Numerical Linear Algebra

Knut–Andreas Lie

Dept. of Informatics, University of Oslo

Numerical Linear Algebra

One of the most important issues in numerical simulation

- large systems of linear equations result from the discretization of ODEs and PDEs
- linear systems are typically sparse because of local discretization stencils

Direct solvers are often not suited

- computing time grows quickly with the number of unknowns, typically $\mathcal{O}(n^2)$ of $\mathcal{O}(n^3)$
- classical elimination destroys sparisity by fill-in $\longrightarrow$ need for additional storage
- exact solution not necessarily needed as the solution of the linear system itself is an approximation

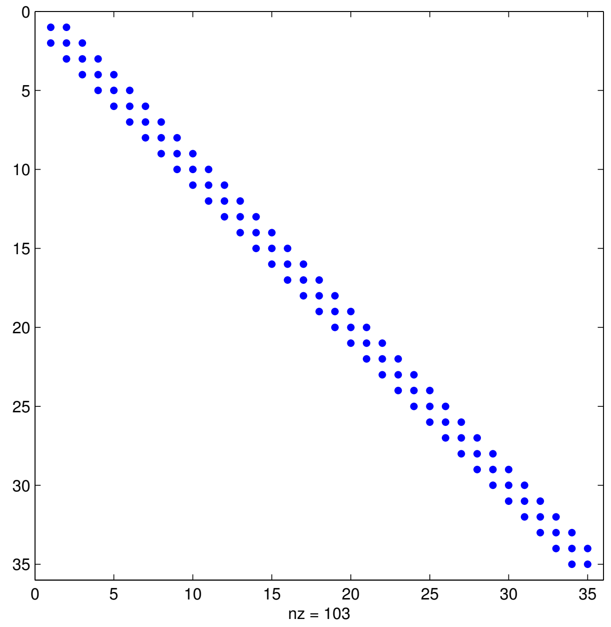
Motivation: Heat Equation Revisited

$$-u''(x) = f(x),$$

$$u_{i+1} - 2u_i + u_{i-1} = -h^2 f_i,$$

Tridiagonal linear system:

- $3n - 2$ nonzero entries



Linear algebra:

- $\mathcal{O}(n^3)$ operations for Gaussian elimination
- $\mathcal{O}(n)$ operations using tridiagonal structure (LU decomposition)

Table: CPU time in seconds for grid with n unknowns

	125	250	500	1000	2000
heat	0.07	0.6	5.9	48.9	391.5
heatTri	0.01	0.01	0.01	0.03	0.05

