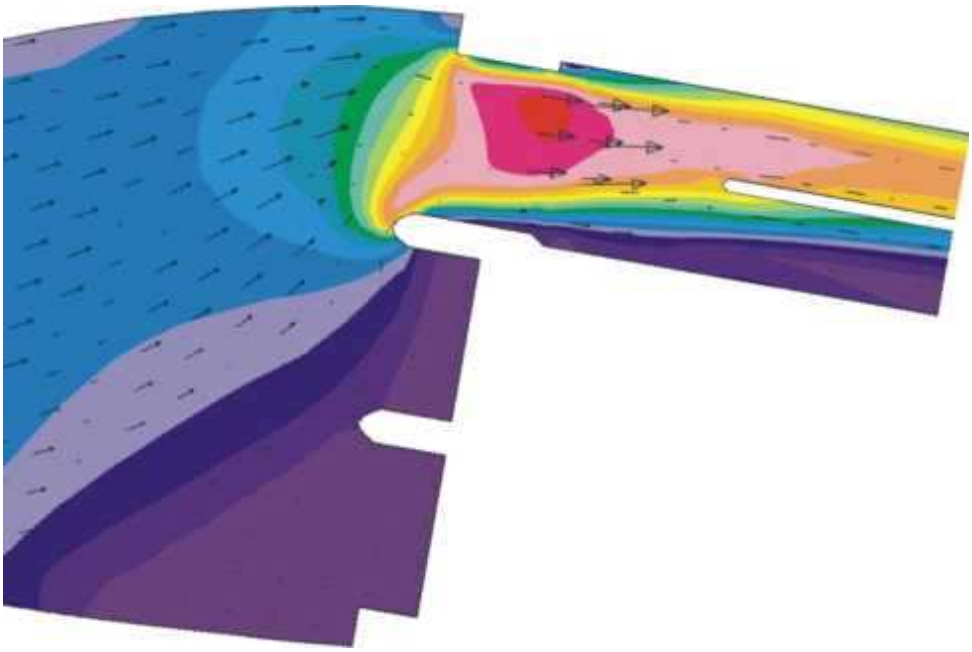
Simulation und wissenschaftliches Rechnen (SiwiR I) 2013/2014
Christoph Pflaum

Contents

1 Performance Optimization

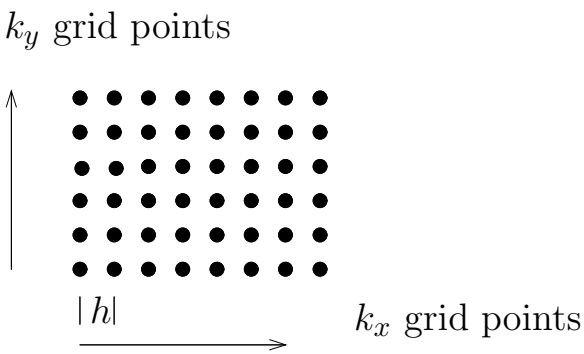
1.1 Motivation

Assume that we want to compute the flow of water in a hydroelectric power plant or the flow of air around a car.



- It is impossible to compute the flow exactly.
- We have to compute an approximate solution on a discretization grid.

Example of a 2D discretization grid:



In 3D, $O(k_x * k_y * k_z)$ data and
$O(k_x * k_y * k_z * k_t)$ floating-point operations are needed.

Example: $k_x = k_y = k_z = 200$ and $k_t = 10000$.
This leads to: $k_x * k_y * k_z = 8 * 10^6$ data and
$k_x * k_y * k_z * k_t = 8 * 10^{10}$ operations.

The computational amount of numerical simulations can be very large. Depending on the problem the computational amount can even be arbitrary large.

Therefore, very fast computers are needed. This leads to the following problems in computer architecture:

- Due to technical reasons the clock rate cannot be arbitrary high.
- In the last years the CPU performance (clock rate, ...) of processors increased more than the performance of memory (bandwidth, ...).

1.2 Basic Concepts of Computer Architecture

1.2.1 Pipeline-Concept

Definition 1 (Latency and bandwidth, access time)**.**

- *The latency L is the time needed until the execution of an instruction can start.*
- *The execution of every instruction needs a certain computational time.*
- *The bandwidth B is the maximum speed of message transfer in* `Mbyte/sec` *(or* `Gbps`*) for an infinitely large message.*

  Thus, the time T for sending a message of size M is:

  $$T = L + M/B.$$

- *The time for reading a certain amount of data from memory is often called* `access time`*.*

Figure 3 depicts a simple pipeline with 5 cycles. Modern processors often contain longer pipelines. Observe, that the latency of a single instruction is 2 cycles.

Example 1.
AMD Opteron: 15 pipeline stages
Intel Nehalem: 16 pipeline stages

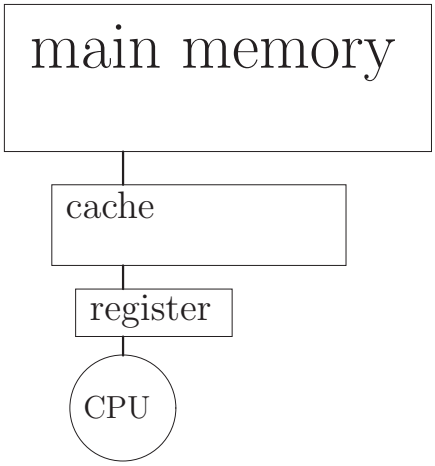


Figure 1: A simple serial computer.

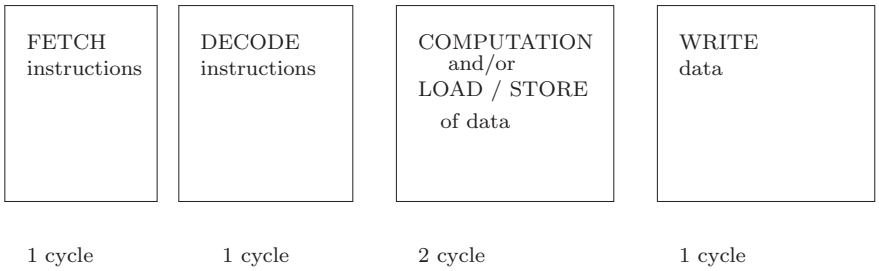


Figure 2: Pipeline concept.

This means FETCH, DECODE, ... can be executed in every cycle, if this leads to correct computations.

Bypassing

Example: Computation of $x * (a + b)$.

By a "bypassing concept", the result of $a + b$ can be used directly after computing it for a multiplication with x.

Fusion of Multiply and Add

Example: Computation of $x * a + b$.

Several processors are able to compute one multiplication and one addi-

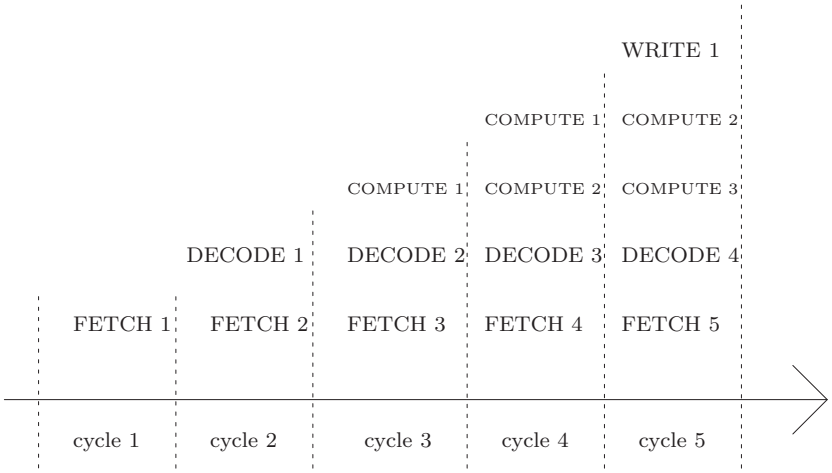


Figure 3: Parallel computations in a pipeline.

tion by one instruction.

Parallel Computations in a Processor

Modern processors are able to perform several instructions in parallel. This can be obtained by

- superscalar processors and
- VLIW processors (very long instruction word)
  e.g. EPIC-concept
  (Explicitly Parallel Instruction Computing)

Superscalar processors are able to perform the parallelization automatically. VLIW processors require certain instructions for performing a parallel computation.

Example 2.

- *superscalar processors: usually:*
  2 floating point operations and 2 integer operations and
  1 read or write of data.
- *Itanium 2: EPIC*

- *Radeon R600: GPU*

Stalls of Pipeline-Processes

If a pipeline cannot accept a new instruction at a certain stage, than this is called "stalled". There exist several reasons for this. One is that certain data are needed which are not contained in registers. Another may be that a previous computation has to end until the new computation can be performed.

$\longrightarrow$ This increases the latency time.

1.2.2 Memory-Concepts

Figure 4 depicts the access time of data of different memories. A high performance can be obtained only if the data are contained in the cache or register.

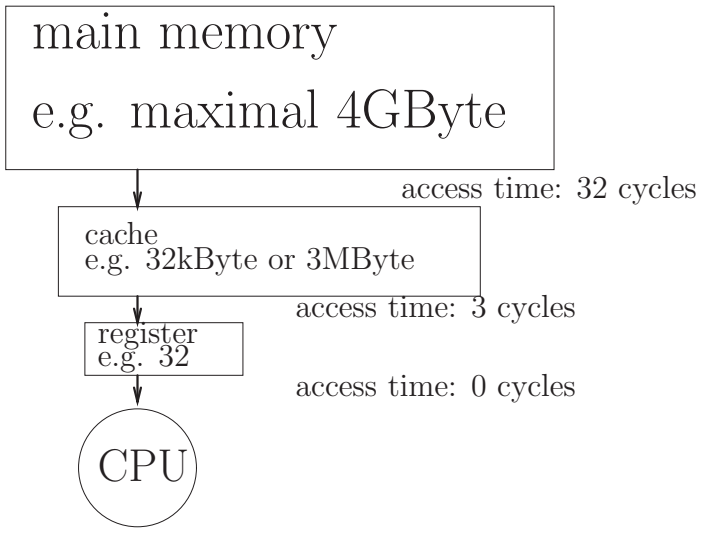


Figure 4: Access time of a processor

The cache consists of several parts of different sizes.

A large cache implies a higher access time (see Figure 5) .

The performance of a computer program is influenced by the access time of data from cache or main memory. Therefore, we have to know how this

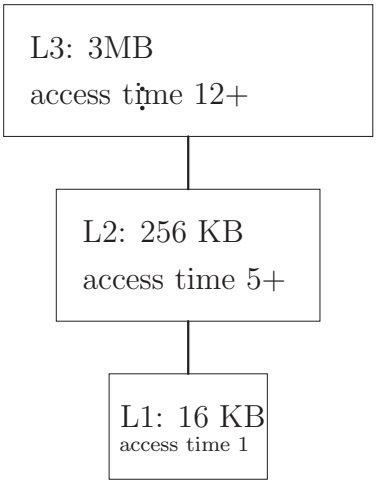


Figure 5: Cache of Itanium 2

access time can be decreased and how data are stored in a memory and copied from one memory to the other.

Important is that data are copied by blocks of a certain fixed size. Let us assume that we want to copy data to a memory with n blocks. Then, there are three concepts to store a new block of data at the block with number s in the memory:

- fully associative: The block can be stored everywhere and can get any number s (or at a free block).
- direct mapped: The number s is k mod n.
- set associative with l sets: The number s can be chosen arbitrary in between $(k \bmod l) * n/l$ and $((k \bmod l) + 1) * n/l - 1$

For each of these cases there is an example in Figure 6. In particular, in case of a "direct mapped cache" the size of a vector has an influence of the performance.

Example: Intel 'Nehalem' Architektur:

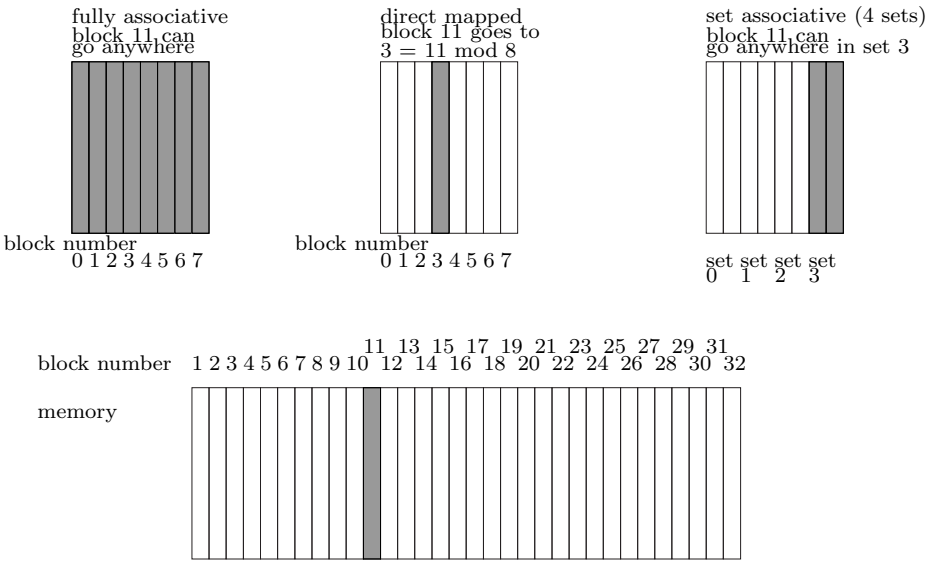


Figure 6: Blocks in a memory.

Nehalem	L1	L2	L3
size	32KB	256 KB	2MB
line size	64	128	128
number of lines	512	2048	16,384
associative sets	64	256	1024
associative	8-way	8-way	16-way

Example: Itanium 2:

Itanium 2	L1	L2	L3
size	16KB	256 KB	3MB
line size	64	128	128
number of lines	256	2048	24,576
associative sets	4	8	12
associative	64-way	256-way	2048-way
update policy	write through	write back	write back

Example: IBM Power 3:

IBM Power 3	L1	L2
size	65KB	4MB
line size	128	128
number of lines		
associative sets		
associative	128-way	direct mapped
update policy	write through	write back

A cash miss occurs, if data from the cache are needed, but they are not in the cache. Therefore these data have to be fetched from a larger memory.

- Compulsory cache misses:
  Every data have to be fetched a first time to the cash. These cash misses cannot be avoided.
- Capacity cache misses: Every cache has a maximal size. Therefore it might happen, that a cash line was overwritten by another cash line.
- Conflict cache misses: If the cache is a directly mapped or set associative cache, then it may happen, that the cache cannot completely be used. Thus, cache lines will be overwritten, however there are free cache lines.

1.3 Methods of Performance Optimization

1.3.1 Basic Optimization done by the Compiler

In this section, we describe basic optimization, which the compiler can do and sometimes cannot do. So this section may help to avoid time-consuming manual optimizations and it can help to change the code such that the compiler is able to do the optimization.

Common Subexpression Elimination

Instead of

```
q = a+b+c;
p = a+b+d;
```

the compiler evaluates

```
t = a+b;
q = t+c;
p = t+d;
```

But the compiler will not replace

```
q = a+b+c;
p = a+d+b;
```

by

```
t = a+b;
q = t+c;
p = t+d;
```

since this is not the same computation in floating arithmetic. Futhermore, the compiler will not simplify

```
q = f(x)+b*f(x);
```

by

```
t = f(x);
q = t+b*t;
```

Loop-Invariant Code Motion

The compiler optimizes

```
for(i=0;i<n;++i)
   a[i] = r*s+b[i];
```

by

```
t = r*s;
for(i=0;i<n;++i)
   a[i] = t+b[i];
```

Evaluation of Constants

The compiler optimizes

```
x = 3*4.0 + y;
```

by

```
x = 12.0 + y;
```

$\longrightarrow$ Optimization by meta-programming in C++! (See section 1.5.2)

Strength Reduction

For an integer i the compiler replaces

```
2*i
```

by

```
i+i
```

In FORTRAN, the compiler replaces

```
x**2
```

by

```
x*x
```

Instruction Scheduling

Instead of

```
a = b+c;
d = 2.0*a+e;
g = 2.0*c;
q = g+b*2.0;
```

the compiler could evaluate

```
a = b+c;
g = 2.0*c;
d = 2.0*a+e;
q = g+b*2.0;
```

and try to optimize the use of the registers. This is a very complex optimization problem.

1.3.2 Performance Optimization: Mult-Add

Several processors perform `a+b*c` as fast as one multiplication. Thus,

```
a = b+c*d+f*g;
```

often is faster than

```
a = f*g+c*d+b;
```

1.3.3 Eliminating Overheads

There exist a lot of ways to avoid overheads.

A simple example is the following. Replace

```
if(sqrt(tt) < eps) { ... }
```

by

```
if(tt < eps*eps) { ... }
```

1.3.4 Loop Unrolling

> Loop unrolling is the general concept to improve performance!

Instead

```
for(int i=0;i<n*m;++i) Comp(i);
```

perform

```
for(int i=0;i<m*n;i=i+n)
   for(int j=0;j<n;j=j+1) Comp(i+j);
```

or

```
for(int i=0;i<m*n;i=i+n)    {
   Comp(i+0);
   Comp(i+1);
      ...
   Comp(i+n-1);             }
```

Perform additional changes of the computations in the interior loop!

Loop unrolling can optimize the performance of a code by

- software pipelining
- instruction parallelization
- improvement of the memory access.

Instruction-parallelization

Consider the algorithm to compute the l_2 norm of a vector. Then,

```
norm =0.0;
for(int i=0;i<n;++i) {
  norm = norm + a[i]*a[i];
}
norm = sqrt(norm);
```

is slower than

```
t1 = t2 = 0.0;
for(int i=0;i<n;i=i+2) {
  t1  =  t1 + a[i+0]*a[i+0];
  t2  =  t2 + a[i+1]*a[i+1];
}
norm = t1 + t2;
norm = sqrt(norm);
```

Figure 7 depicts the performance for $k = 1, 2, 3, 4$ parallel instructions.

Pentium 4 - Vectorization

The Pentium 4 architecture allows two floating point instructions per cycle by SSE2 floating point instructions. Using the option `-xW` for the Intel-Compiler this leads to a so called "vectorization". For example the compiler shows the output:

```
cpc -O3 -xW  -c main.cc
main.cc(32) : (col. 3) remark: LOOP WAS VECTORIZED.
main.cc(84) : (col. 5) remark: LOOP WAS VECTORIZED.
icpc -O3 -xW -o run main.o  -lm
```

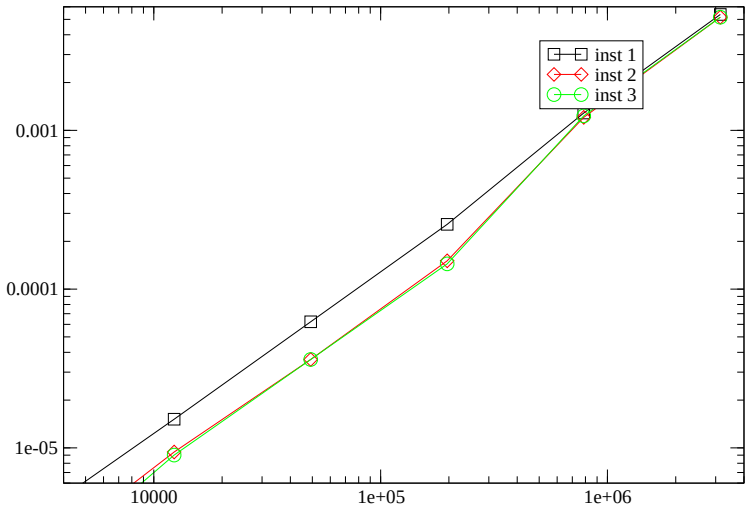


Figure 7: Computational time of a scalar product with k parallel instructions.
(

Figure 8 shows the computational time of

```
for(i=0;i<n;++i)
   c[i] = a[i]*a[i] + b[i]*b[i]*b[i];
```

with respect to n and Figure 9 shows the computational time of

```
for(i=0;i<n;++i)
   c[i] = a[i] +  cos(b[i]);
```

with respect to n for a Pentium 4 . The speed up by a factor of roughly 2 is caused by the SSE2 floating point instructions, which lead to a vectorization of the code.
Improvement of memory access
Consider the matrix vector multiplication. Then,

```
for(int i=0;i<n;++i) {
   t =0.0;
```

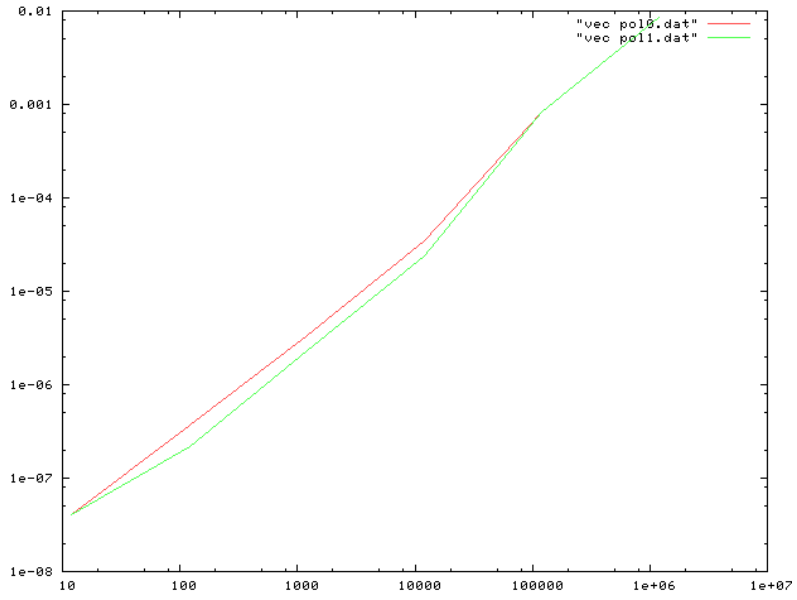


Figure 8: Improvement of performance for a simple vector polynomial.

```
   for(int j=0;j<n;++j) {
      t = t + a(i,j)*x[j];
   }
   y[i]=t;
}
```

is slower than

```
for(int i=0;i<n;i=i+2) {
   t0 =0.0;
   t1 =0.0;
   for(int j=0;j<n;++j) {
      t0 = t0 + a(i  ,j)*x[j];
      t1 = t1 + a(i+1,j)*x[j];
   }
   y[i]   = t0;
   y[i+1] = t1;
}
```

Figure 10 depicts the performance with loop unrolling for $k = 1, 2, 3$.
Such an improvement of the memory access can also be applied to improve

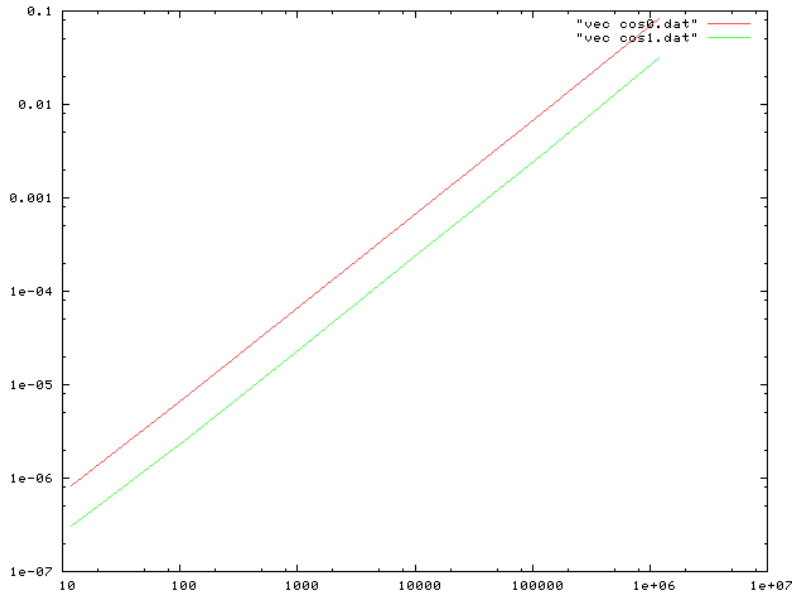


Figure 9: Improvement of performance for a vector expression with cos.

several Gauss-Seidel relaxations. This is called blocking.

