Solving PDEs: the Poisson problem

TMA4280—Introduction to Supercomputing

Based on 2016v slides by Eivind Fonn

NTNU, IMF

March 9. 2018

The Poisson problem

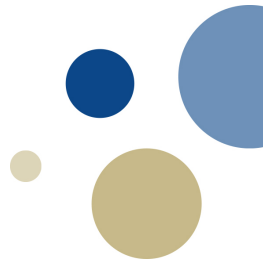
- The Poisson equation is an elliptic partial differential equation.
- The Poisson *problem* is the solution of the Poisson *equation* equipped with a set of boundary conditions.
- The equation is

$$-\Delta u = f, \qquad \text{in } \Omega$$

  where u is the unknown, f is the load on the system and Ω denotes some domain.
- We remark that Δ is the sum of the second order partial derivatives (i.e trace of Hessian matrix), e.g. in one and two dimensions the equation is

$$-u_{xx} = f, \qquad -(u_{xx} + u_{yy}) = f.$$

The Poisson problem



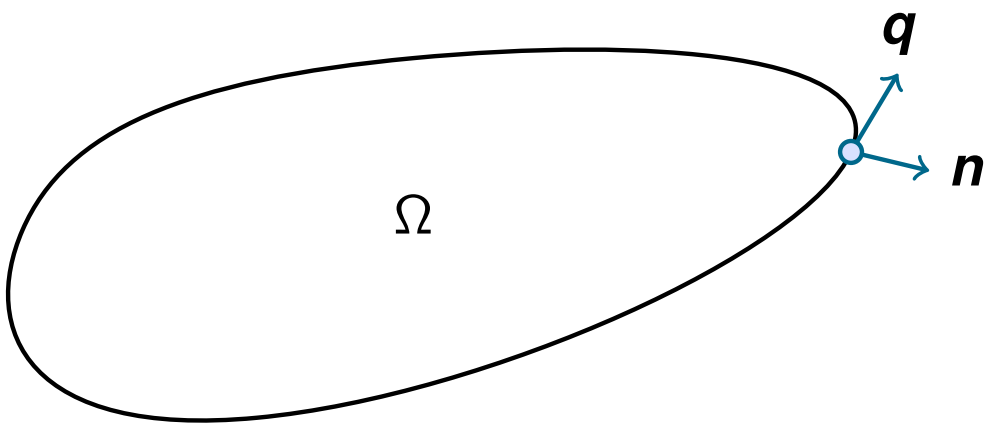
- This problem is important because a number of physical processes are modelled either entirely or in part by the Poisson equation.
- The technical term is a *diffusion process*: u is to be interpreted as the concentration of a physical quantity, and f as the rate at which is introduced ($f > 0$) or removed ($f < 0$) from the domain.
- The physical quantity (typically intensive) may be a concentration, temperature, potentials, ...
- The solution in Ω is uniquely determined by the boundary data and f: this problem involves an elliptic equation.
- Intuitively, the differential operator has a *smoothing* effect.

Steady heat transfer

Energy transfered out of an arbitrary domain V can be expressed as

$$\int_{\partial V} \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}S = \int_{V} f \, \mathrm{d}V.$$

where $\boldsymbol{q}$ is the heat flux, $\boldsymbol{n}$ is the outward surface normal along the boundary ∂V and f represents a volumetric heat source. This basically says that the net energy generation inside the domain must equal the net energy flowing out of the domain.



Steady heat transfer

— Applying the Gauss divergence theorem, we can write

$$\int_{\partial V} \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}S = \int_V \nabla \cdot \boldsymbol{q} \, \mathrm{d}V = \int_V f \, \mathrm{d}V,$$

yielding

$$\nabla \cdot \boldsymbol{q} = f.$$

— Applying Fourier's law, i.e. $\boldsymbol{q} = -\kappa \nabla u$, $\kappa > 0$, we get

$$-\nabla \cdot \kappa \nabla u = f \qquad \text{in } \Omega,$$

to be solved for the temperature u.

— For a constant isotropic heat conductivity κ, we regain the Poisson equation,

$$-\kappa \Delta u = f.$$

Applications: Electrostatics

— The differential forms for the electric field $\boldsymbol{E}$ are

$$\nabla \cdot \boldsymbol{E} = 4\pi\rho$$
$$\nabla \times \boldsymbol{E} = 0,$$

where ρ is the charge density.

— The electric field $\boldsymbol{E}$ can be expressed as the gradient of a scalar potential φ, i.e. $\boldsymbol{E} = -\nabla\varphi$. Thus

$$\nabla \cdot \boldsymbol{E} = -\nabla \cdot \nabla\varphi = -\Delta\varphi = 4\pi\rho.$$

Applications: Potential flow

— Likewise, potential flow in fluid mechanics can be modelled by this equation.

