Technical University Berlin
Scientific Computing WS 2017/2018
Script
Version November 28, 2017
Jürgen Fuhrmann

Contents

1. Preface

This script is composed from the lecture slides with possible additional text. In order to relate to the slides, the slide titles are retained in the skript and printed in bold face blue color aligned to the right.

2. Introduction

2.1. Motivation

There was a time when “computers” were humans

Harvard Computers, circa 1890

By Harvard College Observatory - Public Domain

https://commons.wikimedia.org/w/index.php?curid=10392913

HARVARD COLLEGE OBSERVATORY.

CIRCULAR 173.

PERIODS OF 25 VARIABLE STARS IN THE SMALL MAGELLANIC CLOUD.

The following statement regarding the periods of 25 variable stars in the Small Magellanic Cloud has been prepared by Miss Leavitt.

A Catalogue of 1777 variable stars in the two Magellanic Clouds is given in H.A. 60, No. 4. The measurement and discussion of these objects present problems of unusual difficulty, on account of the large area covered by the two regions, the extremely crowded distribution of the stars contained in them, the faintness of the variables, and the shortness of their periods. As

It was about science – astronomy

Computations of course have been performed since ancient times. One can trace back the termin “computer” applied to humans at least until 1613. The “Harvard computers” became very famous in this context. Incidently, they were mostly female. They predate the NASA human computers of recent movie fame.

Does this scale ?

WEATHER PREDICTION

BY

NUMERICAL PROCESS

Second edition

BY

LEWIS F. RICHARDSON, B.A., F.R.MET.SOC., F.INST.P.

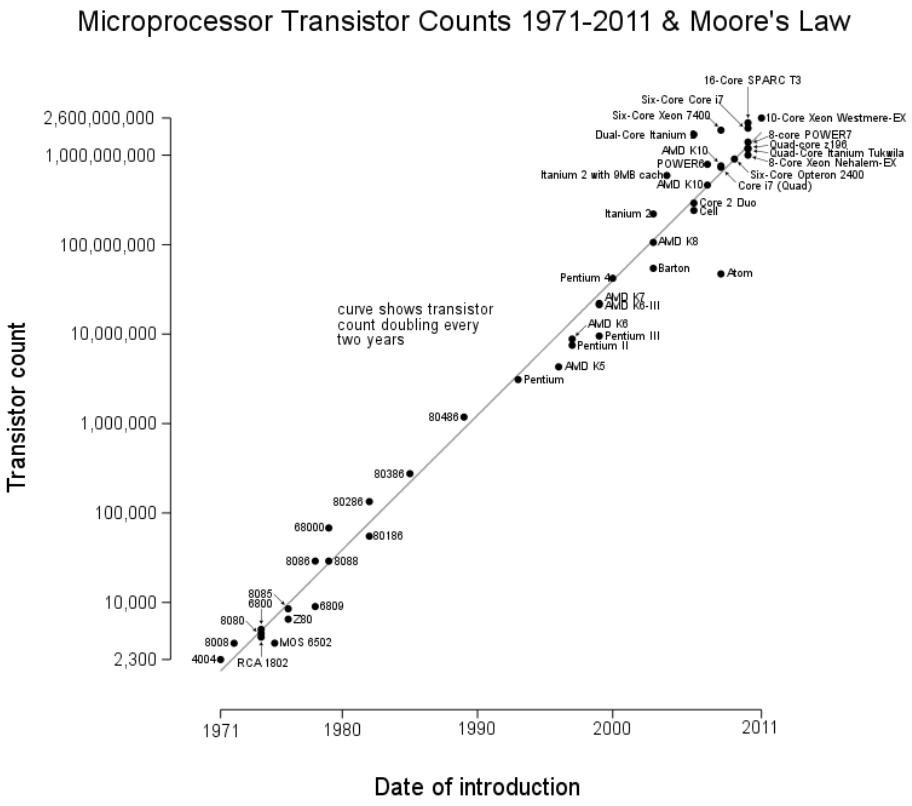
FORMERLY SUPERINTENDENT OF ESKDALEMUIR OBSERVATORY
LECTURER ON PHYSICS AT WESTMINSTER TRAINING COLLEGE

L.F.Richardson 1922

64000 computers predicting weather (1986 Illustration of L.F. Richardson’s vision by S. Conlin)

- This was about weather, not science in the first place
- Science *and* Engineering need computing

Computing was taken over by machines



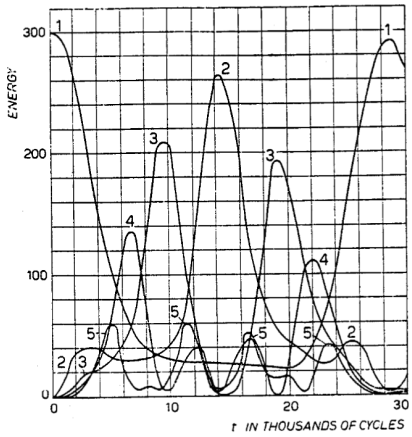
By Wgsimon - Own work, CC BY-SA 3.0 - Public Domain https://commons.wikimedia.org/w/index.php?curid=15193542

Computational engineering

- Starting points: Nuclear weapons + rocket design, ballistic trajectories, weather . . .
- Now ubiquitous:
  - Structural engineering
  - Car industry
  - Oil recovery
  - . . .
- Use of well established, verfied, well supported commercial codes
  - Comsol
  - ANSYS
  - Eclipse
  - . . .

As soon as computing machines became available . . .

. . . Scientists “misused” them to satisfy their curiosity



266.

STUDIES OF NON LINEAR PROBLEMS

E. Fermi, J. Pasta, and S. Ulam
Document LA-1940 (May 1955).

Abstract.

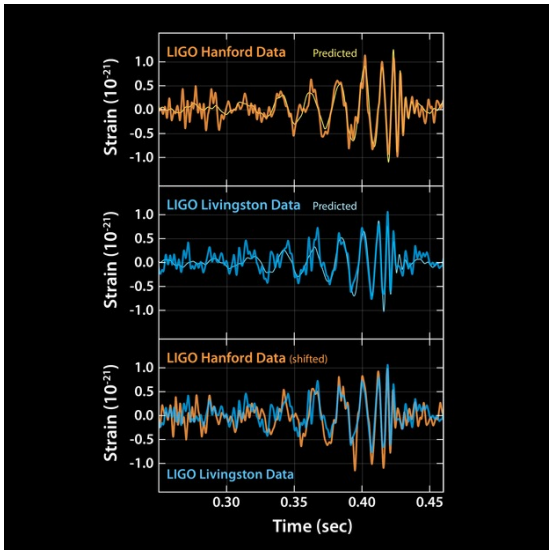
A one-dimensional dynamical system of 64 particles with forces between neighbors containing nonlinear terms has been studied on the Los Alamos computer Maniac I. The nonlinear terms considered are quadratic, cubic, and broken linear types. The results are analyzed into Fourier components and plotted as a function of time.

“. . . Fermi became interested in the development and potentialities of the electronic computing machines. He held many discussions [. . .] of the kind of future problems which could be studied through the use of such machines.”

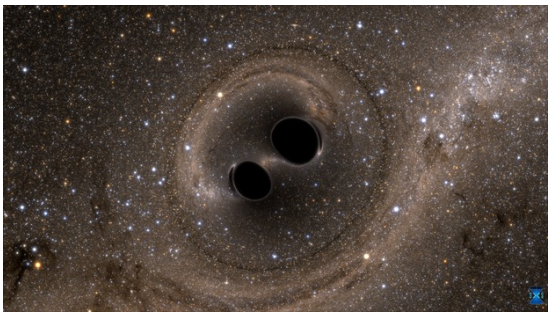
Fermi,Pasta and Ulam studied particle systems with *nonlinear* interactions

Calculations were done on the MANIAC-1 computer at Los Alamos

And they still do. . .