Automatic loop unrolling

Loop unrolling is often automatically performed by the compiler. But in some cases it is impossible for the compiler to unroll a loop. An example is:

```
sum = 0.0;
for(int i=0;i<n;i=i+1)
   for(int j=0;j<n;j=j+1) {
      sum = sum + x[i][j] * (i*i + j*j);
}
```

By hand it is possible to do a loop unrolling with respect to i and j.

```
sum0 =0.0;
sum1 =0.0;
sum2 =0.0;
sum3 =0.0;
for(int i=0;i<n;i=i+2) {
```

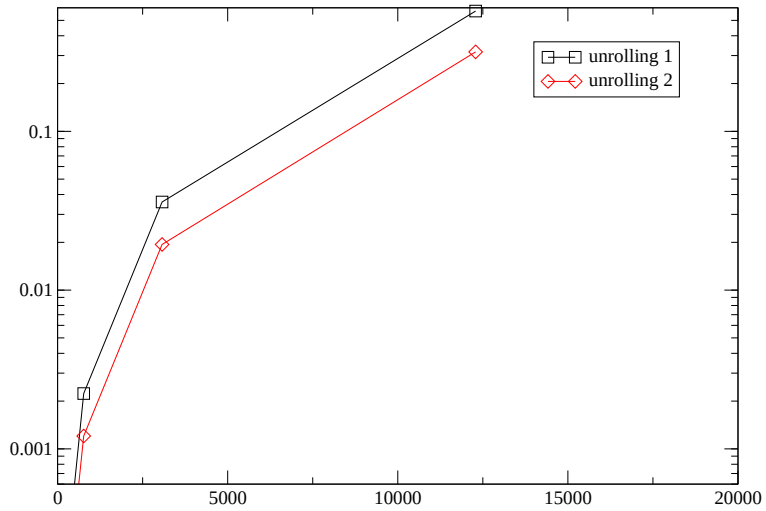


Figure 10: Computational Time of Matrix Vector Multiplication.

```
   for(int j=0;j<n;j=j+2) {
      sum0 = sum0 + x[i][j]     * (i*i          +  j*j);
      sum1 = sum1 + x[i+1][j]   * ((i+1)*(i+1) +  j*j);
      sum2 = sum2 + x[i][j+1]   * (i*i          + (j+1)*(j+1));
      sum3 = sum3 + x[i+1][j+1] * ((i+1)*(i+1) + (j+1)*(j+1));
   }
}
sum = sum0+sum1+sum2+sum3;
```

There is a limit for the size of loop unrolling. This limit is caused by

- a limited number of registers and
- overhead caused by too small loops and a loop length, which is not a multiple of the size of the interior loop.

1.3.5 Optimization of Memory Access

$\longrightarrow$ Optimization of memory access is very important in HPC!

The general rule is:

$\longrightarrow$ Optimize data locality!

This means compute with data from cache!

Let us consider the following two examples in FORTRAN:

```
      dimension a(n,n),b(n,n)
      LOOP A
      do 10,i=1,n
      do 10,j=1,n
10       a(i,j)=b(i,j)*b(i,j)+1.
```

and

```
      dimension a(n,n),b(n,n)
      LOOP B
      do 10,j=1,n
      do 10,i=1,n
10       a(i,j)=b(i,j)*b(i,j)+1.
```

Which version is faster?

The ordering of data in the matrix `a` for $n = 4$ is

```
1  5  9   13
2  6  10  14
3  7  11  15
4  8  12  16
```

Therefore, the loop B is faster.

Let us consider the following two examples in C:

```
      double a[n][n], b[n][n];
      // LOOP A
      for(i=0;i<n;++i)
         for(j=0;j<n;++j)
            a[i][j]=b[i][j]*b[i][j]+1.0;
```

and

```
double a[n][n], b[n][n];
// LOOP B
for(j=0;j<n;++j)
   for(i=0;i<n;++i)
      a[i][j]=b[i][j]*b[i][j]+1.0;
```

Now, loop A is faster!

Let us consider the following example in C++:

```
double **a, **b;
a = new double* [n];
b = new double* [n];
for(i=0;i<n;++i) {
  a[i] = new double[n];
  b[i] = new double[n];
}

// LOOP C
for(i=0;i<n;++i)
   for(j=0;j<n;++j)
      a[i][j]=b[i][j]*b[i][j]+1.0;
```

Now, the data of `a` are cut in several pieces. This leads to less data locality and optimizations as vectorization cannot be performed in an optimal way.

The following code leads to a better data allocation:

```
double *a, *b;
a = new double[n*n];
b = new double[n*n];

// LOOP C
for(i=0;i<n;++i)
   for(j=0;j<n;++j)
      a[i*n+j]=b[i*n+j]*b[i*n+j]+1.0;
```

Using such a data structure, an optimal performance can be obtained on vector machines.

Loop Fusion Consider the follwing code.

Instead of

```
for(i=0;i<n;++i)
   u[i] = u[i] + tau   * g[i];
for(i=0;i<n;++i)
   r[i] = b[i] + alpha * g[i];
```

implement

```
for(i=0;i<n;++i) {
   u[i] = u[i] + tau   * g[i];
   r[i] = b[i] + alpha * g[i];
}
```

Data Layout

Construct a data layout such that the computations can be done locally.

As an example consider the coordinates of particles. In FORTRAN write

```
dimension r(3,n)
```

instead of

```
dimension rx(n), ry(n), rz(n)
```

Blocking

Blocking is similar to loop unrolling. Consider the matrix transposition

```
      dimension a(n,n),b(n,n)
      LOOP A
      do 10,i=1,n
      do 10,j=1,n
10       b(i,j)=a(j,i)
```

Subdivide the index set in small blocks of size $s * s$:

$$\begin{array}{ccc} (1,1) & \ldots & (1,n) \\ \vdots & \ldots & \vdots \\ (n,1) & \ldots & (n,n) \end{array} \qquad \begin{array}{ccc} (k_1,k_2) & \ldots & (k_1,k_2+s) \\ \vdots & \ldots & \vdots \\ (k_1+s,k_2) & \ldots & (k_1+s,k_2+s) \end{array}$$

Then, perform the matrix transposition on each of these blocks.

The size of the cache must be larger than $2 * s * s$.

1.3.6 Automatic Optimization

Compilers try to perform an automatic optimization. In particular, FORTRAN compilers are able to optimize a code by loop unrolling and automatic instruction parallelization.

Using C or C++, there is a problem with *aliasing*.

Let us consider the program

```
void f(double *a,double *b,double *c,double *d){
   for(int i=0;i<n;++i)
      a[i] = b[i] + c[i] * d[i];
   } }
```

Then, the C compiler does not know whether `b[i]` and `a[i-1]` point to the same value or not. Therefore, some compilers cannot perform an automatic optimization in this case.

To avoid the problem with *aliasing* some compilers support the keyword `restrict` or `__restrict` for pointers as follows:

```
   double * restrict a;
   double * restrict b;
   double * restrict c;
   double * restrict d;

   for(int i=0;i<n;++i)
      a[i] = b[i] + c[i] * d[i];
   }
```

1.4 Shared Memory Parallelization

1.4.1 Shared Memory Computer Architecture

A parallel computer architecture with a shared memory consists of several processors and one common memory (see Figure 11). Data are exchanged via a crossbar. This limits the number of processors

1.4.2 Parallelization with OpenMP

The parallelization with OpenMP is based on

- threads

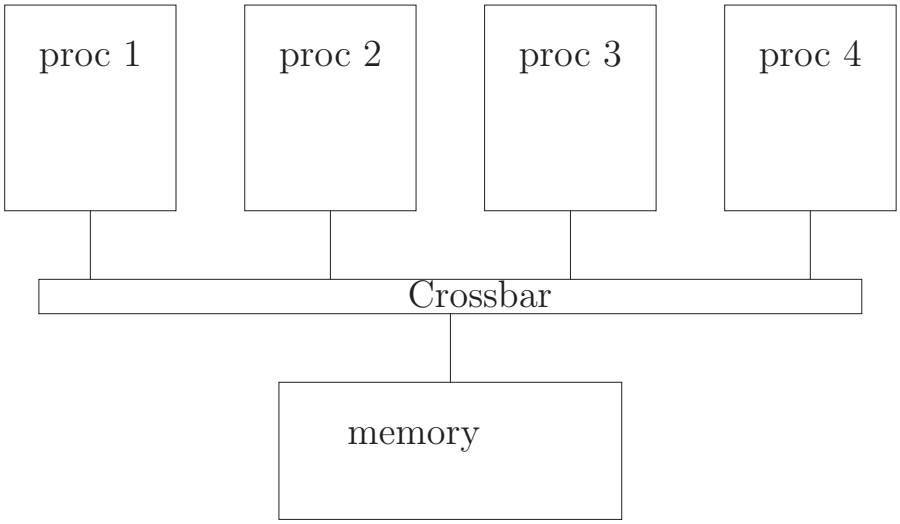


Figure 11: Shared Memory Computer Architecture.

- the usage of pragmas like `# pragma omp parallel for`

A simple parallelization of `for` loops in OpenMP can be obtained as follows:

```
#include <omp.h>
...
int main() {
  ...
  double * __restrict a;
  double * __restrict b;
  double * __restrict c;
  ...
#pragma omp parallel for
  for(int i=0;i<n;++i) {
     c[i] = a[i]*a[i] + b[i]*b[i]*b[i];
  }
```

Figure 12 depicts the computational time of this simple `for`-loop with and without `__restrict` .

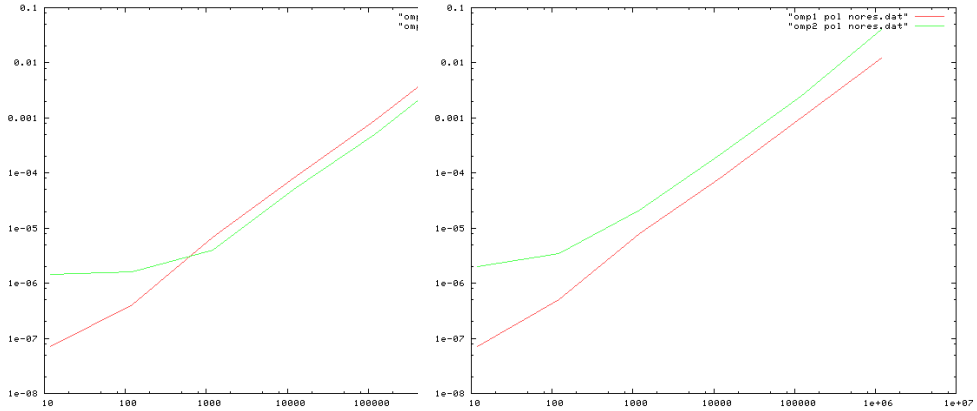


Figure 12: Parallelization with and without `__restrict`

For more complicated constructions the simple pragma
`# pragma omp parallel for`
is not sufficient to obtain an efficient parallelization. One reason for poor performance of an OpenMP parallelization might be that the threads often need the same data from main memory.

One way to avoid this is tell the compiler, that a variable is only used *private* by every thread. This can be done by the private construction as follows:

```
  ...
  double * __restrict a;
  double * __restrict c;
  double sum;
  int i,j;
  ...
\\ good version
#pragma omp parallel for  private(j,sum)
  for(i=0;i<n;++i) {
    sum = 0.0;
    for(j=0;j<n;++j) {
      sum = sum + a[i*n+j];
    }
    c[i] = sum;
  }
```

```
}
```

Computational time for OpenMP parallelization with 2 threads:

n	12	120	1200	12000
sec	3.9e-7	4.6e-5	4.7e-3	4.9e-1
sec parallel (good version)	1.5e-6	2.4e-5	2.3e-3	2.4e-1

The shared memory concept of OpenMP leads to poor performance, if very often the same same piece data is needed by both threads as shown in the following code:

```
  ...
  double * __restrict a;
  double * __restrict b;
  double * __restrict c;
  double sum;
  int i,j;
  ...
// bad version
#pragma omp parallel for  private(j,sum)
  for(i=0;i<n;++i) {
    sum = 0.0;
    for(j=0;j<n;++j) {
      sum = sum + a[j*n+i];
    }
    c[i] = sum;
  }
}
```

This parallelization increases the computational time:

n	12	120	1200	12000
sec	3.9e-7	4.6e-5	4.7e-3	4.9e-1
sec parallel (bad version)	1.5e-6	1.1e-4	1.9e-2	3.0

`reduction` Construction in OpenMP

Let us assume we want to calculate the euclidian norm of a vector

$$\|v\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Then, the following code leads to the wrong result:

```
  ...
  double norm;
  norm = 0.0;
#pragma omp parallel for
  for(i=0;i<n;++i) {
    norm = norm + a[i]*a[i];
  }
  norm = sqrt(norm);
}
```

A correct code can be obtained by the `reduction` construction in OpenMP as follows:

```
  ...
  double norm;
  norm = 0.0;
#pragma omp parallel for reduction(+ : norm)
  for(i=0;i<n;++i) {
    norm = norm + a[i]*a[i];
  }
  norm = sqrt(norm);
}
```

`reduction` can be applied to the operators

`+,*,-,&,|,&&,^ ,||` .

Here, `||` reduces a maximum calculation of a variable.

Not Parallelizable Loops

Consider the loop

```
  ...
  for(i=1;i<n;++i)
    a[i] = a[i-1]+b[i];
  ...
}
```

OpenMP will not parallelize this loop in a correct way.

Not Parallelizable Relaxation Loop
OpenMP cannot parallelize the following loop in a correct way:

```
  ...
  for(i=1;i<n-1;++i)
    a[i] = 0.5*(a[i-1]+a[i+1]);
  ...
}
```

Parallelizable Relaxation Loop
The following loop can be parallelized in a correct way by OpenMP:

```
  ...
#pragma omp parallel for
  for(i=1;i<n-1;i=i+2)
    a[i] = 0.5*(a[i-1]+a[i+1]);

#pragma omp parallel for
  for(i=2;i<n-1;i=i+2)
    a[i] = 0.5*(a[i-1]+a[i+1]);
  ...
```

1.5 Optimization in C and C++

1.5.1 Inlining and Const

The call of a function requires computational times. To avoid this problem a function can be defined to be inlined.

Example:

```
inlining double f(double x) { ... };
```

Advantage:

- optimization of the code in the area where the function is called (such as common subexpression elimination and vectorization)
- no overhead by calling the function

Disadvantage:

- longer compilation time
- longer executable code

Parameters of functions which will not be changed should be defined to be `const`.

Example:

```
inlining double f(const double x) { ... };
```

Member functions of a class which do not modify member values of the class should be defined to be `const` member functions:

Example:

```
class A {
 ...
 inlining double f(const double x) const { ... };
};
```

`const` can help a compiler to optimize a code.

1.5.2 Meta-Programming in C++

The compiler optimizes

```
x = 3*4.0 + y;
```

by

```
x = 12.0 + y;
```

Can we obtain such an optimization for

```
x = Factorial(4) + y;
```

where `Factorial(4)` mathematically means

$$4! = 1 * 2 * 3 * 4 = 24$$

Consider the C++ construction

```
template<int N>
class Factorial {
public:
    enum { value = N * Factorial<N-1>::value };
};
class Factorial<1> {
public:
    enum { value = 1 };
};
```

Then, the compiler replaces

```
x = Factorial<4>::value + y;
```

by

```
x = 24 + y;
```

Meta-Programming means to write a program, which is evaluated during compile-time and not during runtime.

2 Finite Difference Discretization

2.1 Model Problem: Poisson's Equation

Figure 13 shows the construction of a wall. The insulation property of the wall depends on the wall construction. The following mathematical model can be used to calculate the insulation property of a wall:

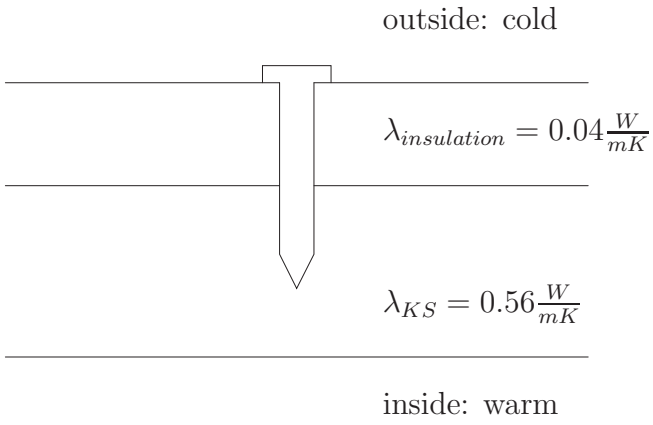


Figure 13: Construction of a wall

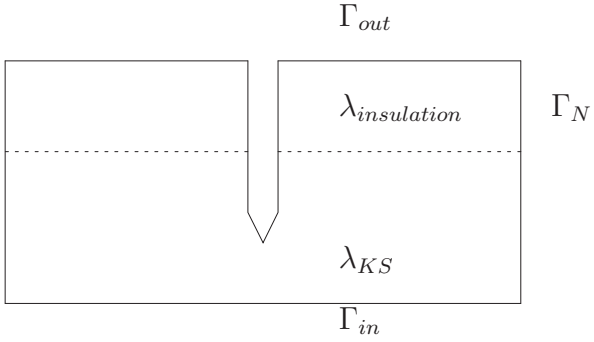


Figure 14: Model of a wall

$$
\begin{array}{rcll}
-\text{div } \lambda \text{ grad} T & = & 0 & \text{on } \Omega \\
T|_{\Gamma_{out}} & = & -10 & \text{on } \Gamma_{out} \\
T|_{\Gamma_{in}} & = & 20 & \text{on } \Gamma_{in} \\
\frac{\partial T}{\partial \vec{n}}|_{\Gamma_N} & = & 0 & \text{on } \Gamma_N
\end{array}
$$

See Figure 14 for the definition of the domain Ω and the boundaries $\Gamma_{out}, \Gamma_{in}, \Gamma_N$. An optimal discretization of such a problem is the finite element discretization. For reason of simplicity, let us apply the finite difference discretization. To this end, let us additionally consider the following simplified partial differential equation:

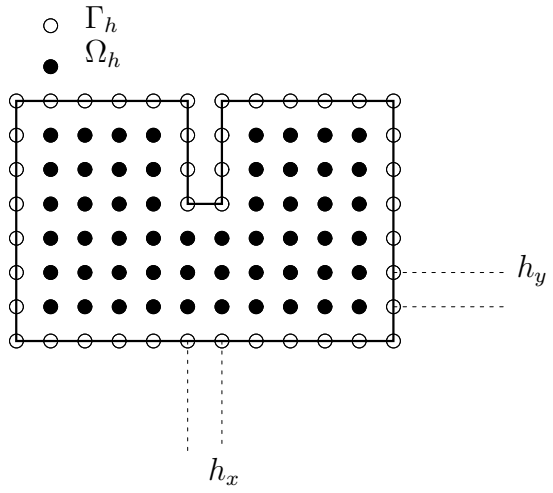


Figure 15: Finite difference discretization grid

$$\begin{aligned} -\triangle T + cT &= f && \text{on } \Omega \\ T|_\Gamma &= g && \text{on } \Gamma \\ \Omega &= (0, L)^2, \end{aligned}$$

where $c > 0$ is a constant and $L > 0$ is the size of the domain. Observe that

$$-\text{div grad}\, T = -\triangle T = -\frac{\partial^2 T}{\partial x^2} - \frac{\partial^2 T}{\partial y^2}.$$

2.2 Finite Differences

The first step in a finite difference discretization is the construction of a discretization grid (see Figure 15). In case of $\Omega = (0, L)^2$, we obtain:

$$\begin{aligned} \Omega_h &= \{(ih, jh) | i, j = 1, \ldots, m-1\} \\ \bar{\Omega}_h &= \{(ih, jh) | i, j = 0, \ldots, m\} \\ \Gamma_h &:= \bar{\Omega}_h \backslash \Omega_h. \end{aligned}$$

where $h = \frac{L}{m}$.

