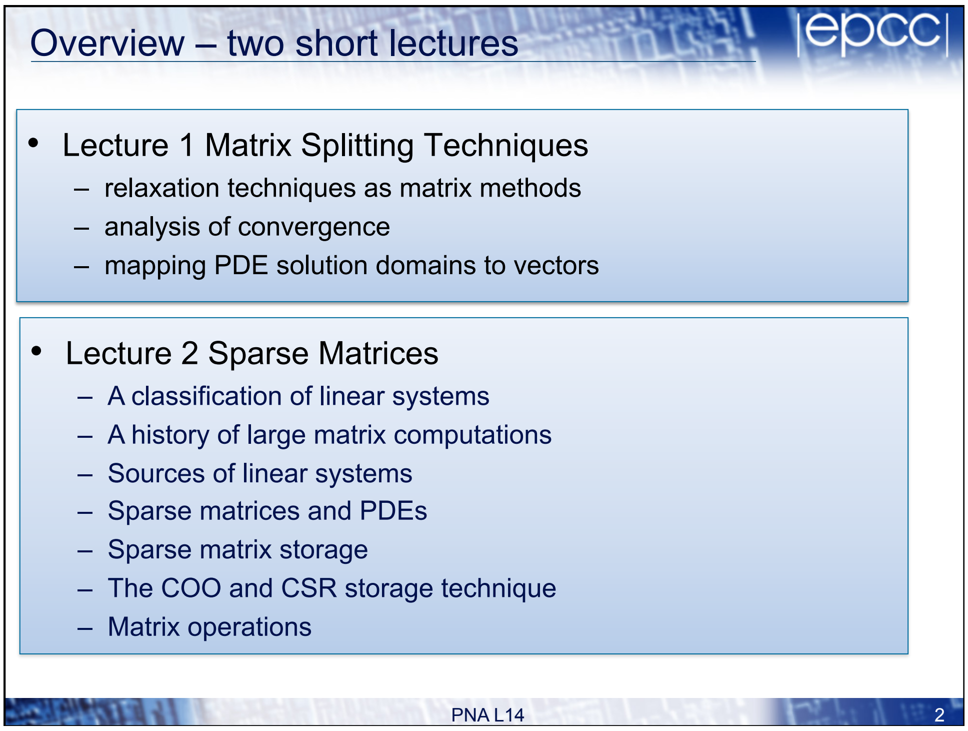
(i) Matrix splitting techniques (ii) Sparse Matrices

Sparse Linear Algebra



Overview – two short lectures

- Lecture 1 Matrix Splitting Techniques
  - relaxation techniques as matrix methods
  - analysis of convergence
  - mapping PDE solution domains to vectors
- Lecture 2 Sparse Matrices
  - A classification of linear systems
  - A history of large matrix computations
  - Sources of linear systems
  - Sparse matrices and PDEs
  - Sparse matrix storage
  - The COO and CSR storage technique
  - Matrix operations

Summary of Relaxation Methods

- So far presented as a pragmatic approach
  - write down the discrete PDE that each solution point must satisfy
  - loop over every interior point and solve the PDE locally
  - apply the boundary conditions by hand
  - iterate many times and hope for convergence
- Some extensions to this Jacobi method
  - eg in-place solution (Gauss-Seidel) and over-relaxation
- Not obviously related to matrix methods

Matrix Splitting

- Any linear problem is of the form *A x* = *b*
  - *A* encodes the precise form of the PDE
  - *b* contains any fixed boundary conditions
- Split *A* into three parts
  - Diagonal, Strictly Upper and Strictly Lower triangular: *A* = *L* + *D* + *U*
  - *not* the same as the LU factors!

$$(L + D + U)x = b, \quad Dx = -(L+U)x + b$$

  - view these as iterative expressions, eg Jacobi corresponds to

$$Dx^{(n+1)} = -(L+U)x^{(n)} + b$$

$$x^{(n+1)} = -D^{-1}(L+U)x^{(n)} + D^{-1}b$$

Consider 1D Pollution Problem

- *A* represents: $-d^2/dx^2$

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

- Splitting into *L*, *D* and *U*
  - a Jacobi iteration is: $D\, x^{(n+1)} = -(L+U)\, x^{(n)} + b$

$$\begin{bmatrix} 2 & \cdot & \cdot & \cdot \\ \cdot & 2 & \cdot & \cdot \\ \cdot & \cdot & 2 & \cdot \\ \cdot & \cdot & \cdot & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}^{(n+1)} = \begin{bmatrix} \cdot & 1 & \cdot & \cdot \\ 1 & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & 1 \\ \cdot & \cdot & 1 & \cdot \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}^{(n)} + \begin{bmatrix} b_1 \\ 0 \\ 0 \\ b_4 \end{bmatrix}$$

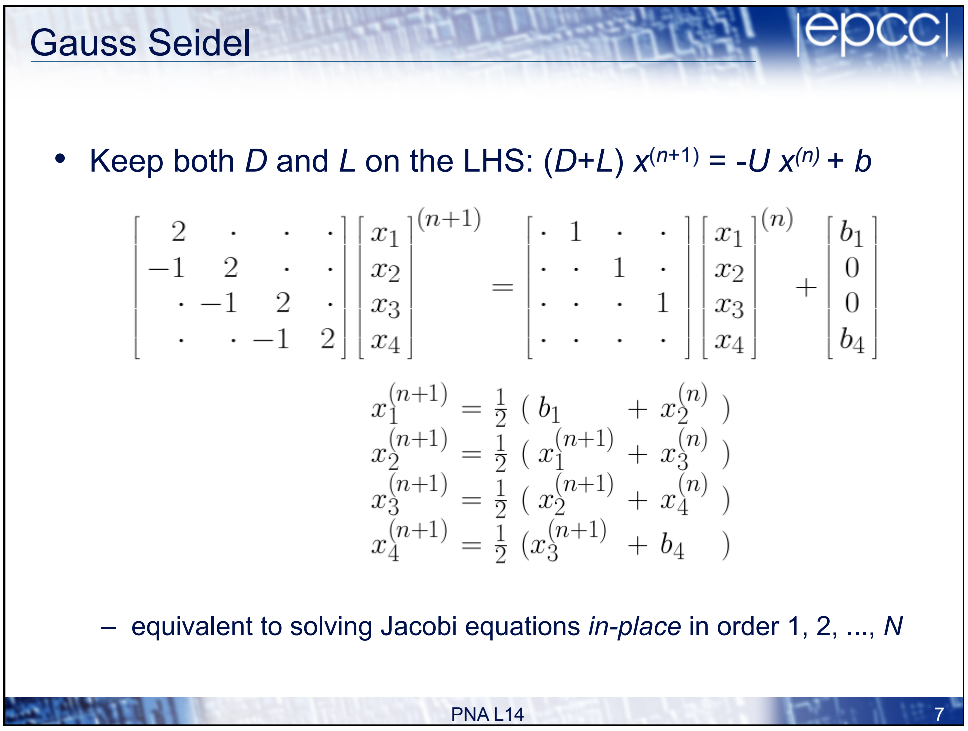
Jacobi Equations

- Equations the same as in previous lectures
  - with u_i replaced by x_i
  - exterior boundary values u_0 and u_{N+1} replaced by b_1 and b_N

-

$$\begin{aligned} x_1^{(n+1)} &= \tfrac{1}{2}\,(\,b_1 + x_2^{(n)}\,) \\ x_2^{(n+1)} &= \tfrac{1}{2}\,(\,x_1^{(n)} + x_3^{(n)}\,) \\ x_3^{(n+1)} &= \tfrac{1}{2}\,(\,x_2^{(n)} + x_4^{(n)}\,) \\ x_4^{(n+1)} &= \tfrac{1}{2}\,(\,x_3^{(n)} + b_4\,) \end{aligned}$$

- Procedure
  - impose PDE at each interior point
  - new value is the average of the old neighbouring points