Caltech/MIT/LIGO Lab



SXS, the Simulating eXtreme Spacetimes (SXS) project (http://www.black-holes.org)

Verification of the detection of gravitational waves by numerical solution of Einstein’s equations of general relativity using the “Spectral Einstein Code”

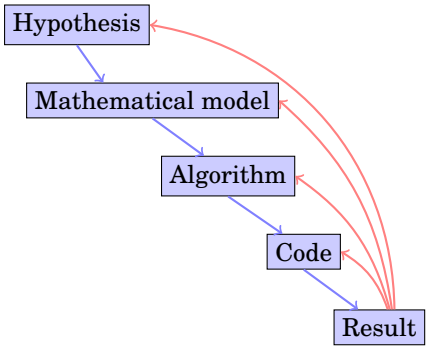
Computations significantly contributed to the 2017 Nobel prize in physics

Scientific computing

"The purpose of computing is insight, not numbers."
(https://en.wikiquote.org/wiki/Richard_Hamming)

- Frontiers of Scientific Computing
  - Insight into complicated phenomena not accessible by other methods
  - Improvement of models to better fit reality
  - Improvment of computational methods
  - Generate testable hypothesis
  - Support experimentation in other scientific fields
  - Exploration of new computing capabilities
  - Prediction, optimization of complex systems
- Good scientifc practice
  - Reproducibility
  - Sharing of ideas and knowledge
- Interdisciplinarity
  - Numerical Analysis
  - Computer science
  - Modeling in specific fields

General approach



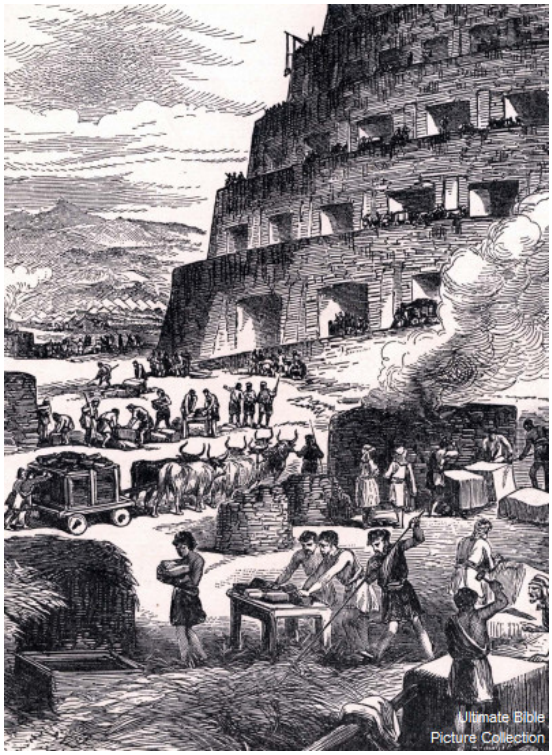
- Possible (probable) involvement of different persons, institutions
- It is important to keep interdisciplinarity in mind

Scientific computing tools

Many of them are Open Source

- General purpose environments
  - Matlab
  - COMSOL
  - Python + ecosystem
  - R + ecosystem
  - Julia (evolving)
- "Classical" computer languages + compilers
  - Fortran
  - C, C++
- Established special purpose libraries
  - Linear algebra: LAPACK, BLAS, UMFPACK, Pardiso
  - Mesh generation: triangle, TetGen, NetGen
  - Eigenvalue problems: ARPACK
  - Visualization libraries: VTK
- Tools in the "background"
  - Build systems Make, CMake
  - Editors + IDEs (emacs, jedit, eclipse)
  - Debuggers
  - Version control (svn, git, hg)

Confusio Linguarum



"And the whole land was of one language and of one speech. ... And they said, Go to, let us build us a city and a tower whose top may reach unto heaven. ... And the Lord said, behold, the people is one, and they have all one language. ... Go to, let us go down, and there confound their language that they may not understand one another's speech. So the Lord scattered them abroad from thence upon the face of all the earth." (Daniel 1:1-7)

Once again Hamming

. . . of "Hamming code" and "Hamming distance" fame, who started his carrier programming in Los Alamos:

"Indeed, one of my major complaints about the computer field is that whereas Newton could say,"If I have seen a little farther than others, it is because I have stood on the shoulders of giants," I am forced to say, "Today we stand on each other's feet." Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things." (1968)

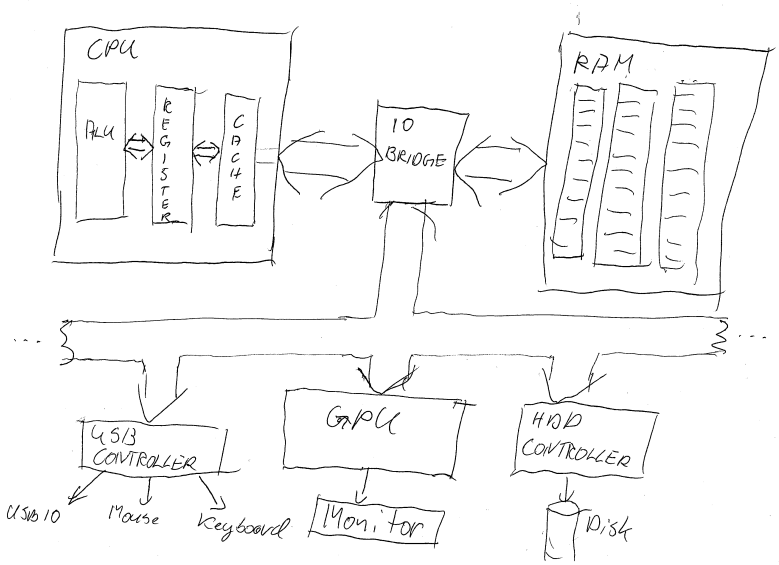
- 2017 this is still a problem

Intended aims and topics of this course

- Indicate a reasonable path within this labyrinth
- Relevant topics from numerical analysis
- Introduction to C++ ($\approx$ 3 lectures) and Python (short, mostly for graphics purposes)
- Provide technical skills to understand a part of the inner workings of the relevant tools
- Focus on partial differential equation (PDE) solution
  - Finite elements
  - Finite volumes
  - Mesh generation
  - Nonlinear if time permits – so we can see some real action
  - Parallelization
  - A bit of visualization
- Tools/Languages
  - C++, Python
  - Parallelization: Focus on OpenMP, but glances on MPI, C++ threads
  - Visualization: Python, VTK

2.2. Sequential Hardware

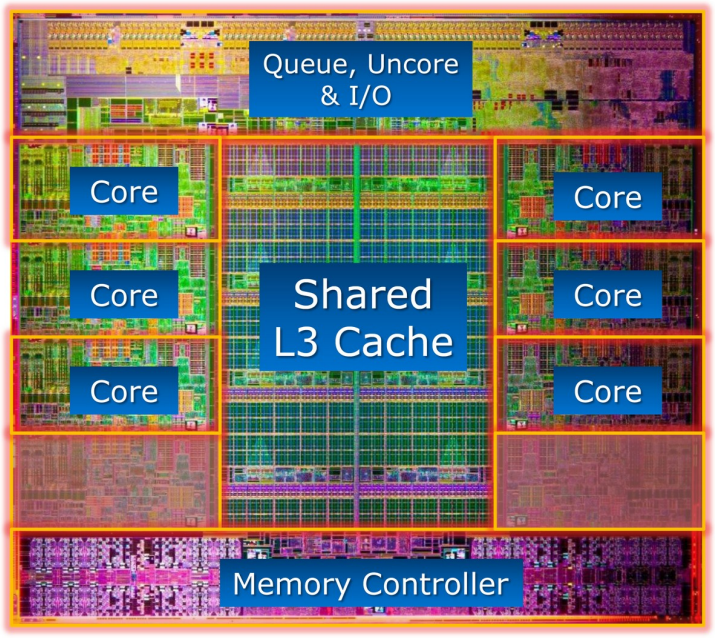
von Neumann Architecture



- Data and instructions from same memory
  - Instruction decode: determine operation and operands
  - Get operands from memory
  - Perform operation
  - Write results back
  - Continue with next instruction

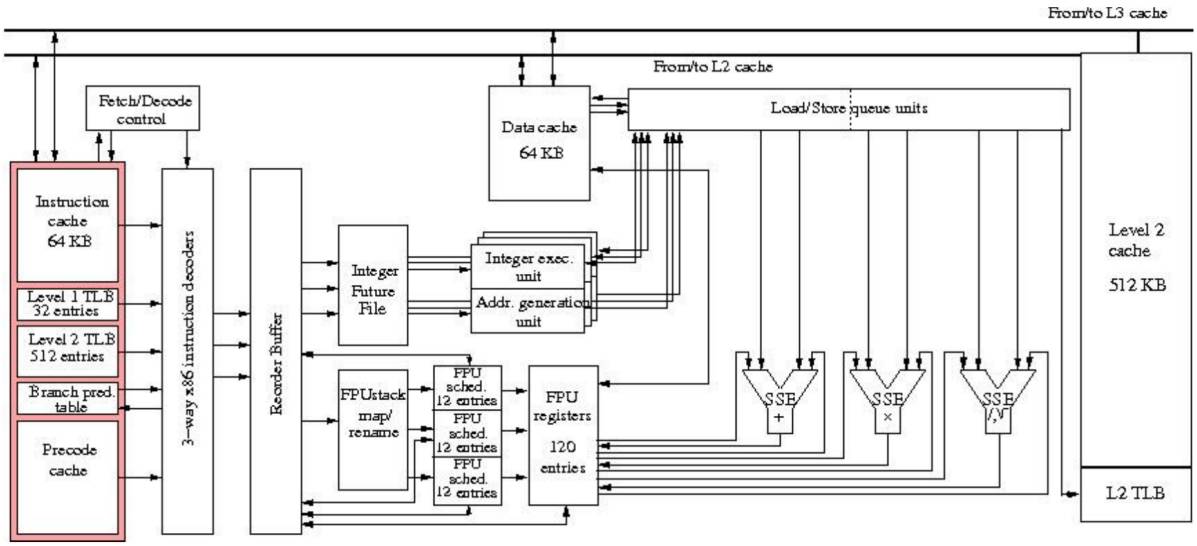
Contemporary Architecture

- Multiple operations simultaneously “in flight”
- Operands can be in memory, cache, register
- Results may need to be coordinated with other processing elements
- Operations can be performed speculatively



Modern CPU. From: https://www.hartware.de/review_1411_2.html

What is in a “core” ?