The second step is to replace derivatives by finite differences:
Let $e_x = (1, 0)$ and $e_y = (0, 1)$, then

$$\begin{aligned}
\frac{\partial u}{\partial x}(x,y) &\approx \frac{u(z+he_x)-u(z)}{h} \\
\frac{\partial u}{\partial x}(x,y) &\approx \frac{u(z)-u(z-he_x)}{h} \\
\frac{\partial u}{\partial x}(x,y) &\approx \frac{u(z+he_x)-u(z-he_x)}{2h}
\end{aligned}$$

and

$$\begin{aligned}
\frac{\partial u}{\partial y}(x,y) &\approx \frac{u(z+he_y)-u(z)}{h} \\
&\vdots
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 u}{\partial x^2}(x,y) &\approx \frac{u(z+he_x)-2u(z)+u(z-he_x)}{h^2} \\
\frac{\partial^2 u}{\partial y^2}(x,y) &\approx \frac{u(z+he_y)-2u(z)+u(z-he_y)}{h^2}.
\end{aligned}$$

Thus, we get

$$-\Delta u(z) = \left(-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2}\right)(z) \approx$$

$$\approx -\frac{u(z+he_x)+u(z+he_y)-4u(z)+u(z-he_x)+u(z-he_y)}{h^2}$$

Using these finite difference formulas, we get the following discretization:

$$-\Delta u(z) = \left(-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2}\right)(z) = f(z) \quad \text{for } z \in \Omega_h$$

$$\begin{aligned}
&-\frac{u_h(z+he_x)-2u_h(z)+u_h(z-he_x)}{h^2} \\
&-\frac{u_h(z+he_y)-2u_h(z)+u_h(z-he_y)}{h^2} &= f(z)
\end{aligned}$$

$$\begin{array}{rrcll} \text{and} & u(z) & = & g(z) & \\ & \approx & = & & \text{for } z \in \Gamma_h = \overline{\Omega_h} \backslash \Omega_h \\ & u_h(z) & = & g(z) & \end{array}$$

Here u_h is the approximate solution on the discretization grid Ω_h.

The finite difference discretization leads to a linear equation system

$$L_h \, U_h \;=\; F_h, \tag{1}$$

where $U_h = (u_h(z))_{z \in \Omega_h}$, and L_h is a $|\Omega_h| \times |\Omega_h|$ matrix.

The discretization can be described by the stencil

$$\begin{pmatrix} & -\frac{1}{h^2} & \\ -\frac{1}{h^2} & \frac{4}{h^2} & -\frac{1}{h^2} \\ & -\frac{1}{h^2} & \end{pmatrix} = \begin{pmatrix} m_{-1,1} & m_{0,1} & m_{1,1} \\ m_{-1,0} & m_{0,0} & m_{1,0} \\ m_{-1,-1} & m_{0,-1} & m_{1,-1} \end{pmatrix}.$$

Let us abbreviate $U_{i,j} := u_h(ih, jh)$ and $f_{i,j} := f(ih, jh)$. Then, the matrix equation (1) is equivalent to:

$$\sum_{k,l=-1}^{1} m_{kl} U_{i+k,j+l} = f_{i,j} \quad \text{for } i, j = 1, \ldots, m-1.$$

This implies that the entries of the matrix $L_h = (l_{(i,j),(o,p)})_{1 \le i,j,o,p \le m-1}$ are

$$l_{(i,j),(o,p)} = \begin{cases} m_{k,l} & \text{if } o = i + k \text{ and } p = j + l \text{ and } |k| \le 1, |l| \le 1 \\ 0 & \text{else.} \end{cases}$$

To describe the right hand side F_h in (1), define

$$\tilde{g}(x, y) = \begin{cases} g(x, y) & \text{if } (x, y) \in \Gamma = \partial\Omega \\ 0 & \text{else.} \end{cases}$$

Then, we get for $F_h = (F_{i,j})_{1 \le i,j \le m-1}$

$$F_{i,j} = f_{i,j} - \sum_{k,l=-1}^{1} m_{kl} \; \tilde{g}((i+k)h, (j+l)h)$$

This implies, that in case of $g = 0$ we obtain

$$F_{i,j} = f_{i,j} = f(ih, jh).$$

Let $\Omega = (0,1)^2$. Number the points of the discretization grid Ω_h by:

$$h(1,1), ..., h(1,m-1), h(2,1), ...$$

Then, the FD discretization leads to an equation $L_h\, U_h \;=\; F_h$, where

$$L_h \;=\; \frac{1}{h^2} \begin{pmatrix} D_h & -E & & \\ -E & D_h & \ddots & \\ & \ddots & \ddots & -E \\ & & -E & D_h \end{pmatrix}, \quad \text{and where}$$

$$D_h \;=\; \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}.$$

2.3 Convergence of the Finite Difference Discretization

Before we describe suitable norms for FD let us make a remark to norms of the finite element (FE) method.

The finite element method leads to approximations $u_h \in \mathcal{C}(\Omega)$ of an exact solution $u \in \mathcal{C}(\Omega)$ of a PDE.

Suitable norms for calculating the discretization error are

$$\|u - u_h\|_{L_\infty} \;:=\; \max_{x \in \Omega} |(u - u_h)(x)|$$

$$\|u - u_h\|_{L_2} \;:=\; \sqrt{\int_\Omega |(u - u_h)(x)|^2 \; dx}$$

- These norms have the *normalization* property

$$(u - u_h)(x) = 1 \;\; \forall x, h \quad \Rightarrow \quad \|u - u_h\| \;=\; \text{const} \;\; \forall h$$

- In case of solutions with singularities onecan expect a better convergence for the $\|.\|_{L_2}$ norm.

Let Ω_h be a sequence of discretization grids.

We are looking for a sequence of norms on $\mathbb{R}^{|\Omega_h|}$ with similar properties for the FD method.

Example: A not suitable norm is

$$\|w\| := \sqrt{\sum_{z\in\Omega_{h_i}} |w(z)|^2}.$$

Definition 2. *We call the sequence of norms $\|\cdot\|_{h_i}$ on $\mathbb{R}^{|\Omega_i|}$ normalized, if*

$$\|\underline{1}\|_{h_i} = 1,$$

where $\underline{1}$ is the constant function $x \mapsto 1$.

Example 3.

$$\begin{aligned} \|x\|_2 &:= \sqrt{\frac{1}{|\Omega_i|}\sum_{z\in\Omega_i} x_z^2} \\ \|x\|_\infty &:= \max_{z\in\Omega_i} |x_z| \end{aligned}$$

Theorem 1. *Consider the finite difference discretization of Poisson's equation on $\Omega = (0, L)^2$ with meshsize h. Then, there is a constant $C > 0$ such that*

$$\|u - u_h\|_\infty \le Ch^2 \left(\left\| \frac{\partial^4 u}{\partial x^4} \right\|_\infty + \left\| \frac{\partial^4 u}{\partial y^4} \right\|_\infty \right).$$

Example:

- If $u = x^2 * y^3$, then $u = u_h$,
- If $u = x^4$, then $\|u - u_h\|_\infty \le Ch^2$.

Example: Poisson's equation on $(0, 1)^2$.
Let $f(x, y) = -12.0 * x^2 - \exp(y)$. Then, the exact solution of

$$\begin{aligned} -\triangle u &= f = -12.0 * x^2 - \exp(y) && \text{on } \Omega \\ u|_{\partial\Omega} &= x^4 + \exp(y) && \text{on } \partial\Omega \end{aligned}$$

is

$$u(x,y) = x^4 + \exp(y).$$

The following table depicts the error $e_{h,\max} := \|u - u_h\|_\infty$:

$h =$	0.5	0.25	0.125	0.0625	0.03125
$N =$	1	9	49	225	961
$e_{h,\max} \approx$	0.033	0.0094	0.0024	0.00061	0.00015
$e_{h/2,\max}/e_{h,\max} \approx$		0.28	0.26	0.25	0.25

How to Choose the Meshsize h:
Assume that the discretization error converges according

$$\|u - u_h\| \le Ch^p.$$

How should we choose h to obtain a discretization error $\|u - u_h\| \le \eta$?

Assume that we can calculate $\|u_{h/2} - u_h\|$.
Then, the assumption $\|u - u_h\| \approx Ch^p$ leads to

$$\|u - u_h\| \approx \frac{1}{1 - 2^{-p}} \|u_{h/2} - u_h\|. \tag{2}$$

Thus, we have to choose h such that

$$\|u_{h/2} - u_h\| \le \eta(1 - 2^{-p}).$$

Let us show (2). Let us assume the asymptotic behavior

$$\|u - u_h\| \approx Ch^p.$$

Then, we get

$$\|u - u_{h/2}\| \approx C \left(\frac{h}{2}\right)^p.$$

and

$$\|u - u_{h/2}\| \approx \|u - u_h\| \frac{1}{2^p}.$$

This implies

$$\begin{aligned} \|u - u_h\| &\le \|u - u_{h/2}\| + \|u_{h/2} - u_h\| \\ &\le \frac{1}{2^p}\|u - u_h\| + \|u_{h/2} - u_h\|. \end{aligned}$$

$$\left(1 - \frac{1}{2^p}\right) \|u - u_h\| \le \|u_{h/2} - u_h\|$$

This shows (2).

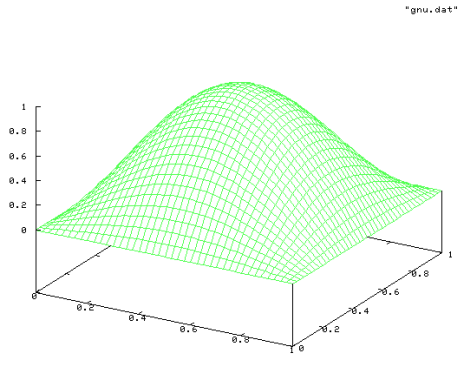


Figure 16: Eigenvector $e_{1,1}$

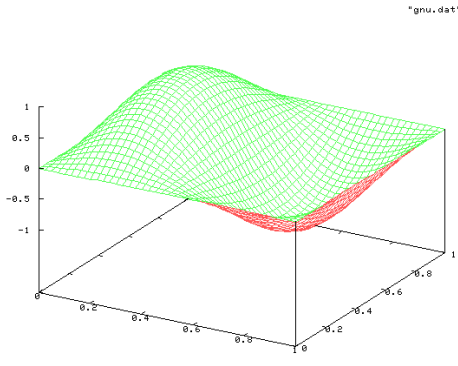


Figure 17: Eigenvector $e_{2,1}$

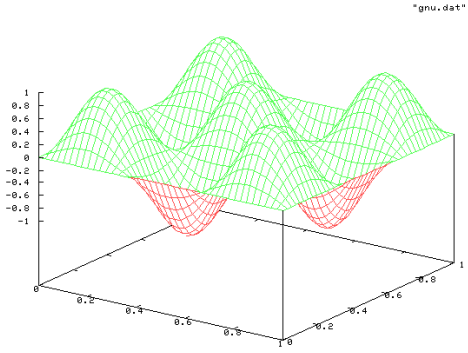


Figure 18: Eigenvector $e_{3,3}$

2.4 Eigenvectors and Eigenvalues of L_h

Consider the FD discretization of Poisson's equation on the unit square.

Then, the matrix L_h has the eigenvalues

$$\lambda_{\nu,\mu} = \frac{4}{h^2}\left(\sin^2\left(\frac{\pi\nu h}{2}\right) + \sin^2\left(\frac{\pi\mu h}{2}\right)\right)$$

with eigenvectors

$$e_{\nu,\mu} = \Big(\sin(\nu\pi x_i)\sin(\mu\pi y_j)\Big)_{(x_i,y_j)\in\Omega_h} \quad \text{where} \quad \nu,\mu = 1,\cdots,m-1$$

and $h = \frac{1}{m}$.

- Smallest eigenvalue: $\frac{4}{h^2} 2 \sin^2\left(\frac{\pi h}{2}\right) \approx 2\pi^2$.
- Largest eigenvalue: $\frac{4}{h^2} 2 \sin^2\left(\frac{\pi(m-1)}{2m}\right) \approx \frac{8}{h^2}$.

3 Basic Solvers

3.1 Introduction

The FD discretization leads to an equation system

$$A_h u_h = b_h,$$

where A_h is an $n \times n$ matrix and $u_h, b_h \in \mathbb{R}^n$ are vectors.

There are

- direct methods and
- iterative methods

for solving such an equation system.

$\downarrow$

- Both methods lead to an approximation $\tilde{u}_h$ of u_h.
- $\|u_h - \tilde{u}_h\|$ is called algebraic error.
- $\|u - u_h\|$ is called discretization error.
- $\|u - \tilde{u}_h\|$ is called total error.

The algebraic error should satisfy the property

$$\|u_h - \tilde{u}_h\| \leq \|u - u_h\|\alpha, \quad \text{where } \alpha \approx 0.1.$$

Let the discretization satisfy $\|u - u_h\| \leq Ch^2$.
Then, this implies

$$\|u - \tilde{u}_h\| \leq C(1 + \alpha)h^2.$$

It is very difficult to calculate numerically the

- algebraic error $\|u_h - \tilde{u}_h\|$ and the

- discretization error $\|u - u_h\|$.

Therefore, one often calculates

- the residuum norm $\|A_h \tilde{u}_h - b_h\|$ or
- the norm $\|\tilde{u}_h - \tilde{u}_{h/2}\|$.

If the exact solution is known, then one can numerically calculate the total error

- total error $\|u - \tilde{u}_h\|$.

Let us recall the basic properties of the Gauss-elimination:

- The Gauss-elimination applied to a full matrix requires
  - $O(n^3)$ operations
  - $O(n^2)$ data

  for solving a linear equation system with n unknowns.
- The Gauss-elimination applied to a band matrix of bandwidth $2k-1$ requires
  - $O(n * k * k)$ operations
  - $O(n * k)$ data.

A band matrix of bandwidth k has the form:

$$\begin{pmatrix} a_{11} & \dots & a_{1k} & & \\ \vdots & a_{22} & \ddots & \ddots & \\ a_{k1} & \ddots & \ddots & \ddots & a_{n-k+1,n} \\ & \ddots & \ddots & a_{n-1,n-1} & \vdots \\ & & a_{n,n-k+1} & \dots & a_{nn} \end{pmatrix}$$

Now consider the matrix of the FD discretization of Poisson's equation on $\Omega = (0,1)^2$. The discretization matrix is a band matrix of size $n = (m-1)^2$ and bandwidth $2m-1$, since $h = \frac{1}{m}$

$$L_h = \frac{1}{h^2} \begin{pmatrix} D_h & -E & & \\ -E & D_h & \ddots & \\ & \ddots & \ddots & -E \\ & & -E & D_h \end{pmatrix}, \quad \text{where } D_h = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}$$

- Then, the Gauss-elimination applied to the band matrix L_h requires
  - $O(n^2)$ operations
  - $O(n^{1.5})$ data.

An iterative solver for solving a linear equation system is a mapping

$$\mathcal{S} : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

with start vector $x_0 \in \mathbb{R}$ such that the sequence $(x_i)_{i \in \mathbb{N}}$ defined by

$$x_{i+1} = \mathcal{S}(x_i)$$

converges to x:

$$\lim_{i \to \infty} x_i = x.$$

Obviously, x should satisfy the fix point property $x = \mathcal{S}(x)$.

3.2 Gauss-Seidel Relaxation

Relaxation of the i-th unknown x_i:
Correct x_i^{old} by x_i^{new} such that the i-th equation of the equation system

$$A \cdot x = b$$

is correct.

Jacobi-iteration:
"Calculate the relaxations simultaneously for all $i = 1, \ldots, n$"
This means:

If $x^{old} = x^k$, then let $x^{k+1} = x^{new}$.

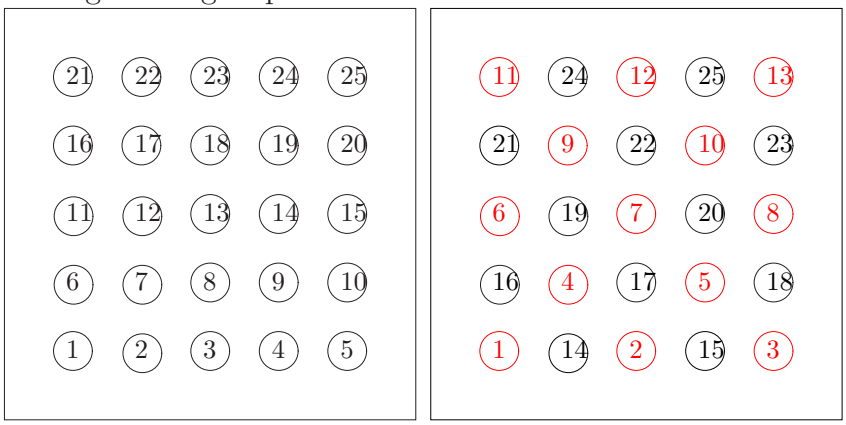
Gauss-Seidel-iteration:
"Calculate relaxation for $i = 1, \ldots, n$ and use the new values "
This means:

$x^{old,1} = x^k$
Iterate for $i = 1, \ldots, n$:
  Calculate $x^{new,i}$ by relaxation of the i-th component.
  Put $x^{old,i+1} = x^{new,i}$.
$x^{k+1} = x^{new,n}$

Remark.

- Jacobi-iteration is independent of the numbering of the grid points
- The convergence rate of the Gauss-Seidel iteration depends on the numbering of the grid points

For Gauss-Seidel iteration one often applies lexicographical and red-black numbering of the grid points.



3.3 cg Iteration

Let A be a symmetric positive definite $n \times n$ matrix
and $b \in \mathbb{R}^n$.
The gradient method for solving

$$Ax = b$$

is

Start with x_0 and calculate the sequence x_k by:

$$\begin{aligned} d_k &= b - Ax_k \\ \alpha_k &= \frac{d_k^T d_k}{d_k^T A d_k} \\ x_{k+1} &= x_k + \alpha_k d_k, \end{aligned}$$

where $k = 0, 1, 2, ...$

The cg iteration is:

Let the start solution be x_0. Then:	g_0	$=$	$Ax_0 - b$
	δ_1	$=$	$g_0^T g_0$ if $\delta_1 \leq \epsilon$ stop
	d_1	$=$	$-g_0$
recursion: $k = 1, \ldots :$	h_k	$=$	Ad_k
	α	$=$	$\frac{\delta_k}{d_k^T h_k}$
	x_k	$:=$	$x_{k-1} + \alpha d_k$
	g_k	$:=$	$g_{k-1} + \alpha h_k$
	δ_{k+1}	$=$	$g_k^T g_k$ if $\delta_{k+1} \leq \epsilon$ stop
	β_k	$=$	δ_{k+1}/δ_k
	d_{k+1}	$=$	$-g_k + \beta_k d_k$

3.4 Estimation of the Algebraic Error

Assume we want to solve

$$Ax = b$$

and we get the approximation $\tilde{x}$. A practical problem is:

- How large is the algebraic error $\|\tilde{x} - x\|$?
- Assume, we apply an iterative solver.
  How many iterations do we have to perform to obtain a small algebraic error?
- How to choose ϵ in the cg-iteration?
- Assume, you have implemented two iterative solvers. Which one is faster?

But: We do not know x !

Estimation of the Algebraic Error for Tests:
For testing a code one does the following test:
Construct right hand sides b such that the exact solution x is well-known.
Example:

- Choose $b = 0$.
- FD on a unit square: Choose $u = x^2y^3$.

Start with $x_0 = \underline{1}$.

Then, one can compute the algebraic error $\|\tilde{x} - x\|$ and one can compare two different codes.

A Hard Approach:
If the exact solution is unknown, one applies the following difficult approach:

Calculate a very good approximation x_e of x by a time consuming solver. Then, consider

$$\|\tilde{x} - x_e\|$$

as the algebraic error $\|\tilde{x} - x\|$.

Estimation of Algebraic Error by Residuum:
The residuum is defined as

$$r := A\tilde{x} - b$$

Then, $\|\tilde{x} - x\| = \|A^{-1}r\| \leq \|A^{-1}\|\|r\|$.
Example:
FD, Poisson on $]0,1[^2$: $\|A^{-1}\|_2 = \lambda_1^{-1} \approx \frac{1}{2\pi^2}$.

- Assume that $\tilde{x} - x = e_{m-1,m-1} + h^2 e_{1,1}$. Then,

$$\|\tilde{x} - x\|_2 \approx 1 \quad \text{and} \quad \|A^{-1}\|_2\|r\|_2 \approx h^{-2}.$$

  Since,

$$\|A^{-1}\|_2\|r\|_2 \approx \left(\frac{\lambda_{m-1,m-1}}{\lambda_{1,1}} + h^2\right) \approx h^{-2}.$$

- Assume that $\tilde{x} - x = h^2 e_{m-1,m-1} + e_{1,1}$. Then,

$$\|\tilde{x} - x\|_2 \approx 1 \quad \text{and} \quad \|A^{-1}\|_2\|r\|_2 \approx 1.$$

Thus, if $\|r\|$ is small, then $\|\tilde{x} - x\|_2$ can be large or small!

Therefore, do not use the size of the residuum to compare two different iterative algorithms.

Example:
FD, Poisson on $]0,1[^2$: We want to obtain $\|\tilde{x} - x\|_2 = O(h^2)$.

- MG: Iterate such that $\|r\| = O(1)$.
- SSOR: Iterate such that $\|r\| = O(h^2)$.

To find a more appropriate approach to estimate the algebraic error, we have to study iterative solvers in more detail.

Property of Iterative Solvers:
Let $x_0 \in \mathbb{R}^n$ and

$$\mathcal{S} : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

be an iterative solver such that the sequence $(x_i)_{i\in\mathbb{N}}$ defined by

$$x_{i+1} = \mathcal{S}(x_i)$$

converges to x. Most of the iterative solvers have the following property:
There exists a constant $0 < q < 1$ and $s, i_{\min} \in \mathbb{N}$ such that

$$\|x_{i+s+1} - x_{i+s}\| \leq q^s \|x_{i+1} - x_i\|$$

for every $i > i_{\min}$. q is called convergence rate of $\mathcal{S}$.