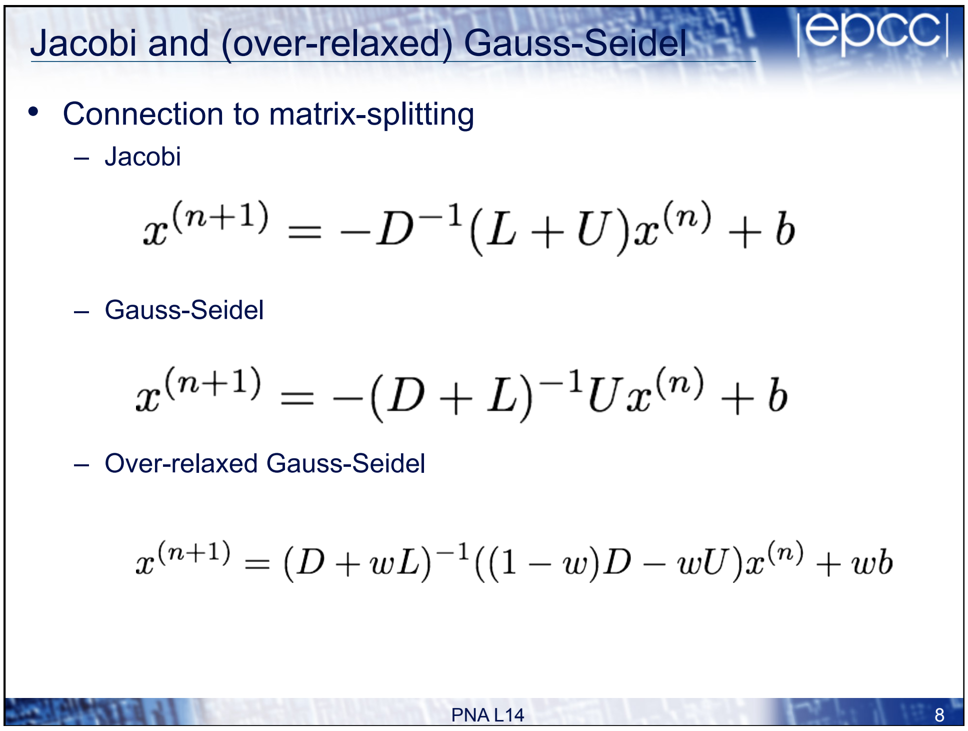
Gauss Seidel

- Keep both *D* and *L* on the LHS: (*D*+*L*) $x^{(n+1)}$ = -*U* $x^{(n)}$ + *b*

$$\begin{bmatrix} 2 & \cdot & \cdot & \cdot \\ -1 & 2 & \cdot & \cdot \\ \cdot & -1 & 2 & \cdot \\ \cdot & \cdot & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}^{(n+1)} = \begin{bmatrix} \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}^{(n)} + \begin{bmatrix} b_1 \\ 0 \\ 0 \\ b_4 \end{bmatrix}$$

$$\begin{aligned} x_1^{(n+1)} &= \tfrac{1}{2}\,(\,b_1 + x_2^{(n)}\,) \\ x_2^{(n+1)} &= \tfrac{1}{2}\,(\,x_1^{(n+1)} + x_3^{(n)}\,) \\ x_3^{(n+1)} &= \tfrac{1}{2}\,(\,x_2^{(n+1)} + x_4^{(n)}\,) \\ x_4^{(n+1)} &= \tfrac{1}{2}\,(x_3^{(n+1)} + b_4\,) \end{aligned}$$

  - equivalent to solving Jacobi equations *in-place* in order 1, 2, ..., *N*



Jacobi and (over-relaxed) Gauss-Seidel

- Connection to matrix-splitting
  - Jacobi

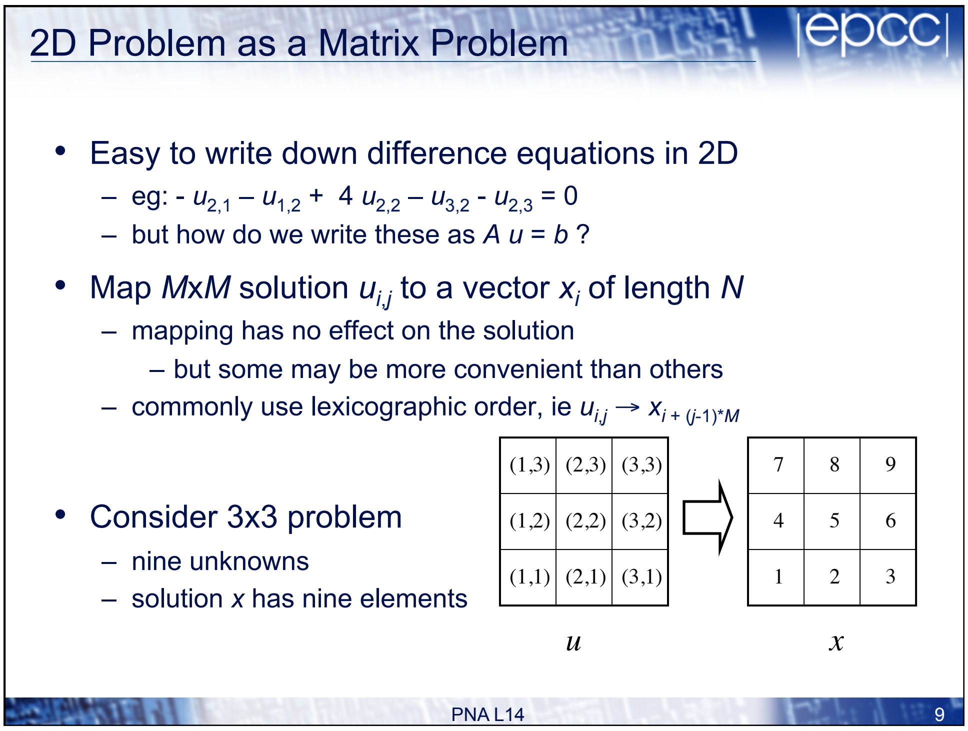
$$x^{(n+1)} = -D^{-1}(L+U)x^{(n)} + b$$

  - Gauss-Seidel

$$x^{(n+1)} = -(D+L)^{-1}Ux^{(n)} + b$$

  - Over-relaxed Gauss-Seidel

$$x^{(n+1)} = (D+wL)^{-1}((1-w)D - wU)x^{(n)} + wb$$



2D Problem as a Matrix Problem

- Easy to write down difference equations in 2D
  - eg: $- u_{2,1} - u_{1,2} + 4\, u_{2,2} - u_{3,2} - u_{2,3} = 0$
  - but how do we write these as $A\ u = b$?
- Map MxM solution $u_{i,j}$ to a vector x_i of length N
  - mapping has no effect on the solution
    - but some may be more convenient than others
  - commonly use lexicographic order, ie $u_{i,j} \rightarrow x_{i + (j-1)*M}$
- Consider 3x3 problem
  - nine unknowns
  - solution x has nine elements

(1,3)	(2,3)	(3,3)
(1,2)	(2,2)	(3,2)
(1,1)	(2,1)	(3,1)

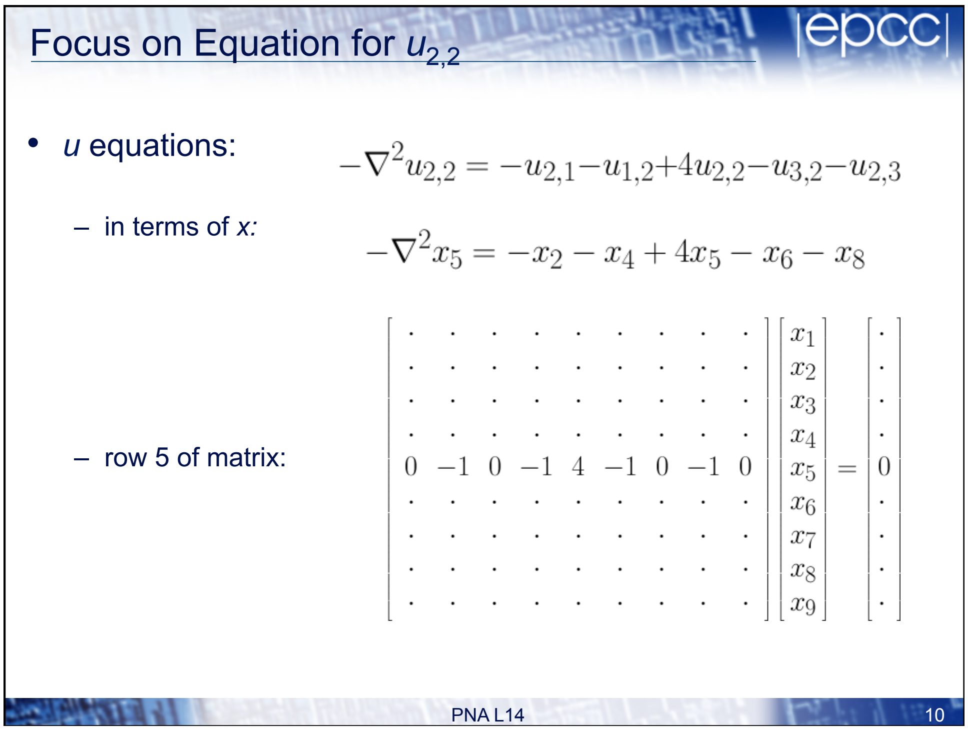
u

$\Rightarrow$

7	8	9
4	5	6
1	2	3

x

PNA L14 9



Focus on Equation for $u_{2,2}$

- u equations:

$$-\nabla^2 u_{2,2} = -u_{2,1} - u_{1,2} + 4u_{2,2} - u_{3,2} - u_{2,3}$$

  - in terms of x:

$$-\nabla^2 x_5 = -x_2 - x_4 + 4x_5 - x_6 - x_8$$

  - row 5 of matrix:

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

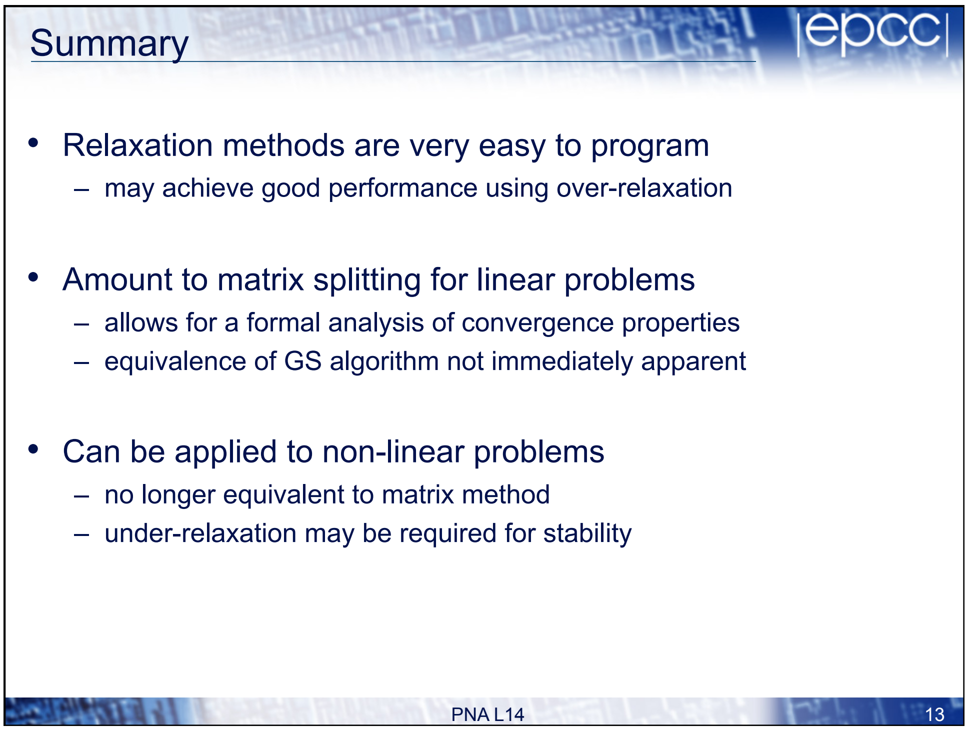
PNA L14 10

Choice of Ordering

- Has some effect on performance
  - may want to reflect red/black nature of chequerboard update
  - ie try to achieve linear access patterns on *x*
- Has a major effect in parallel
  - parallel matrix-vector performed by decomposing over vectors
  - eg $y = Ax$ is parallelised by regular domain decomposition of *y*
  - amount of communications determined by form of mapping
  - optimising the mapping is a *mesh decomposition* problem
  - standard algorithms exist (recursive spectral bisection, ...)

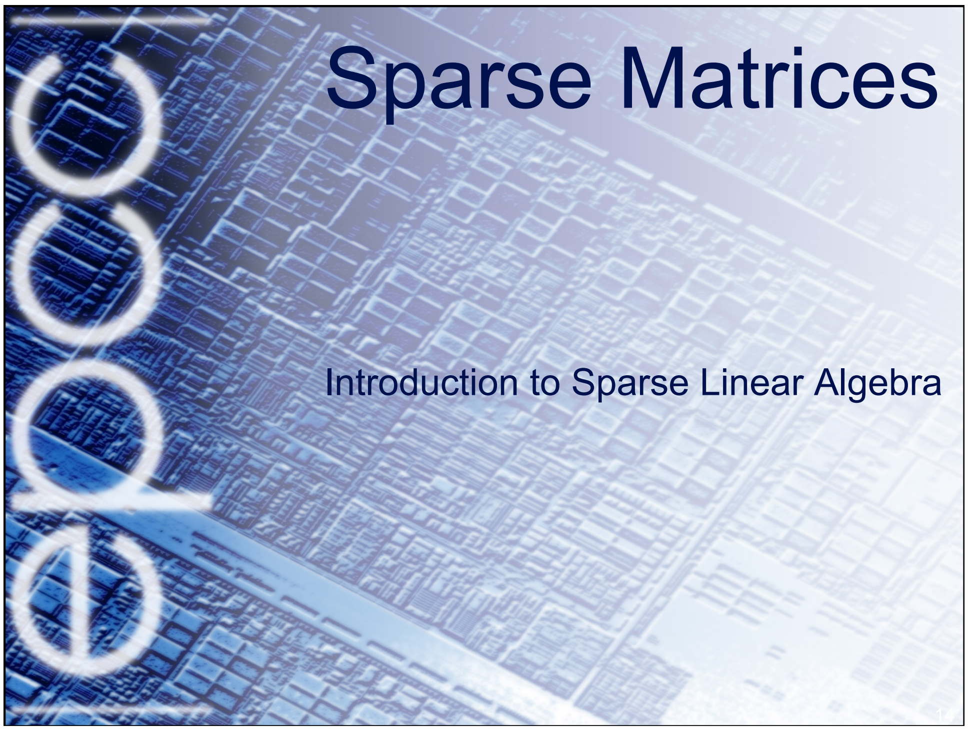
Note: Convergence of Splitting

- General matrix splitting eqns: $E\,x^{(n+1)} = F\,x^n + b$
  - actual solution is perfect solution plus correction
  - $x^n = \hat{x} + \delta x^n$ where $E\hat{x} = F\hat{x} + b$
  - substituting into main equation gives $E\,\delta x^{n+1} = F\,\delta x^n$
  - error in solution evolves according to $\delta x^n = (E^{-1}F)^n\,\delta x^0$
- Convergence depends on eigenvalues of $E^{-1}F$
  - must all be less than one in order to get a solution
  - speed of convergence depends on condition number
- Iteration matrix
  - $E^{-1}F$ is $-D^{-1}(L+U)$ for Jacobi and $-(D+L)^{-1}U$ for Gauss Seidel
  - can show that latter is better conditioned
  - heuristically, GS inverts more of the matrix at each step



Summary

- Relaxation methods are very easy to program
  - may achieve good performance using over-relaxation
- Amount to matrix splitting for linear problems
  - allows for a formal analysis of convergence properties
  - equivalence of GS algorithm not immediately apparent
- Can be applied to non-linear problems
  - no longer equivalent to matrix method
  - under-relaxation may be required for stability



