Simulation and High-Performance Computing

Part 5: Finite Difference Methods for Partial Differential Equations

Steffen Börm

Christian-Albrechts-Universität zu Kiel

September 30th, 2020

Poisson's equation

Model problem: Poisson's equation on the unit square

$$-\Delta u(x) = f(x) \qquad \text{for all } x \in \Omega := (0,1) \times (0,1),$$

Laplace operator: Partial differential operator of second order,

$$\Delta u(x) = \frac{\partial^2 u}{\partial x_1^2}(x) + \frac{\partial^2 u}{\partial x_2^2}(x).$$

Poisson's equation

Model problem: Poisson's equation on the unit square

$$\begin{aligned} -\Delta u(x) &= f(x) && \text{for all } x \in \Omega := (0,1) \times (0,1), \\ u(x) &= 0 && \text{for all } x \in \partial\Omega = \{0,1\} \times [0,1] \cup [0,1] \times \{0,1\}, \end{aligned}$$

with Dirichlet boundary conditions.

Laplace operator: Partial differential operator of second order,

$$\Delta u(x) = \frac{\partial^2 u}{\partial x_1^2}(x) + \frac{\partial^2 u}{\partial x_2^2}(x).$$

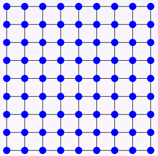
Applications: Electrostatic fields, heat dissipation, wave propagation, fluid dynamics, ...

Grids and grid functions

Problem: The domain Ω contains infinitely many points, but a computer can only store finitely many values.

Grid: Replace the domain by a finite number of points.
Choose $N \in \mathbb{N}$, let $h := \frac{1}{N+1}$ and

$$\Omega_h := \{(ih, jh) \ : \ i, j \in [1 : N]\}, \qquad \bar{\Omega}_h := \Omega_h \cup \partial\Omega_h,$$
$$\partial\Omega_h := \{(ih, jh) \ : \ i, j \in [0 : N+1], \ i \in \{0, N+1\} \vee j \in \{0, N+1\}\}.$$

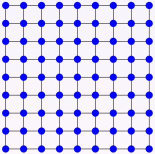


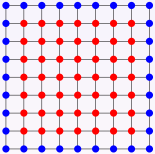
Grids and grid functions

Problem: The domain Ω contains infinitely many points, but a computer can only store finitely many values.

Grid: Replace the domain by a finite number of points.
Choose $N \in \mathbb{N}$, let $h := \frac{1}{N+1}$ and

$$\Omega_h := \{(ih, jh) \ : \ i, j \in [1 : N]\}, \qquad \bar{\Omega}_h := \Omega_h \cup \partial\Omega_h,$$
$$\partial\Omega_h := \{(ih, jh) \ : \ i, j \in [0 : N+1], \ i \in \{0, N+1\} \vee j \in \{0, N+1\}\}.$$



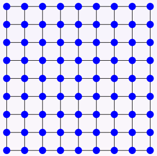


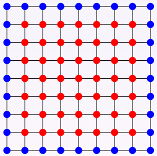
Grids and grid functions

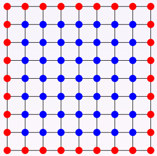
Problem: The domain Ω contains infinitely many points, but a computer can only store finitely many values.

Grid: Replace the domain by a finite number of points.
Choose $N \in \mathbb{N}$, let $h := \frac{1}{N+1}$ and

$$\Omega_h := \{(ih, jh) \ : \ i,j \in [1 : N]\}, \qquad \bar{\Omega}_h := \Omega_h \cup \partial\Omega_h,$$
$$\partial\Omega_h := \{(ih, jh) \ : \ i,j \in [0 : N+1], \ i \in \{0, N+1\} \vee j \in \{0, N+1\}\}.$$





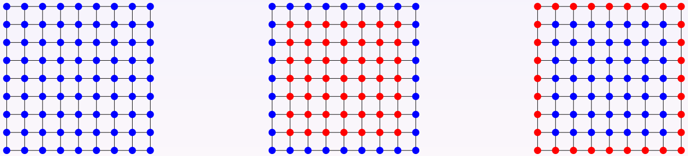


Grids and grid functions

Problem: The domain Ω contains infinitely many points, but a computer can only store finitely many values.