Algebraic Error of an Iterative Solver:

Theorem 2. *Let $0 < q < 1$, $s, i_{\min} \in \mathbb{N}$, $x_0 \in \mathbb{R}^n$ and*

$$\mathcal{S} : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

be an iterative solver such that the sequence $(x_i)_{i\in\mathbb{N}}$ defined by

$$x_{i+1} = \mathcal{S}(x_i)$$

converges to x and satisfies

$$\|x_{i+2} - x_{i+1}\| \leq q\|x_{i+1} - x_i\|$$

for every $i > i_{\min}$. Then, the algebraic error can be estimated by

$$\|x - x_i\| \leq \|x_{i+1} - x_i\|(1-q)^{-1}.$$

Proof:

$$\begin{aligned}\|x - x_i\| &\leq \sum_{k=i}^{\infty} \|x_{k+1} - x_k\| \\ &\leq \sum_{k=i}^{\infty} q^{k-i} \|x_{i+1} - x_i\| \\ &\leq \|x_{i+1} - x_i\|(1-q)^{-1}.\end{aligned}$$

q.e.d.

3.5 Estimation of the Convergence Rate

We want to find a small parameter such that

$$\|x_{i+2} - x_{i+1}\| \leq q\|x_{i+1} - x_i\|.$$

Several iterative solvers have the following property:
There exists a constant $0 < q < 1$ *and* $s, i_{\min} \in \mathbb{N}$ *such that*

$$\|x_{i+s+1} - x_{i+s}\| \leq q^s \|x_{i+1} - x_i\|.$$

for every $i > i_{\min}$.

- Calculate $\tilde{q} = \frac{\|x_{i+2} - x_{i+1}\|}{\|x_{i+1} - x_i\|}$ for large i.
- Calculate $\tilde{q} = \left(\frac{\|x_{i+s+1} - x_{i+s}\|}{\|x_{i+1} - x_i\|}\right)^{\frac{1}{s}}$ for large i, $s \approx 5 - 20$.

Take $\tilde{q}$ as an approximation of q.

Convergence Rate of Linear Iterative Solvers:
Let

$$\mathcal{S}(x_i) = Cx_i + d$$

be an iterative solver (C matrix and d vector).
Then, the convergence rate q does not depend on the right hand side b and not on the start value x_0
(with the exception of choosing an eigenvector as $x - x_0$).

Example 4. *The Gauss-Seidel iteration is a linear iterative solver. To estimate the convergence rate, choose the right hand side* 0 *and the start vector* $x_0 = \underline{1}$. *Then,*

$$\tilde{q} = \frac{\|x_{i+1}\|}{\|x_i\|}$$

is an approximate value of the convergence rate q *for large values* i.

Remark: *To avoid overflow and underflow, additionally normalize the vectors* x_i.

3.6 Estimation of the Convergence Rate by Residuum

We want to find a small parameter q such that

$$\|x_{i+2} - x_{i+1}\| \leq q\|x_{i+1} - x_i\|.$$

Another way to estimate the convergence rate is to study the behavior of the residuum as follows:

Let

$$r_i = Ax_i - b$$

- Calculate $\tilde{q} = \frac{\|r_{i+1}\|}{\|r_i\|}$ for large i.
- Calculate $\tilde{q} = \left(\frac{\|r_{i+s}\|}{\|r_i\|}\right)^{\frac{1}{s}}$ for large i, $s \approx 1 - 20$.

Take $\tilde{q}$ as an approximation of q.

3.7 Finding the Meshsize and the Number of Iterations

Assume we want to obtain a total error $\|u - u_{h,i}\| \leq \eta$!

1. For every meshsize h calculate $\tilde{q}$ by

$$\tilde{q} = \left(\frac{\|u_{i_h+s+1} - u_{i_h+s}\|}{\|u_{i_h+1} - u_{i_h}\|}\right)^{\frac{1}{s}}$$

for large i_h and suitable $s \approx 1 - 20$.

2. Calculate i_h such that $\|u_{h,i_h+1} - u_{h,i_h}\|(1-\tilde{q})^{-1} \leq \frac{1}{4}\eta(1-2^{-p})$. This implies
$$\|u_h - u_{h,i_h}\| \leq \frac{1}{4}\eta(1-2^{-p}).$$

3. Choose h such that $\|u_{h,i_h} - u_{h/2,i_{h/2}}\| \leq \frac{1}{4}\eta(1-2^{-p})$.

Then, we obtain $\|u_h - u_{h/2}\| \leq \frac{3}{4}\eta(1-2^{-p})$ and thus

$$\|u - u_h\| \leq \frac{3}{4}\eta.$$

This implies

$$\|u - u_{h,i_h}\| \leq \eta.$$

4 Software Development

4.1 Debugging

Is there a Bug in the Code or not?:
If a simulation program does not simulate a physical process in a correct way, there can be different reasons for this:

- inaccuracy of the model.
- error in the mathematical solver.
- error (bug) in the code.

There exist different bugs:

- syntax error,
- wrong usage of memory,
- logical sequence of the code is not correct, or
- the mathematical formula is not implemented in a correct way.

Debugging with gdb:
Use debuggers like `gdb`.
To this end, compile with option `-g` and execute `gdb code`.
Commands of this debugger are

- `b ln` Set breakpoint at line number *ln*.
- `r` Run code.
- `s` Make one step.
- `S` Make one step and do not go into functions.
- `p u` Print `u`.
- `b` Backtrace how the code went to a certain point in the code.
- `up` Go up the stack frame.
- `down` Go down the stack frame.
- `c` Continue running the code.

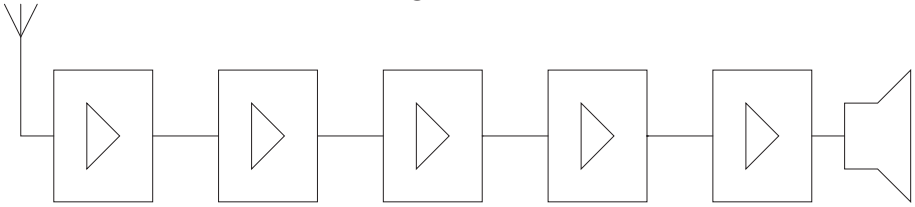
Apply intelligent print statements!:
Instead of using the command `p` in `gdb` write your own intelligent print statements, which `gdb` does not contain.

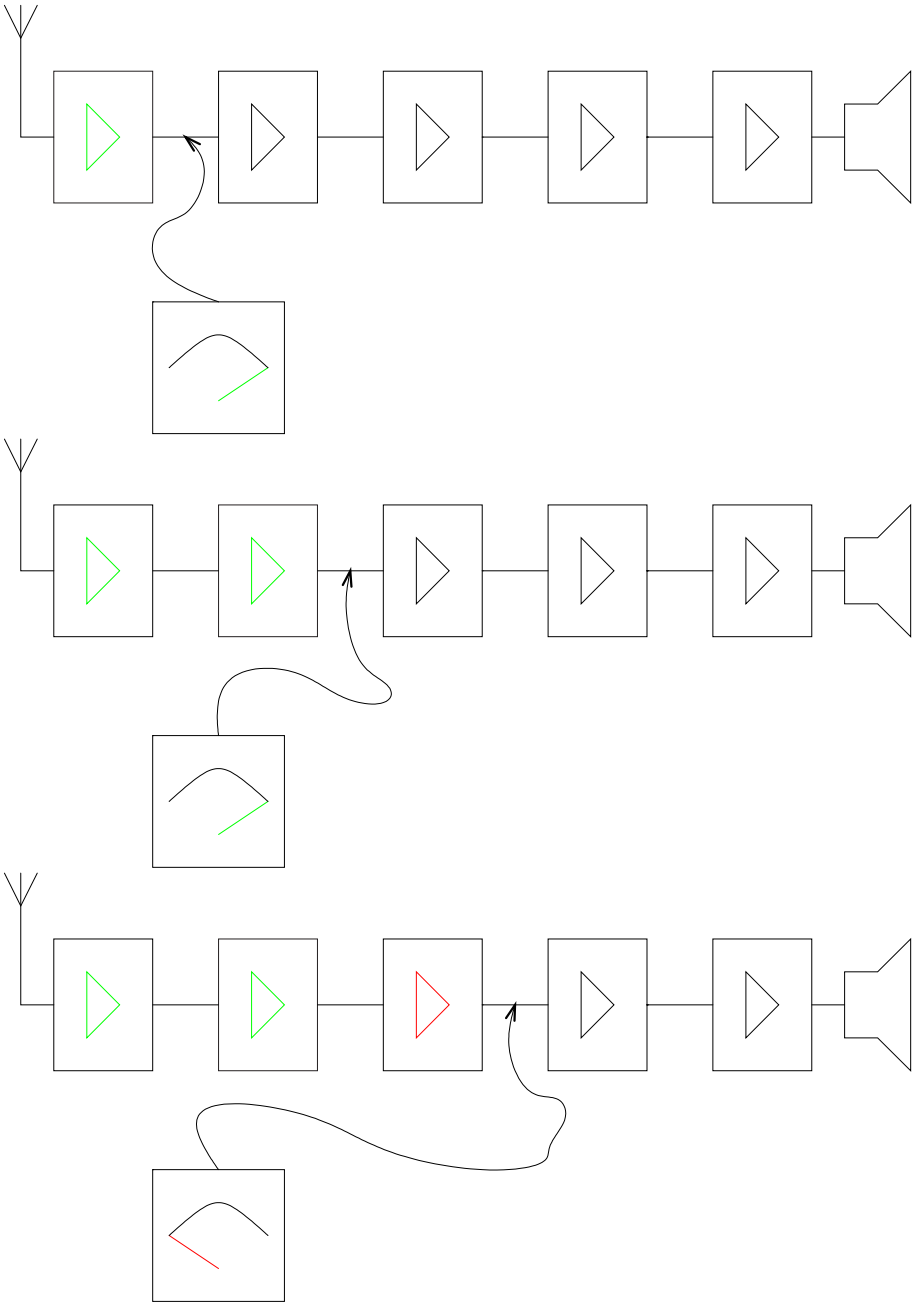
Example:

```
Print_L_infty(u);
```

Prints the L^∞ norm of a vector u.

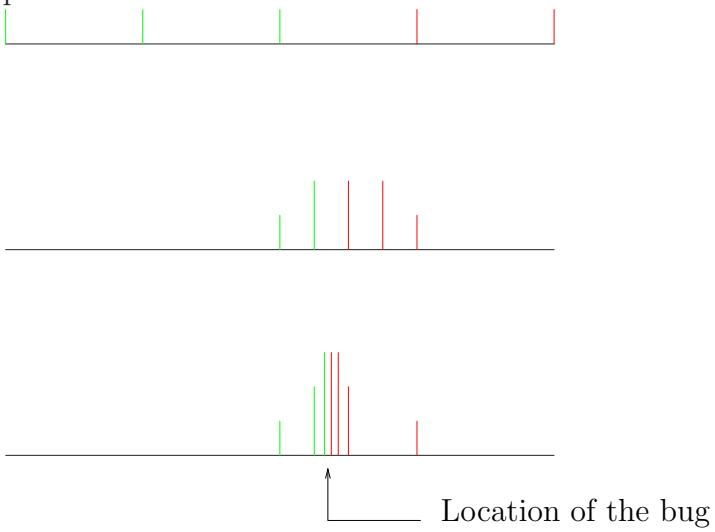
Finding a Bug in a Radio:
To find a bug in a code, we assume that the code runs in a sequential way similar to the amplification of a signal in a radio. The following pictures simplify an approach to find a bug in a radio. Similarly, we insert print statements in a code to find a bug.





Hierarchical Search of Bug:
If a code consists of N of statements, then inserting a print statement after each statement is too complicated for large numbers N. In this case a hier-

archical search with a small number of print statements is more appropriate. See the following picture:



Then, a bug contained in N statements can be found by $\ln(N)$ print statements.

Reduction of the Problem:
A very important concept is the concept to

reduce a *big* problem to a *smaller* one.

Of course the hierarchical search can be treated as such a concept. But there are also other ways to reduce a problem.

- Skip parts of the code in an hierarchical way such that the resulting code still contains the bug.
- Comment out statements in the code. As an example omit coarse grid correction in a multigrid code.
- Write a smaller code which contains the bug.
- Find a problem with a smaller problem size, such that the bug appears!
- Find a problem with known exact solution or a more simple solution! (see Section 4.2 testing)

If a reduction of the code is not possible any more, then analyze the code.

Memory check by valgrind:

Call valgrind by

```
valgrind --tool=memcheck --leak-check=yes run
```

Use of valgrind for:

- finding causes for segmentation faults.
- finding memory leaks

Warnings in an HPC code:
Warnings in a code are very useful to avoid bugs in a code.

```
class vector {
  public:
    vector(int dim_);
    double operator[] (int i) {
      if(i<   0) cout << ‘‘i negative ‘‘ << endl;
      if(i>=dim) cout << ‘‘i too large‘‘ << endl;
      return a[i];
    }
  private:
    int dim;
    double *a;
}
```

But this implementation of `operator[](int i)` is very inefficient.
To increase performance implement a developer version as in the following example:

```
#define developer_version true
// #define developer_version false

  ...
```

```
double vector::operator[] (int i) {
  if(developer_version) {

    if(i<   0) cout << ``i negative `` << endl;
    if(i>=dim) cout << ``i too large`` << endl;
  }
  return a[i];
}
```

or one can use assert as follows:

```
double vector::operator[] (int i) {
   assert(i<0 && i>=dim);
  return a[i];
}
```

Avoid == Sign:
Try to avoid the == sign. Instead use $\geq$ or $\leq$.

Avoid `double` Comparison:
Example:
Instead of

```
double x,h;
h = 1.0 / 10.0;
for(x=0.0;x<=1.0;x=x+h) {
  ...
}
```

write

```
double x,h;
h = 1.0 / 10.0;
for(int i=0;i<10;++i) {
  x = i*h;
  ...
}
```

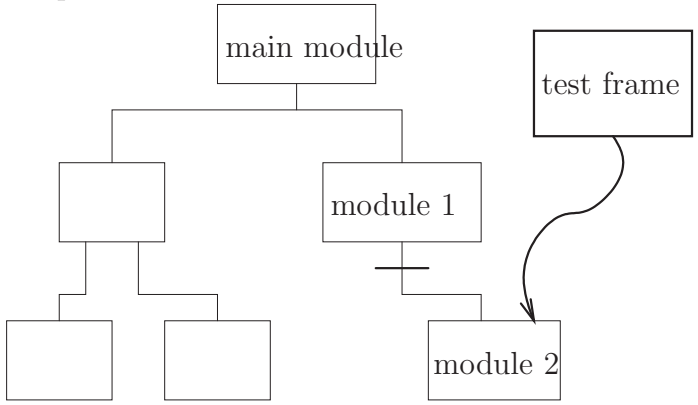
4.2 Testing

Mathematical Error or Bug in the Code?:
Often it is difficult to decide, if there is

- an inaccuracy in the model,
- an error in the mathematical solver, or
- a bug in the code.

This is one of the reasons why a simulation code must be developed in several modules. Each of the modules must be tested in detail.

Test Frame for Module Test:
Implement a test frame for modules. This frame gives a module certain input data and requires certain output data. If the output data are correct, then the module is expected to be correct.



Integration and Regression Test:

- Using the test frame, apply the same tests for each module while developing the code and adding new modules.

  This is called *integration test*.

- Store results of your tests in a data file. Compare new test results with older test results.

  This is called *regression test*.

Finding Test Functions:
The general concept is to calculate the right hand side for a given exact solution. These exact solution are the *test functions*.

There are different kinds of test functions:

- function $\underline{0}, \underline{1}, x, y, ...$
- functions with special properties:
  - $u = \sin(x * \pi)\sinh(y * \pi)$, then $\triangle u = 0$ and u has nice Dirichlet boundary conditions on $\Omega =]0,1[^2$. $u = e^t \sin(x * \pi)$ satisfies the PDE $\frac{\partial u}{\partial t} = -\pi^{-2}\frac{\partial^2 u}{\partial x^2}$.
  - $u = x^2 * y^3$ is an exact solution for a FD discretization (see Section 2.3).
- symmetric solutions like $u = x^5 * y^5$. For symmetric problems, changing the coordinates x and y must lead to the same result.
- general functions. Calculate right hand side by a computer manipulation program (maple,mathematica).

First, test your code with the simplest one!

Test Parameters:
Consider the PDE:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\triangle u + aw - f \\ \frac{\partial w}{\partial t} &= -\triangle w + bu - g \end{aligned}$$

Parameters in a FD discretization are:

- physical parameters a, b.
- meshsize h, timestep τ.
- number of grid points N, number of timesteps m.

First, test your code for physically not correct parameters:

- $a, b = 0, \pm 1, \pm 10, ...$

- $N = 1$ and $m = 1$, ...

Test Part of the PDE:
Instead of

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\triangle u + aw - f \\ \frac{\partial w}{\partial t} &= -\triangle w + bu - g \end{aligned}$$

first test the stationary scalar equation:

$$-\triangle u + aw = f$$

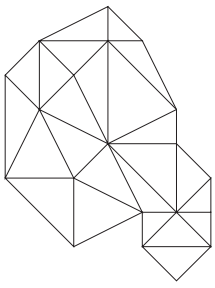
and the stationary system:

$$\begin{aligned} -\triangle u + aw &= f \\ -\triangle w + bu &= g. \end{aligned}$$

Test Convergence:

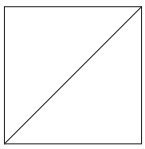
- Test the convergence of your discretization for different test functions and parameters in the equation.
- Test the convergence rate of your iterative solver for different parameters.

Test of an Unstructured Grid Code:
Assume there is a bug in your unstructured grid code with a complicated unstructured grid like:

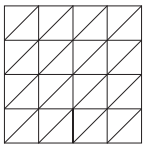


$\longrightarrow$ To Calculate the matrix elements in each step of the code by hand is too complicated!

To test your code let your unstructured grid generator generate a simple structured grid like



or

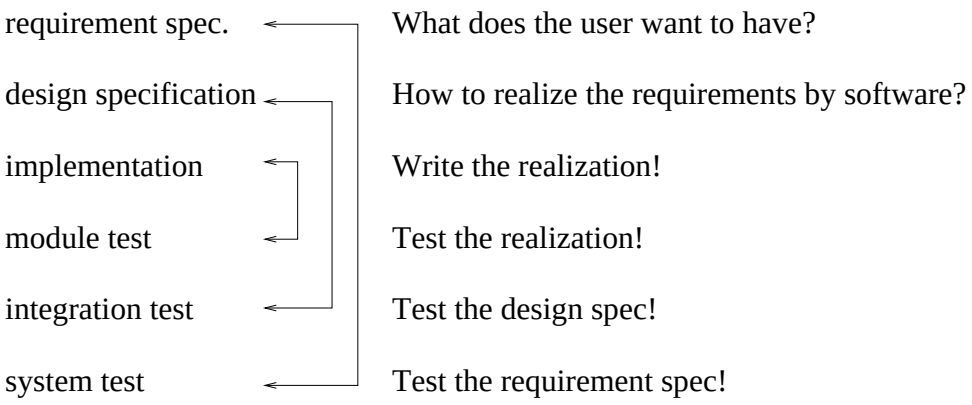


and test your code.

$\longrightarrow$ Change x and y coordinates and test symmetry of your code!

4.3 Software Development

The following picture describes the general concept of software development:



Implement and Test:
Build up your code step by step!
Example: Write a code for

$$-\triangle u + aw = f$$

and then for

$$-\triangle u + aw \quad = \quad f$$

$$-\triangle w + bu = g$$

and at last for:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\triangle u + aw - f \\ \frac{\partial w}{\partial t} &= -\triangle w + bu - g. \end{aligned}$$

Implement one module and test it!

Types of Modules:

- vector library (contains matrix multiplication, use libraries like LAPACK)
- grid generator
- linear equation solver
- calculation of stiffness matrix
- parallelization module
- input, output
- applications (different fluid dynamics applications)

Problems in PDE Software Development:

- Black box solvers which are independent of the PDE and the discretization would be very helpful for the software development (algebraic multigrid (AMG), direct solver). But the optimal solver depends on the PDE and its discretization.
- Optimal solvers use the data structure of the discretization.
- Complicated data structure is needed for adaptive parallel solvers with load balancing.
- It is difficult to describe suitable interfaces between solvers.
- A clear software design often is in contradiction to efficiency. Therefore, expression templates and other template constructions are needed!

4.4 Basic Concept of Expression Templates

Operator Overloading for Vector Class:

Consider the vector class

```
class vector {
 public:
   vector(int l);
   double operator[](int i) { return p[i]; }
   ...
 private:
   int length;
   double *p;
};
```

How should we implement an operator

```
vector operator+(vector &a, vector &b)
```

in an efficient way?

Operator Overloading for Small Vectors:

In case of small vectors, one can implement a vector class as in the following example for the vector class `complex`:

Example: vector class complex:

```
class complex {
 public:
   complex(double& re, double& im);
   ...
   double Re, Im;
};

complex operator+(complex &a, complex& b) {
  return complex(a.Re + b.Re,a.Im + b.Im);
}
```

In case of longer vectors introduce the length of the vector as a template parameter.

Vector Class for Long Vectors:

In case of large vectors, the storage has to be allocated by `new`. The addition of vectors has to be performed by a `for` loop.

```
class vector {
 public:
   vector(int l) { p = new double[l];
                   length = l; };
   double operator[](int i) { return p[i]; }
   ...
 private:
   int length;
   double *p;
};
```

Problem:

- Should `vector operator+(vector &a, vector &b)` allocate an auxiliary vector?
- Efficient implementation of `c = a+b+d;` requires only one loop!

Realization of an Efficient Operator+:

Implement `operator+` such that it gives back an object, which is able to add two vectors:

```
class add_vector {
 public:
   add_vector(double& *a, double& *b)
       : pa(a), pb(b) {};
   double operator[](int i) const
      { return pa[i] + pb[i]; }
   ...
 private:
   double *pa, *pb;
};
```

Now, the expression is evaluated in the `operator +` as follows:

```
class vector {
 public:
   ...
   double operator[](int i) { return p[i]; }
   void   operator=(add_vector& av) {
     for(int i=0;i<lenghth;++i) {
       p[i] = av[i];
     }
   }
   ...
 private:
   int length;
   double *p;
};
```

By this construction, we can add only two vectors. To evaluate larger expressions, we need expression templates.