From: Eijkhout

Modern CPU functionality

- Traditionally: one instruction per clock cycle
- Modern CPUs: Multiple floating point units, for instance 1 Mul + 1 Add, or 1 FMA
  - Peak performance is several operations /clock cycle
  - Only possible to obtain with highly optimized code
- Pipelining
  - A single floating point instruction takes several clock cycles to complete:
  - Subdivide an instruction:
    - Instruction decode
    - Operand exponent align
    - Actual operation
    - Normalize
  - Pipeline: separate piece of hardware for each subdivision
  - Like assembly line

Data and code

- Stored in the same memory ⇒ encoded in the same way
- Stored as binary numbers, most often written in hexadecimal form

Machine code

- Detailed instructions for the actions of the CPU
- Not human readable
- Sample types of instructions:
  - Transfer data between memory location and register
  - Perform arithmetic/logic operations with data in register
  - Check if data in register fulfills some condition
  - Conditionally change the memory address from where instructions are fetched ≡ "jump" to address
  - Save all register context and take instructions from different memory location until return ≡ "call"
- Instructions are very hard to handle, although programming started this way...

```
534c 29e5 31db 48c1 fd03 4883 ec08 e85d
feff ff48 85ed 741e 0f1f 8400 0000 0000
4c89 ea4c 89f6 4489 ff41 ff14 dc48 83c3
0148 39eb 75ea 4883 c408 5b5d 415c 415d
415e 415f c390 662e 0f1f 8400 0000 0000
f3c3 0000 4883 ec08 4883 c408 c300 0000
0100 0200 4865 6c6c 6f20 776f 726c 6400
011b 033b 3400 0000 0500 0000 20fe ffff
8000 0000 60fe ffff 5000 0000 4dff ffff
```

Assembler code

- Human readable representation of CPU instructions
- Some write it by hand ...
  - Code close to abilities and structure of the machine
  - Handle constrained resources (embedded systems, early computers)
- Translated to machine code by a programm called *assembler*

```
.file   "code.c"
.section    .rodata
.LC0:
.string "Hello world"
.text
...
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
subq    $16, %rsp
movl    %edi, -4(%rbp)
movq    %rsi, -16(%rbp)
movl    $.LC0, %edi
movl    $0, %eax
call    printf
```

Memory Hierachy

- Main memory access is slow compared to the processor
  - 100–1000 cycles latency before data arrive
  - Data stream maybe 1/4 floating point number/cycle;
  - processor wants 2 or 3
- Faster memory is expensive
- *Cache* is a small piece of fast memory for intermediate storage of data
- Operands are moved to CPU *registers* immediately before operation
- Memory hierarchy:

  Registers in different cores Fast on-CPU cache memory (L1, L2, L3) Main memory

Registers

Processor instructions operate on registers directly

- have assembly language names names like: `eax`, `ebx`, `ecx`, etc.
- sample instruction: `addl %eax, %edx`
- Separate instructions and registers for floating-point operations

Data caches

- Between the CPU Registers and main memory
- L1 Cache: Data cache closest to registers
- L2 Cache: Secondary data cache, stores both data and instructions
- Data from L2 has to go through L1 to registers
- L2 is 10 to 100 times larger than L1
- Some systems have an L3 cache, ≈10x larger than L2

Cache line

- The smallest unit of data transferred between main memory and the caches (or between levels of cache)
- N sequentially-stored, multi-byte words (usually N=8 or 16).
- If you request one word on a cache line, you get the whole line
  - make sure to use the other items, you've paid for them in bandwidth
  - Sequential access good, "strided" access ok, random access bad
- Cache hit: location referenced is found in the cache
- Cache miss: location referenced is not found in cache
  - triggers access to the next higher cache or memory
- Cache thrashing
  - Two data elements can be mapped to the same cache line: loading the second "evicts" the first
  - Now what if this code is in a loop? "thrashing": really bad for performance
- Performance is limited by data transfer rate
  - High performance if data items are used multiple times

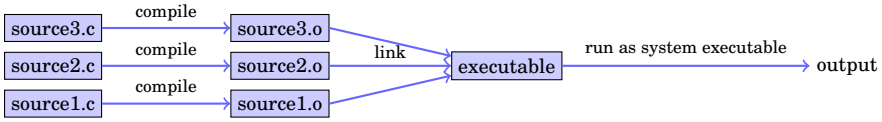
2.3. Computer Languages

Compiled high level languages

- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Translated to machine code (resp. assembler) by *compiler*

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  printf("Hello world");
}
```

- "Far away" from CPU ⇒ the compiler is responsible for creation of optimized machine code
- Fortran, COBOL, C, Pascal, Ada, Modula2, C++, Go, Rust, Swift
- Strongly typed
- Tedious workflow: compile - link - run

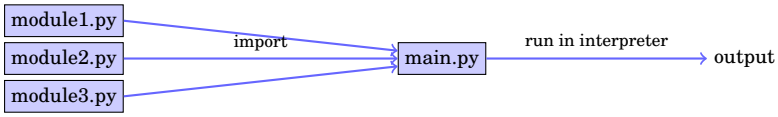


High level scripting languages

- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Need intepreter in order to be executed

```
print("Hello world")
```

- Very far away from CPU ⇒ usually significantly slower compared to compiled languages
- Matlab, Python, Lua, perl, R, Java, javascript
- Less strict type checking, often simple syntax, powerful introspection capabilities
- Immediate workflow: "just run"
  - in fact: first compiled to *bytecode* which can be interpreted more efficiently



JITting to the future ?

- As described, all modern interpreted language first compile to bytecode wich then is run in the interpreter
- Couldn't they be compiled to machine code instead? – Yes, they can: Use a just in time (JIT) compiler!
  - V8 → javascript
  - LLVM Compiler infrastructure → Python/NUMBA, **Julia** (currently at 0.5)
  - LuaJIT
  - Java
  - Smalltalk
- Drawback over compiled languages: compilation delay at every start, can be mediated by caching
- Potential advantage over compiled languages: *tracing* JIT, i.e. optimization at runtime
- Still early times, but watch closely...

Compiled languages in Scientific Computing

- Fortran: FORmula TRANslator (1957)
  - Fortran4: really dead
  - Fortran77: large number of legacy libs: BLAS, LAPACK, ARPACK ...
  - Fortran90, Fortran2003, Fortran 2008
    - Catch up with features of C/C++ (structures,allocation,classes,inheritance, C/C++ library calls)
    - Lost momentum among new programmers
    - Hard to integrate with C/C++
    - In many aspects very well adapted to numerical computing
    - Well designed multidimensional arrays
- C: General purpose language
  - K&R C (1978) weak type checking
  - ANSI C (1989) strong type checking
  - Had structures and allocation early on
  - Numerical methods support via libraries
  - Fortran library calls possible
- C++: *The* powerful object oriented language
  - Superset of C (in a first approximation)
  - Classes, inheritance, overloading, templates (generic programming)
  - C++11: Quantum leap: smart pointers, threads, lambdas, initializer lists in standard
  - With great power comes the possibility of great failure...

Summary

- Compiled languages important for high performance
- Fortran lost its momentum, but still important due to huge amount of legacy libs
- C++ highly expressive, ubiquitous, significant improvements in C++11

3. Introduction to C++

3.1. C++ basics

Evolution

- Essentially, C++ started as “C with classes”
- Current standard is C++11, C++14 and C++17 are evolving.
- Almost all of the C language is part of C++
- C standard library is part of C++ standard librar
- As most computer languages, C++ has variables, flow control, functions etc. which will be discussed first

Printing stuff

Printing is not part of the language itself, but is performed via functions from libraries. As we need printing very early in the examples, we show how to do it.

- IOStream library
  - “Official” C++ output library
  - Type safe, easy to extend
  - Clumsy syntax for format control