Grid: Replace the domain by a finite number of points.
Choose $N \in \mathbb{N}$, let $h := \frac{1}{N+1}$ and

$$\Omega_h := \{(ih, jh) \ : \ i, j \in [1:N]\}, \qquad \bar{\Omega}_h := \Omega_h \cup \partial\Omega_h,$$
$$\partial\Omega_h := \{(ih, jh) \ : \ i, j \in [0:N+1], \ i \in \{0, N+1\} \vee j \in \{0, N+1\}\}.$$



Grid function: Replace a function $u \colon \bar{\Omega} \to \mathbb{R}$ by a mapping $u_h \colon \bar{\Omega}_h \to \mathbb{R}$.

Finite difference approximation

Problem: We cannot evaluate derivatives of a grid function, since it is only defined in discrete points.

Approach: Replace differential operators by difference quotients.

$$g''(t) \approx \frac{g(t+h) - 2\,g(t) + g(t-h)}{h^2},$$

Finite difference approximation

Problem: We cannot evaluate derivatives of a grid function, since it is only defined in discrete points.

Approach: Replace differential operators by difference quotients.

$$
\begin{aligned}
g''(t) &\approx \frac{g(t+h) - 2\,g(t) + g(t-h)}{h^2}, \\
\Delta u(x) &= \frac{\partial^2 u}{\partial x_1^2}(x) + \frac{\partial^2 u}{\partial x_2^2}(x) \\
&\approx \frac{u(x_1+h, x_2) - 2\,u(x) + u(x_1-h, x_2)}{h^2} \\
&\qquad + \frac{u(x_1, x_2+h) - 2\,u(x) + u(x_1, x_2-h)}{h^2} =: \Delta_h u(x).
\end{aligned}
$$

Finite difference approximation

Problem: We cannot evaluate derivatives of a grid function, since it is only defined in discrete points.

Approach: Replace differential operators by difference quotients.

$$
\begin{aligned}
g''(t) &\approx \frac{g(t+h) - 2\,g(t) + g(t-h)}{h^2}, \\
\Delta u(x) &= \frac{\partial^2 u}{\partial x_1^2}(x) + \frac{\partial^2 u}{\partial x_2^2}(x) \\
&\approx \frac{u(x_1+h, x_2) - 2\,u(x) + u(x_1-h, x_2)}{h^2} \\
&\qquad + \frac{u(x_1, x_2+h) - 2\,u(x) + u(x_1, x_2-h)}{h^2} =: \Delta_h u(x).
\end{aligned}
$$

Finite difference approximation

Problem: We cannot evaluate derivatives of a grid function, since it is only defined in discrete points.

Approach: Replace differential operators by difference quotients.

$$\begin{aligned}
g''(t) &\approx \frac{g(t+h) - 2\,g(t) + g(t-h)}{h^2}, \\
\Delta u(x) &= \frac{\partial^2 u}{\partial x_1^2}(x) + \frac{\partial^2 u}{\partial x_2^2}(x) \\
&\approx \frac{u(x_1+h, x_2) - 2\,u(x) + u(x_1-h, x_2)}{h^2} \\
&\qquad + \frac{u(x_1, x_2+h) - 2\,u(x) + u(x_1, x_2-h)}{h^2} =: \Delta_h u(x).
\end{aligned}$$

Finite difference approximation

Problem: We cannot evaluate derivatives of a grid function, since it is only defined in discrete points.

Approach: Replace differential operators by difference quotients.