Expression Template - Wrapper Class:

To construct expression templates, we first need a wrapper class, which represents all possible expressions:

```
template<class A>
class DExpr {
 private:
  A a_;
 public:
  DExpr(const A& x)
    : a_(x)  {}
  double operator[](int i) const
        { return a_.[i]; }
};
```

Expression Template - Operator +:
The following class represents an object which is able to add two expressions.

```
template<class A, class B>
class DExprSum {
 const A a_;  const B b_;
 public:
  DExprSum(const A& a, const B& b)
    : a_(a), b_(b)  {}
  double operator[](int i) const {
    return a_.[i] + b_.[i];   };
};

template<class A, class B>
DExpr<DExprSum<DExpr<A>, DExpr<B> > >
operator+(const DExpr<A>& a,const  DExpr<B>& b) {
  typedef DExprSum<DExpr<A>, DExpr<B> > ExprT;
  return DExpr<ExprT>(ExprT(a,b));
}
```

The expression is evaluated in the `operator +` as follows:

```
  class vector {
   public:
     ...
     double operator[](int i) { return p[i]; }

     template <class A>
     void operator=(const Expr<A>& a) {
       for(int i=0;i<lenghth;++i) {
         p[i] = a[i];
       }
     }
     ...
   private:
     int length;
     double *p;
  };
```

Additionally, we need operators like

```
template<class B>
```

```
DExpr<DExprSum<Dvector, DExpr<B> > >
operator+(const vector& v,const  DExpr<B>& b) {
  typedef DExprSum<Dvector, DExpr<B> > ExprT;
  return DExpr<ExprT>(ExprT(v,b));
}
...
```

Properties of Expression :

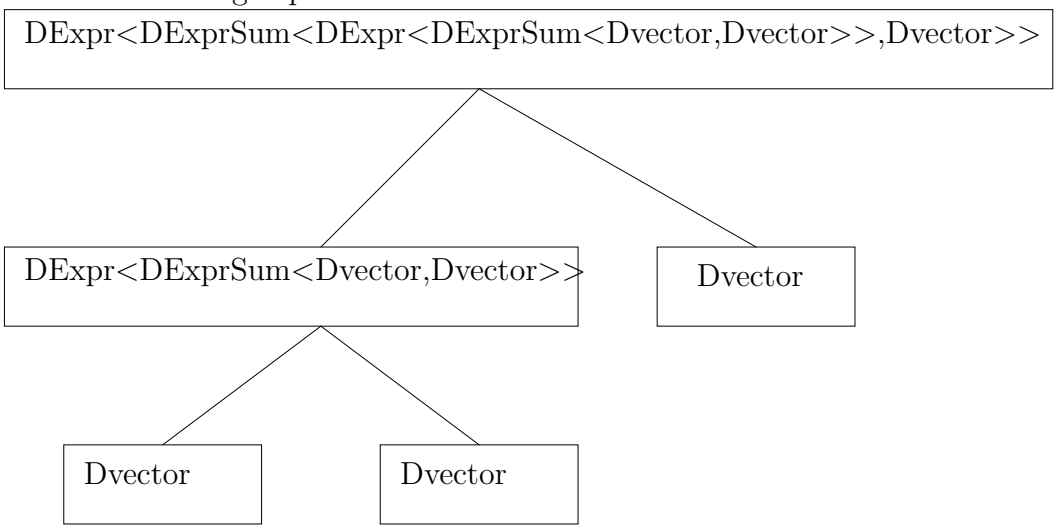
- efficient implementation by inlining.
- parallelization by OpenMP is possible.
- user friendly interface.

Expression Tree:

The expression

```
d = a + b + c;
```

leads to the following expression tree:



First Simplification:

```
template<class A, class B, class Op>
class DExprBinOp {
 const A a_;  const B b_;
 public:
  DExprBinOp(const A& a, const B& b) : a_(a), b_(b)  {}
  double operator[](int i) const {
    return Op::apply(a_.[i], b_.[i]);};
};
class DApSum {
 public:
  DApSum() { }
  static inline double apply(double a, double b)
    { return a+b; }
};

template<class A, class B>
DExpr<DExprBinOp<DExpr<A>, DExpr<B>, DApSum> >
operator+(const DExpr<A>& a,const  DExpr<B>& b)
{
  typedef DExprBinOp<DExpr<A>, DExpr<B>, DApSum> ExprT;
  return DExpr<ExprT>(ExprT(a,b));
}
```

Second Simplification:

```
template <class A> struct Expr{
  inline const A& operator~() const{
    return static_cast<const A&>(*this);}
};
class vector : public Expr<vector> {
  public:
     ...
     template <class A>
```

```
     void operator=(const Expr<A>& a) {
       for(int i=0;i<length;++i) {
         p[i] = (~a).[i];
     }}
     ...
};

template <class A, class B>
class  DExprSum : public Expr<DExprSum<A,B> >{
  const A& a_; const B& b_;
 public:
  DExprSum(const A& a, const B& b)
    : a_(a), b_(b){}
  double operator[](int i) const {
    return a_.[i] + b_.[i];   };
}
```

Now, we can implement the `operator+` in the following short way:

```
template <class A, class B>
inline DExprSum<A,B> operator+ (const Expr<A>& a, const Expr<B>& b){
  return DExprSum<A,B>(~a,~b);
}
```

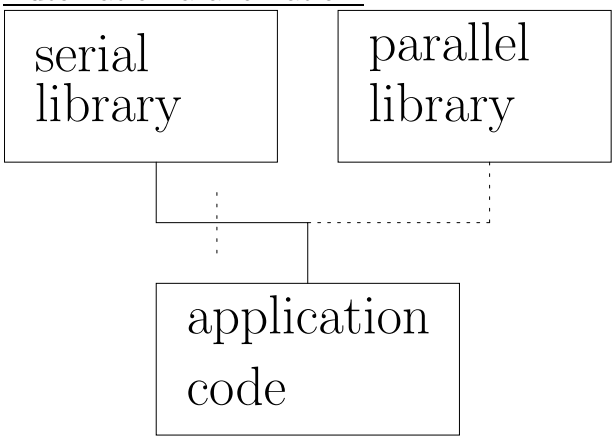
Observe, that we do not need an additional implementation of the `operator+` for arguments like `vector` and `Expr` or `vector` and `vector`.

cg with Expression Templates:
This example shows how to implement the cg algorithm by expression templates:

```
  r = A*u - f;
  d = -r;
  delta = product(r,r);
  for(i=1;i<=iteration && delta > eps;++i) {
    g = A*d;
    tau = delta / product(d,g);
    r = r + tau*g;
```

```
    u = u + tau * d;
    delta_prime = product(r,r);
    beta = delta_prime / delta;
    delta = delta_prime;
    d = beta*d - r;
  }
```

Automatic Parallelization:



Automatic parallelization means that only a change of the included library leads to a parallel code.

Example:

```
template <class A>
void vector::operator=(const Expr<A>& a) {
  #pragma omp parallel for
  for(int i=0;i<length;++i) {
     p[i] = (~a).[i];
  }
}
```

In some cases a straight forward implementation of expression templates leads to less an efficient codes than a direct implementation. The reason is that the compiler cannot see a difference between expressions like

```
  a = b+b+b+b;
```

and

```
a = b+c+d+e;
```

To avoid this problem one can construct enumerated variables.

```
variable<1> a;
variable<2> b;
....
```

Here the class `variable<n>` has an additional template parameter `n`.

4.5 Interfaces with Expression Templates

Expression Templates for Vectors:

Construct operators for operations between

- vectors
- matrix and vector and
- matrices.

`Blitz++` is such a library.

Expression Templates on Structured Grids:

Let us assume that we want to perform finite difference operations on a 2D-structured grid Ω_h.

Implement expression templates such that

```
u[I][J] = 0.25*(u[I+1][J]+u[I-1][J]+
                u[I][J+1]+u[I][J-1]);
```

performs a red black Gauss-Seidel iteration for Poisson's equation on Ω_h. Here,

- u a vector on the grid Ω_h
- $u[I][J]$ represents $u(ih, jh)$

- $u[I+1][J]$ represents $u((i+1)h, jh)$
- ...

Automatic parallelization of the above expression template implementation is possible.

A Jacobi-iteration for Poisson's equation has to be implemented as follows:

```
r[I][J] =  0.25*(u[I+1][J]+u[I-1][J]+
                 u[I][J+1]+u[I][J-1]);
u[I][J] = r[I][J];
```

One also can implement an operator `Laplace_FD(u)` representing the mathematical operator

$$\frac{1}{h^2}\Big(4 * u(ih, jh) \quad -u((i+1)h, jh) - u(ih, (j+1)h) \\ -u((i-1)h, jh) - u(ih, (j-1)h)\Big).$$

Let `Laplace_FD_diag()` be the corresponding diagonal coefficient vector of `Laplace_FD(u)` :

$$\left(\frac{h^2}{4}\right)^{-1}.$$

Then, a Gauss-Seidel iteration for $-\triangle u = f$ can be implemented as follows

```
u = u - (Laplace_FD(u)+f)/  Laplace_FD_diag();
```

and Jacobi by

```
r = u - (Laplace_FD(u)+f)/  Laplace_FD_diag();
u = r;
```

Consider the following implementation of Gauss-Seidel:

```
u[I][J] =  0.25*(u[I+1][J]+u[I-1][J]+
                 u[I][J+1]+u[I][J-1]);
```

Problems:

- What is the range of I and J?
- How, to set values at the boundary?
- How, to implement boundary conditions?

A suitable language for implementing PDE solvers is a current research topic. An optimal interface language is unknown up to now!

Suggestions:

- geometric objects - algebraic objects
- restriction operator to connect geometric objects and algebraic objects.
- vectors on grids and pure algebraic vectors.

Geometric objects are objects in $\mathbb{R}^3$. To describe geometric objects one can apply any method which is used in CAD (computer added design). Let us present a simple example:

Geometric objects:

```
vector3D Ma(0.0,2.0,1.0);
vector3D Mb(0.0,0.0,1.0);
Ball ball_a(1.0, Ma);
   // domain with radius 1.0 at point Ma
Ball ball_b(1.2, Mb);
   // domain with radius 1.2 at point Mb
 ...
Domain domain = ball_a || ball_b;
```

Here the operator `||` calculates the union of two geometric objects.

Algebraic objects are vectors for example. There exist natural mathematical operators for these objects like `+`:

Algebraic objects:

```
vector v1(1000), v2(1000), v3(1000);
...
v3 = v1 + v2;
```

In scientific simulation we need discretization grids on domains. Thus, discretization grids are objects which depend on a geometric object and a discretization method. The simplest discretization is a rectangular grid of mesh size h. In the following example $\bar{\Omega}_h^a$ is a rectangular discretization grid on the domain Ω^a with meshsize h.

```
// Geometric objects:
Domain domain_a = ...; //  Ω^a
Domain domain_b = ...; //  Ω^b

    Grid grid(domain_a,h); // Ω̄_h^a
// grid on domain_a with meshsize h
Subgrid subgrid(grid,domain_b);            // Ω̄_h^a ∩ Ω^b
Boundary_subgrid boundary(grid);           // Γ_h^a = Ω̄_h^a ∩ ∂Ω^a
Interior_subgrid interior(grid);           // Ω_h^a
Boundary_subgrid Dirichlet(boundary,domain_b);
// Γ_h^a ∩ Ω^b
```

For implementing algorithms on a discretization grid we need variables on discretization grid. These are vectors, such that every component of the vector corresponds to a grid point. Then, the operator `|` connects an algebraic and a geometric object.

```
// Geometric objects:
Domain domain_a = ...; //  Ω^a
Grid grid(domain_a,h); // Ω̄_h^a
// grid on domain_a with meshsize h
Boundary_subgrid boundary(grid);           // Γ_h^a = Ω̄_h^a ∩ ∂Ω^a
Interior_subgrid interior(grid);           // Ω_h^a = Ω̄_h^a \ ∂Ω^a
// Variable: vector on a grid (algebraic vector with geometric information)
Variable u(&grid), f(&grid);               // u, f ∈ R^|Ω̄_h^a|
coordinate_x X; coordinate_y Y; // coordinates
// Application of the restriction operator

  u =  X*X*Y*Y  | boundary;
```

Here the operator `|` connects the algebraic expression `u = X*X*Y*Y` with the geometric object `boundary`. Therefore, the mathematical meaning of the

last line of the above code is:

$$u(x,y) \;=\; x*x*y*y \quad \forall (x,y) \in \Gamma_h^a = \bar{\Omega}_h^a \cap \partial\Omega^a$$

This mathematical expression can also be described in the following notation:

$$u(x,y) \;=\; x*x*y*y\Big|_{\Gamma_h^a}.$$

Now, let us present an example for solving approximatively Poisson's equation on a domain using the finite difference discretization and 50 Gauss-Seidel relaxations.

```
   // Geometric objects:
Domain domain_a = ...; //  Ω^a

     Grid grid(domain_a,h); // Ω̄^a_h
// grid on domain_a with meshsize h
Boundary_subgrid boundary(grid);          // Γ^a_h = Ω̄^a_h ∩ ∂Ω^a
Interior_subgrid interior(grid);          // Ω^a_h = Ω̄^a_h \ ∂Ω^a
// Variable: vector on a grid (algebraic vector with geometric information)
Variable u(&grid), f(&grid);              // u, f ∈ R^|Ω̄^a_h|
coordinate_x X; coordinate_y Y; // coordinates
// Application of the restriction operator
  u =  X*X*Y*Y          | boundary;
  f = -2*(X*X+Y*Y)      | interior;
  for(int i=1;i<50;++i)
    u = u-(Laplace_FD(u)+f) / Laplace_FD_diag() | interior;
```

5 Parallelization

5.1 Introduction

One can distinguish the following parallelization concepts:

- Shared memory parallelization
  - Parallelization with one main memory and several different processors (see Figure 11)

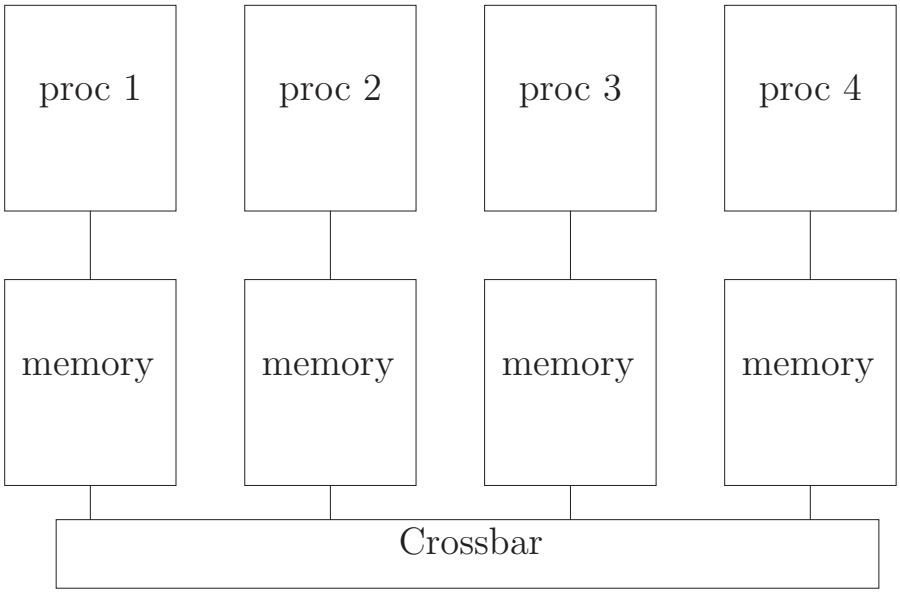


Figure 19: Distributed Memory Computer Architecture.

  - NUMA architecture (Non-Uniform Memory Access).

- Distributed memory parallelization (see Figure 19)
- Hybrid parallelization with a shared memory and a distributed memory
- Vectorization. One processor can perform parallel computations on long vectors.

5.2 MPI - Message Passing Interface

- MPI is a library language for C, C++ and FORTRAN.
- There exist different MPI libraries. MPICH and MPI-LAM are one of them.
- The MPI library is included by `mpi.h`.

Run an MPI code by

```
mpirun -np p code
```

Here `p` is the number of processors.

- Every processor runs the same program with a different *rank*.
- Data are send by MPI-functions from one processor to the other. All MPI-functions have the prefix `MPI_`.
- Data are send from one processor to the other of a certain *communicator*. The *rank* of the processor depends on the communicator. Here, we use only the communicator `MPI_COMM_WORLD` which is of type `MPI_Comm`.

First MPI - Functions:
Let us describe the most elementary MPI functions:

```
int MPI_Init(int *argc, char ***argv);
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Comm_Finalize();
```

`size` is the total number p of processors and
`rank` the number from $0, ..., p-1$.

The return value of these function is an information about the error. This will be discussed later.

MPI_Bcast and MPI_Reduce:
`MPI_Bcast` sends data from processor with number `root` to all other processors.

`MPI_Reduce` applies an operation to data of all processors. The result is sent to root.

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm);
```

Pointers (like `buf`) point to arrays of type `datatype` and length `count`.
Possible data types for `MPI_Datatype` are:

```
MPI_INT, MPI_DOUBLE, MPI_LONG, MPI_CHAR, ...
```

Example Numerical Integration:
The trapezoidal rule is the following rule for numerical integration:

$$\int_0^1 f(x)dx \approx h\sum_{i=1}^{n} f\Big(h(i-0.5)\Big)$$

where $h = \frac{1}{n}$. To parallelize this formula let us assume that $n = kp$, where p is the number of processors. Then, we get

$$\int_0^1 f(x)dx \approx \sum_{j=1}^{p} h\sum_{i=1}^{k} f\Big(h(((j-1)k+i)-0.5)\Big)$$

A parallel code using MPI is:

```
#include "mpi.h"
#include <math.h>

double f(double x) {
  return x*x+sin(x);
}

int main(int argc, char** argv) {
  int n,k;
  double h, my_integral, integral;
  ifstream PARAMETER;

  int my_rank;                  // Rank of process
  int p;                        // Number of processes

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if(my_rank==0) {
    PARAMETER.open("para.dat",ios :: in);
```

```
    PARAMETER >> k;               // problem size
    PARAMETER.close();
  }

  MPI_Bcast(&k,1,MPI_INT,0,MPI_COMM_WORLD);

  n = p*k;
  h = 1.0 / n;

  my_integral = 0.0;
  for(int i=1;i<=k;++i)
     my_integral = my_integral + f(h*((k*my_rank+i)-0.5));

  MPI_Reduce(&my_integral,&integral,1,MPI_DOUBLE,
             MPI_SUM,0,MPI_COMM_WORLD);

  integral = integral *h;
  if(my_rank==0)
    cout << "Integral is: " << integral << endl;

  MPI_Finalize();
  return 0;
}
```

Send and Receive with Blocking:

`MPI_send` sends data to the processor with destination rank `dest` and with tag (*german: Anhänger, Etikett*): `tag`.

Valid tags are values from 0 to 32767.

`MPI_Recv` receives data from processor with source rank `source`. This function returns the status `status`.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest,   int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm, MPI_Status *status);
```

`status` provides the following informations:

- `status.MPI_SOURCE`
- `status.MPI_TAG`

For other functions, `status` can provide informations about the error.

Example for Send and Receive:

```
MPI_Reduce(&my_integral,&integral,1,MPI_DOUBLE,
           MPI_SUM,0,MPI_COMM_WORLD);

if(my_rank!=0)
  MPI_Send(&my_integral,1, MPI_DOUBLE,0,
           10+my_rank, MPI_COMM_WORLD);
else {
  double source_integral;
  MPI_Status status;

  integral = my_integral
  for(int source=1;source<p;++source) {
     MPI_Recv(&source_integral,1, MPI_DOUBLE,source,
              10+source, MPI_COMM_WORLD, &status);
     integral = integral + source_integral;
     cout << " I got message from: " << source << endl;
  }
}
```

Improvement by MPI_ANY...:
The computations of processor 1 might be more time consuming than the computations of processor p. In this case, the following code is more efficient:

```
if(my_rank!=0)
  MPI_Send(&my_integral,1, MPI_DOUBLE,0,
           10+my_rank, MPI_COMM_WORLD);
else {
  double source_integral;
  MPI_Status status;
```

```
    integral = my_integral
    for(int source=1;source<p;++source) {
       MPI_Recv(&source_integral,1, MPI_DOUBLE,MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
       integral = integral + source_integral;
       cout << " I got message from: "
            << status.MPI_SOURCE << endl;
    }
  }
```

Send and Receive without Blocking:
The difference between `MPI_Isend` and `MPI_send` is that `MPI_Isend` does not block the execution of commands after calling `MPI_Isend`.

The handler `request` provides informations about finishing `MPI_Isend`. This information can be obtained by `MPI_Test` using `*flag`.

`MPI_Wait` waits until `MPI_Isend` is finished.

`MPI_Waitall` waits until several `MPI_Isend` and `MPI_Irecv` are finished.