— Given a velocity field $\boldsymbol{U}$ which is irrotational and incompressible,

$$\nabla \times \boldsymbol{U} = 0$$
$$\nabla \cdot \boldsymbol{U} = 0,$$

it follows that $\boldsymbol{U} = \nabla\varphi$ where φ is a scalar velocity potential, which satisfies the Laplace equation

$$\Delta\varphi = 0.$$

Applications: Numerical methods

Use of the Poisson equation can also be guided by the numerics and not explicitly by the physical model:

— Projection methods for incompressible Navier–Stokes: Helmoltz decomposition of L^2 function into divergence free L^2 function and gradients of H^1 function.

1. Velocity prediction: solving the momentum equation.
2. Projection step: project the predicted velocity onto solenoidal space.

— Homogenization methods for multiscale problems: solving a local problem, typically a Poisson problem on a cell, to model small scales.

In any case, computing an approximation of a solution to the Poisson requires:

— a discretization (physical space, frequential space)

— and a discretization in time in the case of evolution problems.

Unsteady heat transfer: discretization in time

— Unsteady heat transfer is modelled by the heat equation

$$\frac{\partial u}{\partial t} = \kappa \Delta u + f \qquad \text{in } \Omega.$$

— Discretizing in time using Backward Euler, we otain

$$\frac{1}{\Delta t}(u^{n+1} - u^n) = \kappa \Delta^{n+1} + f^{n+1}$$

where superscript n refers to a quanity at time t^n, $n = 0, 1, \ldots$.

— This can be written as

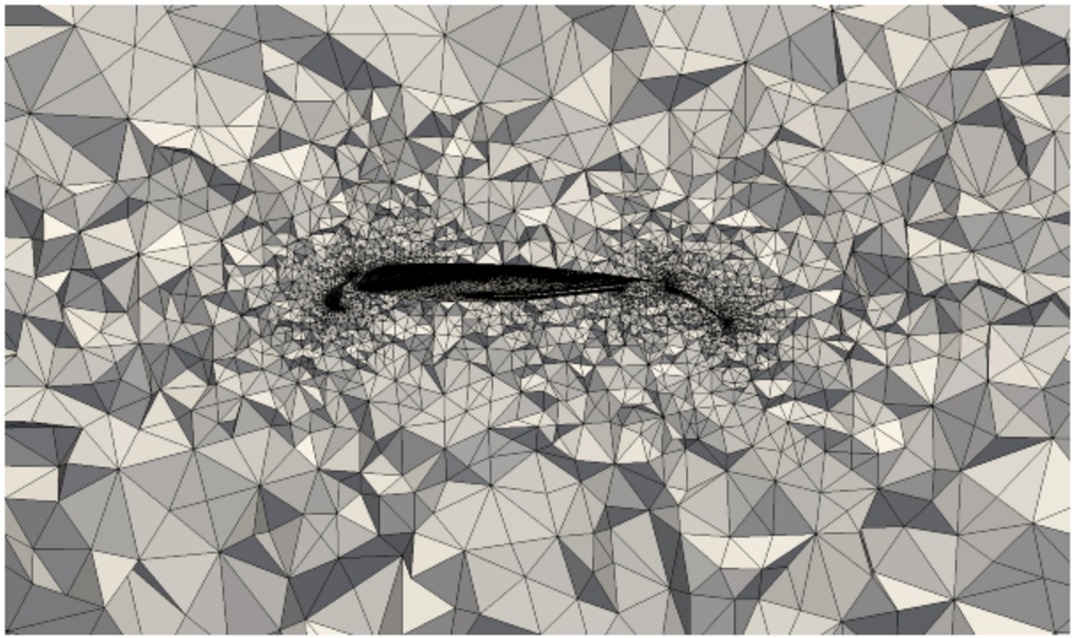
$$\left(-\kappa\Delta + \frac{1}{\Delta t}\right) u^{n+1} = \frac{u^n}{\Delta t} + f^{n+1}.$$

— This is a *Helmholtz* equation: Laplacian plus a multiple of the identity.

Solving PDEs: mesh generation

Depending on the discretization, cartesian grids or simplicial meshes may be used.