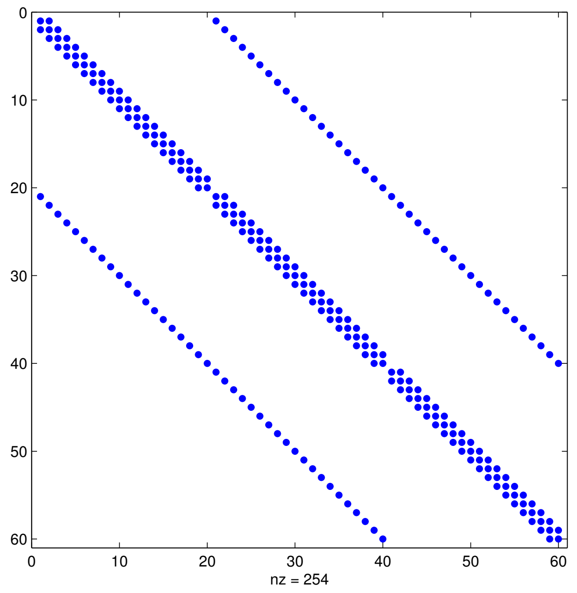
Motivation, cont'd

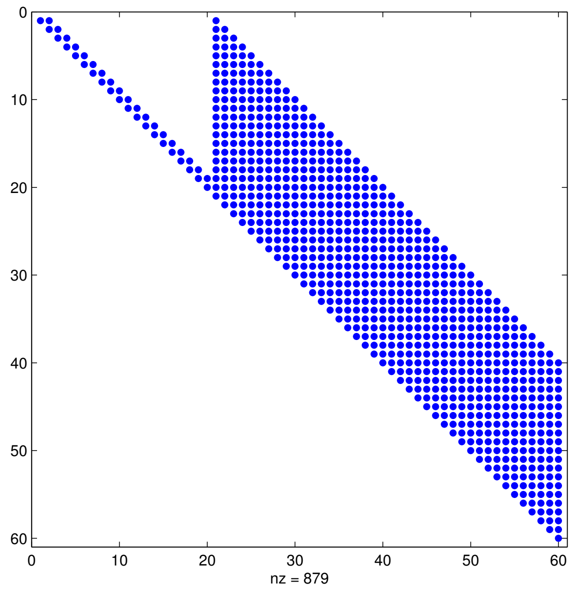
$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

Discretization: FDM or FEM

Pentadiagonal system

- Only $5n$ out of n^2 entries are nonzero
- LU decomposition gives fill-in: going from from $5n$ to $\mathcal{O}(N^{3/2})$ nonzero elements $\longrightarrow$ need for extra storage
- Even worse in 3D..





Iterative Solvers

Goals for approximate, iterative solvers

- compute a series of approximations

$$x^0, x^1, x^2, \ldots$$

  that converge to the correct solution x

- take advantage of the sparsity pattern and introduce no or little extra storage requirements
- try to keep the growth in computational time as low as possible, i.e., as close to $\mathcal{O}(n)$ as possible

Iterative Solvers, cont'd

Different types:

- stationary relaxation methods
  - Jacobi, Gauss–Seidel, (S)SOR
  - older methods
  - easy to understand and implement
  - slow convergence
- nonstationary (Krylov) methods:
  - steepest descent, conjugate gradient
  - minimal residual, ...
  - based upon orthogonal vectors
  - may give fast convergence
- domain decomposition, multigrid/multilevel,..
  - advanced, complex, but fast methods

Relaxation Techniques

Assume an approximate solution x^k. The residual $r^k = b - Ax^k$ gives an indicator for the error.

Relaxation: use residual to improve upon approximation

Residual equation:

$$r^k = b - Ax^k = A(x - x^k) = Ae^k$$

Basic idea: Solve a modified residual equation, which is easier to solve

$$Be^k = r^k, \qquad B \sim A$$

Then set: $\quad x^{k+1} = x^k + e^k$

Relaxation Techniques, cont'd

Question: How should we choose the matrix B?

- B should be chosen similar to A, i.e., such that $B^{-1} \approx A^{-1}$
- B should be chosen such that $Be = r$ is easy to solve

Algorithm:

$$\begin{aligned} Be^k = B(x^{k+1} - x^k) = r^k = b - Ax^k \\ \longrightarrow Bx^{k+1} = b - (A - B)x^k \end{aligned}$$

Thus, in each iteration we have to perform a matrix multiplication and solve a linear system which is easier to solve than the original

Example: Jacobi Iteration

Choose $B = \mathrm{diag}(A)$, i.e., diagonal elements of A

$$x_i^{k+1} = \frac{1}{A_{ii}} \left[b_i - \sum_{j \neq i} A_{ij} x_j^k \right]$$

Remarks:

- all x_i^{k+1} are computed from old values x_i^k
- the update of different x_is are independent and can be performed in parallel
- implementation requires one extra vector
- works good only for strongly diagonally-dominant systems

Example: Gauss–Seidel Iteration

Choose $B = L_A$, i.e., lower tridiagonal part of A

$$x_i^{k+1} = \frac{1}{A_{ii}}\left[b_i - \sum_{j<i} A_{ij}x_j^{k+1} - \sum_{j>i} A_{ij}x_j^{k}\right]$$

Remarks:

- new approximations x_i^{k+1} are used to compute the rest of the approximation
- the update of different x_is are not independent, but depend on the order and is therefore a *serial* algorithm
- does not require extra memory, as updates can be performed directly in the vector x
- usually faster convergence than the Jacobi iteration

Example: Successive Over-Relaxation (SOR)

Write $x^{k+1} = \omega \bar{x}^{k+1} + (1-\omega)x^k, \quad \omega \in [0,2]$

$$x_i^{k+1} = \frac{\omega}{A_{ii}}\left[b_i - \sum_{j<i} A_{ij}x_j^{k+1} - \sum_{j>i} A_{ij}x_j^k\right] + (1-\omega)x_i^k$$

Remarks:

- the method is similar to the Gauss–Seidel algorithm
- the value of ω is used to accelerate the convergence
- for a given discretization: red-black ordering may turn the algorithm from serial to parallel
- problem: choosing the parameter ω optimally (advanced implementations try to estimate optimal ω during iteration)
- SSOR: alternating forward-backward sweeps

Nonstationary Iterative Methods

= methods that involve information that changes at each iteration

Important class — Krylov subspace methods:

- assume simple basic iteration: $x^{k+1} = x^k + r^k$
- observe that $r^{k+1} = b - Ax^{k+1} = b - A(x^k + r^k) = (I - A)r^k$
- now if we assume that $x^0 = 0$

$$x^{k+1} = r^0 + r^1 + \cdots + r^k = \sum_{n=1}^{k} (I - A)^n r_0$$
$$\in \operatorname{span}\{r^0, Ar^0, \ldots A^k r^0\}$$

This space is called the Krylov subspace. The iterative method gives elements of Krylov subspaces of increasing order

Idea: use better approximations from the Krylov subspace

Example: the Conjugate Gradient Method

Update by a multiple of an optimal search vector

$$x^{k+1} = x^k + \alpha^{k+1} p^{k+1}$$

Similarly, update residuals

$$r^{k+1} = r^k - \alpha^k q^{k+1}, \qquad q^{k+1} = Ap^{k+1}$$

where

$$\alpha^{k+1} = \frac{(r^k)^T \cdot r^k}{(p^{k+1})^T A p^{k+1}},$$

and the search directions are obtained from

$$p^{k+1} = r^k + \beta^k p^k, \qquad \beta^k = \frac{(r^k)^T \cdot r^k}{(r^{k-1})^T r^{k-1}}$$

Conjugate Gradients, cont'd

- The method of choice for symmetric, positive definite systems
- Construction principle: orthogonal residuals

$$(r^\ell)^T r^k = 0, \quad \text{if} \quad \ell \neq k$$

- Common to use *preconditioner*, i.e., solve an equation of the form

$$M^{-1}Ax = M^{-1}b$$

  where M approximates A and $M^{-1}A$ has a better condition number.

  Examples:

  – Jacobi: $M = \mathrm{diag}(A)$
  – SSOR: $M = (D+L)D^{-1}(D+L)^T$, if $A = L + D + L^T$

How Do We Store Sparse Matrices?

$$A = \begin{pmatrix} a_{1,1} & 0 & 0 & a_{1,4} & 0 \\ 0 & a_{2,2} & a_{2,3} & 0 & a_{2,5} \\ 0 & a_{3,2} & a_{3,3} & 0 & 0 \\ a_{4,1} & 0 & 0 & a_{4,4} & a_{4,5} \\ 0 & a_{5,2} & 0 & a_{5,4} & a_{5,5} \end{pmatrix}$$

- Only a small fraction of the entries are nonzero
- Utilizing sparsity is essential for computational efficiency!
- Implementation:

$$\begin{aligned} \texttt{A} &= (a_{1,1}, a_{1,4}, a_{2,2}, a_{2,3}, a_{2,5}, \ldots) \\ \texttt{irow} &= (1, 3, 6, 8, 11, 14), \\ \texttt{jcol} &= (1, 4, 2, 3, 5, 2, 3, 1, 4, 5, 2, 4, 5). \end{aligned}$$