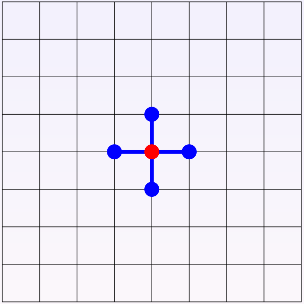
$$
\begin{aligned}
g''(t) &\approx \frac{g(t+h) - 2\,g(t) + g(t-h)}{h^2}, \\
\Delta u(x) &= \frac{\partial^2 u}{\partial x_1^2}(x) + \frac{\partial^2 u}{\partial x_2^2}(x) \\
&\approx \frac{u(x_1+h, x_2) - 2\,u(x) + u(x_1-h, x_2)}{h^2} \\
&\qquad + \frac{u(x_1, x_2+h) - 2\,u(x) + u(x_1, x_2-h)}{h^2} =: \Delta_h u(x).
\end{aligned}
$$

Important: If $x \in \Omega_h$, the approximation $\Delta_h u_h(x)$ is well-defined.

Discrete Laplace operator

Five-point stencil: The Laplace operator is approximated by

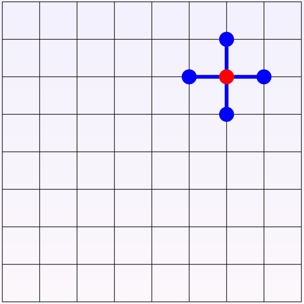
$$\Delta_h u_h(x) = \frac{u(x_1 + h, x_2) + u(x_1 - h, x_2) + u(x_1, x_2 + h) + u(x_1, x_2 - h) - 4u(x)}{h^2}$$



Discrete Laplace operator

Five-point stencil: The Laplace operator is approximated by

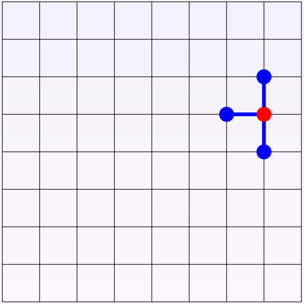
$$\Delta_h u_h(x) = \frac{u(x_1 + h, x_2) + u(x_1 - h, x_2) + u(x_1, x_2 + h) + u(x_1, x_2 - h) - 4u(x)}{h^2}$$



Discrete Laplace operator

Five-point stencil: The Laplace operator is approximated by

$$\Delta_h u_h(x) = \frac{u(x_1 + h, x_2) + u(x_1 - h, x_2) + u(x_1, x_2 + h) + u(x_1, x_2 - h) - 4u(x)}{h^2}$$

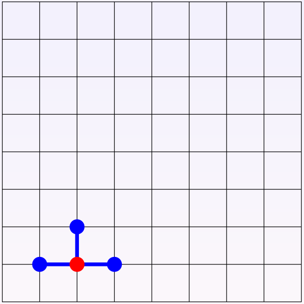


Boundary points are a special case, since they are not degrees of freedom.

Discrete Laplace operator

Five-point stencil: The Laplace operator is approximated by

$$\Delta_h u_h(x) = \frac{u(x_1 + h, x_2) + u(x_1 - h, x_2) + u(x_1, x_2 + h) + u(x_1, x_2 - h) - 4u(x)}{h^2}$$

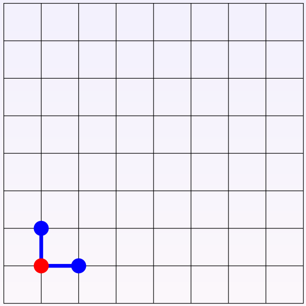


Boundary points are a special case, since they are not degrees of freedom.

Discrete Laplace operator

Five-point stencil: The Laplace operator is approximated by

$$\Delta_h u_h(x) = \frac{u(x_1+h,x_2) + u(x_1-h,x_2) + u(x_1,x_2+h) + u(x_1,x_2-h) - 4u(x)}{h^2}$$



Boundary points are a special case, since they are not degrees of freedom.