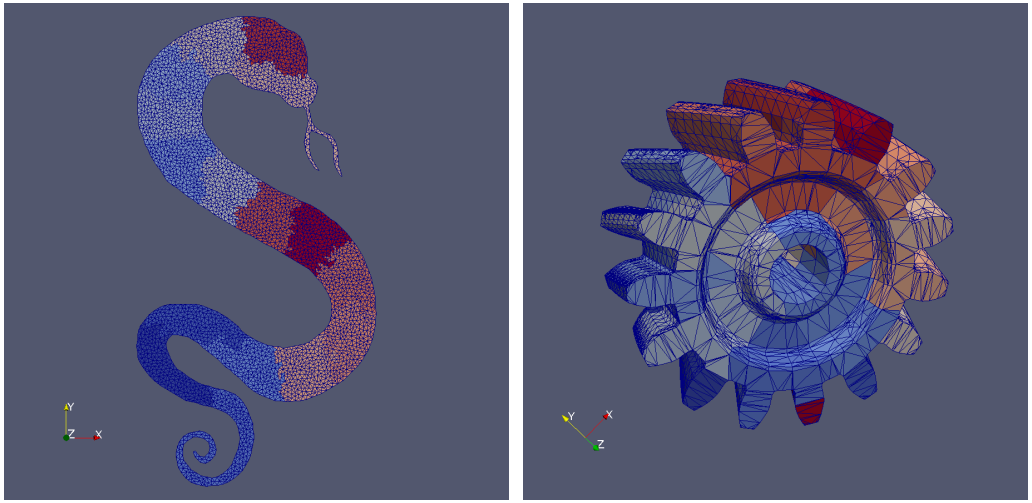
Complex geometries are usually more suited for simplicial meshes:



but numerical methods like immersed boundaries or fictitious domain approaches maybe used.

Solving PDEs: mesh distribution, domain decomposition

1. Solving in parallel requires mesh distribution:



Different partitioning software packages: ParMetis, Zoltan, Scotch.

2. Domain decomposition: not to be confused with mesh distribution! Such methods can be used for:
   - accelerate the resolution,
   - deal with multiphysics,
   - solve on composite domains.

Solving PDEs: adaptivity

1. *h-adaptivity*: refine the mesh by dividing cells
2. *p-adaptivity*: increase the polynomial order,
3. *r-adaptivity*: move mesh to increase accuracy locally,

Goal-oriented adaptivity: adaptation w.r.t a goal functional

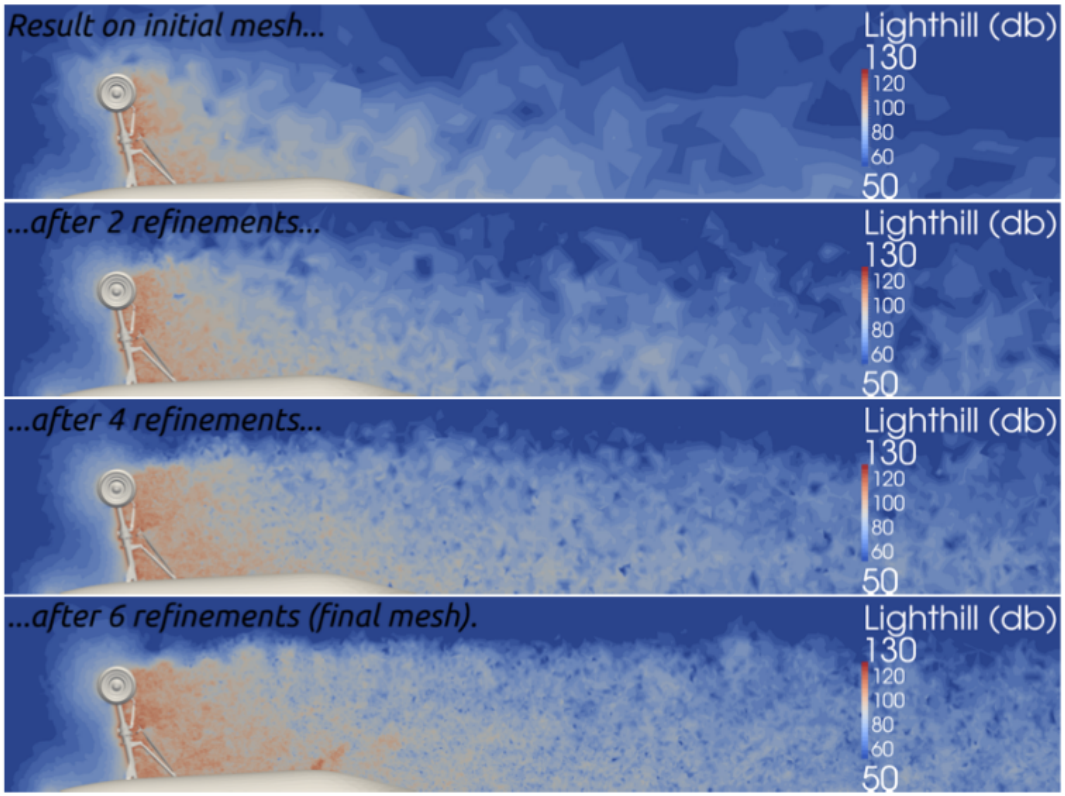
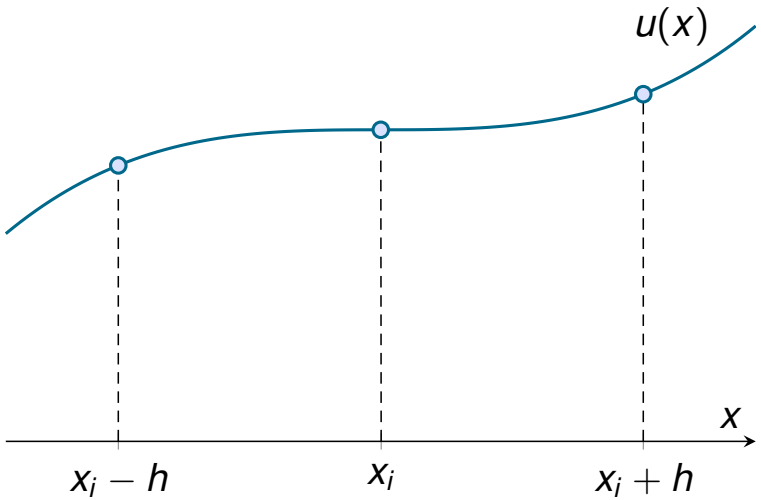