Sparse Matrices

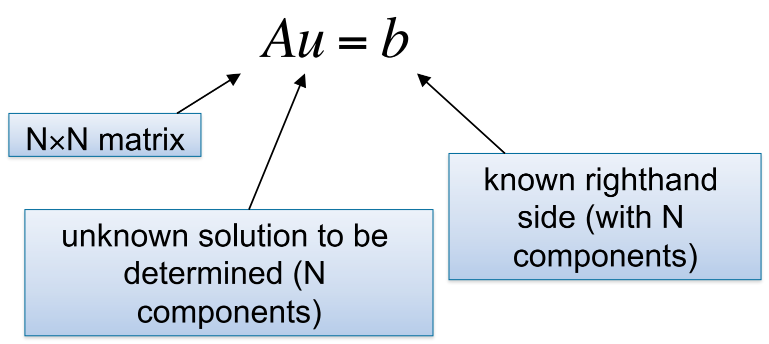
Introduction to Sparse Linear Algebra

Overview

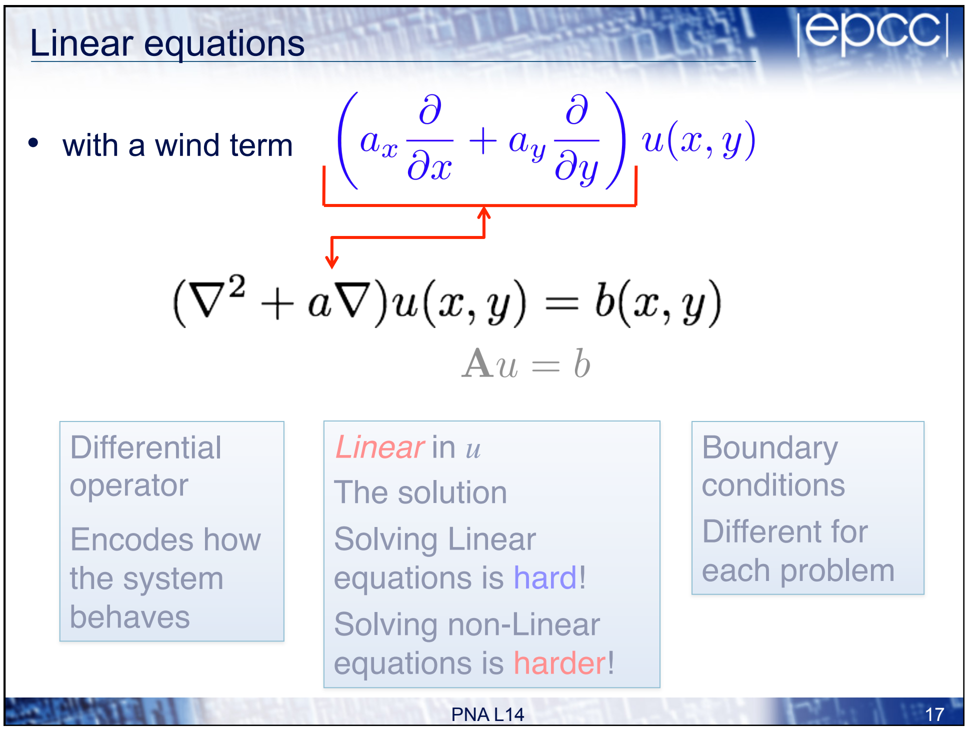
- Motivation
  - A classification of linear systems
  - A history of large matrix computations
  - Sources of linear systems
  - Sparse matrices and PDEs
  - Sparse matrix storage
  - The COO and CSR storage technique
  - Matrix operations

Recall what a linear system is

- Recall that a linear system (of size **N**) can be represented by a matrix equation, of the form:


$$Au = b$$
N×N matrix
unknown solution to be determined (N components)
known righthand side (with N components)

- This is simply a representation of **N** equations linking **N** unknown quantities: $\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_N}$.



Linear equations

- with a wind term $\underbrace{\left(a_x \frac{\partial}{\partial x} + a_y \frac{\partial}{\partial y}\right)}\, u(x,y)$

$$(\nabla^2 + a\nabla)u(x,y) = b(x,y)$$

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

Differential operator

Encodes how the system behaves

Linear in u

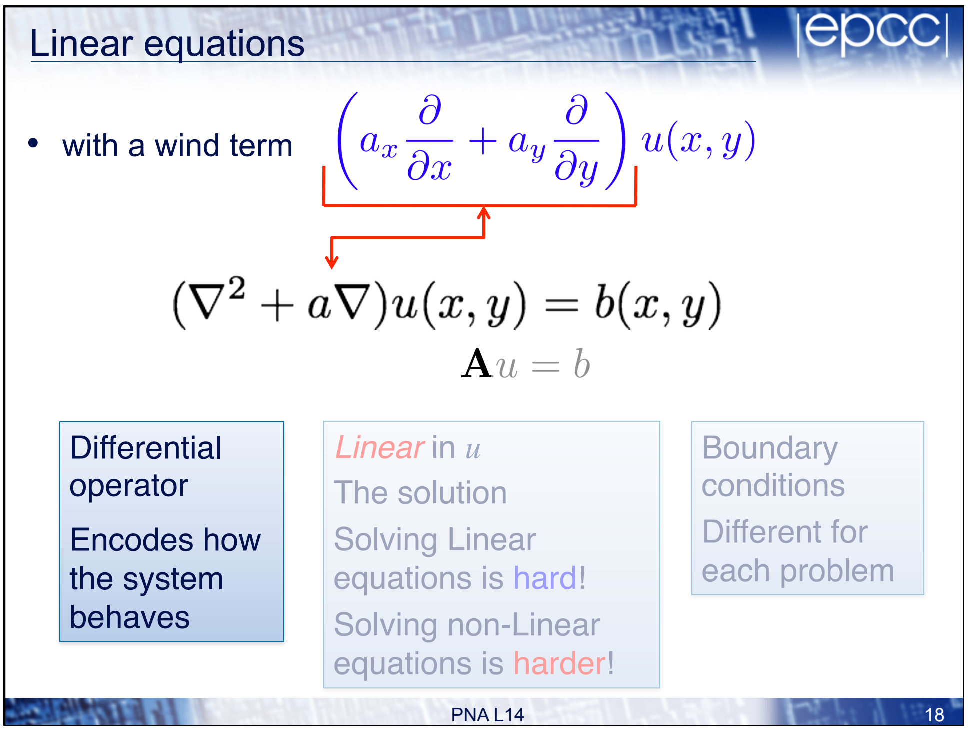
The solution

Solving Linear equations is hard!

Solving non-Linear equations is harder!

Boundary conditions

Different for each problem



Linear equations

- with a wind term $\underbrace{\left(a_x \frac{\partial}{\partial x} + a_y \frac{\partial}{\partial y}\right)}\, u(x,y)$

$$(\nabla^2 + a\nabla)u(x,y) = b(x,y)$$

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

Differential operator

Encodes how the system behaves

Linear in u

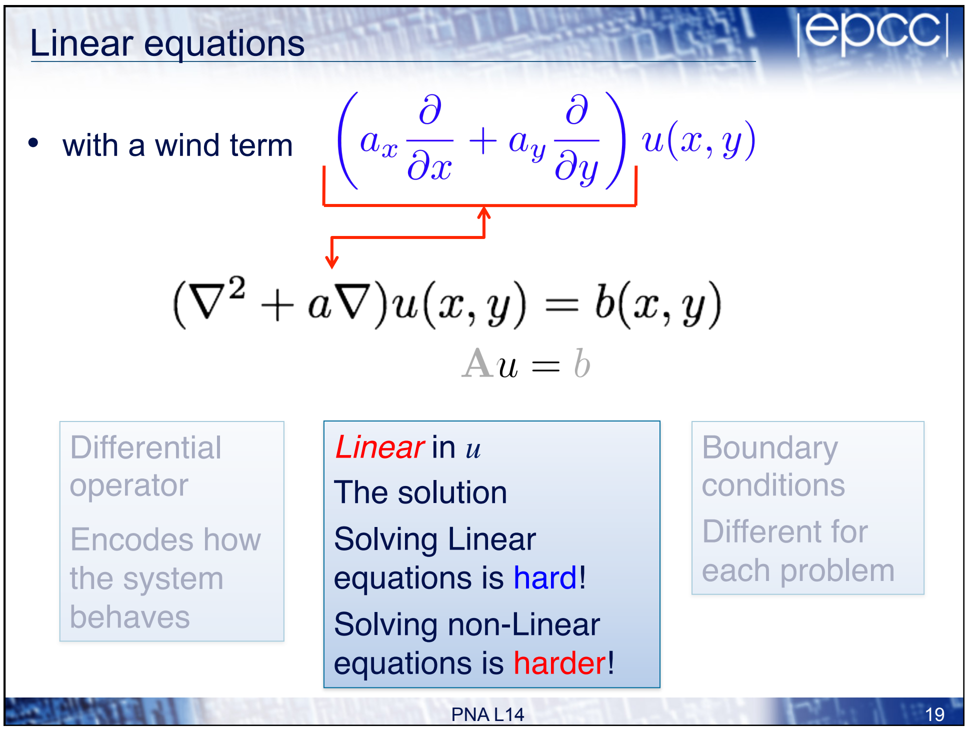
The solution

Solving Linear equations is hard!