Linear system

Boundary value problem: Find $u \in C(\bar{\Omega})$ with $u|_\Omega \in C^2(\Omega)$ such that

$$\begin{aligned} -\Delta u(x) &= f(x) && \text{for all } x \in \Omega, \\ u(x) &= 0 && \text{for all } x \in \partial\Omega. \end{aligned}$$

Finite difference approximation: Find $u_h \colon \bar{\Omega}_h \to \mathbb{R}$ such that

$$\begin{aligned} -\Delta_h u_h(x) &= f(x) && \text{for all } x \in \Omega_h, \\ u_h(x) &= 0 && \text{for all } x \in \partial\Omega_h. \end{aligned}$$

Linear system

Boundary value problem: Find $u \in C(\bar{\Omega})$ with $u|_\Omega \in C^2(\Omega)$ such that

$$\begin{aligned} -\Delta u(x) &= f(x) && \text{for all } x \in \Omega, \\ u(x) &= 0 && \text{for all } x \in \partial\Omega. \end{aligned}$$

Finite difference approximation: Find $u_h \colon \bar{\Omega}_h \to \mathbb{R}$ such that

$$\begin{aligned} -\Delta_h u_h(x) &= f(x) && \text{for all } x \in \Omega_h, \\ u_h(x) &= 0 && \text{for all } x \in \partial\Omega_h. \end{aligned}$$

Result: We approximate the partial differential equation by a linear system of dimension $n := N^2$.

Lexicographic enumeration

Problem: The index set $\bar{\Omega}_h$ is two-dimensional, but standard programming languages prefer one-dimensional structures.

Approach: Enumerate row by row.

$$x = (ih, jh) \quad \mapsto \quad i + (N+2)\,j \qquad\qquad i, j \in [0 : N+1].$$

Implementation: Grid functions represented by arrays with $(N+2)^2$ elements, $u_h(ih, jh) =$`u[i+j*yinc]` with `yinc` $= N + 2$.

Lexicographic enumeration

Problem: The index set $\bar{\Omega}_h$ is two-dimensional, but standard programming languages prefer one-dimensional structures.

Approach: Enumerate row by row.

$$x = (ih, jh) \quad \mapsto \quad i + (N+2)\,j \qquad\qquad i, j \in [0 : N+1].$$

Implementation: Grid functions represented by arrays with $(N+2)^2$ elements, $u_h(ih, jh) =$`u[i+j*yinc]` with `yinc` $= N + 2$.

Boundary points can be eliminated, since they are not degrees of freedom.

$$x = (ih, jh) \quad \mapsto \quad (i-1) + N\,(j-1) \qquad\qquad i, j \in [1 : N],$$

this leads to arrays with $n := N^2$ elements containing only unknown grid values $u_h(ih, jh) =$`u[(i-1)+(j-1)*yinc]` with `yinc` $= N$.

Matrix structure

Approach: Degrees of freedom in the j-th row of a grid function u_h are collected in a vector $u^{(j)} \in \mathbb{R}^N$ with $u_i^{(j)} := u_h(ih, jh)$, $i, j \in [1 : N]$.

Linear system is now n-dimensional with $n = N^2$, given by

$$\frac{1}{h^2} \begin{pmatrix} T & -I & & \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{pmatrix} \begin{pmatrix} u^{(1)} \\ u^{(2)} \\ \vdots \\ u^{(N)} \end{pmatrix} = \begin{pmatrix} f^{(1)} \\ f^{(2)} \\ \vdots \\ f^{(N)} \end{pmatrix}$$

with the tridiagonal matrix

$$T := \begin{pmatrix} 4 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{N \times N}.$$

Direct solvers

Problem: We have to solve the linear system $Au = f$ with the matrix

$$A := \frac{1}{h^2} \begin{pmatrix} T & -I & & \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{pmatrix}, \qquad T := \begin{pmatrix} 4 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}.$$

First approach: If n is not too large, we can use a direct solver, e.g., Gaussian elimination.

Direct solvers

Problem: We have to solve the linear system $Au = f$ with the matrix

$$A := \frac{1}{h^2} \begin{pmatrix} T & -I & & \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{pmatrix}, \qquad T := \begin{pmatrix} 4 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}.$$

First approach: If n is not too large, we can use a direct solver, e.g., Gaussian elimination.

Refined approach: Use Krylov methods or multigrid iteration, these are topics in parts 8 and 9 of this course.

Example: LU factorization

Triangular matrices:

- $L \in \mathbb{R}^{n\times n}$ is lower triangular if $\ell_{ij} = 0$ for all $i < j$.
- $U \in \mathbb{R}^{n\times n}$ is upper triangular if $u_{ij} = 0$ for all $i > j$.