Figure: Lighthill tensor. Vilela De Abreu/N. Jansson/Hoffman (2012)

Finite difference methods

- Consider a continuous function 1D function $u(x)$.
- Introduce a grid, $\{x_i\}_{i=0}^{N}$, with $x_i = x_0 + ih$.
- Here h is the grid spacing. For simplicity we consider equidistant grids (constant h).



- Want to approximate derivatives of the function only using data on the grid.

Finite differences

— First idea: linear approximation of slope

$$u'(x_i) \approx \frac{1}{h}\left(u(x_i + h) - u(x_i)\right).$$

— This is called a one-sided difference (a *forward* difference). Invoking Taylor we find

$$\begin{aligned}\frac{1}{h}\left(u(x_i + h) - u(x_i)\right) &= \frac{1}{h}\left(u(x_i) + hu'(x_i) + \mathcal{O}(h^2) - u(x_i)\right) \\ &= u'(x_i) + \mathcal{O}(h)\end{aligned}$$

In other words, this is a *first order* approximation to $u'(x_i)$.

Finite differences

— Second idea: a centered difference

$$u'(x_i) \approx \frac{1}{2h}\left(u(x_i + h) - u(x_i - h)\right).$$

— Invoking Taylor we find

$$\frac{1}{2h}\left(u(x_i + h) - u(x_i - h)\right) = u'(x_i) + \mathcal{O}(h^2).$$

In other words, this is a *second order* approximation to $u'(x_i)$.

Finite difference methods

— A centered difference for the second derivative:

$$u''(x_i) \approx \frac{1}{h}\left(u'(x_i + {}^{h}/_{2}) - u'(x_i - {}^{h}/_{2})\right)$$
$$\approx \frac{1}{h^2}\left(u(x_i + h) - 2u(x_i) + u(x_i - h)\right)$$

— Invoking Taylor we find

$$\frac{1}{h^2}\left(u(x_i + h) - 2u(x_i) + u(x_i - h)\right) = u''(x_i) + \mathcal{O}(h^2).$$

In other words, this is a *second order* approximation to $u''(x_i)$.

Stencils

Finite differences are often illustrated using *stencils*.

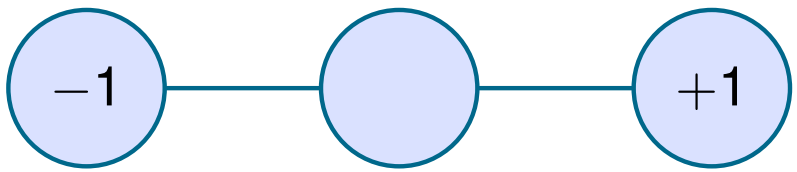


Figure: Stencil for the first derivative (central difference)

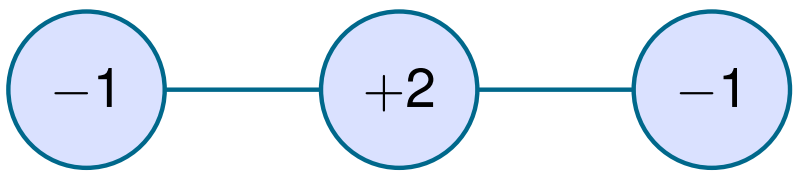
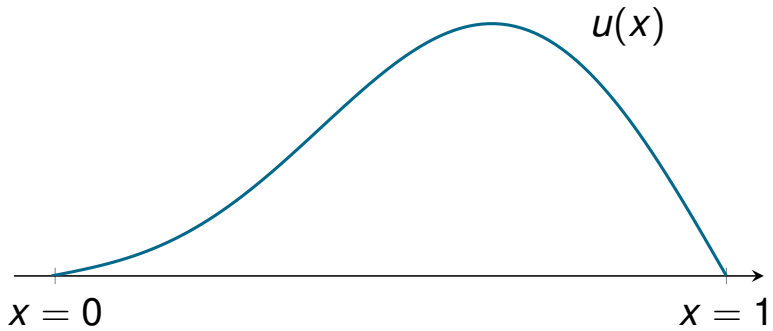


Figure: Stencil for the second derivative

Discretization in 1D

— We now consider the 1D Poisson problem

$$-u_{xx} = f, \qquad \text{in } \Omega = (0,1),$$
$$u(0) = u(1) = 0.$$



— These are called homogenous Dirichlet boundary conditions: the solution is prescribed to be zero on the boundaries of the domain.