Solving non-Linear equations is harder!

Boundary conditions

Different for each problem



Linear equations

- with a wind term $\left(a_x \frac{\partial}{\partial x} + a_y \frac{\partial}{\partial y}\right) u(x,y)$

$$(\nabla^2 + a\nabla)u(x,y) = b(x,y)$$

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

Differential operator

Encodes how the system behaves

Linear in u

The solution

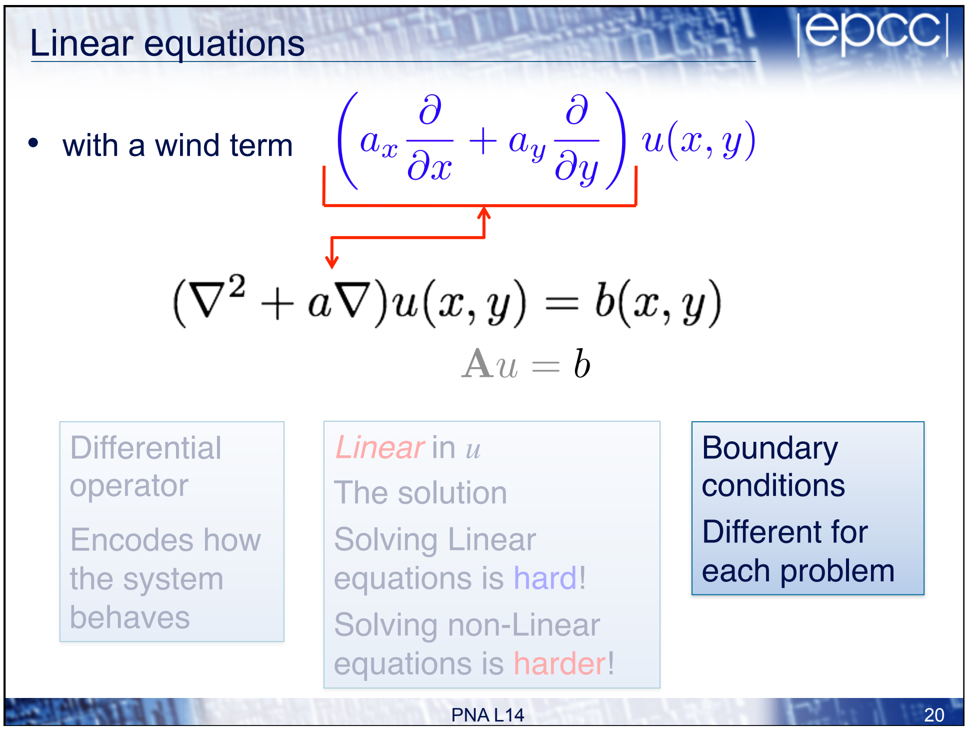
Solving Linear equations is hard!

Solving non-Linear equations is harder!

Boundary conditions

Different for each problem

PNA L14 19



Linear equations

- with a wind term $\left(a_x \frac{\partial}{\partial x} + a_y \frac{\partial}{\partial y}\right) u(x,y)$

$$(\nabla^2 + a\nabla)u(x,y) = b(x,y)$$

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

Differential operator

Encodes how the system behaves

Linear in u

The solution

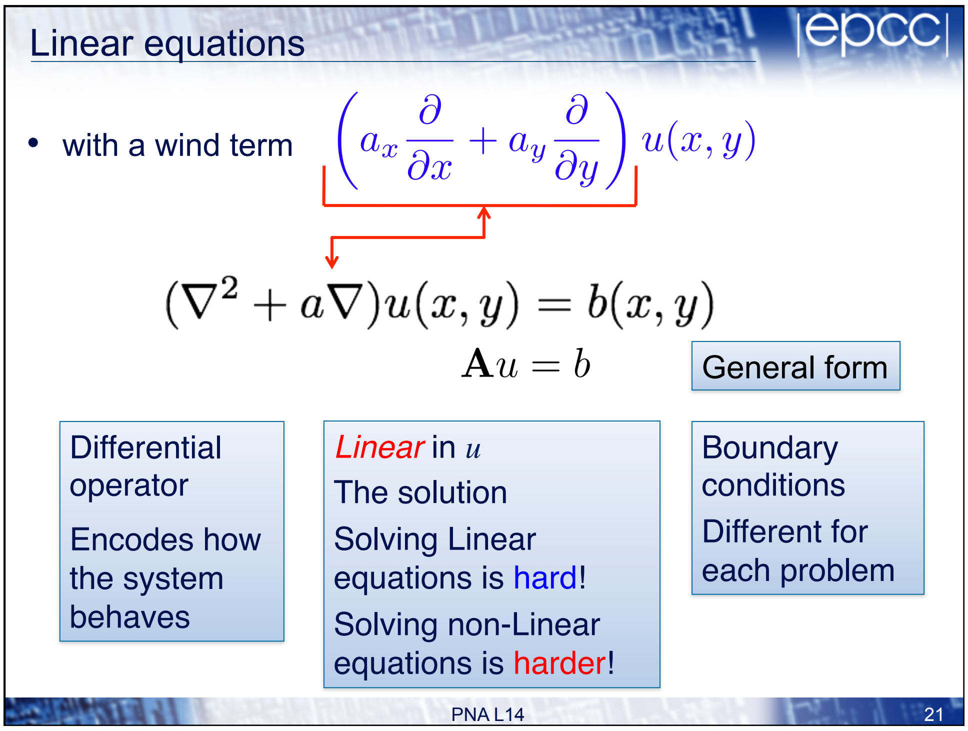
Solving Linear equations is hard!

Solving non-Linear equations is harder!

Boundary conditions

Different for each problem

PNA L14 20


Linear equations

- with a wind term $\left(a_x \frac{\partial}{\partial x} + a_y \frac{\partial}{\partial y}\right) u(x,y)$

$$(\nabla^2 + a\nabla)u(x,y) = b(x,y)$$

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

General form

Differential operator

Encodes how the system behaves

Linear in u

The solution

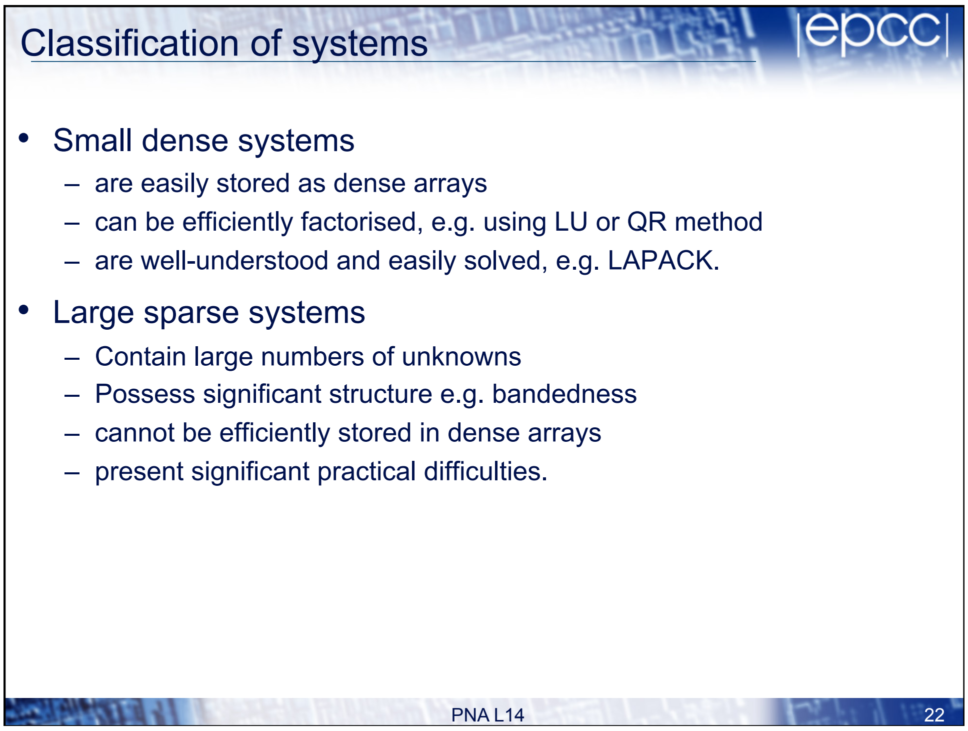
Solving Linear equations is hard!