```
#include <iostream>
...
std::cout << "Hello world" << std::endl;
```

- C Output library
  - Supported by C++-11 standard
  - No type safety, Hard to extend
  - Short, relatively easy syntax for format control
  - Same format specifications as in Python

```
#include <cstdio>
...
std::printf("Hello world\n");
```

C++ : scalar data types

- Store character, integer and floating point values of various sizes
- Type sizes are the "usual ones" on 64bit systems

name	printf	bytes	bits	Minimum value	Maximum value
char	%c (%d)	1	8	-128	127
unsigned char	%c (%d)	1	8	0	255
short int	%d	2	16	-32768	32767
unsigned short int	%u	2	16	0	65535
int	%d	4	32	-2147483648	2147483647
unsigned int	%u	4	32	0	4294967295
long int	%ld	8	64	-9223372036854775808	9223372036854775807
unsigned long int	%lu	8	64	0	18446744073709551615
float	%e	4	32	1.175494e-38	3.402823e38
double	%e	8	64	2.225074e-308	1.797693e308
long double	%Le	16	128	3.362103e-4932	1.189731e4932
bool	%d	1	8	0	1

- The standard only guarantees that `sizeof(short ...) <= sizeof(...) <=sizeof(long ...)`
- E.g. on embedded systems sizes may be different
- Declaration and output (example)

```
#include <cstdio>
...
int i=3;
double x=15.0;
std::printf("i=%d, x=%e\n",i,x);
```

Typed constant expressions

- C++ has the ability to declare variables as constants:

```
const int i=15;
i=i+1; // attempt to modify value of const object leads to
     // compiler error
```

Scopes, Declaration, Initialization

- **All variables are typed and must be declared**
  - Declared variables "live" in scopes defined by braces `{ }`
  - Good practice: initialize variables along with declaration
  - "auto" is a great innovation in C++11 which is useful with complicated types which arise in template programming
    - type of *lvalue* (left hand side value) is detected from type of *rvalue* (value at the right hand side)

```
{
   int i=3;
   double x=15.0;
   auto y=33.0;
}
```

Arithmetic operators

- Assignment operator

```
a=b;
c=(a=b);
```

- Arithmetic operators `+`, `-`, `*`, `/`, modulo (%)
- Beware of precedence which (mostly) is like in math!
- If in doubt, use brackets, or look it up!

- Compund assignment: `+=`, `-=`, `*=`, `/=`, `%=`

```
x=x+a;
x+=a; // equivalent to =x+a
```

- Increment and decrement:
  `++`,`--`

```
y=x+1;
y=x++; // equivalent to y=x; x=x+1;
y=++x; // equivalent to x=x+1; y=x;
```

Further operators

- Relational and comparison operators `==`, `!=`, `>`, `<`, `>=`, `<=`
- Logical operators `!`, `&&`, `||`
  - short circuit evaluation:
    - if `a` in `a&&b` is false, the expression is false and `b` is never evaluated
    - if `a` in `a||b` is true, the expression is true and `b` is never evaluated
- Conditional ternary operator `?`

```
c=(a<b)?a:b; // equivalent to the following
if (a<b) c=a; else c=b;
```

- Comma operator `,`

```
c=(a,b); // evaluates to c=b
```

- Bitwise operators `&`, `|`, `^`, `~`, `<<`, `>>`
- `sizeof`: memory space (in bytes) used by the object resp. type

```
n=sizeof(char); //  evaluate
```

Functions

- Functions have to be *declared* and given names as other variables:
  ```
  type name(type1 p1, type2 p2,...);
  ```
- `(...)` holds parameter list
  - each parameter has to be defined with its type
- `type` part of declaration describes type of return value
  - `void` for returning nothing

```
  double multiply(double x, double y);
```

- Functions are *defined* by attaching a scope to the declaration
  - *Values of parameters are copied into the scope*

```
double multiply(double x, double y)
{
   return x*y;
}
```

- Functions are *called* by statements invoking the function with a particular set of parameters

```
{
    double s=3.0, t=9.0;
    double result=multiply(s,t);
    printf("s=%e, t=%e, s*t= %e\n",s,t,result); // s and t keep their values
}
```

Functions: inlining

- Function calls sometimes are expensive compared to the task performed by the function
  - Remember: save all register context and take instructions from different memory location until return, restore register context after return
  - The compiler may include the content of functions into the instruction stream instead of generating a call

```
inline double multiply(double x, double y)
{
   return x*y;
}
```

Flow control: Statements and conditional statements

- Statements are individual expressions like declarations or instructions or sequences of statements enclosed in curly braces:

```
{ statement1; statement2; statement3; }
```

- Conditional execution: `if`

```
if (condition) statement;
if (condition) statement; else statement;
```

```
if (x>15)
{
    printf("error");
}
else
{
   x++;
}
```

Equivalent but less safe:

```
if (x>15)
    printf("error");
else
   x++;
```

Flow control: Simple loops

- While loop:
  `while (condition) statement;`

```
i=0;
while (i<9)
{
  printf("i=%d\n",i);
  i++;
}
```

- Do-While loop: `do statement while (condition);`

Flow control: for loops

- This is the most important kind of loops for numerical methods.
  `for (initialization; condition; increase) statement;`
  1. initialization is executed. Generally, here, one declares a counter variable and sets it to some initial value. This is executed a single time, at the beginning of the loop.
  2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
  3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces `{ }`
  4. increase is executed, and the loop gets back to step 2.
  5. The loop ends: execution continues at the next statement after it.
- All elements (initialization, condition, increase, statement) can be empty

```
for (int i=0;i<9;i++)   printf("i=%d\n",i); // same  as on previous slide
for(;;);  // completely valid, runs forever
```

Flow control: break, continue

- break statement: “premature” end of loop

```
for (int i=1;i<10;i++)
{
   if (i*i>15) break;
}
```

- continue statement: jump to end of loop body

```
for (int i=1;i<10;i++)
{
   if (i==5) continue;
   else do_someting_with_i;
}
```

Flow control: switch

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  ...
  default:
     default-group-of-statements
}
```

equivalent to

```
if      (expression==constant1) {group-of-statements-1;}
else if (expression==constant2) {group-of-statements-2;}
...
else                            {default-group-of-statements;}
```

Execution of switch statement can be faster than the hierarchy of if-then-else statement

The Preprocessor

- Before being sent to the compiler, the source code is sent through the *preprocessor*
- It is a legacy from C which is slowly being squeezed out of C++
- Preprocessor commands start with #
- Include contents of file `file.h` found on a default search path known to the compiler:

```
#include <file.h>
```

- Include contents of file `file.h` found on user defined search path:

```
#include "file.h"
```

- Define a piece of text (mostly used for constants in pre-C++ times)
  (avoid, use `const` instead):

```
#define N 15
```

- Define preprocessor macro for inlining code
  (avoid, use inline functions instead):

```
#define  MAX(X,Y) (((x)>(y))?(x):(y))
```

Conditional compilation and pragmas

- Conditional compilation of pieces of source code, mostly used to dispatch between system dependent variant of code. Rarely necessary nowadays...

```
#ifdef MACOSX
statements to be compiled only for MACOSX
#else
statements for all other systems
#endif
```

- There can be more complex logic involving constant expressions
- A pragma gives directions to the compiler concerning code generation:

```
#pragma omp parallel
```

Headers

- If we want to use functions from the standard library we need to include a *header file* which contains their declarations
  - The `#include` statement invokes the C-Preprocessor and leads to the inclusion of the file referenced therein into the actual source
  - Include files with names in < > brackets are searched for in system dependent directories known to the compiler

```
#include <iostream>
```

Namespaces

- Namespaces allow to prevent clashes between names of functions from different projects
  - All functions from the standard library belong to the namespace `std`

```
namespace foo
{
  void cool_function(void);
}

namespace bar
{
  void cool_function(void);
}

...

{
  using namespace bar;
  foo::cool function()
  cool_function() // equivalent to bar::cool_function()
}
```

Modules ?

- Currently, C++ has no well defined module system.
- A module system usually is emulated using the preprocessor and namespaces.

Emulating modules

- File `mymodule.h` containing interface declarations

```
#ifndef MYMODULE_H // Handle multiple #include statements
#define MYMODULE_H
namespace mymodule
{
  void my_function(int i, double x);
}
#endif
```

- File `mymodule.cpp` containing function definitions

```
#include "mymodule.h"
namespace mymodule
{
  void my_function(int i, double x)
  {
    ...body of function definition...
  }
}
#endif
```

- File using `mymodule`:

```
#include "mymodule.h"
...
mymodule::my_function(3,15.0);
```

main

Now we are able to write a complete program in C++

- `main()`
  is the function called by the system when running the program. Everything else needs to be called from there.