— Introduce the grid, $\{x_i\}_{i=0}^{N}$, with $x_i = x_0 + ih$, $h = 1/N$.

Discretization in 1D

— Let u_i denote $u(x_i)$ and f_i denote $f(x_i)$.

— Due to the boundary conditions, we know that $u_0 = u_N = 0$.

— We thus have $N - 1$ unknowns that we collect in a vector $\boldsymbol{u}$.

— We then apply the second order finite difference formula in each grid point.

$$-\frac{1}{h^2}\left(u_{i+1} - 2u_i + u_{i-1}\right) = f_i, \qquad i = 1, \ldots, n-1,$$

$$u_0 = u_N = 0.$$

The algebraic unknowns are also called degrees of freedoms: computing a discrete solution consists of determining a numerical for each degree of freedom.

Discretization in 1D

These equations can also be expressed as the system

$$
\begin{aligned}
2u_1 - u_2 &= h^2 f_2 \\
-u_1 + 2u_2 - u_3 &= h^2 f_3 \\
&\vdots \\
-u_{N-2} + 2u_{N-1} &= h^2 f_{N-1},
\end{aligned}
$$

or in matrix form

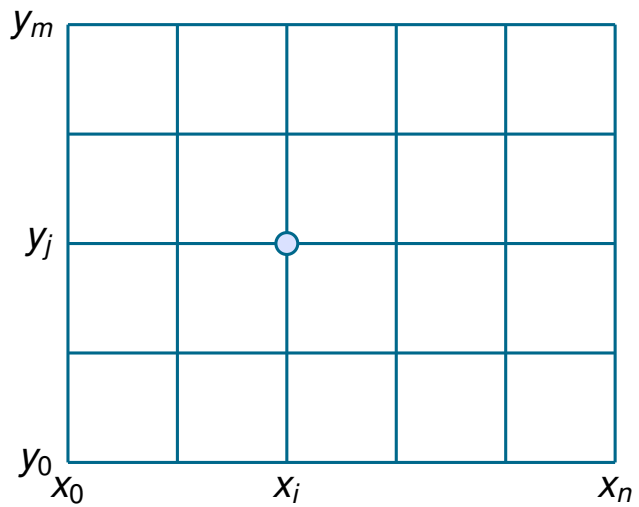
$$
\underbrace{\begin{pmatrix}
2 & -1 & & & \\
-1 & 2 & -1 & & \\
 & \ddots & \ddots & \ddots & \\
 & & -1 & 2 & -1 \\
 & & & -1 & 2
\end{pmatrix}}_{\boldsymbol{A}}
\underbrace{\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix}}_{\boldsymbol{u}}
= h^2
\underbrace{\begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-2} \\ f_{n-1} \end{pmatrix}}_{\boldsymbol{f}},
$$

Discretization in 1D

- The matrix $\boldsymbol{A}$ is sparse (tridiagonal).
- The matrix $\boldsymbol{A}$ is symmetric, that is $\boldsymbol{A} = \boldsymbol{A}^\mathsf{T}$.
- The matrix $\boldsymbol{A}$ is positive definite, that is, $\boldsymbol{v}^\mathsf{T}\boldsymbol{A}\boldsymbol{v} > 0$ for all vectors $\boldsymbol{v} \in \mathbb{R}^{N-1}$, $\boldsymbol{v} \neq \boldsymbol{0}$.
- Thus, the system of $N-1$ equations is solvable and has a unique solution.
- The error in the grid points is of second order, i.e. $|u(x_i) - u_i| \sim \mathcal{O}(h^2)$.

Discretization in 2D

If the domain Ω is rectangular, we can use two independent grids $\{x_i\}_{i=0}^{N}$ and $\{y_j\}_{j=0}^{M}$ in the x and y-directions.