Solving non-Linear equations is harder!

Boundary conditions

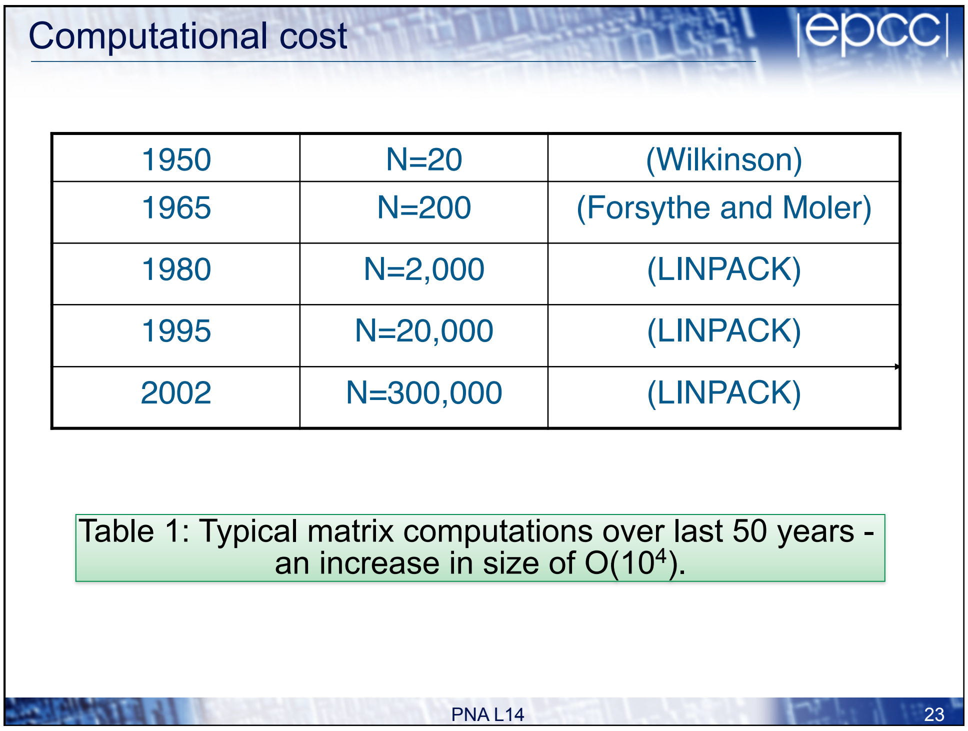
Different for each problem

PNA L14 21


Classification of systems

- Small dense systems
  - are easily stored as dense arrays
  - can be efficiently factorised, e.g. using LU or QR method
  - are well-understood and easily solved, e.g. LAPACK.
- Large sparse systems
  - Contain large numbers of unknowns
  - Possess significant structure e.g. bandedness
  - cannot be efficiently stored in dense arrays
  - present significant practical difficulties.

PNA L14 22

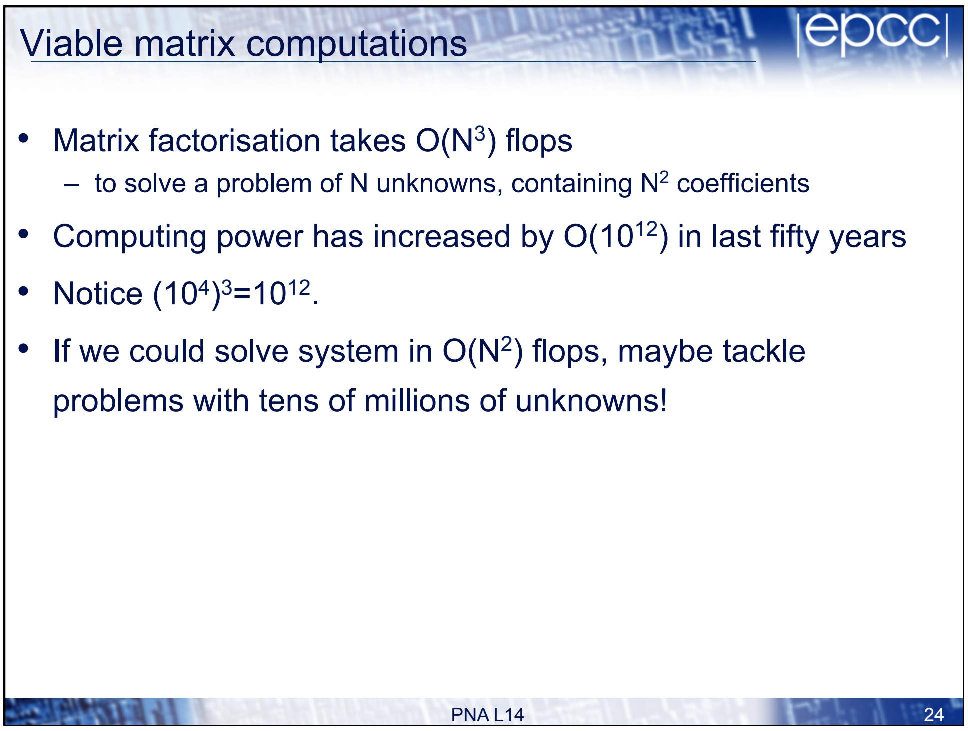


Computational cost

1950	N=20	(Wilkinson)
1965	N=200	(Forsythe and Moler)
1980	N=2,000	(LINPACK)
1995	N=20,000	(LINPACK)
2002	N=300,000	(LINPACK)

Table 1: Typical matrix computations over last 50 years - an increase in size of O(10^4).

PNA L14 23



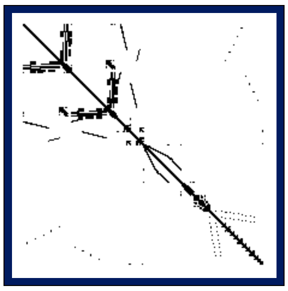
Viable matrix computations

- Matrix factorisation takes O(N^3) flops
  - to solve a problem of N unknowns, containing N^2 coefficients
- Computing power has increased by O(10^{12}) in last fifty years
- Notice $(10^4)^3=10^{12}$.
- If we could solve system in O(N^2) flops, maybe tackle problems with tens of millions of unknowns!

PNA L14 24

Sources of linear systems

- Small dense systems (<O(10^4) unknowns)
  - forces between bodies in a mechanical structure
  - interaction between chemicals in a reaction
- Large sparse systems (>O(10^4) unknowns)
  - models from CFD, financial markets, biology, particle physics, environmental: discretisation of PDEs
    - finite difference, finite element, or finite volume methods

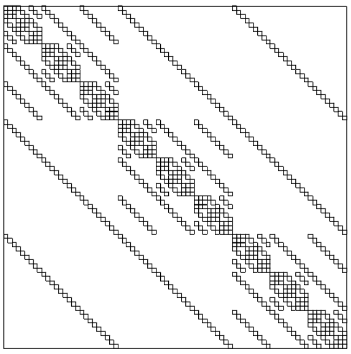

Gear box model: 153,746 unknowns, 9,080,404 nonzeros, T. Davis, University of Florida Sparse Matrix Collection

QCD

- Continuous space-time is replaced with a 4D lattice
- For a lattice with 24^3 x 48 space-time points with 4 spin and 3 colour components each matrix size is
  - 7,962,624 x 7,962,624 = 63,403,380,965,376 elements
  - This corresponds to 1,014,454,095,446,016 Bytes ≈ 1 PB c.f. HECToR (ARCHER predecessor) which had total of 90 TB RAM
- The matrix represents a Dirac operator
  - essentially a complicated differential operator so mostly we're dealing with nearest-neighbour interactions.

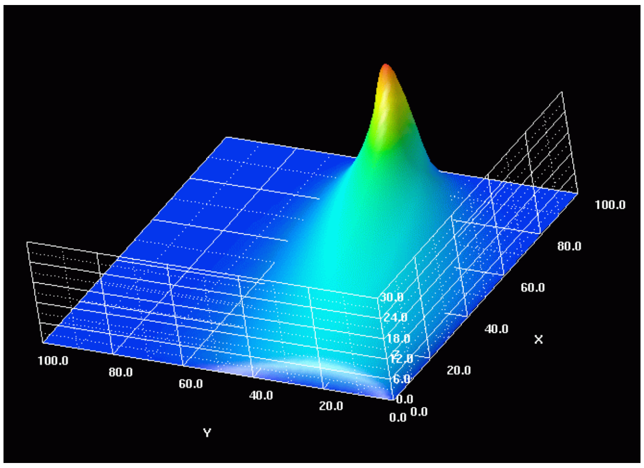
Wilson-Dirac matrix

- Wilson-Dirac operator
  - Hermitian
  - Well-conditioned
  - Well-structured
  - Sparse!