Assume the follwing content of the file `run42.cxx`:

```
#include <cstdio>

int main()
{
   int i=4,j=2;
   int answer=10*i+j;
   printf("Hello world, the answer is %d!\n",answer);
   return 0;
}
```

Then the sequence of command line commands

```
$ g++ -o run42 run42.cxx
$ ./run42
```

gives the right answer to (almost) anything.

3.2. C++ – the hard stuff

Addresses and pointers

- Objects are stored in memory, in order to find them they have an *address*
- We can determine the address of an object by the `&` operator
  - The result of this operation can be assigned to a variable called *pointer*
  - "pointer to type x" is another type denoted by `*x`
- Given an address (pointer) object we can refer to the content using the `*` operator

```
int i=15;  // i is an object
int *j= &i; // j is a pointer
int k=*j; // now, k=15
```

- The `nullptr` object can be assigned to a pointer in order to indicate that it points to "nothing"

```
int *p=nullptr;
```

Passing addresses to functions

- Instead of values, addresses can be passed to functions

```
void do_multiplication(double x, double y, double *result)
{
    *result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,&result) // result now contains 45
```

Arrays

- Focusing on numerical methods for PDEs results in work with finite dimensional vectors which are represented as *arrays* - sequences of consecutively stored objects
- Stemming from C, in C++ array objects represent just the fixed amount of consecutive memory. No size info or whatsoever
- No bounds check
- First array index is always 0

```
double x[9]; // uninitialized array of 9 elements
double y[3]={1,2,3}; // initialized array of 3 elements
double z[]={1,2,3};  // Same
double z[]{1,2,3};   //Same
```

- Accessing arrays
  - `[]` is the array access operator in C++
  - Each element of an array has an index

```
double a=x[3];  // undefined value because x was not initialized
double b=y[12]; // undefined value because out of bounds
y[12]=19;       // may crash program ("segmentation fault"),
double c=z[0];  // Acces to first element in array, now c=1;
```

Arrays, pointers and pointer arithmetic

- Arrays are strongly linked to pointers
- Array object can be treated as pointer

```
double x[]={1,2,3,4};
double b=*x; // now x=1;
double *y=x+2; // y is a pointer to third value in arrax
double c=*y;   // now c=3
ptrdiff_t d=y-x; // We can also do differences between pointers
```

- Pointer arithmetic is valid only in memory regions belonging to the same array

Arrays and functions

- Arrays are passed by passing the pointer referring to its first element
- As they contain no length information, we need to pass that as well

```
void func_on_array1(double[] x, int len);
void func_on_array2(double* x, int len); // same
void func_on_array3(const double[] x, int len);//same, but prevent changing x
...
double x[]={3,4,5};
int len=sizeof(x)/sizeof(x[0]);
func_on_array1(x,len);
```

- Be careful with array return

```
double * some_func(void)
{
  double a[]={-1,-2,-3};
  return a; // illegal as with the end of scope, the life time of a is over
            // smart compilers at least warn
}
```

Arrays with length detected at runtime ?

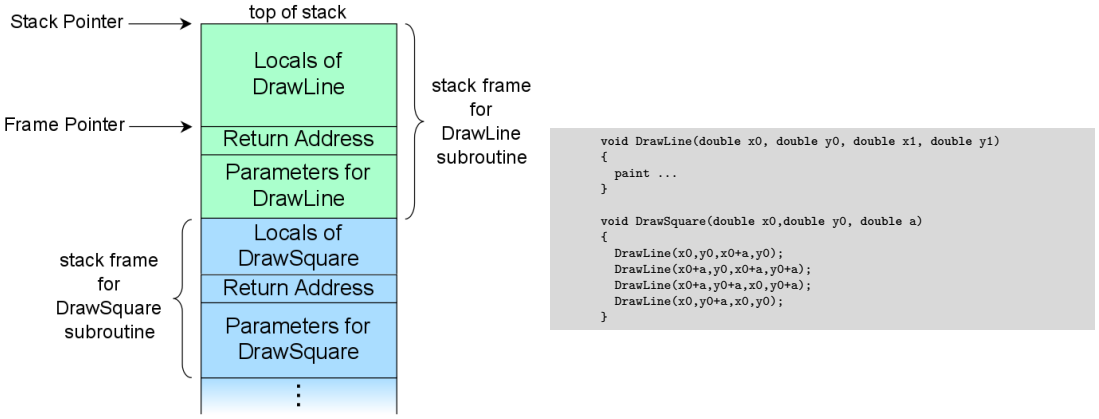
- This one is illegal (not part of C++ standard), though often compilers accept it

```
void another_func(int n)
{
    int b[n];
}
```

- Even in main() this will be illegal.
- How to work on problems where size information is obtained only during runtime, e.g. user input ?

Memory: stack

- pre-allocated memory where `main()` and all functions called from there put their data.
  - Any time a function is called, the current position in the instruction stream is stored in the stack as the return address, and the called function is allowed to work with the memory space after that



By R. S. Shaw, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1956587

Stack space is scarce

- Variables declared in {} blocks get placed on the stack
- All previous examples had their data on the stack, even large arrays
- Stack space should be considered scarce
- Stack size for a program is fixed, set by the system
- On UNIX, use `ulimit -s` to check/set stack size default

Memory: heap

- Chunks from free system memory can be reserved – "allocated" – on demand in order to provide memory space for objects
- The operator `new` reserves the memory and returns an address which can be assigned to a pointer variable
- The operator `delete` (`delete[]` for arrays) releases this memory and makes it available for other processes
- Compared to declarations on the stack, these operations are expensive
- Use cases:
  - Problem sizes unknown at compile time
  - Large amounts of data
  - . . . so, yes, we will need this. . .

```
double *x= new double(5); // allocate space for a double, initialize it with 5
double *y=new double[5];  // allocate space of five doubles, uninitialized
x[3]=1;                   // Segmentation fault
y[3]=1;                   // Perfect...
delete x;                 // Choose the right delete!
delete[] y;               // Choose the right delete!
```

Multidimensional Arrays

- Multidimensional arrays are useful for storing matrices, tensors, arrays of coordinate vectors etc.
- It is easy to declare a multidimensional array on the stack when the size the array is known at compile time

```
double x[5][6];

for (int i=0;i<5;i++)
   for (int j=0;j<6;j++)
      x[i][j]=0;
```

- Determining array dimensions from function parameters may work with some compilers, but are not backed by the standard
- One option to have 2D arrays with arbitrary, run-time defined dimensions is to allocate a an array of pointers to double, and to use `new` to allocate each (!) row
  . . . this leads to nowhere . . .

Intermediate Summary

- This was mostly all (besides structs) of the C subset of C++
  - Most "old" C libraries and code written in previous versions of C++ are mostly compatible to modern C++
- Many "classical" programs use the `(int size, double * data)` style of passing data, especially in numerics
  - UMFPACK, Pardiso direct solvers
  - PetsC library for distributed linear algebra
  - triangle, tetgen mesh generators
  - . . .
- On this level it is possible to call Fortran programs from C++
  - BLAS vector operations
  - LAPACK dense matrix linear algebra
  - ARPACK eigenvalue computations
  - . . .
- Understanding these interfaces is the main reason to know about plain C pointers and arrays
- Modern C++ has easier to handle and safer ways to do these things, so they should be avoided as much as possible in new programs

Classes and members

- Classes are data types which collect different kinds of data, and methods to work on them.

```
class class_name
{
  private:
    private_member1;
    private_member2;
    ...
  public:
    public_member1;
    public_member2;
    ...
};
```

- If not declared otherwise, all members are private
- `struct` is the same as `class` but by default all members are public
- Accessing members of a class object:

```
   class_name x;
   x.public_member1=...
```

- Accessing members of a pointer to class object:

```
   class_name *x;
   (*x).public_member1=...
   x->public_member1=...
```

Example class

- Define a class `vector` which holds data and length information and thus is more comfortable than plain arrays

```
class vector
{
  private:
      double *data;
  public:
       int size;
      double get_value( int i) {return data[i];};
      void set_value( int i, double value); {data[i]=value;};
};

...

{
  vector v;
  v.data=new double(5);  // would work if data would be public
  v.size=5;
  v.set_value(3,5);

  b=v.get_value(3); // now, b=5
  v.size=6;  // size changed, but not the length of the data array...
             // and who is responsible for delete[] at the end of scope ?
}
```

- Methods of a class know all its members
- It would be good to have a method which constructs the vector and another one which destroys it.

