier to solve by:
  - Exchanging two equations (changes sign of |A|).
  - Multiplying an equation by a non-zero constant (multiplies |A| by the same constant).
  - Multiplying an equation by a nonzero constant and then subtracting it from another equation (leaves |A| unchanged).
- Iterative methods – start from a guess and then improve and refine it until convergence.

Gauss elimination

We want to solve the linear system Ax=b, or:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad & \qquad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned}$$

Gauss elimination: use first equation to remove the x_1 term from all other equations by subtracting $\lambda=a_{j\,1}/a_{1\,1}$ times the first equation from the j-th equation (j>1):

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a'_{22}x_2 + \ldots + a'_{2n}x_n &= b'_2 \\ &\vdots \\ a'_{n2}x_2 + \ldots + a'_{nn}x_n &= b'_n \end{aligned}$$

Gauss elimination

This process is called "Gauss elimination". We can then repeat it with the 2nd to nth sub-matrix:

$$\text{Eq. } (i) \leftarrow \text{Eq. } (i) - \lambda \times \text{ Eq. } (j)$$

Finally resulting in:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= b_1 \\
a'_{22}x_2 + a'_{23}x_3 + \ldots + a'_{2n}x_n &= b'_2 \\
a''_{33}x_3 + \ldots + a''_{3n}x_n &= b''_3 \\
&\vdots \\
a'''_{nn}x_n &= b'''_n
\end{aligned}$$

→ we end up with an (upper) triangular matrix -- (forward) elimination phase, followed by back-substitution phase: solve last equation for x_n and work your way back up.

Example

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 2 & 3 \\ 3 & 3 & 3 \end{pmatrix} \qquad b = \begin{pmatrix} 6 \\ 7 \\ 9 \end{pmatrix}$$

on board

Gaussian elimination: algorithm

$$\left[\begin{array}{cccccccccc|c}
A_{11} & A_{12} & A_{13} & \cdots & A_{1k} & \cdots & A_{1j} & \cdots & A_{1n} & & b_1 \\
0 & A_{22} & A_{23} & \cdots & A_{2k} & \cdots & A_{2j} & \cdots & A_{2n} & & b_2 \\
0 & 0 & A_{33} & \cdots & A_{3k} & \cdots & A_{3j} & \cdots & A_{3n} & & b_3 \\
\vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
0 & 0 & 0 & \cdots & A_{kk} & \cdots & A_{kj} & \cdots & A_{kn} & & b_k \\
\hline
\vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
0 & 0 & 0 & \cdots & A_{ik} & \cdots & A_{ij} & \cdots & A_{in} & & b_i \\
\vdots & \vdots & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
0 & 0 & 0 & \cdots & A_{nk} & \cdots & A_{nj} & \cdots & A_{nn} & & b_n
\end{array}\right]$$

$$A_{ij} \leftarrow A_{ij} - \lambda A_{kj}, \quad j = k, k+1, \ldots, n$$

$$b_i \leftarrow b_i - \lambda b_k$$

Elimination phase

Gaussian elimination: algorithm

$$x_n = b_n / A_{nn}$$

Back substitution phase

$$x_k = \left(b_k - \sum_{j=k+1}^{n} A_{kj} x_j \right) \frac{1}{A_{kk}}, \quad k = n-1, n-2, \ldots, 1$$