Sparse Matrices in FORTRAN

Code example for $y = Mx$

```
integer p, q, nnz
integer irow(p+1), jcol (nnz)
double precision M(nnz), x(q), y(p)
 ...
call  prodvs (M, p, q, nnz, irow,  jcol , x, y)
```

Two major drawbacks:

- Explicit transfer of storage structure (5 args)
- Different name for two functions that perform the same task on two different matrix formats

Sparse Matrices using Objects

```
class MatSparse
{
  private:
    double* A;      // long vector with the nonzero matrix entries
    int*    irow;   // indexing array
    int*    jcol;   // indexing array
    int     m, n;   // A is (logically) m times n
    int     nnz;    // number of nonzeroes
  public:
    // the same functions as in the example above
    // plus functionality for initializing the data structures
};
```

What has been gained?

- Users cannot see the sparse matrix data structure
- Matrix-vector product syntax remains the same
- Easy to switch between `MatDense` and `MatSparse`

Programming with Matrices

What is a matrix?

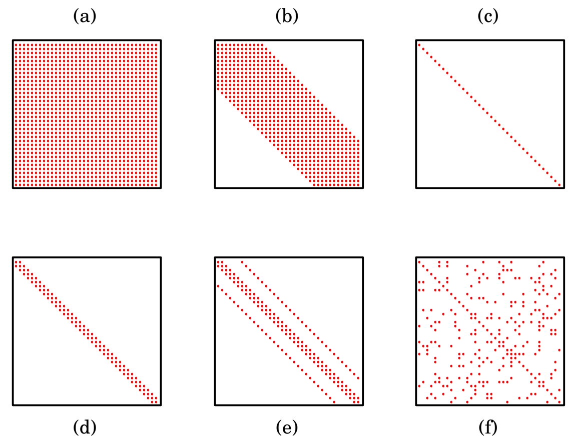
A well defined mathematical quantity, containing a table of numbers and a set of legal operations

How do we program with matrices?

- Do standard arrays in any computer language give good enough support for matrices?
- No! We need a jungle of formats for different types of matrices
- Why? The efficiency and memory requirements of numerical methods depend strongly upon the structure of the matrix and the storage scheme.

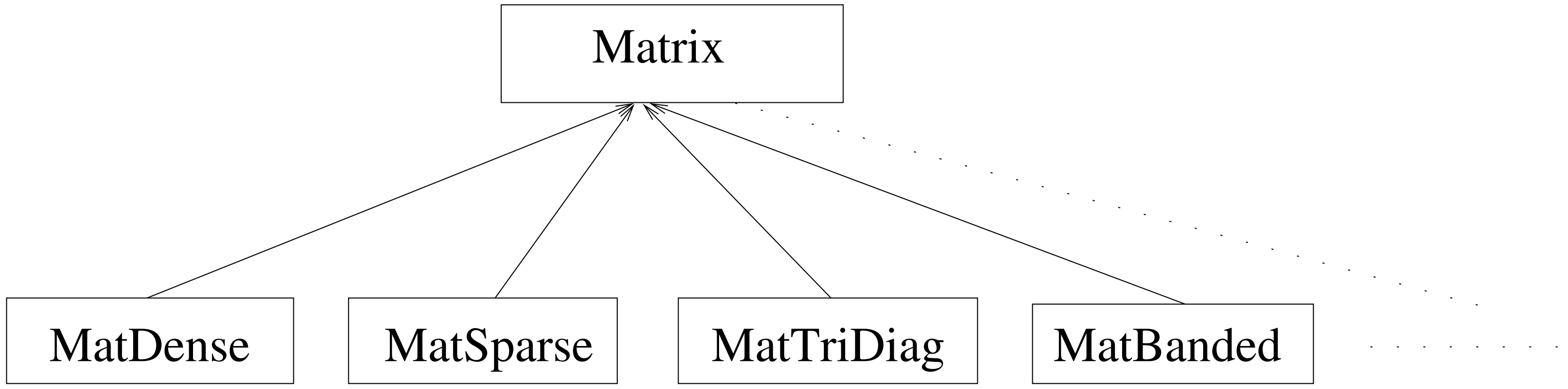
The Jungle of Matrix Formats

- Suppose we want to hide details of storage in a class
- Unfortunately there are a lot of formats:
  - dense matrices
  - banded matrices
  - n-diagonal matrices
  - general sparse matrices
  - structured sparse matrices
  - finite difference stencil as matrices, ...