Constructors and Destructors

```
class vector
{ private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size(){ return size;};
      double get_value( int i ) { return data[i]; };
      void set_value( int i, double value ) { data[i]=value; };
      Vector( int new_size ) { data = new double[new_size];
                               size=new_size; };
      ~Vector() { delete [] data; };
};
...
{ vector v(5);
  for (int i=0;i<5;i++) v.set_value(i,0.0);
  v.set_value(3,5);
  b=v.get_value(3); // now, b=5
  v.size=6;  // Size is now private and can not be set;
  vector w(5);
  for (int i=0;i<5;i++) w.set_value(i,v.get_value(i));
  // Destructors automatically called at end of scope.
}
```

- Constructors are declared as `classname(...)`
- Destructors are declared as `~classname()`

Interlude: References

- C style access to objects is direct or via pointers
- C++ adds another option - references
  - References essentially are alias names for already existing variables
  - Must always be initialized
  - Can be used in function parameters and in return values
  - No pointer arithmetics with them
- Declaration of reference

```
double a=10.0;
double &b=a;

b=15; // a=15 now as well
```

- Reference as function parameter: no copying of data!

```
void do_multiplication(double x, double y, double &result)
{
    result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,result) // result now contains 45
```

Vector class again

- We can define `()` and `[]` operators!

```
class vector
{
  private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      double & operator()(int i) { return data[i]; };
      double & operator[](int i) { return data[i]; };
      vector( int new_size) { data = new double[new_size];
                              size=new_size;}
      ~vector() { delete [] data;}
};
...
{
  vector v(5);
  for (int i=0;i<5;i++) v[i]=0.0;
  v[3]=5;
  b=v[3]; // now, b=5
  vector w(5);
  for (int i=0;i<5;i++) w(i)=v(i);
}
```

Matrix class

- We can define `(i,j)` but not `[i,j]`

```
class matrix
{ private:
      double *data=nullptr;
      int size=0;  int nrows=0;
      int ncols=0;
  public:
      int get_nrows( return nrows);
      int get_ncols( return ncols);
      double & operator()(int i,int j) { return data[i*nrow+j]);
      matrix( int new_rows,new_cols)
      { nrows=new_rows; ncols=new_cols;
        size=nrows*ncols;
        data = new double[size];
      }
      ~matrix() { delete [] data;}
};
...
{
  matrix m(3,3);
  for (int i=0;i<3;i++)
    for (int j=0;j<3;j++)
        m(i,j)=0.0;
}
```

Inheritance

- Classes in C++ can be extended, creating new classes which retain characteristics of the base class.
- The *derived class* inherits the members of the *base class*, on top of which it can add its own members.

```
class vector2d
{  private:
     double *data;
     int nrow, ncol;
     int size;
   public:
     double & operator(int i, int j);
     vector2d(int nrow, ncol);
     ~vector2d();
}
class matrix: public vector2d
{ public:
     apply(const vector1d & u, vector1d &v);
     solve(vector1d &u, const vector1d &rhs);
}
```

- All operations which can be performed with instances of `vector2d` can be performed with instances of `matrix` as well
- In addition, `matrix` has methods for linear system solution and matrix-vector multiplication

Generic programming: templates

- Templates allow to write code where a data type is a parameter
- We want do be able to have vectors of any basic data type.
- We do not want to write new code for each type

```
template <typename T>
class vector
{
  private:
      T *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      T & operator[](int i) { return data[i]; };
      vector( int new_size) { data = new T[new_size];
                              size = new_size;};
      ~vector() { delete [] data;};
};
...
{
  vector<double> v(5);
  vector<int> iv(3);
}
```

C++ template libray

- The standard template library (STL) became part of the C++11 standard
- Whenever you can, use the classes available from there
- For one-dimensional data, std::vector is appropriate
- For two-dimensional data, things become more complicated
  - There is no reasonable matrix class
    - std::vector< std::vector> is possible but has to allocate each matrix row and is inefficient
  - it is hard to create a std::vector from already existing data

Smart pointers

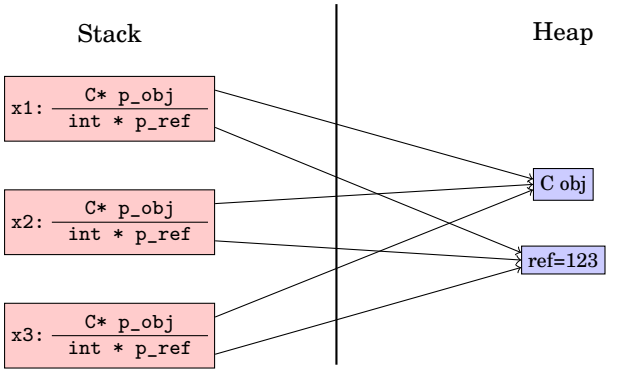
. . . with a little help from Timo Streckenbach from WIAS who introduced smart pointers into our simulation code.

- Automatic book-keeping of pointers to objects in memory.
- Instead of the meory addres of an object aka. pointer, a structure is passed around *by value* which holds the memory address and a pointer to a *reference count* object.
- It delegates the member access operator `->` and the address resolution operator `*` to the pointer it contains.
- Each assignment of a smart pointer increases this reference count.
- Each destructor invocation from a copy of the smart pointer structure decreases the reference count.
- If the reference count reaches zero, the memory is freed.
- `std::shared_ptr` is part of the C++11 standard

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```




```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
std::shared_ptr<C> x3= x1;
```

Smart pointers vs. *-pointers

- When writing code using smart pointers, write

```
#include <memory>
class R;
std::shared_ptr<R> ReturnObjectOfClassR(void);
void PassObjectOfClassR(std::shared_ptr<R> pR);
...
{  auto pR=std::make_shared<R>();
   PassObjectOfClassR(pR)
   // Smart pointer object is deleted at end of scope and frees memory
}
```

instead of

```
class R;
R* ReturnObjectOfClassR(void);
void PassObjectOfClassR(R* o);
...
{  R* pR=new R;
   PassObjectOfClassR(pR);
   delete pR; // never forget this here!!!
}
```

Smart pointer advantages vs. *-pointers

- "Forget" about memory deallocation
- Automatic book-keeping in situations when members of several different objects point to the same allocated memory
- Proper reference counting when working together with other libraries

3.3. C++ – some code examples

C++ code using vectors, C-Style, with data on stack

File /net/wir/numcxx/examples/00-cxx-basics/01-c-style-stack.cxx

```
#include <cstdio>
void initialize(double *x, int n)
{
    for (int i=0;i<n;i++) x[i]= 1.0/(double)(1+n-i);
}
double sum_elements(double *x, int n)
{
    double sum=0;
    for (int i=0;i<n;i++) sum+=x[i];
    return sum;
}
int main()
{
    const int n=12345678;
    double x[n];
    initialize(x,n);
    double s=sum_elements(x,n);
    printf("sum=%e\n",s);
}
```

- Large arrays may not fit on stack
- C-Style arrays do not know their length

C++ code using vectors, C-Style, with data on heap

File /net/wir/numcxx/examples/00-cxx-basics/02-c-style-heap.cxx

```
#include <cstdio>
#include <cstdlib>
#include <new>
void initialize(double *x, int n)
{    for (int i=0;i<n;i++) x[i]= 1.0/(double)(1+n-i);
}
double sum_elements(double *x, int n)
{   double sum=0;
    for (int i=0;i<n;i++) sum+=x[i];
    return sum;
}
int main()
{   const int n=12345678;
    try  {  x=new double[n]; // allocate memory for vector on heap }
    catch (std::bad_alloc)  { printf("error allocating x\n");
                              exit(EXIT_FAILURE); }
    initialize(x,n);
    double s=sum_elements(x,n);
    printf("sum=%e\n",s);
    delete[] x;}
```

- Arrays passed in a similar way as in previous example
- Proper memory management is error prone

C++ code using vectors, (mostly) modern C++-style

File /net/wir/numcxx/examples/00-cxx-basics/03-cxx-style-ref.cxx

```
#include <cstdio>
#include <vector>
void initialize(std::vector<double>& x)
{ for (int i=0;i<x.size();i++) x[i]= 1.0/(double)(1+n-i);
}
double sum_elements(std::vector<double>& x)
{   double sum=0;
    for (int i=0;i<x.size();i++)sum+=x[i];
    return sum;}
int main()
{   const int n=12345678;
    std::vector<double> x(n); // Construct vector with n elements
                              // Object "lives" on stack, data on heap
    initialize(x);
    double s=sum_elements(x);
    printf("sum=%e\n",s);
    // Object destructor automatically called at end of lifetime
    // So data array is freed automatically
}
```

- Heap memory management controlled by object lifetime

C++ code using vectors, C++-style with smart pointers

File /net/wir/numcxx/examples/00-cxx-basics/05-cxx-style-sharedptr.cxx

```
#include <cstdio>
#include <vector>
#include <memory>
void initialize(std::vector<double> &x)
{  for (int i=0;i<x.size();i++) x[i]= 1.0/(double)(1+n-i);}
double sum_elements(std::vector<double> & x)
{    double sum=0;
    for (int i=0;i<x.size();i++)sum+=x[i];
    return sum;
}
int main()
{   const int n=12345678;
    // call constructor and wrap pointer into smart pointer
    auto x=std::make_shared<std::vector<double>>(n);
    initialize(*x);
    double s=sum_elements(*x);
    printf("sum=%e\n",s);
    // smartpointer calls destructor if reference count reaches zero
}
```

- Heap memory management controlled by smart pointer lifetime
- If method or function does not store the object, pass by reference ⇒ API stays the same as for previous case.