Gauss Elimination: code

```
## module gaussElimin
''' x = gaussElimin(a,b).
    Solves [a]{b} = {x} by Gauss elimination.
'''
from numpy import dot

def gaussElimin(a,b):
    n = len(b)
  # Elimination Phase
    for k in range(0,n-1):
        for i in range(k+1,n):
           if a[i,k] != 0.0:
               lam = a [i,k]/a[k,k]
               a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
               b[i] = b[i] - lam*b[k]
  # Back substitution
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

Performance of Gauss elimination

Rough estimate:

forward elimination: 3 loops → n^3

back-substitution: 2 loops → n^2

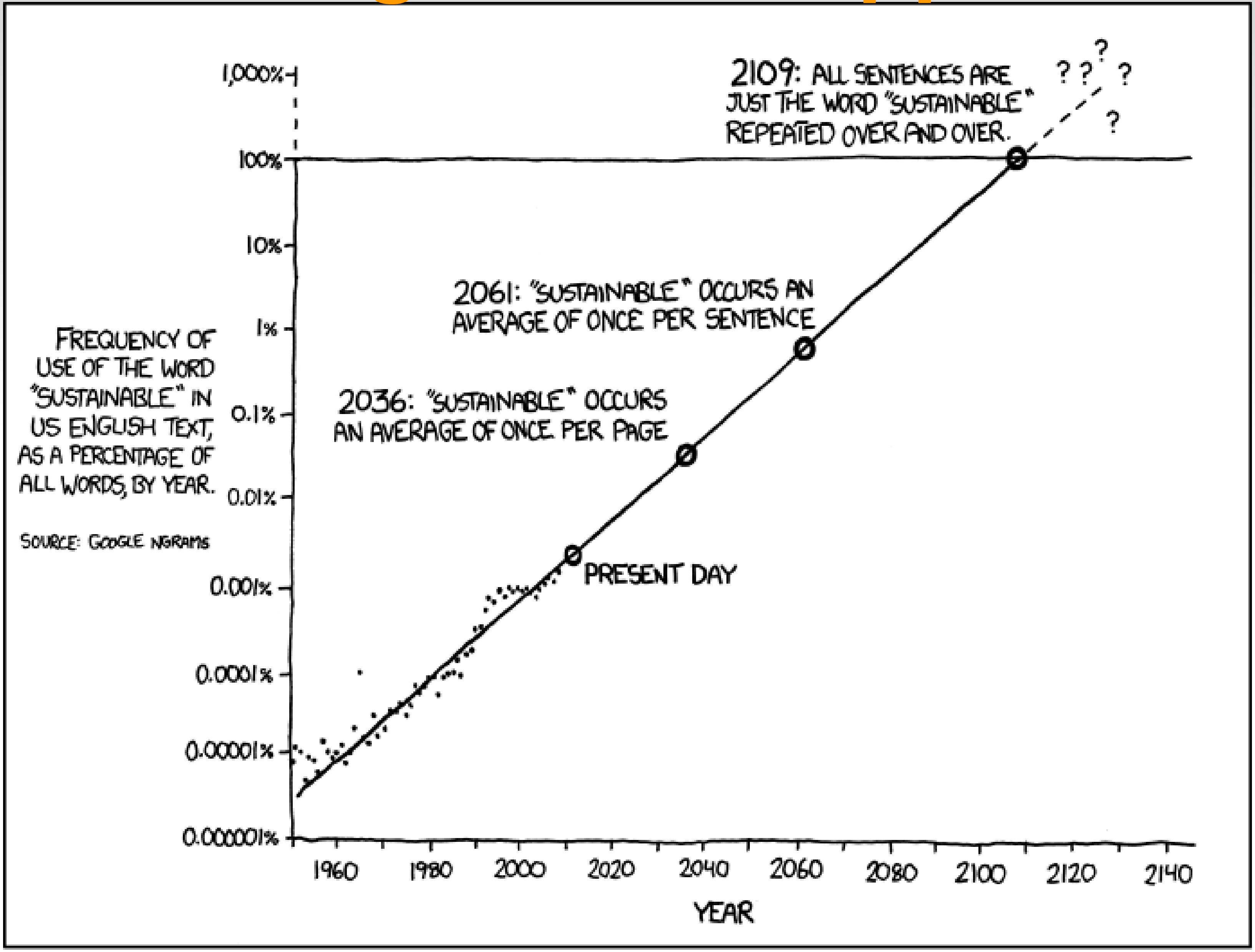
More detailed analysis: $n^3/3$ multiplications and $n^2/2$ summations in total.

→ $O(n^3)$ → $n<1000$ easy on a PC

For a general matrix no better scaling is possible – can only hope to improve pre-factor (and numerical stability)

Very brute-force methods may not work, e.g. Cramer's rule scales as $n!$ → already $n=20$ is numerically impossible!

Linear Algebra: an Application

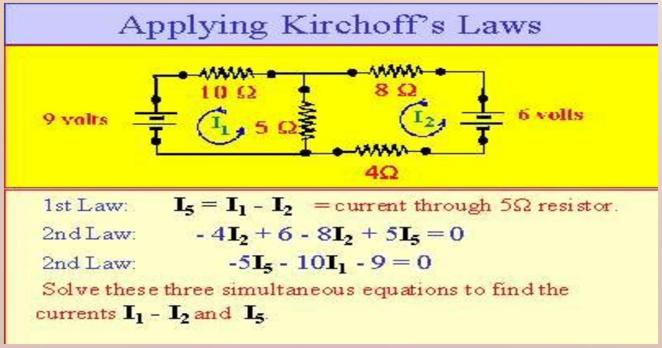


THE WORD "SUSTAINABLE" IS UNSUSTAINABLE.

LU factorization

- Often we have to solve a linear system Ax=b for the same A (i.e. physical system), but different b's (inputs).

  Q: What does this correspond to in our Kirchoff's Laws example?



- Solving the system again as a new one is possible, but is not the most efficient approach.
- Instead, we can re-use the expensive bit of the first solution, while getting the rest cheaply → LU factorization:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

LU factorisation: solving the equations

Any square matrix A can be expressed as a product of a lower- and an upper-triangular matrix, such that A=LU. This is called LU factorisation. If A is nonsingular, so are L and U. If the split is available, obtaining the solution of Ax=b is done as follows:

1) split A=LU (the expensive bit, $\sim n^3$ oper., independent of b)

2) solve Lz=b (cheap, $\sim n^2$ operations)

3) solve Ux=z, (cheap, $\sim n^2$ operations)

then, Ax=LUx=Lz=b, so we have the solution. The two triangular systems are easy to solve:

$l_{11}z_1=b_1 \rightarrow$ find z_1, then back-substitute in previous equations. Same for solving Ux=z, once we know z.

LU factorisation: method

How do we do the LU factorization itself? We have already done it during Gauss elimination!
$\rightarrow$ Just requires us to store the Gauss multipliers λ.

U is just the usual upper triangular matrix from the forward Gauss elimination.

L has 1 on diagonal and $L_{j\,i} = \lambda_{j\,i}$ where λ_{ji} is the multiplier required to eliminate $A_{j\,i}$. Just overwrite $A_{i\,j}$ with λ instead of setting to zero in Gauss elimination algorithm.

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \longrightarrow [\mathbf{L}\backslash\mathbf{U}] = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & U_{33} \end{bmatrix}$$

LU factorisation: expense

How many operations does LU factorization require?

Same as the Gaussian elimination!

Count them and remember that elimination is $O(n^3)$, while back-substitution is $O(n^2)$. We do same operation in each case, just in a different order.