$$x_i = x_0 + ih_x,$$
$$y_j = y_0 + jh_y.$$

Finite differences in 2D

We need to approximate $u_{xx} + u_{yy}$ at a point (x_i, y_j). Therefore, we use two one-dimensional finite differences,

$$u_{xx}(x_i, y_i) \approx \frac{1}{h_x^2} (u_{i-1,j} - 2u_{i,j} + u_{i+1,j})$$

$$u_{yy}(x_i, y_i) \approx \frac{1}{h_y^2} (u_{i,j-1} - 2u_{i,j} + u_{i,j+1})$$

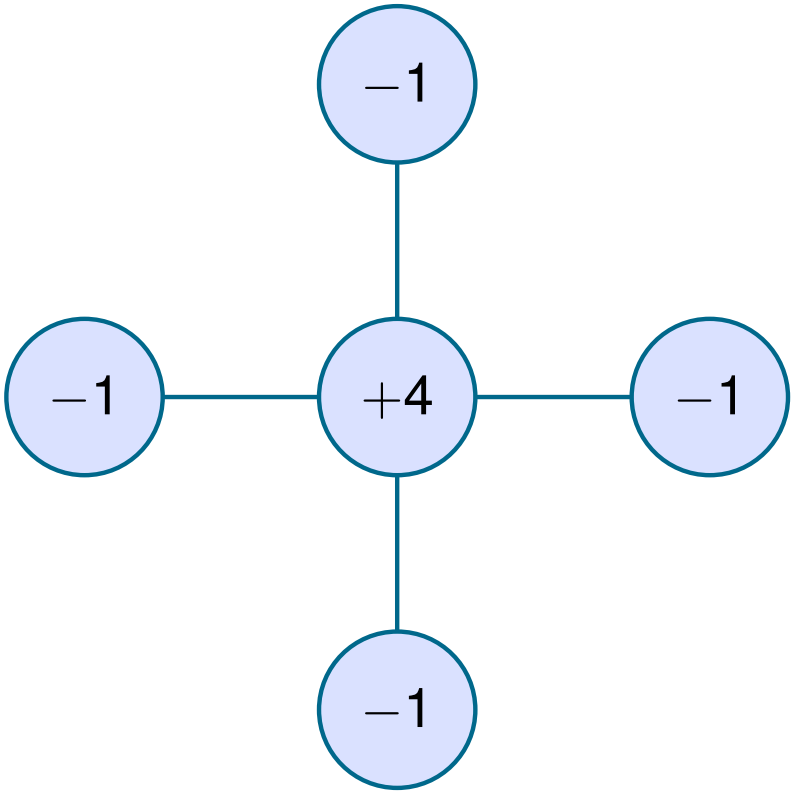
If $h_x = h_y = h$ we get

$$\nabla u(x_i, y_i) \approx \frac{1}{h^2} (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j})$$

2D finite differences result from the discretization of the differential operator along each axis.

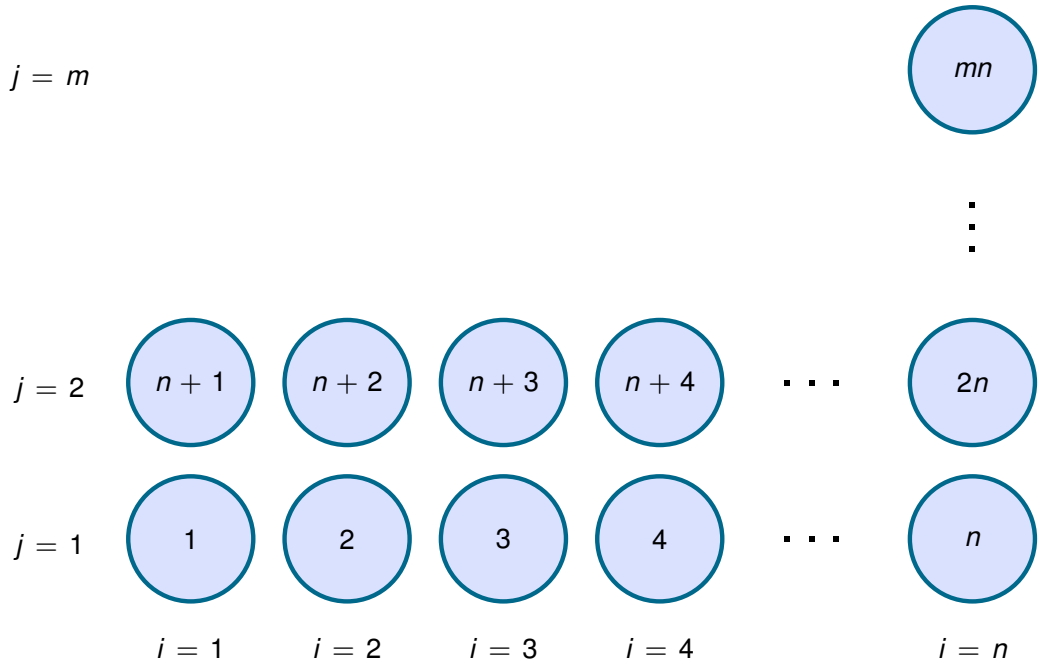
Finite differences in 2D

This is known as the *five-point* stencil. (The signs are flipped because the Poisson equation involves $-\nabla$.)



Node numbering in 2D

Numbering nodes is not as trivial in 2D as in 1D. We use a “natural” ordering of the unknowns: we first number all the *internal* nodes along “row” 1 (in the x-direction), followed by the nodes in “row” 2, etc.