4. Direct solution of linear systems of equations

4.1. Recapitulation from Numerical Analysis

Floating point representation

- Scientific notation of floating point numbers: e.g. $x = 6.022 \cdot 10^{23}$
- Representation formula:

$$x = \pm \sum_{i=0}^{\infty} d_i \beta^{-i} \beta^e$$

  - $\beta \in \mathbb{N}, \beta \geq 2$: base
  - $d_i \in \mathbb{N}, 0 \leq d_i \leq \beta$: mantissa digits
  - $e \in \mathbb{Z}$: exponent
- Representation on computer:

$$x = \pm \sum_{i=0}^{t-1} d_i \beta^{-i} \beta^e$$

  - $\beta = 2$
  - t: mantissa length, e.g. $t = 53$ for IEEE double
  - $L \leq e \leq U$, e.g. $-1022 \leq e \leq 1023$ (10 bits) for IEEE double
  - $d_0 \neq 0 \Rightarrow$ normalized numbers, unique representation

Floating point limits

- symmetry wrt. 0 because of sign bit
- smallest positive normalized number: $d_0 = 1, d_i = 0, i = 1 \ldots t-1$
  $x_{min} = \beta^L$
- smallest positive denormalized number: $d_i = 0, i = 0 \ldots t-2, d_{t-1} = 1$
  $x_{min} = \beta^{1-t} \beta^L$
- largest positive normalized number: $d_i = \beta - 1, 0 \ldots t-1$
  $x_{max} = \beta(1 - \beta^{1-t}) \beta^U$

Machine precision

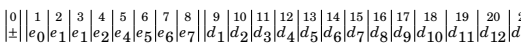
- Exact value x
- Approximation $\tilde{x}$
- Then: $|\frac{\tilde{x}-x}{x}| < \epsilon$ is the best accuracy estimate we can get, where
  - $\epsilon = \beta^{1-t}$ (truncation)
  - $\epsilon = \frac{1}{2}\beta^{1-t}$ (rounding)
- Also: ϵ is the smallest representable number such that $1 + \epsilon > 1$.
- Relative errors show up in partiular when
  - subtracting two close numbers
  - adding smaller numbers to larger ones

Machine epsilon

- Smallest floating point number ϵ such that $1+\epsilon > 1$ in floating point arithmetic
- In exact math it is true that from $1+\varepsilon = 1$ it follows that $0+\varepsilon = 0$ and vice versa. In floating point computations this is not true
- Many of you used the right algorithm and used the first value or which $1+\varepsilon = 1$ as the result. This is half the desired quantity.
- Some did not divide start with 1.0 but by other numbers. E.g. 0.1 is not represented exactly in floating point arithmetic
- Recipe for calculation:

```
Set ϵ = 1.0;
while 1.0 + ϵ/2.0 > 1.0 do
    ϵ = ϵ/2.0
end
```

Normalized floating point number

- IEEE 754 32 bit floating point number – normally the same as C++ `float` 
- Storage layout for a normalized number ($d_0 = 1$)
  - bit 0: sign, $0 \rightarrow +$, $\quad 1 \rightarrow -$
  - bit 1…8: $r = 8$ exponent bits, value $e + 2^{r-1} - 1 = 127$ is stored
    $\Rightarrow$ no need for sign bit in exponent
  - bit 9…31: $t = 23$ mantissa bits $d_1 \ldots d_{23}$
  - $d_0 = 1$ not stored ≡ "hidden bit"
- Examples

1	0_01111111_00000000000000000000000	$e = 0$, stored 127
2	0_10000000_00000000000000000000000	$e = 1$, stored 128
0.5	0_01111110_00000000000000000000000	$e = -1$, stored 126
0.1	0_01111011_10011001100110011001101	infinite periodic
0	0_00000000_00000000000000000000000	

- Numbers which are exactly represented in decimal system may not be exactly represented in binary system.

How Addition $1+\epsilon$ works ?

- 1. Adjust exponent of number to be added:
  - Until both exponents are equal, add one to exponent, shift mantissa to right by one bit
- 2. Add both numbers
- 3. Normalize result

We have at maximum t bit shifts of normalized mantissa until mantissa becomes 0, so $\epsilon = 2^{-t}$.

Data of IEEE 754 floating point representations

	size	t	r	ϵ
`float`	32	23	8	1.1920928955078125e-07
`double`	64	53	11	2.2204460492503131e-16
`long double`	128	63	15	1.0842021724855044e-19

- Floating point format not standardized by language but by IEEE comitee
- Implementation of `long double` varies, may even be the same as `double`, or may be significantly slower
- Intended answer: sum in reverse order. Start with adding up many small values which would be cancelled out if added to an already large sum value.
- Results for float:

n	forward sum	forward sum error	reverse sum	reverse sum error
10	1.5497677326202392e+00	9.51664447784423828e-02	1.5497677326202392e+00	9.51664447784423828e-02
100	1.6349840164184570e+00	9.95016098022460937e-03	1.6349838972091674 8e+00	9.95028018951416015e-03
1000	1.6439348459243774e+00	9.99331474304199218e-04	1.6439344882965087 8e+00	9.99689102172851562e-04
10000	1.6447253227233886e+00	2.08854675292968750e-04	1.6448340415954589 8e+00	1.00135803222656250e-04
100000	1.6447253227233886e+00	2.08854675292968750e-04	1.6449240446090698 2e+00	1.01327896118164062e-05
1000000	1.6447253227233886e+00	2.08854675292968750e-04	1.6449329853057861 3e+00	1.19209289550781250e-06
10000000	1.6447253227233886e+00	2.08854675292968750e-04	1.6449339389801025 3e+00	2.38418579101562500e-07
100000000	1.6447253227233886e+00	2.08854675292968750e-04	1.6449340581893920 8e+00	1.19209289550781250e-07

- No gain in accuracy for forward sum for $n > 10000$
- `long double` mostly not a good option

Matrix + Vector norms

- Vector norms: let $x = (x_i) \in \mathbb{R}^n$
  - $||x||_1 = \sum_i =^n |x_i|$: sum norm, l_1-norm
  - $||x||_2 = \sqrt{\sum_{i=1}^n x_i^2}$: Euclidean norm, l_2-norm
  - $||x||_\infty = \max_{i=1...n} |x_i|$: maximum norm, l_∞-norm
- Matrix $A = (a_{ij}) \in \mathbb{R}^n \times \mathbb{R}^n$
  - Representation of linear operator $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^n$ defined by $\mathcal{A} : x \mapsto y = Ax$ with

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

  - Induced matrix norm:

$$\begin{aligned} ||A||_v &= \max_{x \in \mathbb{R}^n, x \neq 0} \frac{||Ax||_v}{||x||_v} \\ &= \max_{x \in \mathbb{R}^n, ||x||_v = 1} \frac{||Ax||_v}{||x||_v} \end{aligned}$$

Matrix norms

- $||A||_1 = \max_{j=1...n} \sum_{i=1}^n |a_{ij}|$ maximum of column sums
- $||A||_\infty = \max_{i=1...n} \sum_{j=1}^n |a_{ij}|$ maximum of row sums
- $||A||_2 = \sqrt{\lambda_{max}}$ with λ_{max}: largest eigenvalue of $A^T A$.

Matrix condition number and error propagation

Problem: solve $Ax = b$, where b is inexact.

$$A(x + \Delta x) = b + \Delta b.$$

Since $Ax = b$, we get $A\Delta x = \Delta b$. From this,

$$\left\{ \begin{array}{ll} \Delta x & = A^{-1}\Delta b \\ Ax & = b \end{array} \right\} \Rightarrow \left\{ \begin{array}{ll} ||A|| \cdot ||x|| & \geq ||b|| \\ ||\Delta x|| & \leq ||A^{-1}|| \cdot ||\Delta b|| \end{array} \right.$$

$$\Rightarrow \frac{||\Delta x||}{||x||} \leq \kappa(A) \frac{||\Delta b||}{||b||}$$

where $\kappa(A) = ||A|| \cdot ||A^{-1}||$ is the *condition number* of A.