- SM Pickles PhD thesis UoE 1998

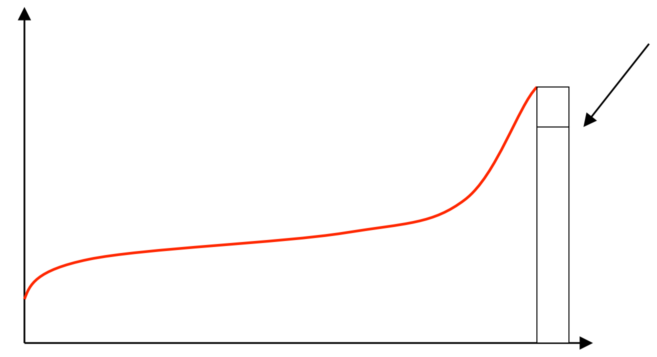
Example PDE: spread of pollution



Spread of pollution -- solution of large, sparse linear system.

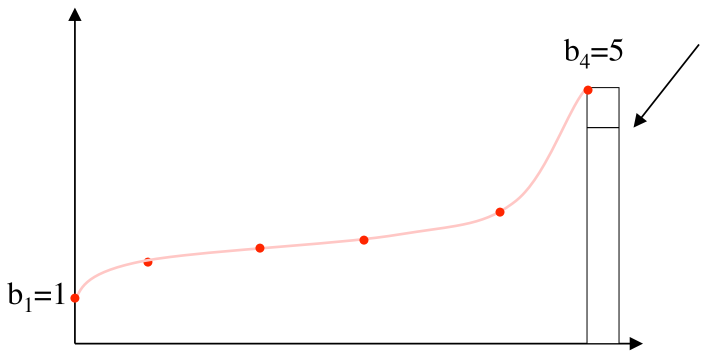
One-dimensional pollution model

Solution might look like:



Discretisation of model

Replace continuous equation by discrete set of values, defined on a mesh (M=4):



Finite difference matrix

$$\begin{bmatrix} d & u & 0 & & & 0 \\ l & d & \ddots & & & \\ 0 & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & 0 \\ & & & \ddots & \ddots & u \\ 0 & & & 0 & l & d \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} -lb_1 \\ 0 \\ \vdots \\ \vdots \\ 0 \\ -ub_N \end{bmatrix}$$

$$l = -\frac{1}{h^2}, \qquad d = 2\frac{1}{h^2} + \frac{a}{h}, \quad u = -\frac{1}{h^2} - \frac{a}{h}$$

Sparse matrix storage

- As majority of entries in matrix A are zero, it is inefficient to store a sparse matrix in a 2-D array.
  - makes elimination of redundant calculations more difficult
  - large amounts of memory for storage
- Dozens of storage techniques -- we consider 2 popular choices:
  - COO format (most obvious technique)
  - CSR/ CSC format (as used by Sun S3L, MATLAB, PETSc etc)

COO format

- Co-ordinate (COO) storage involve 3 scalars and 3 arrays:

int nRow	- number of rows
int nCol	- number of columns
int nNz	- number of nonzero elements
double value[nNz]	- contents of matrix location
int rowIdx[nNz]	- row index of each element
int colIdx[nNz]	- likewise for columns

- Data usually in row-major order
- Not particularly efficient
  - – redundant storage of column entries (when ordered)
  - – easy for user, not optimised for programming
  - – expensive to add/modify data or change capacity
  - – very easy to perform matrix transpose operation

Example of COO format

- Matrix

$$A = \begin{pmatrix} 1.0 & 0.1 & 0.5 & \bullet \\ 0.4 & \bullet & \bullet & 0.7 \\ \bullet & 0.6 & \bullet & \bullet \\ 0.3 & \bullet & 0.2 & 0.8 \end{pmatrix}$$

```
nRow = 4; nCol = 4; nNz = 9;
value[9] = {1.0, 0.1, 0.5, 0.4, 0.7, 0.6,
0.3, 0.2, 0.8};
colIdx[9] = {0, 1, 2, 0, 3, 1, 0, 2, 3};
rowIdx[9] = {0, 0, 0, 1, 1, 2, 3, 3, 3};
```

Matrix-vector multiplication

```
public void timesv(double[] v, double[] w){
   for(int j=0; j<this.nRow; j++)
      w[j] = 0.0;

   for(int j=0; j<this.nNz; j++)
      w[this.rowIdx[j]] +=
        this.value[j]*v[this.colIdx[j]];
   return;
}
```

double indirection for assignment and data access

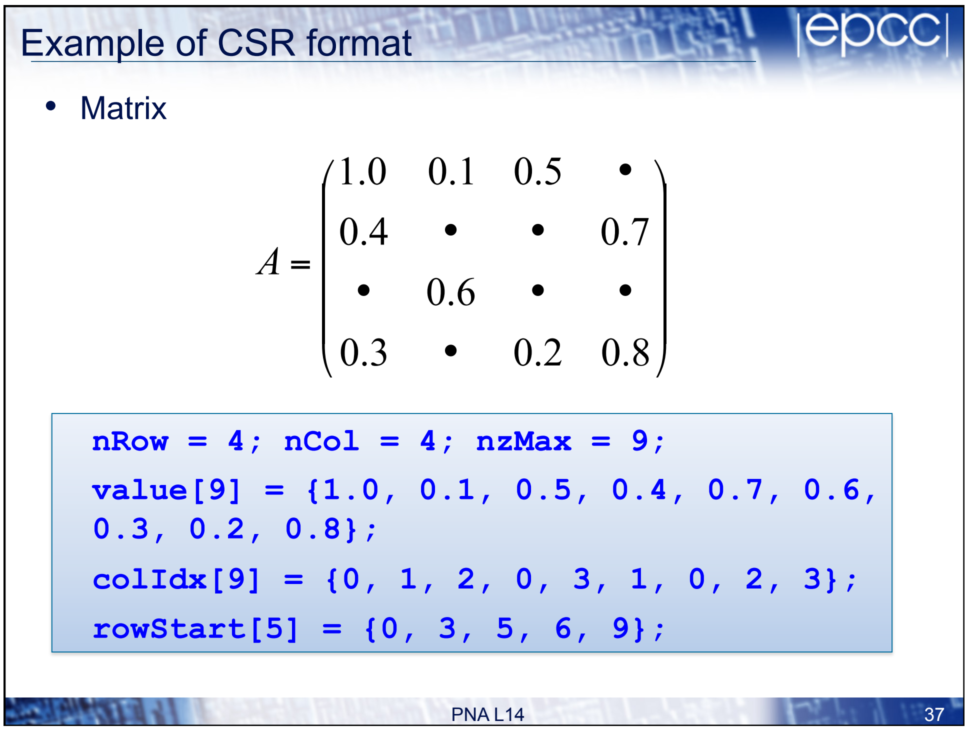
CSR format

- Compressed sparse row (CSR) format - 3 scalars and 3 arrays (data stored in row-major order).

int nRow	- number of rows
int nCol	- number of columns
int nzMax	- maximum number of elements
double value[nzMax]	- contains nonzero entries, stored in row-major order
int colIdx[nzMax]	- columns index of corresponding element in value
int rowStart[nRow+1]	- index into value[] and colIdx[] of the first entry in each row

(nNz = rowStart[nRow])

- Good for matrix-vector multiplication, and reasonable for transpose operation.
- Flexible design in terms of expansion, modification.