Each additional solution of LUx=b for a new 'b' requires just $O(n^2)$ operations, i.e. is relatively cheap (for large n, anyway).

LU factorisation: notes and uses

The LU factorization of a matrix A is not unique, the one we discussed is called “Doolittle reduction”. Others are discussed in the book.

A special case: for symmetric and positive-definite matrix: can choose L = U’, A = L L^T $\rightarrow$ Cholesky factorisation.

Uses of LU:

inverse X of matrix A: $\sum_j A_{ij} X_{jk} = \delta_{ik}$ $\rightarrow$ think of X_{jk} as $X_j(k)$ and $\delta_{ik} = b_i(k)$ k = 1,...,n, then solve LUx=b for n unit vector right-hand sides, then the solution vectors x are the columns of A^{-1} - matrix inverse is $O(n^3)$ in operation count.

determinant of A: just product of diagonal elements U_{ii}:
det(A)=det(L)*det(U) = $\prod_i U_{ii}$

Tri-diagonal matrices

- Some special (but important!) matrices can lead to systems of linear equations that are solved much faster than $O(n^3)$.
- An important example are tridiagonal matrices (which we will encounter later in this course in e.g. cubic spline interpolation and also in finite-difference DE solvers). These are a special case of so-called sparse matrices (i.e. matrices where a large number of values are zeros).
- LU decomposition preserves the banded structure:

$$\mathbf{A} = \begin{bmatrix} X & X & 0 & 0 & 0 \\ X & X & X & 0 & 0 \\ 0 & X & X & X & 0 \\ 0 & 0 & X & X & X \\ 0 & 0 & 0 & X & X \end{bmatrix} \longrightarrow \mathbf{L} = \begin{bmatrix} X & 0 & 0 & 0 & 0 \\ X & X & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 \\ 0 & 0 & X & X & 0 \\ 0 & 0 & 0 & X & X \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} X & X & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 \\ 0 & 0 & X & X & 0 \\ 0 & 0 & 0 & X & X \\ 0 & 0 & 0 & 0 & X \end{bmatrix}$$

Tri-diagonal matrices: storage

- Normally we store e, d, and c as vectors, rather than the full matrix, which provides huge savings in storage.

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & 0 & \cdots & 0 \\ c_1 & d_2 & e_2 & 0 & \cdots & 0 \\ 0 & c_2 & d_3 & e_3 & \cdots & 0 \\ 0 & 0 & c_3 & d_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & c_{n-1} & d_n \end{bmatrix} \longrightarrow \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{n-1} \end{bmatrix}$$

- Example:

1000x1000 full matrix → 10^6 numbers

3-band only → 2998 numbers, 334x less!!

Tri-diagonal matrices: solution

- Standard methods like Gaussian elimination are easily adopted (but much faster) to this case. This can be solved by Gauss elimination with O(n) operations:

$$\text{row } k \leftarrow \text{row } k - (c_{k-1}/d_{k-1}) \times \text{row } (k-1), \quad k = 2, 3, \ldots, n$$

$$d_k \leftarrow d_k - (c_{k-1}/d_{k-1})e_{k-1}$$

$$c_{k-1} \leftarrow c_{k-1}/d_{k-1}$$

LU: Doolitle reduction

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & 0 & \cdots & 0 \\ c_1 & d_2 & e_2 & 0 & \cdots & 0 \\ 0 & c_2 & d_3 & e_3 & \cdots & 0 \\ 0 & 0 & c_3 & d_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & c_{n-1} & d_n \end{bmatrix}$$

In Python:

```
for k in range(1,n):
    lam = c[k-1]/d[k-1]
    d[k] = d[k] - lam*e[k-1]
    c[k-1] = lam
```

Then, the back substitution phase proceeds as before...

Tri-diagonal matrices: code from book

```
## module LUdecomp3
''' c,d,e = LUdecomp3(c,d,e):  LU decomposition of tridiagonal matrix [c\d\e].
    On output{c},{d} and {e} are the diagonals of the decomposed matrix.
    x = LUsolve3(c,d,e,b): Solves [c\d\e]{x} = {b}, where {c}, {d} and {e}
    are the vectors returned from LUdecomp3.
'''
def LUdecomp3(c,d,e):
    n = len(d)
    for k in range(1,n):
        lam = c[k-1]/d[k-1]
        d[k] = d[k] - lam*e[k-1]
        c[k-1] = lam
    return c,d,e
def LUsolve3(c,d,e,b):
    n = len(d)
    for k in range(1,n):
        b[k] = b[k] - c[k-1]*b[k-1]
    b[n-1] = b[n-1]/d[n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - e[k]*b[k+1])/d[k]
    return b
```

Tridiagonal matrices: Scipy

- Naturally, there already exists and internal Scipy routine for solving banded problems: scipy.linalg.solve_banded((l,u), cm, rhs)
- The arguments are:
  - (l,u) → how many diagonals below and above the main diagonal – e.g. for tridiagonal matrix we have (l,u)=(1,1).
  - rhs=a vector holding the right-hand side of the system
  - cm = a matrix storing (economically) the non-zero matrix elements, as follows:

$$\begin{pmatrix} 1 & -5 & 3 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & -3 & 5 & 0 & 0 & 0 & 0 \\ 0 & -2 & 1 & 5 & 9 & 0 & 0 & 0 \\ 0 & 0 & 9 & -1 & 5 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -3 & -2 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 1 & -6 \\ 0 & 0 & 0 & 0 & 0 & -3 & 2 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 9 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 3 & 5 & 9 & 4 & -2 & -6 \\ 0 & -5 & -3 & 5 & 5 & -3 & 1 & 7 \\ 1 & 2 & 1 & -1 & 2 & 0 & 2 & 1 \\ 3 & -2 & 9 & 0 & 2 & -3 & 9 & 0 \end{pmatrix}$$

Gauss elimination with pivoting

Consider the following example:

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

The algorithm already fails after first step, since $\lambda = 14/0 \rightarrow$ division by zero! Similarly, a mix of very small and very large numbers will cause serious precision problems. Remedy: exchange 2nd and 3rd row -> "pivoting". The diagonal element used to generate the division coefficient is called "pivot".

Basic algorithm is numerically unstable, since a small pivot can lead to round-off and overflow. Instead, perform row interchanges ("partial pivoting") so that you always place the largest element (on or below diagonal) on diagonal to be used as next pivot.

Gauss with partial pivoting

Pivoting only requires a small change to the algorithm: in each row (after the for i=1:n-1 statement), find p ≥ i such that |A(p,i)| = max(|A(j,i)|,j≥i), then interchange rows i and p.

There is also full pivoting which interchanges also columns, but partial pivoting is numerically nearly as stable, and is simpler, so we limit our discussion to it.

Gauss elimination with partial pivoting is simple to understand and implement, and reasonably stable and fast, and so is a good test-method if a more advanced “canned” routine gives suspect results.

Gauss elimination with partial pivoting