These functions are defined as follows:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest,  int tag, MPI_Comm comm,
              MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request);
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Waitall(int count,MPI_Request *array_of_request,
                MPI_Status *array_of_statuses);
```

:
Instead of

```
MPI_Reduce(&my_integral,&integral,1,MPI_DOUBLE,
           MPI_SUM,0,MPI_COMM_WORLD);
```

one can write

```
double      *source_integral;
MPI_Request *req;           MPI_Status  *status;
req             = new MPI_Request[p-1];
status          = new MPI_Status[p-1];
source_integral = new double[p];
if(my_rank!=0) {
  MPI_Isend(&my_integral,1, MPI_DOUBLE,0,
            10+my_rank, MPI_COMM_WORLD,&req[0]);
  MPI_Waitall(1,req,status); }
else {
  source_integral[0] = my_integral;
  for(int source=1;source<p;++source) {
     MPI_Irecv(&source_integral[source],1, MPI_DOUBLE,source,
               10+source, MPI_COMM_WORLD, &req[source-1]);
  }
  MPI_Waitall(p-1,req,status);
  integral=0.0;
  for(int source=0;source<p;++source) {
     integral = integral + source_integral[source];
} }
```

Error Handlers:
There are two different error handlers:

- `MPI_ERRORS_ARE_FATAL` (default): This error handler forces to abort all MPI processes.
- `MPI_ERRORS_RETURN`: Now, the MPI-function returns an error information.

One can set the handler `MPI_ERRORS_RETURN` by

```
MPI_Errhandler_set(MPI_COMM_WORLD,
                   MPI_ERRORS_RETURN);
```

Error Handler `MPI_ERRORS_RETURN`:
Let `errcode` be a return value of an MPI-function. Then,

- `errcode==MPI_SUCCESS` (This means there is no error.), or
- `errcode` can be decoded by

```
int MPI_Error_class(int errcode,
                    int *errorclass)
```

Possible values for `*errorclass` depend on the MPI implementation. In MPI-1 the following classes are defined:

```
MPI_SUCCESS
MPI_ERR_RANK
MPI_ERR_BUFFER
...
```

Example:

```
int dest, errorclass;
if(developer_version)
   MPI_Errhandler_set(MPI_COMM_WORLD,
                      MPI_ERRORS_RETURN);

...
errcode = MPI_Send(...,dest,...);
if(developer_version)
  if(errcode != MPI_SUCCESS) {
    MPI_Error_class(errcode,&errorclass);
    if(errorclass==MPI_ERR_RANK)
      cout << " MPI send rank error: " << dest << endl;
    if(errorclass==MPI_BUFFER)
      cout << " MPI send buffer error. " << endl;
    ...
  }
```

Test Incoming Message:
Sometimes a process would like to know, whether there is a process sending a message. This can be tested by `MPI_Iprobe`.

Example

```
if(my_rank==0) {
  MPI_Status  status;  int flag = false;
  MPI_Iprobe(MPI_ANY_SOURCE,2,MPI_my_rank,
             &flag, &status );
  if(flag==true) {
     int rank_from = status.MPI_SOURCE;

     MPI_Recv(buffer, num_data,
      MPI_DOUBLE,rank_from,
      2,MPI_my_rank, &status); }}
```

Debugging:
A parallel debugger is totalview.

The running state on every processor is reported on a different window.

5.3 Distributed Memory Parallelization of PDE-Solvers

Let us assume that a PDE is discretized on the discretization grid Ω_h. If $|\Omega_h|$ is very large, than the data stored on this grid cannot be stored in the main memory. In this case, one has to apply a distributed memory parallelization concept.

A distributed memory parallelization of algorithms on Ω_h is based on a partition of Ω_h:

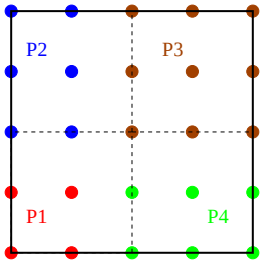
$$\Omega_h = \bigcup_{i=1}^{p} \Omega_h^i.$$

An optimal partitioning depends on

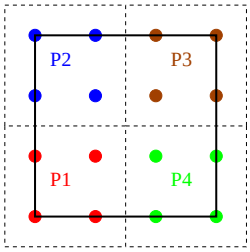
- the "sequential flow" of the algorithm,
- the amount of data to be sent, and
- the amount of computations, which have to be performed on each partition. This computational amount should be balanced on the partitions (load balancing).

Optimal Partitioning for Relaxation Methods:

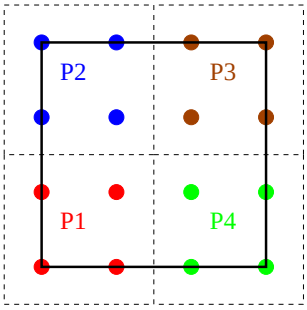
point approach



cell approach



Cell Partitioning:



$$\Omega_h = \{(ih, jh) \mid i, j = 0, ..., N-1\},$$

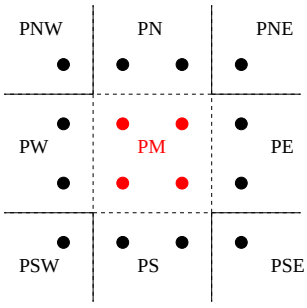
where $h = \frac{1}{N-1}$ and
$N = \sqrt{p}n,\ n, N \in \mathbb{N}.$

$$\Omega_h^{k,s} = \{((kn+i)h, (sn+j)h) \mid i, j = 0, ..., n-1\},$$

where $k, s = 0, ..., \sqrt{p} - 1$. Then,

$$\Omega_h = \bigcup_{k,s=0}^{\sqrt{p}-1} \Omega_h^{k,s}.$$

Cell Partitioning:



For the evaluation of stencil operators, data of points on neighbor processors are needed.
These are the data at ghost points:

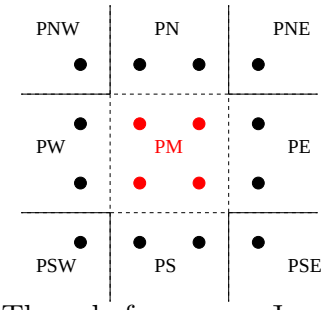
$$\hat{\Omega}_h^{k,s} \backslash \Omega_h^{k,s}$$

where

$$\hat{\Omega}_h^{k,s} = \{((kn+i)h, (sn+j)h) \mid i,j = -1, ..., n\} \cap \Omega_h$$

for $k, s = 0, ..., \sqrt{p} - 1$.

Update of Data for Jacobi Iteration:



In a Jacobi iteration, data have to be sent and received from neighbor processors. Let $N, S, NW, ...$ be the indices of the neighbor processor with index $M = (k, s)$.

Then, before every Jacobi iteration the data at points $\hat{\Omega}_h^M \cap \Omega_h^P$ have to be sent from processor P to processor M.

Let us denote this procedure `Send(P);`

Example Code:
Implement first `MPI_Irecv` then `MPI_Isend`!

```
num_message = 0;
if(rank_source != -1 && number_receive>0) {
   MPI_Irecv(receive_info ,number_receive,
             MPI_DOUBLE,rank_source,26,comm,
             &req[num_message]);
   ++num_message;
}
if(rank_destination != -1 && number_send>0) {
   MPI_Isend(send_info,number_send,
             MPI_DOUBLE,rank_destination,26,comm,
             &req[num_message]);
   ++num_message;
}
MPI_Waitall(num_message,req,status);
```

Two Sending Approaches:

- 1. Approach

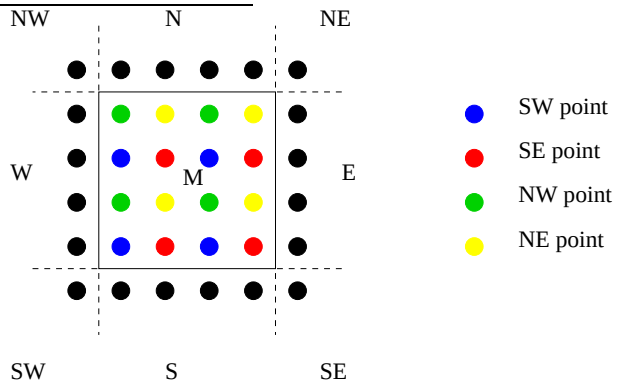
```
Send(E);  Send(W);  Send(N);  Send(S);
Send(NE); Send(NW); Send(SE); Send(SW);
Waitall();
```

- 2. Approach

```
Send(E);   Send(W);
Waitall();
Send_(N);  Send_(S);
Waitall();
```

  This approach updates data also from `NE,NW,...`, if `Send_` also sends the updated data from processor `E,W`.

Sending for 4 Color Gauss-Seidel:



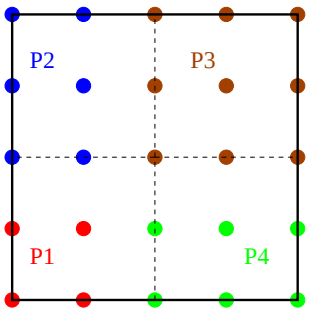
Assume that data at the ghostpoints of `W,S` are updated. Then,

```
Relax(SW);
Send_(E);        Waitall();
Relax(SE);
Send_(N);        Waitall();
Relax(NE);
Send_(W);        Waitall();
Relax(NW);
Send_(S);        Waitall();
```

This is the minimal communication needed Gauss-Seidel relaxation on a structured grid.

Point Partitioning:

point approach



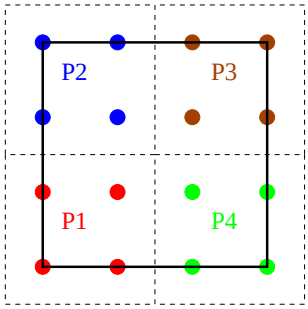
$$\Omega_h = \{(ih, jh) \mid i, j = 0, ..., N\},$$

where $h = \frac{1}{N}$ and $H = \frac{1}{\sqrt{p}}$
$N = \sqrt{p}n$, $n, N \in \mathbb{N}$. Define

$$\begin{aligned} \bar{\Omega}^{k,s} &= [Hk, H(k+1)] \times [Hs, H(s+1)] \\ \hat{\Omega}^{k,s} &= [Hk, H(k+1)[\times[Hs, H(s+1)[\end{aligned}$$

$\Omega_h^{k,s} = \Omega_h \cap \left(\bar{\Omega}^{k,s} \setminus \bigcup_{(k',s') \neq (k,s)} \hat{\Omega}^{k',s'}\right)$. Then, $\Omega_h = \bigcup_{k,s=0}^{\sqrt{p}-1} \Omega_h^{k,s}$.

Load Balancing:



$$\Omega_h = \{(ih, jh) \mid i, j = 0, ..., N-1\},$$

where $h = \frac{1}{N}$. Let $p = p_1 p_2$.
Make a partitioning with
p_1 processors in x-direction and
p_2 processors in y-direction .

- Same load balancing for every processor.
- $D_{send} = 2\frac{N}{p_1} + 2\frac{N}{p_2} = 2N(\frac{1}{p_1} + \frac{1}{p_2})$ data to be sent.

Example:

- $p_1 = \sqrt{p}$, then $D_{send} = 4N\frac{1}{\sqrt{p}}$.
- $p_1 = p$, then $D_{send} = 2N(\frac{1}{p} + 1)$.

5.4 Automatic Parallelization with MPI and Expression Templates

Let us consider the cg iteration:

```
r = A*u - f;
d = -r;
delta = product(r,r);
for(i=1;i<=iteration && delta > eps;++i) {
  g = A*d;
  tau = delta / product(d,g);
  r = r + tau*g;
  u = u + tau * d;
  delta_prime = product(r,r);
  beta = delta_prime / delta;
  delta = delta_prime;
  d = beta*d - r;
}
```

When is an update of ghost values needed?

```
enum Update_typ { no_update, update };
class vector : public Expr<vector> {
 public:
   vector(int l) { update_var = no_update; };
  Update_typ expression_update_typ() const {
      return  update_var; };
 private:
   Update_typ update_var;
   int id;
 ... };
Update_typ DExprSum::expression_update_typ() const {
    return a_.expression_update_typ() ||
           b_.expression_update_typ() };
Update_typ DExprLaplace_FD::expression_update_typ() const {
    return update; };