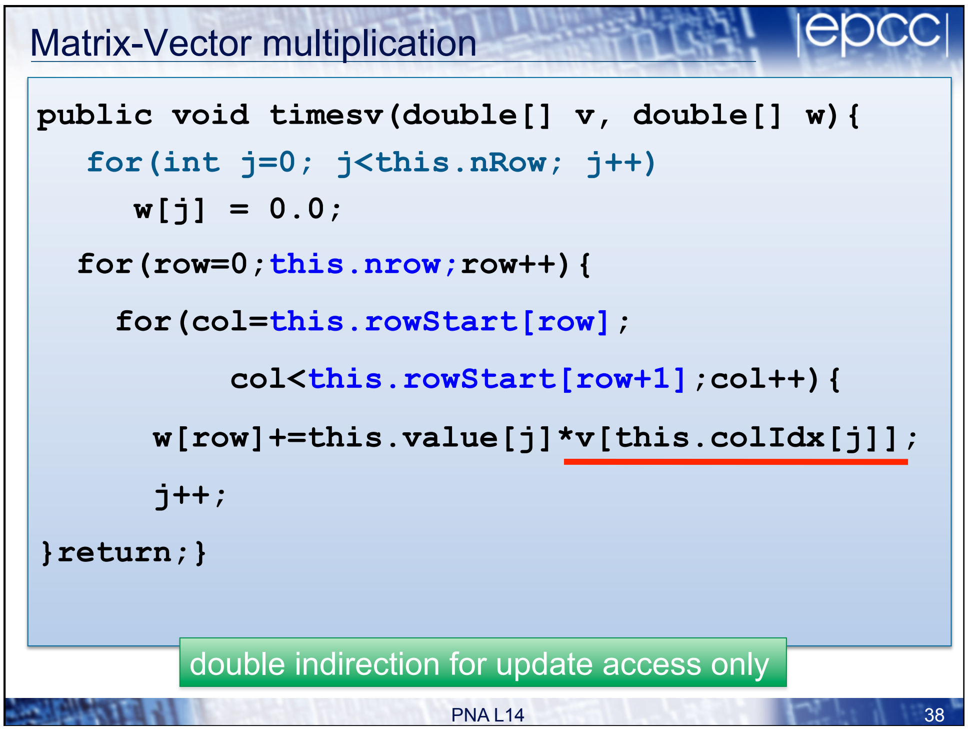
Example of CSR format

- Matrix

$$A = \begin{pmatrix} 1.0 & 0.1 & 0.5 & \bullet \\ 0.4 & \bullet & \bullet & 0.7 \\ \bullet & 0.6 & \bullet & \bullet \\ 0.3 & \bullet & 0.2 & 0.8 \end{pmatrix}$$

```
nRow = 4; nCol = 4; nzMax = 9;
value[9] = {1.0, 0.1, 0.5, 0.4, 0.7, 0.6,
0.3, 0.2, 0.8};
colIdx[9] = {0, 1, 2, 0, 3, 1, 0, 2, 3};
rowStart[5] = {0, 3, 5, 6, 9};
```

PNA L14 37

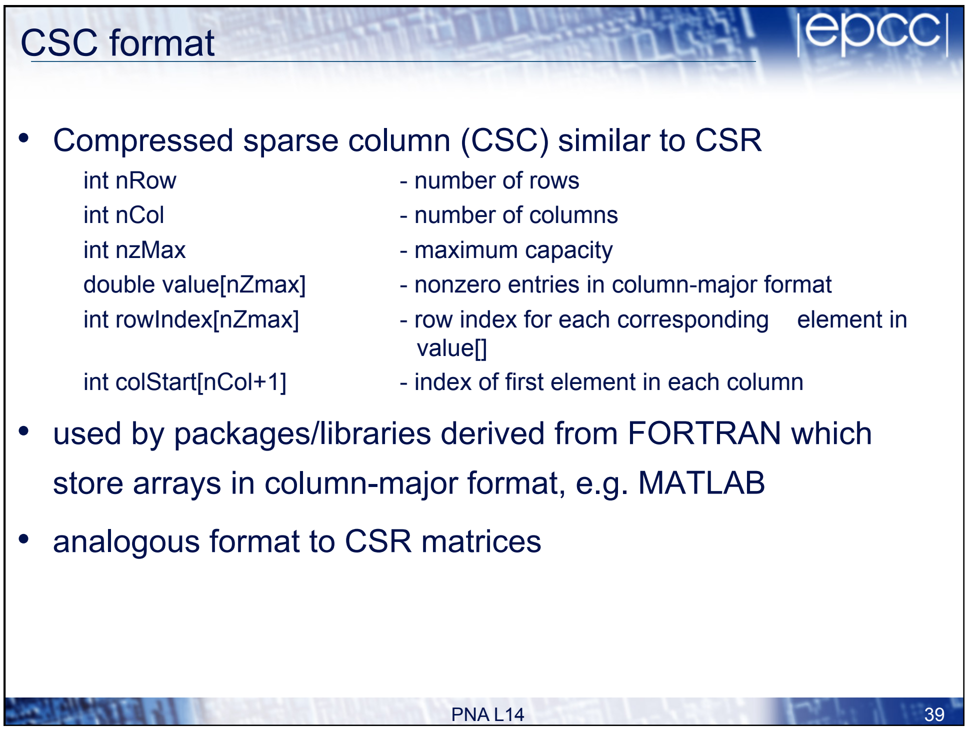


Matrix-Vector multiplication

```
public void timesv(double[] v, double[] w){
   for(int j=0; j<this.nRow; j++)
      w[j] = 0.0;
  for(row=0;this.nrow;row++){
    for(col=this.rowStart[row];
          col<this.rowStart[row+1];col++){
      w[row]+=this.value[j]*v[this.colIdx[j]];
      j++;
}return;}
```

double indirection for update access only

PNA L14 38

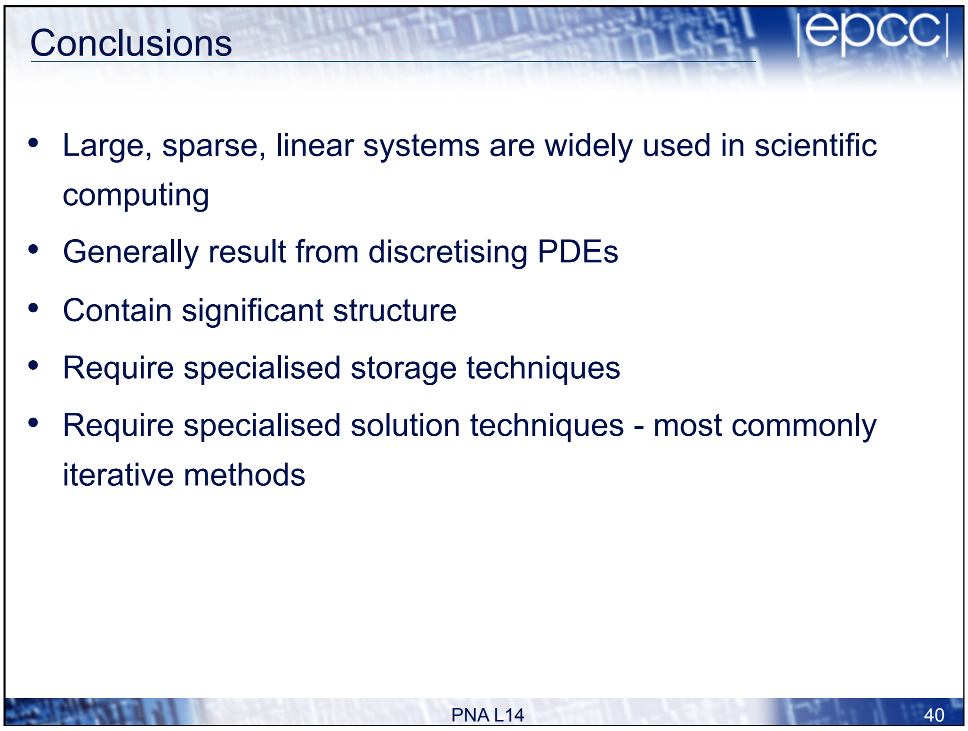


CSC format

- Compressed sparse column (CSC) similar to CSR

int nRow	- number of rows
int nCol	- number of columns
int nzMax	- maximum capacity
double value[nZmax]	- nonzero entries in column-major format
int rowIndex[nZmax]	- row index for each corresponding element in value[]
int colStart[nCol+1]	- index of first element in each column

- used by packages/libraries derived from FORTRAN which store arrays in column-major format, e.g. MATLAB
- analogous format to CSR matrices



Conclusions

- Large, sparse, linear systems are widely used in scientific computing
- Generally result from discretising PDEs
- Contain significant structure
- Require specialised storage techniques
- Require specialised solution techniques - most commonly iterative methods