The numbering of the nodes is important as it is reflected in the structure of the matrix if the same numbering is used for the algebraic system is used.

Matrix block structure

The final linear system will have a block-like structure,

$$\underbrace{\begin{pmatrix} \boldsymbol{A}_0 & \boldsymbol{A}_1 & & & \\ \boldsymbol{A}_1 & \boldsymbol{A}_0 & \boldsymbol{A}_1 & & \\ & \boldsymbol{A}_1 & \boldsymbol{A}_0 & \ddots & \\ & & \ddots & \ddots & \boldsymbol{A}_1 \\ & & & \boldsymbol{A}_1 & \boldsymbol{A}_0 \end{pmatrix}}_{\boldsymbol{A}} \underbrace{\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_N \end{pmatrix}}_{\boldsymbol{u}} = \underbrace{\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_N \end{pmatrix}}_{\boldsymbol{g}}$$

The resulting matrix is *sparse*, which means that it contains only a few non-zero entries.

This structure is the algebraic counterpart of the locality of the stencil: the differential operator only involves neighbours.

Matrix block structure

The blocks $\boldsymbol{A}_0$ and $\boldsymbol{A}_1$ can be written as

$$\boldsymbol{A}_0 = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & -1 & 4 & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 4 \end{pmatrix} \qquad \boldsymbol{A}_1 = \begin{pmatrix} -1 & & & & \\ & -1 & & & \\ & & -1 & & \\ & & & \ddots & \\ & & & & -1 \end{pmatrix}.$$

Assembling linear systems

Assembly of the linear system may be non-negligible in terms of computational time:

- Depending on the time discretization (explicit, implicit, . . .) the matrix may or may not be reassembled.
- Assembling the contributions may be expensive (finite elements).
- Performance may be improved by parallelism (OpenMP).
- Parallelism can be also used for overlapping assembly and solve.

Solving linear systems

Three sources of performance:

1. general improvement of algorithms e.g Krylov solvers and preconditioners,
2. exploiting the structure of the discrete problem by choosing a specific solver,
3. using parallelism in implementations (OpenMP).

Improvement can of course come from improving serial implementations: optimized assembly, template meta-programming, optimization of kernels by code generation . . .

Linear algebra packages

BLAS	Linear algebra routines, reference (FORTRAN)
OpenBLAS	Optimized multithreaded BLAS (C, Assembly)
LAPACK	Direct and eigenvalue solvers (FORTRAN)
Armadillo	Linear algebra package (C++)
SuiteSparse	Factorizations with GPU support
MUMPS	Parallel sparse matrix direct solvers (FORTRAN)

Software packages like PETSc, Trilinos provde general interfaces to these packages.

Linear algebra performance

Performance is measured with the number of floating-point operations per second (FLOPS).

Performance of linear algebra operations is heavily influenced by:

1. ratio between computations and data movement,
2. memory access patterns.

Performance can be further improved by parallelism: some algorithms offer intrisically more opportunities for parallelism than other (think in terms of data dependencies)

Linear algebra performance

Vector-vector operations (AXPY):

1. $2n$ ops
2. $2n$ data

$\rightarrow r_{ops/data} = O(1)$ performance guided by ILP and caching limits.

Matrix-matrix operations (MM): $A \in \mathbb{R}^{mxk}, B \in \mathbb{R}^{kxn}$

1. $mn(2k-1)$ ops
2. $(m+n)k$ data

$\rightarrow r_{ops/data} = O(N)$ data reuse by several floating-point operations.

Such considerations were explained when introducing computational efficiency of dense linear algebra benchmarks (LINPACK) vs. sparse matrix benchmarks (HPCG)

BLAS

— A very helpful library here is *BLAS*: Basic Linear Algebra Subprograms.

— BLAS is an old specification from the late seventies and early eighties.

— It consist of a collection of functions with strangely cryptic and short names.

— Can be installed on Ubuntu with `sudo apt-get install libblas-dev`.

— On Vilje, Intel's implementation of BLAS is available under *MKL*: the Math Kernel Library, and on Lille OpenBLAS is available.

BLAS levels

— BLAS functions are organized by *level*.

— Level 1: vector-vector operations.

$$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$$

```
daxpy(n, alpha, y, 1, x, 1)
```

— Level 2: matrix-vector operations.

$$\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$$

```
dgemv('N', m, n, alpha, A, m, x, 1, beta, y, 1)
```

— Level 3: matrix-matrix operations

$$\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$$

```
dgemm('N', 'N', m, n, k, alpha, A, m, B, k, C, m)
```

BLAS conventions

— All functions in BLAS starts with one of the letters
  - s for *single*.
  - d for *double*.
  - c for *single complex*.
  - z for *double complex*.

— If the operation involves a matrix, two letters describing the matrix format follow. The most important of these are
  - ge for *general* matrices.
  - po for *symmetric* matrices.
  - gb for *general banded* matrices.
  - pb for *symmetric banded* matrices.