Triangular systems $Lx = b$ and $Ux = b$ can be solved by forward and backward substitution.

LU factorization: A matrix $A \in \mathbb{R}^{n\times n}$ is split into $A = LU$.

$$Ax = b \qquad \iff \qquad Ly = b \text{ and } Ux = y.$$

Forward substitution

Goal: Solve $Lx = b$ with a lower triangular matrix L.

Approach: Split into submatrices and -vectors.

$$L = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ x_* \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_* \end{pmatrix}.$$

We obtain

$$\begin{pmatrix} \ell_{11}x_1 \\ L_{*1}x_1 + L_{**}x_* \end{pmatrix} = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} x_1 \\ x_* \end{pmatrix} = Lx = b = \begin{pmatrix} b_1 \\ b_* \end{pmatrix}$$

and can solve $\ell_{11}x_1 = b_1$ directly and

$$L_{**}x_* = b_* - L_{*1}x_1$$

by recursion, since L_{**} is again lower triangular.

Backward substitution

Goal: Solve $Ux = b$ with an upper triangular matrix U.

Approach: Split into submatrices and -vectors.

$$U = \begin{pmatrix} U_{**} & U_{*n} \\ & u_{nn} \end{pmatrix}, \qquad x = \begin{pmatrix} x_* \\ x_n \end{pmatrix}, \qquad b = \begin{pmatrix} b_* \\ b_n \end{pmatrix}.$$

We obtain

$$\begin{pmatrix} U_{**}x_* + U_{*n}x_n \\ u_{nn}x_n \end{pmatrix} = \begin{pmatrix} U_{**} & U_{*n} \\ & u_{nn} \end{pmatrix} \begin{pmatrix} x_* \\ x_n \end{pmatrix} = Ux = b = \begin{pmatrix} b_* \\ b_n \end{pmatrix}$$

and can solve $u_{nn}x_n = b_n$ directly and

$$U_{**}x_* = b_* - U_{*n}x_n$$

by recursion, since U_{**} is again upper triangular.

LU factorization

Goal: Given A, find L lower and U upper triangular with $A = LU$.

Approach: Split into submatrices.

$$A = \begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix}, \qquad L = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix}, \qquad U = \begin{pmatrix} u_{11} & U_{1*} \\ & U_{**} \end{pmatrix}.$$

We obtain

$$\begin{pmatrix} \ell_{11} u_{11} & \ell_{11} U_{1*} \\ L_{*1} u_{11} & L_{**} U_{**} + L_{*1} U_{1*} \end{pmatrix} = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} u_{11} & U_{1*} \\ & U_{**} \end{pmatrix} = \begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix}$$

and can solve $\ell_{11} u_{11} = a_{11}$ directly (usually $\ell_{11} = 1$, $u_{11} = a_{11}$), $L_{*1} u_{11} = A_{*1}$ and $\ell_{11} U_{1*} = A_{1*}$ by scaling, and use recursion for

$$L_{**} U_{**} = A_{**} - L_{*1} U_{1*}.$$

Arrays and pointers in C

Arrays and pointers: In C, arrays and pointers are almost synonymous. If `A` is a pointer, `A[k]` means accessing the element that can be found k steps behind the one `A` points to.

Pointer arithmetic: We can move pointers by adding integers.

```
real A[] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
real *B, *C;

B = A + 2;
printf("%f\n", B[1]);   /* Yields B[1] = A[2+1] = 4.0 */

C = B - 1;
printf("%f\n", C[3]);   /* Yields C[3] = B[2] = A[4] = 5.0 */
```

Array representation of matrices and vectors

Vectors $x \in \mathbb{R}^n$ are represented by

- a pointer `x` to the first coefficient,
- an increment `incx` that takes us to the next coefficient, and
- the dimension `n`.

We can find x_i in `x[(i-1)*incx]`, $i \in [1 : n]$.

Array representation of matrices and vectors

Vectors $x \in \mathbb{R}^n$ are represented by

- a pointer `x` to the first coefficient,
- an increment `incx` that takes us to the next coefficient, and
- the dimension `n`.