```
## module gaussPivot
''' x = gaussPivot(a,b,tol=1.0e-9).
   Solves [a]{x} = {b} by Gauss elimination with scaled row pivoting
'''
from numpy import zeros,argmax,dot
import swap
import error
def gaussPivot(a,b,tol=1.0e-12):
    n = len(b)
    # Set up scale factors
    s = zeros(n)
    for i in range(n):
        s[i] = max(abs(a[i,:]))
    for k in range(0,n-1):
      # Row interchange, if needed
        p = argmax(abs(a[k:n,k])/s[k:n]) + k
        if abs(a[p,k]) < tol: error.err('Matrix is singular')
        if p != k:
            swap.swapRows(b,k,p)
            swap.swapRows(s,k,p)
            swap.swapRows(a,k,p)
```

Gauss elimination with partial pivoting

The rest essentially the same as the simple Gaussian Elimination

```
# Elimination
        for i in range(k+1,n):
            if a[i,k] != 0.0:
                lam = a[i,k]/a[k,k]
                a[i,k+1:n] = a [i,k+1:n] - lam*a[k,k+1:n]
                b[i] = b[i] - lam*b[k]
    if abs(a[n-1,n-1]) < tol: error.err('Matrix is singular')

# Back substitution
    b[n-1] = b[n-1]/a[n-1,n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

→ try this and the old version out on the previous example:

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

Ill-conditioned matrices

Some matrices are inherently close to being singular (i.e. det(A) is close to 0) and lead to systems of equations that are difficult to solve.

Example: Hilbert matrix, $h_{i\,j} = 1/(i+j-1)$

(Ill) conditioning can be quantified, but rules are complex. A simple 'rule of thumb' is that det(A) should not be much smaller than the elements of A.

For ill-conditioned matrices even small changes of the coefficients can result in huge changes in the solution, which can make even small-ish systems of linear equations (e.g. 20) impossible to solve!

Fixes require special treatments, e.g. SVD (singular value decomposition) or extended precision routines.

Comments

We discussed Gauss elimination and the need for (partial) pivoting.

This allows to solve Ax=b with forward elimination followed by back-substitution.

We also looked at LU decomposition, the inverse and the determinant of a matrix

These are direct methods, but for large sparse systems, iterative methods are often better (especially conjugate gradient methods and multigrid methods). We did not discuss the iterative methods here.

Big libraries available

standard: LAPACK -> http://www.netlib.org/lapack/

- callable from Fortran, C, etc., implemented in SciPy Python package.

Generally, netlib.org is a good resource of information.

For C/C++, look at e.g. GSL (gnu scientific library) -> http://www.gnu.org/software/gsl/

Implementation "details" are very important for good performance.

→ commercial packages (e.g. NAG, IMSL) or

→ vendor implementations (intel MKL, AMD ACML)

Python (NumPy/SciPy) commands

Internally, Python uses efficient LAPACK routines. Some are available in the NumPy module, and more are in the SciPy module ('import scipy','from scipy import linalg').

If A is a NumPy matrix and b is a vector, we have:

x=solve(A,b) (or np.linalg.solve(A,b)) – solves system Ax=b using Gauss elimination with partial pivoting (in the LU form) and row interchanges.

linalg.lu(A) – computes the LU factorisation explicitly.

linalg.det(A) – computes the determinant of A

linalg.inv (A) , or A.I – computes the inverse of A

transpose(A) – gives the transpose of A

Look them up in the help if you need them.