class Update_handler;
```

```
template <class A>
void  vector::operator=(const Expr<A>& a) {
   if((~a).expression_update_typ()) {
     Update_handler handler_update;
     (~a).Give_update_data(handler_update);
     handler_update.Make_update();
   }
   for(int i=0;i<lenghth;++i) {
       p[i] = (~a).[i];
   }
}
```

6 Raytracing

Raytracing is used in

- Computer graphics: How does light look at an image plane?
- Simulation of light in engineering applications: How is ight absorded in a medium (example: laser crystal).

The main idea of ray tracing is that light is modeled by several rays of light.

Forward and Backward Raytracing:

- Forward Raytracing: Light propagates from a light source in several directions until either vanishes by absorpion or it impings at the image plane or leaves out of the computational domain.
- Backward Raytracing: Find the rays which imping at the image plane by back tracing rays beginning from all points of the image plane in all possible directions.

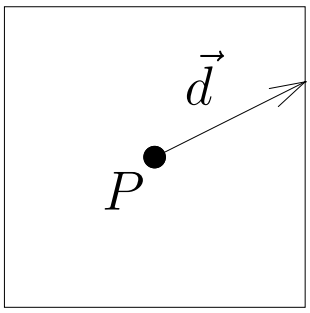
Concept of Forward Raytracing:
A ray starts at a point P and propagates in direction $\vec{d}$ with intensity I. The path of the ray can be described by

$$P + \lambda \vec{d}, \qquad \lambda \in \mathbb{R}$$

The following situations can happen:

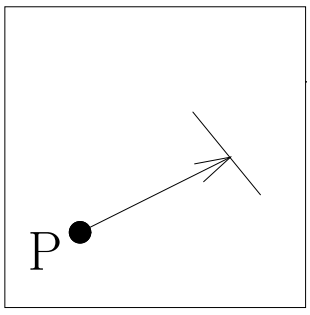
- The ray propagates out of the computational domain.
- The ray impings at an object and vanishes.
- The ray impings at an objects and is reflected in one or more directions.
- The ray progagates from a medium A to medium B with different refraction indices.
- Light of the ray is absorbed while propagating through a medium.

Ray out of the Computational Domain:



computational domain

Ray Impings on Object:

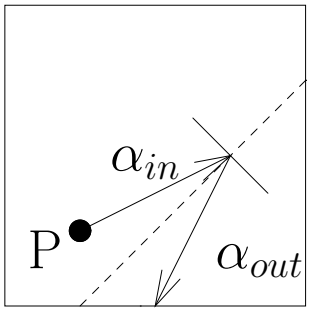


The behaviour of a ray impinging on an object, depends on the surface. To analyze the behaviour of the ray let

- I be the incident ray,
- N normal vector of the surface of the object,
- R reflected ray
- R tansmitted ray.

Obviously $N \cdot N = 1$, since N is normal vector.

Perfect Specular Reflection:



In case of perfect specular reflection, there is only one reflected ray which satisfies:

$$\alpha_{in} = \alpha_{out}.$$

The refected ray R has to be contained in the plane spanned by N and I as follows:

$$R = I + \beta N.$$

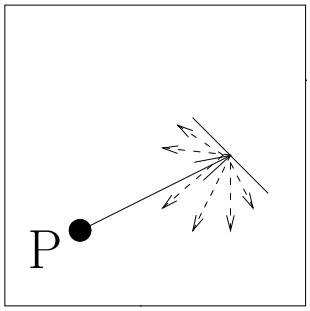
Then, we get

$$\begin{aligned} \cos(\alpha_{in}) &= \cos(\alpha_{out}) \\ -I \cdot N &= N \cdot R \\ &= N \cdot (I + \beta N) \\ &= (N \cdot I) + \beta. \end{aligned}$$

This implies:

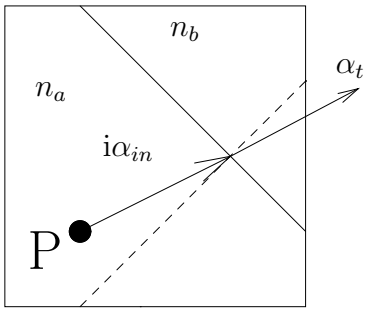
$$R = I - 2(N \cdot I)N.$$

Perfect Diffuse Reflection:
The Lambert refection describes a diffusive reflection of light by several rays:



The intensity of light is the same independent of the angle of the reflected ray. To reduce computational amount one should discretize Lambert refection by a stochastic reflection. This means that the angles of the reflected rays are choosen randomly.

Perfect Specular Transmission:



Perfect specular transmission satisfies Snell's law:

$$\alpha_{in} \cdot n_A = \alpha_t \cdot n_B.$$

To calculate the direction of the transmitted ray, let us choose

$$\|I\| = \|T\| = 1.$$

Then, the refected ray R has to be contained in the plane spanned by N and I as follows:

$$R = \alpha I + \beta N.$$

Furthermore, we use the abbreviations:

$$S_i \;=\; \sin(\alpha_{in}),$$

$$
\begin{aligned}
S_t &= \sin(\alpha_t), \\
C_i &= \cos(\alpha_{in}) \;=\; N \cdot (-I), \\
C_t &= \cos(\alpha_t) \;=\; (-N) \cdot T, \\
\eta_{it} &= \frac{S_t}{S_i}.
\end{aligned}
$$

Since $S^2 + C^2 = 1$, we get

$$
\begin{aligned}
(1 - C_i^2)\eta_{it}^2 &= S_i^2 eta_{it}^2 \\
&= S_t^2 = (1 - C_t^2).
\end{aligned}
$$

This implies

$$
\begin{aligned}
1 - (1 - C_i^2)\eta_{it}^2 &= C_t^2 \\
&= ((-N) \cdot T)^2 \\
&= ((-N) \cdot (\alpha I + \beta N))^2 \\
&= (\alpha C_i - \beta)^2.
\end{aligned}
$$

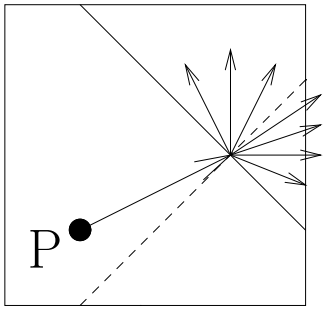
Furthermore, we get

$$
\begin{aligned}
1 &= T \cdot T \\
&= (\alpha I + \beta N) \cdot (\alpha I + \beta N) \\
&= \alpha^2 - 2\alpha\beta C_i + \beta^2.
\end{aligned}
$$

These two equations have 4 solutions for α and β. Comparing these solutions, one can show that only the following solution is physically correct:

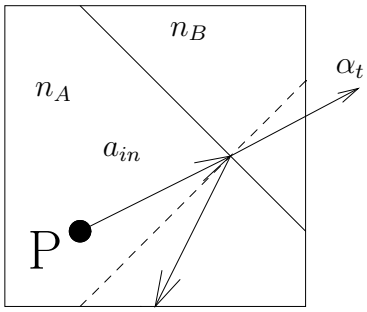
$$
\begin{aligned}
\alpha &= \eta_{it} \\
\beta &= \eta_{it} C_i - \sqrt{1 + \eta_{it}^2 (C_i^2 - 1)} \\
&= \eta_{it} \cos(\alpha_{in}) - \sqrt{1 - \eta_{it}^2 \sin^2(\alpha_{in})} \\
R &= \alpha I + \beta N.
\end{aligned}
$$

Perfect Diffusive Transmission:



In case of perfect diffusive transmission, light is transmitted in all transmitted direction with the same power. One should apply stochastic ray tracing to simulate perfect diffusive transmission.

General Situation:

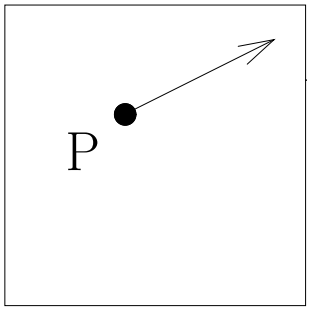


In reality light is often partly reflected and partly transmitted. Furthermore, reflected and transmitted light contains a ceratin portion of specular and diffusive light.

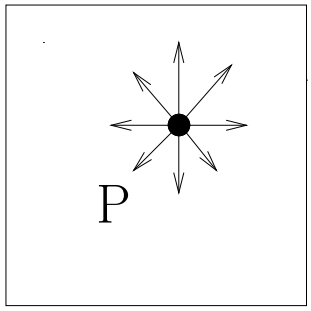
Light Sources:
There exist different kind of light sources:

- point light source
- multimode light source
- Gaussian beam light of low oder (not multimode).
  $\rightarrow$ This kind of light cannot be modeled by ray tracing.

Point Light Sources:

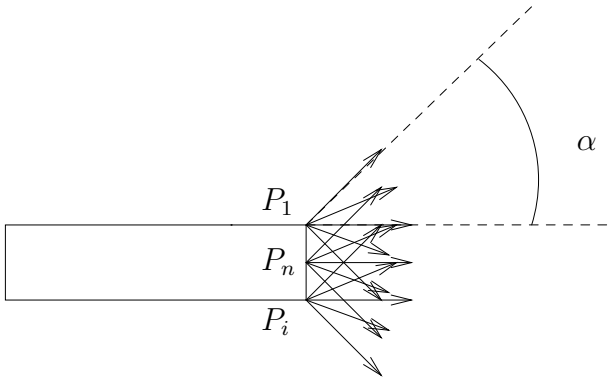


one direction



several directions

Multimode Light Sources:
Set of rays starting at points P_i, $i = 1, ..., n$ to every direction with angle ϕ between $-\alpha$ and α:



The numerical aperture NA is defined by:
$\text{NA} = n_r \cdot \sin(\alpha)$, where n_r refraction index of the medium.
Example: Light of multimode fiber.

Discretization of Light Source:
Assume that a light source consists of an infinite number of rays starting at points $P_i \in \Omega_{source}$ in directions $\vec{d_i} \in \Phi_{P_i}$.

Assume that the intensity of the light source is constant close to the light source.

To discretize the light source, we approximate the light source by a finite number of rays:

N rays starting at P_i in direction $\vec{d}_i$,
where $i = 1, ..., N$.

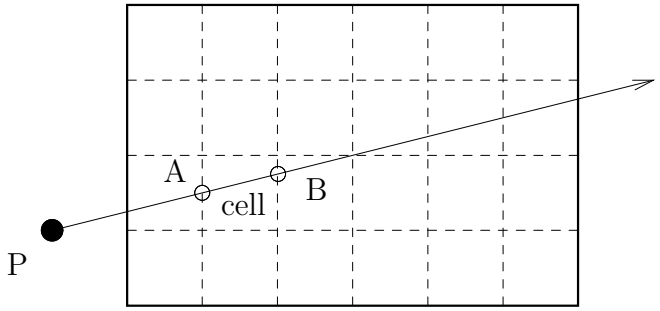
If the total power of the light source is I, then the power of each discretized ray is I/N.

Often, the starting points P_i and the directions $\vec{d}_i$ can be chosen by random numbers.

Random Numbers:
Assume that $P_i \in \Omega_{source} \subset [a_x, b_x] \times [a_y, b_y]$ and
$\vec{d}_i \in \Phi_{P_i} = [a_\phi, b_\phi]$.

Then, random values for P_i and the directions $\vec{d}_i$ can be constructed by a random number generator for an interval $[a, b]$.

Absorption of Light:
Assume that light propagates through absorbing medium.



Discretize absorbing medium by cells of meshsize h. The power of light absorbed in a cell is:

$$P_{abs}(cell) = P(A)(1 - \exp(-\alpha \overline{AB})).$$

7 Finite Differences

7.1 Stability Analysis

7.1.1 Discretization of Stiff ODE's

Let us assume that the ODE

$$\begin{aligned} y'(t) &= f(t, y(t)), \quad t \geq t_0 \\ y(t_0) &= y_0 \end{aligned}$$

is given, where $y : [t_0, \infty[\to \mathbb{R}^n$.

To discretize this ODE, let $\tau > 0$ be a time step.

Let us denote y_i the approximation of $y(t_i)$, where $t_i := \tau i + t_0$.

Types of solvers:

- simplest method: Euler method
- Runge Kutta methods (one step method)
- multi-step methods
- implicit, explicit methods

Examples:

- explicit Euler: $y_{i+1} = y_i + \tau f(t_i, y_i)$.
  $p = 1$. Explicit one step method.
- implicit Euler: $y_{i+1} = y_i + \tau f(t_{i+1}, y_{i+1})$.
  $p = 1$. Implicit one step method.
- classical Runge Kutta method

$$\begin{aligned} k_1 &= \tau f(x_i, y_i) \\ k_2 &= \tau f(x_i + 1/2\tau, y_i + 1/2k_1) \\ k_3 &= \tau f(x_i + 1/2\tau, y_i + 1/2k_2) \\ k_4 &= \tau f(x_i + \tau, y_i + k_3) \\ y_{i+1} &= y_i + 1/6k_1 + 1/3k_2 + 1/3k_3 + 1/6k_4. \end{aligned}$$

  $p = 4$. Explicit one step method.

- Simpson's method:

$$y_{i+1} - y_{i-1} = \frac{\tau}{3}(f(t_{i+1}, y_{i+1}) + 4f(t_i, y_i) + f(t_{i-1}, y_{i-1})).$$

  $p = 4$. Implicit multi-step method.

- Middle point method: $y_{i+1} - y_{i-1} = 2\tau f(t_i, y_i)$.
  $p = 2$. Explicit multi-step method.

Stability of a Multi-Step Method:
To analyze the stability of a multi-step method of length s, consider the ODE

$$y' = 0, \quad y(0) = y_0.$$

Assume that the multi-step method leads to the recursion formula

$$\sum_{i=0}^{s} a_i y_{i+j} = 0 \quad \forall j \in \mathbb{N}_0.$$

for this ODE.

Definition 3. *The multi-step method is stable, if for all start values $y_0, ...y_{s-1}$, the sequence y_i is bounded.*

Theorem 3. *A multi-step method is stable if all roots of the polynomial*

$$\sum_{i=0}^{s} a_i z^i$$

are simple roots and contained in the disc

$$\{z \in \mathbb{C} \mid |z| \leq 1\}.$$

(A more general stability theorem is given in Stoer/Burlisch, Einführung in die Numerische Mathematik II).

To determine whether an ODE is an stiff ODE, one has to linearize the ODE.

Let us linearize $f(y, t')$ at a certain point $\hat{t}, \hat{y}$ by Taylor series in y direction:

$$f(\hat{y}, \hat{t}) \approx b + A(y - \hat{y}).$$

The ODE is a stiff ODE, if A has negative eigenvalues of different size.

Definition 4. *The ODE solver is a stable ODE solver for stiff equation systems, if*

$$\lim_{i\to\infty} y_i \;=\; 0 \quad \forall\tau > 0 \quad \textit{and} \quad y_i > 0 \quad \forall\tau > 0,\ i \in \mathbb{N}$$

for the ODE

$$y' = \lambda y, \qquad y(0) = 1,$$

where $\lambda < 0$.

Example 5. *y_1 concentration of O_3 in atmosphere and y_2 concentration of O in atmosphere.*

$$\begin{aligned} \frac{\partial y_1}{\partial t} &= -y_1 - y_1 y_2^2 + 294 y_2 \\ \frac{\partial y_2}{\partial t} &= (y_1 - y_1 y_2)/98 - 3y_2 \end{aligned}$$

where $\hat{y}_1(0) = 0$ and $\hat{y}_2(0) = 0$. The linearization of this ODE at $(1,1)$ leads to

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} -1 & 294 \\ -1/98 & -3 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}.$$

The eigenvalues are 0 and -2.

Example 6. *Consider the following rate equation in laser simulation:*

$$\begin{aligned} \frac{\partial N}{\partial t} &= -10^{-15} N n - 10^5 N + C \\ \frac{\partial n}{\partial t} &= 7 \cdot 10^{-16} N n - 1.7 \cdot 10^7 n + S \end{aligned}$$

The discretization of this equation leads to numerical difficulties.

Apply a given ODE solver to the ODE

$$y' = \lambda y, \qquad y(0) = 1.$$

Often this leads to an iteration formula of the form

$$y_{i+1} = y_i g(\lambda\tau).$$

Then, stability means

$$|g(z)| < 1 \qquad \forall Re(z) < 0.$$

- explicit Euler: (not stable for stiff ODE's)

$$y_{i+1} = y_i + hf(t_i, y_i)$$

- implicit Euler: (stable for stiff ODE's)

$$y_{i+1} = y_i + hf(t_{i+1}, y_{i+1}).$$

7.1.2 Discretization of Parabolic PDE's

Let $\Omega \subset \mathbb{R}^d$ be a domain.

The standard parabolic PDE is:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \alpha^2 \triangle u + f(t, \vec{x}), \quad \vec{x} \in \Omega, t \geq t_0, \\ u(t_0, \vec{x}) &= u_0(\vec{x}), \quad \vec{x} \in \Omega, \qquad \text{initial condition} \\ u(t, \vec{x}) &= g(t, \vec{x}), \quad \vec{x} \in \partial\Omega, t \geq t_0, \qquad \text{boundary condition} \end{aligned}$$

where, g, f, u_0 are given functions.

Example 7. *$\Omega =]0, \pi[^2$, $f = 0$, $g = 0$, and $u_0(x, y) = \sin(x)\sin(y)$.*

Then, the exact solution is

$$u(t, x, y) = e^{-\alpha^2 t} \sin(x) \sin(y).$$

- Let $\bar{\Omega}_h \subset \bar{\Omega}$ be a discretization grid.

  Let $\Omega_h = \bar{\Omega}_h \cap \Omega$.

- $t_i := \tau i + t_0$.

- Let us denote $\bar{u}_h(t_i, \vec{x}_h)$, $i \in \mathbb{N}_0$, $\vec{x}_h \in \Omega_h$ the approximate solution.

  Furthermore, let us abbreviate $u_h(t_i) = (\bar{u}_h(t_i, \vec{x}_h))_{\vec{x}_h \in \Omega_h}$.

- Let us discretize $\triangle w$ by

$$\bar{L}_h w_h,$$

  where $\bar{L}_h$ is a $|\Omega_h| \times |\bar{\Omega}_h|$ matrix and $w_h \in \mathbb{R}^{|\Omega_h|}$.

  (e.g. finite difference discretization).

- In case of homogeneous boundary conditions ($g = 0$)
  $\bar{L}_h$ can be replaced by the $|\Omega_h| \times |\Omega_h|$ matrix L_h.

$$\begin{aligned} u_h(t_0, \vec{x}_h) &= u_0(\vec{x}_h) \qquad \vec{x}_h \in \bar{\Omega}_h \\ u_h(t_i, \vec{x}_h) &= g(t_i, \vec{x}_h) \qquad \vec{x}_h \in \bar{\Omega}_h \backslash \Omega_h, \quad i \in \mathbb{N}_0. \end{aligned}$$

- forward difference method (explicit Euler)

$$\bar{u}_h(t_{i+1}) = \bar{u}_h(t_i) + \tau(\alpha^2 \bar{L}_h \bar{u}_h(t_i) + f_h(t_i)).$$

- backward difference method (implicit Euler)

$$\bar{u}_h(t_{i+1}) = \bar{u}_h(t_i) + \tau(\alpha^2 \bar{L}_h \bar{u}_h(t_{i+1}) + f_h(t_{i+1})).$$

- Crank-Nicolson

$$\begin{aligned} \bar{u}_h(t_{i+1}) = \bar{u}_h(t_i) + \tau \frac{1}{2} \quad (\quad & \alpha^2 \bar{L}_h \bar{u}_h(t_i) + f_h(t_i) + \\ & \alpha^2 \bar{L}_h \bar{u}_h(t_{i+1}) + f_h(t_{i+1}) \ ). \end{aligned}$$

To analyze the stability of the previous discretization, let us consider

$$\begin{aligned} \frac{\partial u}{\partial t} &= \alpha^2 \triangle u, \quad \vec{x} \in \Omega, t \geq t_0, \\ u(t_0, \vec{x}) &= u_0(\vec{x}), \quad \vec{x} \in \Omega, && \text{initial condition} \\ u(t, \vec{x}) &= 0, \quad \vec{x} \in \partial\Omega, t \geq t_0, && \text{boundary condition} \\ \Omega &:=]0, \pi[^2. \end{aligned}$$

The exact solution of this PDE is

$$\begin{aligned} u(t, (x, y)) &= \sum_{\nu,\mu=0}^{\infty} a_{\nu,\mu} \ \sin(\nu x) \sin(\mu y) \ e^{-\alpha^2(\nu^2+\mu^2)(t-t_0)}, \quad \text{where} \\ u_0(x, y) &= \sum_{\nu,\mu=0}^{\infty} a_{\nu,\mu} \ \sin(\nu x) \sin(\mu y). \end{aligned}$$

Observe that $u(t, (x, y)) \geq 0$, if $a_{\nu,\mu} \geq 0 \forall \nu, \mu$.

Let $\Omega_h := \{h(i, j) \mid i, j = 1, m - 1\}$ be the discretization grid.

Lemma 1. *L_h has the eigenvectors*

$$e_{\nu,\mu} = \Big(\sin(\nu\pi x_i)\sin(\mu\pi y_j)\Big)_{(x_i,y_j)\in\Omega_h}, \quad \textit{where} \quad \nu,\mu = 1,\cdots,m-1,$$

with eigenvalues

$$\lambda_{\nu,\mu} = -\frac{4}{h^2}\left(\sin^2\left(\frac{\pi\nu h}{2}\right) + \sin^2\left(\frac{\pi\mu h}{2}\right)\right).$$

The eigenvalues can be estimated by

$$\frac{8}{h^2} > -\lambda_{\nu,\mu} > 2\pi^2.$$

The functions $(e_{\nu,\mu})_{\nu,\mu=1,\ldots,m-1}$ form a basis of $\mathbb{R}^{|\Omega_h|}$. Thus we can write

$$u_0 = \sum_{\nu,\mu=1}^{m-1} c_{\nu,\mu}(t_0)e_{\nu,\mu}.$$

Definition 5. *The discretization of the parabolic equation is stable, if the following condition holds:*

Let the coefficients $c_{\nu,\mu}(t_0)$ be nonnegative.

Then, the coefficients of the approximate solution for $f = 0$, $g = 0$ are nonnegative

$$c_{\nu,\mu}(t) \geq 0 \quad \forall\nu,\mu,\ t > t_0.$$

Analysis of Forward Difference Method:

- The Fourier analysis of the forward difference method

$$u_h(t_{i+1}) = u_h(t_i) + \tau(\alpha^2 L_h u_h(t_i) + f_h(t_i)).$$

leads to the explicit Euler formula ($f = 0$, $g = 0$):

$$c_{\nu,\mu}(t_{i+1}) = (1 + \tau\alpha^2\lambda_{\nu,\mu})c_{\nu,\mu}(t_i).$$

- Stability is obtained if $|1 + \tau\alpha^2\lambda_{\nu,\mu}| < 1$ and therefore

$$\tau < \frac{2}{\alpha^2|\lambda_{\nu,\mu}|}.$$

Thus the condition

$$\tau < \frac{2}{\alpha^2 \frac{8}{h^2}} = \frac{2h^2}{8\alpha^2}$$

is sufficient for the stability of the forward difference method.

Analysis of Backward Difference Method:

- The analysis of the backward difference method

$$u_h(t_{i+1}) = u_h(t_i) + \tau(\alpha^2 L_h u_h(t_{i+1}) + f_h(t_{i+})).$$

leads to the implicit Euler formula ($f = 0$, $g = 0$):

$$c_{\nu,\mu}(t_{i+1}) = c_{\nu,\mu}(t_i) + \tau\alpha^2\lambda_{\nu,\mu}c_{\nu,\mu}(t_{i+1}).$$

This implies:

$$c_{\nu,\mu}(t_{i+1}) = c_{\nu,\mu}(t_i)\frac{1}{1 - \tau\alpha^2\lambda_{\nu,\mu}}.$$

- Stability is obtained independent of τ since

$$0 < \frac{1}{1 - \tau\alpha^2\lambda_{\nu,\mu}} < 1.$$

Analysis of Crank-Nicolson :

- The analysis of Crank-Nicolson

$$u_h(t_{i+1}) = u_h(t_i) + \tau\frac{1}{2} \quad (\quad \alpha^2\bar{L}_h u_h(t_i) + f_h(t_i) + \alpha^2\bar{L}_h u_h(t_{i+1}) + f_h(t_{i+1}) \ )$$

leads to the formula ($f = 0$, $g = 0$):

$$c_{\nu,\mu}(t_{i+1}) = c_{\nu,\mu}(t_i) + \tau\frac{1}{2}\alpha^2\lambda_{\nu,\mu}(c_{\nu,\mu}(t_i) + c_{\nu,\mu}(t_{i+1})).$$

This implies:

$$c_{\nu,\mu}(t_{i+1}) = c_{\nu,\mu}(t_i)\frac{1 + \frac{1}{2}\tau\alpha^2\lambda_{\nu,\mu}}{1 - \frac{1}{2}\tau\alpha^2\lambda_{\nu,\mu}}.$$

- Stability is obtained independent of τ since

$$\left|\frac{1 + \frac{1}{2}\tau\alpha^2\lambda_{\nu,\mu}}{1 - \frac{1}{2}\tau\alpha^2\lambda_{\nu,\mu}}\right| < 1.$$

But for large $|\alpha^2\lambda_{\nu,\mu}|$: $\left|\frac{1+\frac{1}{2}\tau\alpha^2\lambda_{\nu,\mu}}{1-\frac{1}{2}\tau\alpha^2\lambda_{\nu,\mu}}\right| \rightarrow 1$.

7.1.3 Discretization of Hyperbolic PDE's

Let $\Omega \subset \mathbb{R}^d$ be a domain.

The standard hyperbolic PDE is:

$$\begin{aligned}
\frac{\partial^2 u}{\partial t^2} &= \alpha^2 \triangle u + f(t,\vec{x}), \quad \vec{x} \in \Omega, t \geq t_0, \\
u(t_0,\vec{x}) &= u_0(\vec{x}), \quad \vec{x} \in \Omega, && \text{1. initial condition} \\
\frac{\partial u}{\partial t}(t_0,\vec{x}) &= u_1(\vec{x}), \quad \vec{x} \in \Omega, && \text{2. initial condition} \\
u(t,\vec{x}) &= g(t,\vec{x}), \quad \vec{x} \in \partial\Omega, t \geq t_0, && \text{boundary condition}
\end{aligned}$$

where, g, f, u_0, u_1 are given functions.

- Let $\bar{\Omega}_h \subset \bar{\Omega}$ be a discretization grid.
  Let $\bar{\Omega}_h = \Omega_h \cap \Omega$.
- $t_i := \tau i + t_0$.
- Let us denote $\bar{u}_h(t_i, \vec{x}_h)$, $i \in \mathbb{N}_0$, $\vec{x}_h \in \Omega_h$
  the approximate solution.
  Furthermore, let us abbreviate $u_h(t_i) = (\bar{u}_h(t_i, \vec{x}_h))_{\vec{x}_h \in \Omega_h}$

- Let us discretize $\triangle w$ by
$$\bar{L}_h w_h,$$
where $\bar{L}_h$ is a $|\Omega_h| \times |\Omega_h|$ matrix and $w_h \in \mathbb{R}^{|\Omega_h|}$.
(e.g. finite difference discretization).

- In case of homogeneous boundary conditions ($g = 0$)
$\bar{L}_h$ can be replaced by the $|\Omega_h| \times |\bar{\Omega}_h|$ matrix L_h.

First initial condition and boundary condition:

$$\begin{aligned} u_h(t_0, \vec{x}_h) &= u_0(\vec{x}_h) \qquad \vec{x}_h \in \bar{\Omega}_h \\ u_h(t_i, \vec{x}_h) &= g(t_i, \vec{x}_h) \qquad \vec{x}_h \in \bar{\Omega}_h \backslash \Omega_h, \quad i \in \mathbb{N}_0. \end{aligned}$$

Second initial condition:

$$u_h(t_1, \vec{x}_h) = u_0(\vec{x}_h) + \tau u_1(\vec{x}_h) + \frac{1}{2}\tau^2 \alpha^2 \triangle u_0(\vec{x}_h).$$

This initial condition can be derived by the following Taylor series:

$$u(t_0 + \tau, \vec{x}) = u(t_0, \vec{x}) + \tau \frac{\partial u(t_0, \vec{x})}{\partial t} + \frac{1}{2}\tau^2 \frac{\partial^2 u(t_0, \vec{x})}{\partial t^2} + O(\tau^3).$$

Discretization of the PDE:

$$u_h(t_{i+1}) = 2u_h(t_i) - u_h(t_{i-1}) + \tau^2 \alpha^2 \left(\bar{L}_h u_h(t_i) + f_h(t_i) \right).$$

To analyze the stability of the previous discretizations, let us consider the case $f = 0$, $g = 0$, $u_1 = 0$. Then, the exact solution of this PDE is

$$\begin{aligned} u(t, (x, y)) &= \sum_{\nu,\mu=0}^{\infty} a_{\nu,\mu} \ \sin(\nu x) \sin(\mu y) \ \cos\left(\alpha (t - t_0) \sqrt{\nu^2 + \mu^2} \right), \quad \text{where} \\ u_0(x, y) &= \sum_{\nu,\mu=0}^{\infty} a_{\nu,\mu} \ \sin(\nu x) \sin(\mu y). \end{aligned}$$

Observe that

$$a_{\nu,\mu} \ \sin(\nu x) \sin(\mu y) \ \cos\left(\alpha (t - t_0) \sqrt{\nu^2 + \mu^2} \right)$$

is bounded for $t \to \infty$.

The functions $(e_{\nu,\mu})_{\nu,\mu=1,\ldots,m-1}$ form a basis of $\mathbb{R}^{|\Omega_h|}$. Thus, we can write

$$u_0 = \sum_{\nu,\mu=1}^{m-1} c_{\nu,\mu}(t_0) e_{\nu,\mu}.$$

Definition 6. *The discretization of the hyperbolic equation is stable, if the following condition holds:*

Assume that

$$c_{\nu,\mu} = \begin{cases} c \neq 0 & \text{for } (\nu,\mu) = (\nu',\mu') \\ 0 & \text{for } (\nu,\mu) \neq (\nu',\mu') \end{cases}.$$

Then, the approximate solution for $f = 0$, $g = 0$, $u_1 = 0$ is bounded for $t \to \infty$.

Analysis of the Discretization:

- The Fourier analysis of the discretization

  $$u_h(t_{i+1}) = 2u_h(t_i) - u_h(t_{i-1}) + \tau^2\alpha^2 \left(\bar{L}_h u_h(t_i) + f_h(t_i)\right)$$

  leads to the formula ($f = 0$, $g = 0$, $u_1 = 0$):

  $$c_{\nu,\mu}(t_{i+1}) = (2 + \tau^2\alpha^2\lambda_{\nu,\mu}) c_{\nu,\mu}(t_i) - c_{\nu,\mu}(t_{i-1}).$$

- Stability is obtained, if the roots of

  $$z^2 - (2 + \tau^2\alpha^2\lambda_{\nu,\mu})z + 1$$

  are simple and contained in the disc $\{z \in \mathbb{C} \mid |z| \leq 1\}$. To analyze stability, let $b = 2 + \tau^2\alpha^2\lambda_{\nu,\mu}$. Then,

  $$\begin{aligned} z^2 - bz + 1 &= 0 \\ &\Downarrow \\ z &= \frac{b \pm \sqrt{b^2 - 4}}{2} \end{aligned}$$

Recall that (see Lemma 1)

$$\frac{8}{h^2} > -\lambda_{\nu,\mu} > 2\pi^2.$$

There are 2 cases: $b^2 > 4$ and $b^2 \leq 4$. If $b^2 > 4$, then, there is a root z which is not contained in the unit disc. Thus, we consider only the case $b^2 \leq 4$. This implies

$$\begin{array}{rcl} b^2 & \leq & 4 \\ & \Updownarrow & \\ 2 + \tau^2\alpha^2\lambda_{\nu,\mu} & \geq & -2 \\ & \Updownarrow & \\ 4 & \geq & -\tau^2\alpha^2\lambda_{\nu,\mu} \\ & \Uparrow & \\ 4 & \geq & \tau^2\alpha^2\dfrac{8}{h^2} \\ & \Uparrow & \\ \tau & \leq & \dfrac{h}{\alpha\sqrt{2}}. \end{array}$$

Since $b^2 \leq 4$, we obtain

$$|z|^2 = \left|\frac{b \pm i\sqrt{4-b^2}}{2}\right|^2 = \frac{b^2 + (4-b^2)}{4} = 1.$$

Thus, we get CFL (Courant, Friedrich, Lewy) condition (1928):

$$\tau < \frac{1}{\sqrt{2}} h|\alpha|^{-1}.$$

7.2 Order of Consistency

Order of Consistency and Convergence:

- There are slightly different definitions of consistency for different types of ODE solvers and types of PDE's.

- There are different definitions for stability.
- In numerical analysis one proves:
  consistency + stability $\Rightarrow$ *convergence*

Order of Consistency for Elliptic PDE's:

Definition 7. *Let $L(u)$ be a differential operator on Ω and $L_h(u_h)$ a discrete approximation of this operator on the discretization grid Ω_h. Furthermore, let $R_h : \mathcal{C}(\bar{\Omega}) \to \mathbb{R}^{|\Omega_h|}$ be the pointwise restriction operator. Then, the consistency order of L_h is of order $O(h^p)$, if there exists a constant $C > 0$ such that*

$$\|R_h(L(u)) - L_h(R_h(u))\| \ \leq \ Ch^p.$$

Example 8. *Consider the differential operator $\frac{\partial}{\partial x}$.*
The consistency order of central differences is $O(h^2)$
and the consistency order of upwind or downwind differences is $O(h)$.

Order of Consistency for ODE's:

Definition 8. *Let $y' = f(t, y)$ be an ODE on the domain $[t_0, \infty[$.*

Let $y_i \to \Psi(y_i) = y_{i+1}$ be a mapping, which calculates an approximate solution y_{i+1} at $t_{i+1} = t_i + \tau$ for a given approximation y_i at t_i.

Then, the consistency error is of order $O(\tau^p)$, if there exists a constant $C > 0$ such that

$$\left|\tau^{-1}(y^{ex}(t_{i+1}) - y_{i+1})\right| \ \leq \ C\tau^p,$$

where y^{ex} is an exact solution of the ODE with initial condition $y^{ex}(t_i) = y_i$.

Example 9. *Explicit Euler formula:*

$$y_{i+1} = y_i + \tau f(x_i, y_i).$$

Then,

$$\begin{aligned}
\frac{y(t_i + \tau) - y_{i+1}}{\tau} &= \tau^{-1}(y(t_i + \tau) - y_i - \tau f(t_i, y_i)) \\
&= \tau^{-1}(\tau y'(t_i) + \frac{1}{2}\tau^2 y''(\xi) - \tau f(t_i, y_i)) \\
&= \frac{1}{2}\tau y''(\xi).
\end{aligned}$$

This shows a consistency of order $O(\tau)$.

Consistency for Parabolic PDE:

Definition 9. *Let $y' = f(t, y)$ be a parabolic PDE on the domain $[t_0, \infty[$.*

Let $y_i \to \Psi(y_i) = y_{i+1}$ be a mapping, which calculates an approximate solution y_{i+1} at $t_{i+1} = t_i + \tau$ for a given approximation y_i at t_i.

Then, the consistency error is of order $O(\tau^p)$, if there exists a constant $C > 0$ such that

$$\left\|\tau^{-1}(y^{ex}(t_{i+1}) - y_{i+1})\right\| \ \leq \ C\tau^p,$$

where y^{ex} is an exact solution of the parabolic PDE with initial condition $y^{ex}(t_i) = y_i$.

7.3 Shortly-Weller Discretization for Curvilinear Bounded Domains

Shortly-Weller Discretization:
The following Shortly-Weller discretization is used to discretize elliptic equations on curvilinear bounded domains:

- Let $\Omega \subset]a_x, b_x[\times]a_y, b_y[= Q$ be an open bounded domain.
- Discretize Q by a structured grid Q_h of meshsize h.
- Denote $\Omega_h := Q_h \cap \Omega$ the interior points.
- The set of regular points is:

  $$\Omega_h^r := \{z \in \Omega_h \mid z + (h, 0), z + (-h, 0), z + (0, h), z + (0, -h) \in \Omega_h\}.$$

  and the set of near boundary points: $\Omega_h^n := \Omega_h \backslash \Omega_h^r$.

- Let the set of boundary points Γ_h be the set

  $$\begin{aligned} & \{(x, y+\tau) \in \partial\Omega \mid (x, y) \in \Omega_h^n, \ \ (x, y+h) \notin \Omega_h, \ \ ](x, y), (x, y+\tau)[\subset \Omega\} \\ \cup \ & \{(x, y-\tau) \in \partial\Omega \mid (x, y) \in \Omega_h^n, \ \ (x, y-h) \notin \Omega_h, \ \ ](x, y), (x, y-\tau)[\subset \Omega\} \\ \cup \ & \{(x+\tau, y) \in \partial\Omega \mid ...\} \cup \{(x-\tau, y) \in \partial\Omega \mid ...\}. \end{aligned}$$

- For every point $M = (x, y) \in \Omega_h^i$ denote the north point by

$$N := \begin{cases} (x, y+\tau) & \text{if } (x, y+h) \notin \Omega_h \\ (x, y+h) & \text{if } (x, y+h) \in \Omega_h. \end{cases}$$

  Analogously, define the points N, S, W.

- Let the mesh sizes h_N, h_S, h_W, h_E be defined such that

$$\begin{aligned} N &= (x, y+h_N), \quad \text{where } M = (x, y), \\ S &= (x, y-h_S), \quad \text{where } M = (x, y), \\ E &= (x+h_E, y), \quad \text{where } M = (x, y), \\ W &= (x-h_W, y), \quad \text{where } M = (x, y). \end{aligned}$$

Let us discretize the equation

$$-\triangle u = f, \quad u|_{\partial\Omega} = g$$

as follows

- $u(z) = g(z)$ for all $z \in \Gamma_h$.

- For every $z \in \Omega_h^i$ let

$$\begin{aligned} -\triangle_h u_h(M) &= \left(\frac{2}{h_N h_S} + \frac{2}{h_W h_E}\right) u(M) \\ &\quad - \frac{2}{h_N(h_N+h_S)} u(N) - \frac{2}{h_S(h_N+h_S)} u(S) \\ &\quad - \frac{2}{h_W(h_W+h_E)} u(W) - \frac{2}{h_E(h_W+h_E)} u(E). \end{aligned}$$

Observe that for points $M, N, S, W, E \in \Omega_h$, we obtain

$$-\triangle_h u_h(M) = \frac{4u_M - u_E - u_W - u_N - u_S}{h^2},$$

where $h = h_E = h_W = h_N = h_S$.

Theorem 4. • *In general, the discretization matrix of the Shortly-Weller discretization is not symmetric.*

- *The order of consistency is:*

$$\begin{aligned} \|(R_h(L(u)) - L_h(R_h(u)))(M)\| &= O(h) \quad \forall M \in \Omega_h^n \\ \|(R_h(L(u)) - L_h(R_h(u)))(M)\| &= O(h^2) \quad \forall M \in \Omega_h^i. \end{aligned}$$

- *If $u \in \mathcal{C}^4(\bar{\Omega})$, then the convergence is of order $O(h^2)$:*

$$\|R_h(u) - u_h\|_\infty = O(h^2).$$

8 Nested Dissection

Direct Solvers for PDE's:
FD discretization of Poisson's equation $Mx = b$,
where M is a matrix of size $N = n^d$, d dimension.

	storage	time
Gauss elimination	$N^2 = n^{2d}$	$N^3 = n^{3d}$
Band Gauss elimination	$Nn^{d-1} = n^{2d-1}$	$Nn^{2(d-1)} = n^{3d-2}$
at $d = 2$	n^3	n^4
at $d = 3$	n^5	n^7
Nested dissection $d > 2$	n^{2d-2}	n^{3d-3}
at $d = 2$	$n^2 \log n$	n^3
at $d = 3$	n^4	n^6
Iterative multigrid	n^d	n^d

Block Elimination:
Let us write $Mx = b$ as

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} t \\ x_{co} \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix},$$

where

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \quad b = \begin{pmatrix} q \\ p \end{pmatrix}, \quad x = \begin{pmatrix} t \\ x_{co} \end{pmatrix}$$

Here *co* is an abbreviation for coarse.

Block Elimination:
The block decomposition leads to

$$\begin{aligned} M_{co} &:= D - CA^{-1}B \\ b_{co} &:= p - CA^{-1}q. \end{aligned}$$

One has to solve

$$\begin{aligned} M_{co}x_{co} &= b_{co} && (3)\\ t &= A^{-1}(q - Bx_{co}) && (4) \end{aligned}$$

Here *co* is an abbreviation for coarse.
Equation (3) can be solved recursively or by Gauss-Elimination.
A^{-1} has to be calculated by Gauss-Elimination.

Block Elimination:
Let

$$\{1, 2, ..., N\} = A \cup B$$

Then

$$\mathbb{R}^N = V_A \oplus V_B := \{v + w \mid v \in V_A \text{ and } w \in V_B\}$$

where

$$\begin{aligned} V_A &= \left\{ \sum_{i \in A} e_i \lambda_i \mid \lambda_i \in \mathbb{R} \right\}, \\ V_B &= \left\{ \sum_{i \in B} e_i \lambda_i \mid \lambda_i \in \mathbb{R} \right\}. \end{aligned}$$

Block Elimination:
Spaces decomposition: $V_k = W_k \oplus V_{k-1}$.
Then: $\mathbb{R}^N = W_{k_{max}} \oplus ... \oplus W_1 \oplus V_0$

$$M_k = \begin{pmatrix} A_k & B_k \\ C_k & D_k \end{pmatrix}$$

where

$$\begin{aligned} A_k : W_k \to W_k, \qquad & B_k : V_{k-1} \to W_k, \\ C_k : W_k \to V_{k-1}, \qquad & D_k : V_{k-1} \to V_{k-1}, \\ M_k : V_k \quad \to \quad & V_k \end{aligned}$$

Nested Dissection:

$$\begin{array}{ccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 3 & 2 & 3 & 1 & 3 & 2 & 3 & 0 \\
0 & 2 & 2 & 2 & 1 & 2 & 2 & 2 & 0 \\
0 & 3 & 2 & 3 & 1 & 3 & 2 & 3 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 3 & 2 & 3 & 1 & 3 & 2 & 3 & 0 \\
0 & 2 & 2 & 2 & 1 & 2 & 2 & 2 & 0 \\
0 & 3 & 2 & 3 & 1 & 3 & 2 & 3 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Decomposition of discretization grid:

$$\Omega_h = \Omega^0 \cup \Omega^1 \cup \Omega^2 \cup \Omega^3$$

Nested Dissection:
Number the finest grid $\Omega_h = \{1, 2, ..., N\}$. Then, decompose:

$$\{1, 2, ..., N\} = \Omega^0 \cup \Omega^1 \cup \Omega^2 \cup \Omega^3$$

$$M_k = \begin{pmatrix} A_k & B_k \\ C_k & D_k \end{pmatrix}, \quad x_k = \begin{pmatrix} t_k \\ x_{k-1} \end{pmatrix}, \quad b_k = \begin{pmatrix} q_k \\ p_k \end{pmatrix}.$$

$$\begin{aligned}
M_{k-1} &:= D_k - C_k A_k^{-1} B_k \\
b_{k-1} &:= p_k - C_K A_k^{-1} q_k. \\
M_{k-1} x_{k-1} &= b_{k-1} \\
t_k &= A_k^{-1}(q_k - B_k x_{k-1})
\end{aligned}$$

M_0^{-1} and A_k^{-1} have to be calculated by Gauss-Elimination.

Nested Dissection:
The computational amount of nested dissection is dominated by computation of M_0^{-1} and A_k^{-1}. Let us estimate this this computational amount:
Let $n = 2^{k_{max}}$.

$$M_0$$

is matrix of size $O(2^{(d-1)k_{max}}) = O(n^{d-1})$.
M_0^{-1} computation:

$O(n^{2d-2})$ storage requirement.
$O(n^{3d-3})$ computational requirement.

Nested Dissection:
Let $n = 2^{k_{max}}$.

$$A_k$$

has a block-structure and consists of

$$2^{d(k-1)}$$

blocks of size $O(2^{(d-1)(k_{max}-k)})$.
Storage requirement for A_k^{-1} computation:

$$\begin{aligned} O(\sum_{k=0}^{k_{max}} 2^{d(k-1)}(2^{(d-1)(k_{max}-k)})^2) &= O(N\sum_{k=0}^{k_{max}} 2^{(d-2)k}) \\ = O(Nk_{max}) &= O(n^2\log(n)) \text{ if } d=2 \\ = O(N2^{(d-2)k_{max}}) &= O(n^{2d-2}) \text{ if } d>2 \end{aligned}$$

Nested Dissection:
Let $n = 2^{k_{max}}$.

$$A_k$$

has a block-structure and consists of

$$2^{d(k-1)}$$

blocks of size $O(2^{(d-1)(k_{max}-k)})$.
Computational requirement for A_k^{-1} computation:

$$\begin{aligned} O(\sum_{k=0}^{k_{max}} 2^{d(k-1)}(2^{(d-1)(k_{max}-k)})^3) &= O(N\sum_{k=0}^{k_{max}} 2^{(2d-3)k}) \\ &= O(n^{3d-3}) \end{aligned}$$

Implementation of Nested Dissection:
Implementation has to take into account that all matrices are block matrices.

$\rightarrow$ recursive implementation is needed.

For reasons of simplicity assume $d = 2$, $\Omega = [0,1]^2$.
Define the cells (Zelle)

$$\mathcal{Z}_{i,j}^k = [ih_k, jh_k] \times [(i+1)h_k, (j+1)h_k],$$

where $h_k = 2^{-k}$ and

$$I = (i,j) \in \mathcal{I}^k := \{(i,j) \mid i,j = 0, ..., 2^k - 1\}.$$

Observe that $\mathcal{Z}_{0,0}^0 = [0,1]^2$ and

$$\mathcal{Z}_{i,j}^k = \mathcal{Z}_{i,j}^{k+1} \cup \mathcal{Z}_{i+1,j}^{k+1} \cup \mathcal{Z}_{i,j+1}^{k+1} \cup \mathcal{Z}_{i+1,j+1}^{k+1}.$$

Implementation of Nested Dissection:
Define

$$\begin{aligned}
\mathcal{A}_{i,j}^{k_{max}} &:= \mathcal{Z}_{i,j}^{k_{max}} \cap \Omega_h \\
\mathcal{B}_{i,j}^k &:= \mathcal{A}_{i,j}^k \cap \partial\mathcal{Z}_{i,j}^k \\
\mathcal{I}_{i,j}^k &:= \mathcal{A}_{i,j}^k \backslash \mathcal{B}_{i,j}^k \\
\mathcal{A}_{i,j}^{k-1} &:= \mathcal{B}_{i,j}^k \cup \mathcal{B}_{i+1,j}^k \cup \mathcal{B}_{i,j+1}^k \cup \mathcal{B}_{i+1,j+1}^k \quad \text{for } k \leq k_{max}.
\end{aligned}$$

Furthermore, we can define

$$\begin{aligned}
\Omega_0 &:= \mathcal{B}_{0,0}^0 \\
\Omega_k &:= \mathcal{I}_{i,j}^{k-1}.
\end{aligned}$$

Implementation of Nested Dissection:
Let

$$V(B) := \text{span}\{e_i \mid i \in B\}$$

Then define matrices, which map spaces to spaces:

$$\begin{aligned}
A_I^k : V(\mathcal{I}_I^k) \rightarrow V(\mathcal{I}_I^k), \qquad & B_I^k : V(\mathcal{B}_I^k) \rightarrow V(\mathcal{I}_I^k), \\
C_I^k : V(\mathcal{I}_I^k) \rightarrow V(\mathcal{B}_I^k), \qquad & D_I^k : V(\mathcal{B}_I^k) \rightarrow V(\mathcal{B}_I^k), \\
M_I^k : V(\mathcal{A}_I^k) \quad &\rightarrow \quad V(\mathcal{A}_I^k)
\end{aligned}$$

These matrices are stored with respect to the standard basis $\{e_i\}$. Extend matrix $M : V(B) \to V(A)$ according

$$M(e_i) := \begin{cases} M(e_i) & \text{if } e_i \in V(B) \\ 0 & \text{else.} \end{cases}$$

Implementation of Nested Dissection:

$$\begin{aligned} M_{i,j}^{k-1} &:= \sum_{I=i,j,\ldots,i+1,j+1} D_I^k - C_I^k (A_I^k)^{-1} B_I^k \\ b_{i,j}^{k-1} &:= \sum_{I=i,j,\ldots,i+1,j+1} p_I^k - C_I^K (A_I^k)^{-1} q_I^k. \\ t_I^k &= (A_I^k)^{-1}(q_I^k - B_I^k x_I^{k-1}) \end{aligned}$$

On coarsest grid one has to solve exactly

$$M^0 x^0 = b^0$$

Equation

$$M^{k-1} x^{k-1} = b^{k-1}$$

has to be solved recurively from coarse to fine grid.

Implementation of Nested Dissection:
How to define

$$M_{i,j}^{k_{max}} ???$$

- In case of Finite Elements, these are the local stiffness matrices.
- In case of Poisson's equation take the 4x4 matrix

$$\frac{1}{h^2} \begin{pmatrix} 1 & -0.5 & 0 & -0.5 \\ -0.5 & 1 & -0.5 & 0 \\ 0 & -0.5 & 1 & -0.5 \\ -0.5 & 0 & -0.5 & 1 \end{pmatrix}$$

Lineare Algebra with Indizes:

```
class VectorIndex : public ExprAlg<VectorIndex> {
  public:
    template <class Ind> VectorIndex(const Ind& index) {
       size = index.getSize();     Sn   = index.getIndices();
       data = new double[size];
       s    = new int;             Smy  = new int;
    }
    template <class A> void operator=(const ExprAlg<A>& a );
       ...
  private:
     double* data;
     int size;     // Laenge Vektor
     int *Sn;      // Nummern der globalen Indizes
     int *Smy;     // fuer Auswertung: globaler Index
     int *s;       // fuer Auswertung: lokaler Index   };
```

Lineare Algebra with Indizes:

```
void VectorIndex::startI(int max_size) const  {
     (*s) = 0;      if(size>0) (*Smy) = Sn[(*s)];
}
double VectorIndex::getValueI(int Sglobal) const {
   while(Sglobal > (*Smy) && (*s) < size) {
      ++(*s);         (*Smy) = Sn[(*s)];  }
   if((*Smy) > Sglobal || (*s)>=size) return 0;
   return data[(*s)];
}
template <class A>
void VectorIndex::operator=(const ExprAlg<A>& a) {
   const A& ao(a); ao.startI(size);
   for(int ss = 0;ss < size;++ss ) {
       data[ss] =  ao.getValueI(Sn[ss]); }
}  // ----> sorted Indizes!!!
```

Lineare Algebra with Indizes:
Implementation of matrices with Indizes:

```
class MatrixIndex : public ExprAlg<...> {
   public:
      template <class Ind>
      MatrixIndex(const Ind& indexI,const Ind& indexJ);
    ....
}
```

Operators like =,+,- are implemented such that they can be applied to vectors and matrices with respect to different index set:

$$v = b + c$$

Here iteration is performed for the index set A of v.

If b or c is not defined at a certain index $i \in A$, then `getValueI(i)` return 0.0.

Lineare Algebra with Indizes:

- Observe that if v is defined for a index set A.
  Then v is contained in the corresponding vector space:

$$v \in V(A)$$

- A `class IndexSet` is needed which
  - stores indizes in a sequential order and
  - allows union of two index set by *merge sort*.

Implementation of Nested Dissection:
The sets

$$\mathcal{A}_I^k = \mathcal{B}_I^k \cup \mathcal{I}_I^k$$

have to be represented by objects of `class IndexSet` and constructed recurively.

The matrices

$$\begin{aligned} A_I^k : V(\mathcal{I}_I^k) \to V(\mathcal{I}_I^k), \qquad & B_I^k : V(\mathcal{B}_I^k) \to V(\mathcal{I}_I^k), \\ C_I^k : V(\mathcal{I}_I^k) \to V(\mathcal{B}_I^k), \qquad & D_I^k : V(\mathcal{B}_I^k) \to V(\mathcal{B}_I^k), \\ M_I^k : V(\mathcal{A}_I^k) \quad \to \quad & V(\mathcal{A}_I^k) \end{aligned}$$

have to be represented by objects of `class MatrixIndex` and constructed recurively.

Implementation of Nested Dissection:
The sets $\mathcal{A}_I^k, \mathcal{B}_I^k, \mathcal{I}_I^k$ and matrices $A_I^k, B_I^k, C_I^k, D_I^k$, and M_I^k have to be stored as members of leaves in an *quadtree*.

```
class  Leaf {
 public:
   Leaf(...);        ...
```

```
   VectorIndex* x;    ///> W
   ...
   MatrixIndex* A;   ///> W  -> W
   MatrixIndex* B;   ///> Vb -> W
   ...
private:
  std::vector<Leaf*> children;
  IndexVector  allIndizes;       ///> set A
  IndexVector  interiorIndizes;  ///> set I
  IndexVector  boundaryIndizes;  ///> set B
};
```

Implementation of Nested Dissection:

- Dested Dissection has to be implemented by traversing through a quadtree with leaves of obeject `class Leaf`.
- Efficiency mainly depends on the efficient implementation of
  - matrix multiplication and
  - Gauss-algorithm implementation to compute A^{-1}.

  using a lineare algebra library on index sets. To this end cache efficient implementation is very important!

References

[1] Goedecker und Adolfy Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM, 2001.

[2] J.L. Hennessy and D. A. Patterson *Computer Architecture, A Quantitative Approache. Third Edition.* Morgan Kaufmann Publishers, 2003.

[3] A. S. Glassner . *An Introduction to Ray Tracing.* Morgan Kaufmann Publishers, 2007.

[4] W. Gropp und E. Lusk und A. Skjellum. *Using MPI.* The MIT Press, 1999.

[5] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. *Parallel Programming in OpenMP.* Morgam Kaufmann Publishers, 2001.

[6] A. Alexandrescu. *Modern C++ Design.* Generic Programming and Design Patterns Applied.

[7] W. Hackbusch. *Elliptic Differential Equations. Theory and Numerical Treatment.* Springer Series in Computational Mathematics Vol.18, Springer, 1992.

[8] R.L. Burden, J.D. Faires. *Numerical Analysis.* Brooks/Cole, 2001.