- Who is interested in knowing all details of the data structures? — Very few!
- Scenario: one often has has to try different storage forms to get maximal code efficiency.

Object-Oriented Realization



- Matrix = object
- Common interface to matrix operations
- Base class: define operations, no data
- Subclasses: implement specific storage schemes and algorithms
- Details of storage schemes are hidden
- It is possible to program with the base class only!

Object-Oriented Realization, cont'd

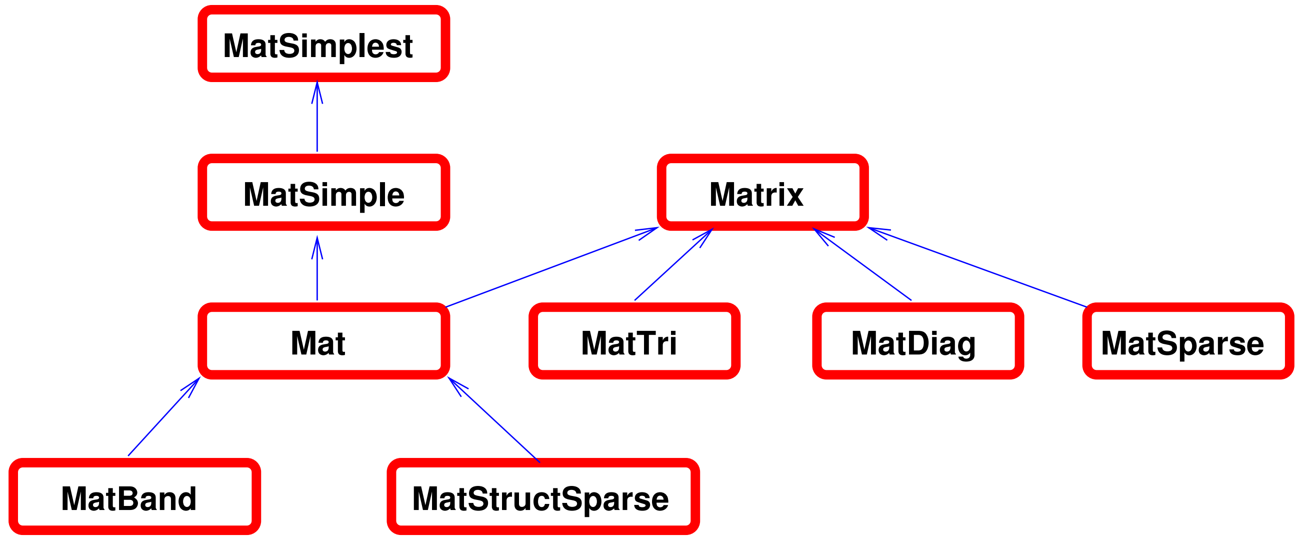
- Generic programming in user code via base class:

```
Matrix& M;
M.prod(x,y);   // y=M*x
```

  i.e., we need not know the structure of `M`, only that it refers to some concrete subclass object.

- Member functions are *virtual* functions

Example from Diffpack:



Every Rose Has its Thorns...

- Object-oriented programming do wonderful things, but might be *inefficient*
- Adjusted picture:
  - The *user* of the class and numerical linear algebra packages does seldom need to know the storage structure
  - For the *develper* it is essential in order to develop efficient metods

$\Rightarrow$ Object-oriented numerics: balance between efficiency and nice object-oriented design