We can find x_i in `x[(i-1)*incx]`, $i \in [1:n]$.

Matrices $A \in \mathbb{R}^{m \times n}$ are represented by

- a pointer `A` to the first coefficient a_{11},
- a leading dimension `ldA` that takes us to the next column, and
- the dimensions `rowsA` and `colsA`.

We can find a_{ij} in `A[(i-1)+(j-1)*ldA]`, $i \in [1:m]$, $j \in [1:n]$.

Array representation of matrices and vectors

Vectors $x \in \mathbb{R}^n$ are represented by

- a pointer `x` to the first coefficient,
- an increment `incx` that takes us to the next coefficient, and
- the dimension `n`.

We can find x_i in `x[(i-1)*incx]`, $i \in [1 : n]$.

Matrices $A \in \mathbb{R}^{m \times n}$ are represented by

- a pointer `A` to the first coefficient a_{11},
- a leading dimension `ldA` that takes us to the next column, and
- the dimensions `rowsA` and `colsA`.

We can find a_{ij} in `A[(i-1)+(j-1)*ldA]`, $i \in [1 : m]$, $j \in [1 : n]$.

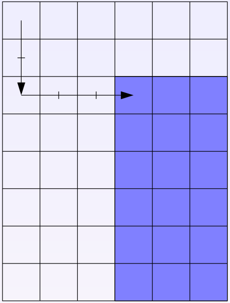
Standard practice: Better to use indices $[0 : n-1]$ instead of $[1 : n]$.

Submatrices and -vectors

Submatrix: If $A \in \mathbb{R}^{n \times m}$ is given,

```
B = A + 2 + 3 * ldA;
ldB = ldA;
rowsB = rowsA - 2;
colsB = colsA - 3;
```

defines the submatrix starting in the third row and the fourth column.

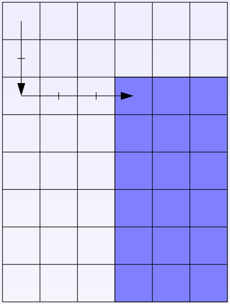


Submatrices and -vectors

Submatrix: If $A \in \mathbb{R}^{n \times m}$ is given,

```
B = A + 2 + 3 * ldA;
ldB = ldA;
rowsB = rowsA - 2;
colsB = colsA - 3;
```

defines the submatrix starting in the third row and the fourth column.



Subvectors: The second row and the third column are represented by

```
x = A + 1;          incx = ldA;   n = colsA;
y = A + 2 * ldA;    incy = 1;     m = rowsA;
```

BLAS

Basic Linear Algebra Subprograms can be used to take advantage of highly optimized implementations of basic operations on matrices and vectors.

BLAS Level 1 contains vector operations, for example

- scaling a vector, $x \leftarrow \alpha x$:

```
void scal(int n, real alpha,
          real *x, int incx);
```

- adding two vectors, $y \leftarrow y + \alpha x$:

```
void axpy(int n, real alpha,
          const real *x, int incx,
          real *y, int incy);
```

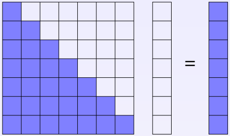
- computing the Euclidean norm $\|x\|_2 = (x_1^2 + \ldots + x_n^2)^{1/2}$:

```
real nrm2(int n, const real *x, int incx);
```

Forward substitution with BLAS

Goal: Solve $Lx = b$. Using the block notation

$$\begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} x_1 \\ x_* \end{pmatrix} = \begin{pmatrix} b_1 \\ b_* \end{pmatrix},$$



equivalent with $x_1 = b_1/\ell_{11}$ and $L_{**}x_* = b_* - L_{*1}x_1$.

Recursion: We have to apply the algorithm to the submatrix L_{**} and its submatrices. $\rightarrow$ Store the row/column offset in a variable `k`.