BLAS

- The BLAS home page can be found at http://netlib.org/blas/
- BLAS is written in Fortran and therefore expects Fortran memory layout (column-major ordering).
- For C, the *CBLAS* implementation is popular. CBLAS supports both row- and column-major orders.

Serial BLAS

```
void MxV(double *u, double *A, double *v, int N)
{
  dgemv('N', N, N, 1.0, A, N, v, 1, 0.0, u, 1);
}

double innerproduct(double *u, double *v, int N)
{
  // Necessary adjustments must be made for MPI
  return ddot(N, u, 1, v, 1);
}
```

Simple example

Computing

$$\alpha = \sum_{i=1}^{K} \boldsymbol{v}_i^{\mathsf{T}} \boldsymbol{A} \boldsymbol{v}_i$$

where $\boldsymbol{v} \in \mathbb{R}^{N \times K}$ and $\boldsymbol{A} \in \mathbb{R}^{N \times N}$. There are K terms, each of which require us to compute a matrix-vector product and an inner product.

Serial

```
void MxV(double *u, double **A, double *v, int N)
{
  for (size_t i = 0; i < N; i++) {
    u[i] = 0.0;
    for (size_t j = 0; j < N; j++)
      u[i] += A[i][j] * v[j];
  }
}

double innerproduct(double *u, double *v, int N)
{
  double result = 0.0;
  for (size_t i = 0; i < N; i++)
    result += u[i] * v[i];
  return result;
}
```

Serial

```
double dosum(double **A, double **v, int K, int N)
{
  double alpha = 0.0, temp[N];
  for (size_t i = 0; i < K; i++) {
    MxV(temp, A, v[i], N);
    alpha += innerproduct(temp, v[i], N);
  }

  return alpha;
}
```

OpenMP micro-version

- It is tempting to exploit all parallelism in sight. However, don't do that.
- Let us use OpenMP for micro-parallelism. That is, we exploit parallelism within the inner product and the matrix-vector operation.
- That means two fork/join operations per term, so $2K$ in total.

OpenMP micro-version

```
void MxV(double *u, double **A, double *v, int N)
{
  #pragma omp parallel for schedule(static)
  for (size_t i = 0; i < N; i++) {
    u[i] = 0.0;
    for (size_t j = 0; j < N; j++)
      u[i] += A[i][j] * v[j];
  }
}
```

OpenMP micro-version

```
double innerproduct(double *u, double *v, int N)
{
  double result = 0.0;
  #pragma omp parallel for schedule(static) \
          reduction(+:result)
  for (size_t i = 0; i < N; i++)
    result += u[i] * v[i];
  return result;
}
```

OpenMP macro-version

— The alternative is to exploit the coarsest parallelism: each iteration in the `dosum` method.

— In this case we perform exactly one fork and one join.

— Problem: the `dosum` method uses a temporary buffer for the matrix-vector multiplication, which cannot be shared between threads. We have to use a separate buffer for each thread.

OpenMP macro-version

```
double dosum(double **A, double **v, int K, int N)
{
  double alpha = 0.0;
  double **temp = createMatrix(K, N);
  #pragma omp parallel for schedule(static) \
          reduction(+:alpha)
  for (size_t i = 0; i < K; i++) {
    MxV(temp[i], A, v[i], N);
    alpha += innerproduct(temp[i], v[i], N);
  }

  return alpha;
}
```

MPI macro-version

```
double dosumMPI(double **A, double **v,
                int myK, int N)
{
  double myalpha = dosum(A, V, myK, N);
  double alpha;
  MPI_Allreduce(&myalpha, &alpha, 1, MPI_DOUBLE,
                MPI_SUM, MPI_COMM_WORLD);
  return alpha;
}
```

In addition to the usual MPI code, you have to decide on a particular division of the work among the nodes. For brevity's sake, it is left out in this example.

Speedup results

Table: $N = 2048$, $K = 1024$

Threads	Micro	Macro	MPI
1	1.00	1.00	1.00
2	1.84	1.83	1.56
4	2.79	2.76	3.46

Table: $N = 16$, $K = 32768$

Threads	Micro	Macro	MPI
1	1.00	1.00	1.00
2	0.50	2.00	2.00
4	0.33	3.49	4.00

Timing results

Table: $N = 2048$, $K = 1024$

Threads	Macro	w/ BLAS	MPI	w/ BLAS
1	35.20	2.06	35.27	2.05
2	17.68	1.06	18.73	1.17
4	9.08	0.66	9.15	0.61
8	4.54	0.36	4.82	0.32

Timing results

Table: $N = 16$, $K = 32768$ (milliseconds)

Threads	Macro	w/ BLAS	MPI	w/ BLAS
1	9.44	9.10	10.71	9.36
2	20.08	24.31	7.62	4.48
4	15.20	28.78	6.20	6.23
8	7.36	23.89	5.58	4.68