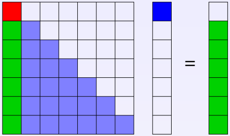
```
for(k=0; k<n; k++) {
  /* Divide by the diagonal element */
  x[k] = b[k] / L[k+k*ldL];

  /* Subtract L_{*1} x_1 from right-hand side */
  axpy(n-k-1, -x[k], L+(k+1)+k*ldL, 1,
       b+(k+1), 1);
}
```

Forward substitution with BLAS

Goal: Solve $Lx = b$. Using the block notation

$$\begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} x_1 \\ x_* \end{pmatrix} = \begin{pmatrix} b_1 \\ b_* \end{pmatrix},$$



equivalent with $x_1 = b_1/\ell_{11}$ and $L_{**}x_* = b_* - L_{*1}x_1$.

Recursion: We have to apply the algorithm to the submatrix L_{**} and its submatrices. $\rightarrow$ Store the row/column offset in a variable `k`.

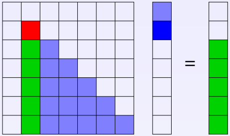
```
for(k=0; k<n; k++) {
  /* Divide by the diagonal element */
  x[k] = b[k] / L[k+k*ldL];

  /* Subtract L_{*1} x_1 from right-hand side */
  axpy(n-k-1, -x[k], L+(k+1)+k*ldL, 1,
       b+(k+1), 1);
}
```

Forward substitution with BLAS

Goal: Solve $Lx = b$. Using the block notation

$$\begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} x_1 \\ x_* \end{pmatrix} = \begin{pmatrix} b_1 \\ b_* \end{pmatrix},$$



equivalent with $x_1 = b_1/\ell_{11}$ and $L_{**}x_* = b_* - L_{*1}x_1$.

Recursion: We have to apply the algorithm to the submatrix L_{**} and its submatrices. $\rightarrow$ Store the row/column offset in a variable k.

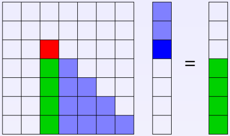
```
for(k=0; k<n; k++) {
  /* Divide by the diagonal element */
  x[k] = b[k] / L[k+k*ldL];

  /* Subtract L_{*1} x_1 from right-hand side */
  axpy(n-k-1, -x[k], L+(k+1)+k*ldL, 1,
       b+(k+1), 1);
}
```

Forward substitution with BLAS

Goal: Solve $Lx = b$. Using the block notation

$$\begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} x_1 \\ x_* \end{pmatrix} = \begin{pmatrix} b_1 \\ b_* \end{pmatrix},$$



equivalent with $x_1 = b_1/\ell_{11}$ and $L_{**}x_* = b_* - L_{*1}x_1$.

Recursion: We have to apply the algorithm to the submatrix L_{**} and its submatrices. $\rightarrow$ Store the row/column offset in a variable `k`.

```
for(k=0; k<n; k++) {
  /* Divide by the diagonal element */
  x[k] = b[k] / L[k+k*ldL];

  /* Subtract L_{*1} x_1 from right-hand side */
  axpy(n-k-1, -x[k], L+(k+1)+k*ldL, 1,
       b+(k+1), 1);
}
```

BLAS Level 2

Goal: In order to compute an LU factorization, we have to work with matrices, not vectors.

BLAS Level 2 contains matrix-vector operations, for example

- computing a matrix-vector product, $y \leftarrow y + \alpha Ax$ or $y \leftarrow y + \alpha A^T x$:

```
void gemv(bool trans, int rows, int cols, real alpha,
          const real *A, int ldA,
          const real *x, int incx,
          real *y, int incy);
```

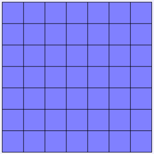
- adding a rank-one product to a matrix, $A \leftarrow A + \alpha xy^T$:

```
void ger(int rows, int cols, real alpha,
         const real *x, int incx,
         const real *y, int incy,
         real *A, int ldA);
```

LU decomposition with BLAS

Goal: Find LU decomposition. Block notation

$$\begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix} = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} u_{11} & U_{1*} \\ & U_{**} \end{pmatrix}$$



leads to $\ell_{11} = 1$, $u_{11} = a_{11}$, $U_{1*} = A_{1*}$,
$L_{*1} u_{11} = A_{*1}$, $L_{**} U_{**} = A_{**} - L_{*1} U_{1*}$

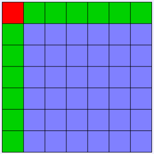
In-place algorithm overwrites A directly with L and U.

```
for(k=0; k<n; k++) {
  scal(n-k-1, 1.0/A[k+k*ldA], A+(k+1)+k*ldA, 1);
  ger(n-k-1, n-k-1, -1.0,
      A+(k+1)+ k   *ldA, 1,
      A+ k   +(k+1)*ldA, ldA,
      A+(k+1)+(k+1)*ldA, ldA);
}
```

LU decomposition with BLAS

Goal: Find LU decomposition. Block notation

$$\begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix} = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} u_{11} & U_{1*} \\ & U_{**} \end{pmatrix}$$



leads to $\ell_{11} = 1$, $u_{11} = a_{11}$, $U_{1*} = A_{1*}$,
$L_{*1} u_{11} = A_{*1}$, $L_{**} U_{**} = A_{**} - L_{*1} U_{1*}$

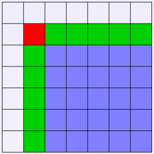
In-place algorithm overwrites A directly with L and U.

```
for(k=0; k<n; k++) {
  scal(n-k-1, 1.0/A[k+k*ldA], A+(k+1)+k*ldA, 1);
  ger(n-k-1, n-k-1, -1.0,
      A+(k+1)+ k   *ldA, 1,
      A+ k   +(k+1)*ldA, ldA,
      A+(k+1)+(k+1)*ldA, ldA);
}
```

LU decomposition with BLAS

Goal: Find LU decomposition. Block notation

$$\begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix} = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} u_{11} & U_{1*} \\ & U_{**} \end{pmatrix}$$



leads to $\ell_{11} = 1$, $u_{11} = a_{11}$, $U_{1*} = A_{1*}$,
$L_{*1} u_{11} = A_{*1}$, $L_{**} U_{**} = A_{**} - L_{*1} U_{1*}$

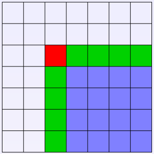
In-place algorithm overwrites A directly with L and U.

```
for(k=0; k<n; k++) {
  scal(n-k-1, 1.0/A[k+k*ldA], A+(k+1)+k*ldA, 1);
  ger(n-k-1, n-k-1, -1.0,
      A+(k+1)+ k    *ldA, 1,
      A+ k    +(k+1)*ldA, ldA,
      A+(k+1)+(k+1)*ldA, ldA);
}
```

LU decomposition with BLAS

Goal: Find LU decomposition. Block notation

$$\begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix} = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} u_{11} & U_{1*} \\ & U_{**} \end{pmatrix}$$



leads to $\ell_{11} = 1$, $u_{11} = a_{11}$, $U_{1*} = A_{1*}$,
$L_{*1} u_{11} = A_{*1}$, $L_{**} U_{**} = A_{**} - L_{*1} U_{1*}$

In-place algorithm overwrites A directly with L and U.

```
for(k=0; k<n; k++) {
  scal(n-k-1, 1.0/A[k+k*ldA], A+(k+1)+k*ldA, 1);
  ger(n-k-1, n-k-1, -1.0,
      A+(k+1)+ k   *ldA, 1,
      A+ k   +(k+1)*ldA, ldA,
      A+(k+1)+(k+1)*ldA, ldA);
}
```

Summary

Finite differences:

- Replace the domain Ω by a grid Ω_h,
- replace functions by grid functions, and
- replace differential operators by finite difference operators.

Potential equation approximated by linear system

$$\begin{aligned} -\Delta_h u_h(x) &= f(x) && \text{for all } x \in \Omega_h, \\ u_h(x) &= 0 && \text{for all } x \in \partial\Omega_h. \end{aligned}$$

Direct solvers like the LU factorization can be used to solve this system.

BLAS offers highly optimized routines that can speed up the implementation significantly.