Approaches to linear system solution

Solve $Ax = b$

Direct methods:

- Deterministic
- Exact up to machine precision
- Expensive (in time and space)

Iterative methods:

- Only approximate
- Cheaper in space and (possibly) time
- Convergence not guaranteed

4.2. Dense matrix problems – direct solvers

Approaches to linear system solution

Let A: $n \times n$ matrix, $b \in \mathbb{R}^n$.

Solve $Ax = b$

- Direct methods:
  - Exact
    - up to machine precision
    - condition number
  - Expensive (in time and space)
    - where does this matter ?
- Iterative methods:
  - Only approximate
    - with good convergence and proper accuracy control, results are not worse than for direct methods
  - May be cheaper in space and (possibly) time
  - Convergence guarantee is problem dependent and can be tricky

Complexity: "big O notation"

- Let $f, g : \mathbb{V} \to \mathbb{R}^+$ be some functions, where $\mathbb{V} = \mathbb{N}$ or $\mathbb{V} = \mathbb{R}$.

  We write
  $$f(x) = O(g(x)) \quad (x \to \infty)$$
  if there exist a constant $C > 0$ and $x_0 \in \mathbb{V}$ such that
  $$\forall x > x_0, \quad |f(x)| \le C|g(x)|$$

- Often, one skips the part "$(x \to \infty)$"
- Examples:
  - Addition of two vectors: $O(n)$
  - Matrix-vector multiplication (for matrix where all entries are assumed to be nonzero): $O(n^2)$

Really bad example of direct method

Solve $Ax = b$ by Cramer's rule

$$x_i = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1i-1} & b_1 & a_{1i+1} & \dots & a_{1n} \\ a_{21} & & \dots & & b_2 & & \dots & a_{2n} \\ \vdots & & & & \vdots & & & \vdots \\ a_{n1} & & \dots & & b_n & & \dots & a_{nn} \end{vmatrix} / |A| \quad (i = 1 \dots n)$$

This takes $O(n!)$ operations...

Gaussian elimination

- Essentially the only feasible direct solution method
- Solve $Ax = b$ with square matrix A.
- While formally, the algorithm is always the same, its implementation depends on
  - data structure to store matrix
  - possibility to ignore zero entries for matrices with many zeroes
  - sorting of elements

Gaussian elemination: pass 1

$$\begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix} x = \begin{pmatrix} 16 \\ 26 \\ -19 \end{pmatrix}$$

Step 1: $\text{equation}_2 \leftarrow \text{equation}_2 - 2\,\text{equation}_1$
$\text{equation}_3 \leftarrow \text{equation}_3 - \frac{1}{2}\,\text{equation}_1$

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -12 & 2 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -27 \end{pmatrix}$$

Step 2: $\text{equation}_3 \leftarrow \text{equation}_3 - 3\,\text{equation}_2$

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix}$$

Gaussian elimination: pass 2

Solve upper triangular system

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & 0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix}$$

$$\begin{aligned} -4x_3 &= -9 && && \Rightarrow x_3 = \frac{9}{4} \\ -4x_2 + 2x_3 &= -6 && \Rightarrow -4x_2 = -\frac{21}{2} && \Rightarrow x_2 = \frac{21}{8} \\ 6x_1 - 2x_2 + 2x_3 &= 2 && \Rightarrow 6x_1 = 2 + \frac{21}{4} - \frac{18}{4} = \frac{11}{4} && \Rightarrow x_1 = \frac{11}{4} \end{aligned}$$

LU factorization

Pass 1 expressed in matrix operation

$$L_1 A x = \begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -12 & 2 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -27 \end{pmatrix} = L_1 b, \qquad L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix}$$

$$L_2 L_1 A x = \begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix} = L_2 L_1 b, \qquad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

- Let $L = L_1^{-1} L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & 3 & 1 \end{pmatrix}$, $U = L_2 L_1 A$. Then $A = LU$
- Inplace operation. Diagonal elements of L are always 1, so no need to store them ⇒ work on storage space for A and overwrite it.

LU factorization

Solve $Ax = b$

- Pass 1: factorize $A = LU$ such that L, U are lower/upper triangular
- Pass 2: obtain $x = U^{-1} L^{-1} b$ by solution of lower/upper triangular systems
  - 1. solve $L\tilde{x} = b$
  - 2. solve $Ux = \tilde{x}$
- We never calculate A^{-1} as this would be more expensive

Problem example

- Consider $\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\epsilon \\ 1 \end{pmatrix}$
- Solution: $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Machine arithmetic: Let $\epsilon << 1$ such that $1 + \epsilon = 1$.
- Equation system in machine arithmetic:
  $1 \cdot \epsilon + 1 \cdot 1 = 1 + \epsilon$
  $1 \cdot 1 + 1 \cdot 1 = 2$
- Still fulfilled!

Problem example II: Gaussian elimination

- Ordinary elimination: equation$_2$ ← equation$_2$ − $\frac{1}{\epsilon}$ equation$_1$
  $\begin{pmatrix} \epsilon & 1 \\ 0 & 1 - \frac{1}{\epsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\epsilon \\ 2 - \frac{1+\epsilon}{\epsilon} \end{pmatrix}$
- In exact arithmetic:
  $$\Rightarrow x_2 = \frac{1 - \frac{1}{\epsilon}}{1 - \frac{1}{\epsilon}} = 1 \Rightarrow x_1 = \frac{1 + \epsilon - x_2}{\epsilon} = 1$$
- In floating point arithmetic: $1 + \epsilon = 1$, $1 - \frac{1}{\epsilon} = -\frac{1}{\epsilon}$, $2 - \frac{1}{\epsilon} = -\frac{1}{\epsilon}$: $\begin{pmatrix} \epsilon & 1 \\ 0 & -\frac{1}{\epsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -\frac{1}{\epsilon} \end{pmatrix}$

  $\Rightarrow x_2 = 1 \Rightarrow \epsilon x_1 + 1 = 1 \Rightarrow x_1 = 0$

Problem example III: Partial Pivoting

- Before elimination step, look at the element with largest absolute value in current column and put the corresponding row "on top" as the "pivot"
- This prevents near zero divisions and increases stability

$$\begin{pmatrix}1 & 1\\ \epsilon & 1\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix}=\begin{pmatrix}2\\ 1+\epsilon\end{pmatrix}\Rightarrow\begin{pmatrix}1 & 1\\ 0 & 1-\epsilon\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix}=\begin{pmatrix}2\\ 1-\epsilon\end{pmatrix}$$

- Independent of ϵ:

$$x_2=\frac{1-1\epsilon}{1-\epsilon}=1,\qquad x_1=2-x_2=1$$

- Instead of A, factorize PA: $PA=LU$, where P is a permutation matrix which can be encoded using an integer vector

Gaussian elimination and LU factorization

- Full pivoting: in addition to row exchanges, perform column exchanges to ensure even larger pivots. Seldomly used in practice.
- Gaussian elimination with partial pivoting is the "working horse" for direct solution methods
- Complexity of LU-Factorization: $O(N^3)$, some theoretically better algorithms are known with e.g. $O(N^{2.736})$
- Complexity of triangular solve: $O(N^2)$
  $\Rightarrow$ overall complexity of linear system solution is $O(N^3)$

Cholesky factorization

- $A=LL^T$ for symmetric, positive definite matrices

BLAS, LAPACK

- BLAS: Basic Linear Algebra Subprograms http://www.netlib.org/blas/
  - Level 1 - vector-vector: $\mathbf{y}\leftarrow\alpha\mathbf{x}+\mathbf{y}$
  - Level 2 - matrix-vector: $\mathbf{y}\leftarrow\alpha A\mathbf{x}+\beta\mathbf{y}$
  - Level 3 - matrix-matrix: $C\leftarrow\alpha AB+\beta C$
- LAPACK: Linear Algebra PACKage http://www.netlib.org/lapack/
  - Linear system solution, eigenvalue calculation etc.
  - `dgetrf`: LU factorization
  - `dgetrs`: LU solve
- Used in overwhelming number of codes (e.g. matlab, scipy etc.). Also, C++ matrix libraries use these routines. Unless there is special need, they should be used.
- Reference implementations in Fortran, but many more implementations available which carefully work with cache lines etc.

Matrices from PDEs

- So far, we assumed that matrices are stored in a two-dimensional, $n\times n$ array of numbers
- This kind of matrices are also called *dense* matrices
- As we will see, matrices from PDEs (can) have a number of structural properties one can take advantage of when storing a matrix and solving the linear system

1D heat conduction

- v_L, v_R: ambient temperatures, α: heat transfer coefficient
- Second order boundary value problem in $\Omega = [0,1]$:

$$\begin{aligned} -u''(x) &= f(x) && \text{in}\Omega \\ -u'(0) + \alpha(u(0) - v_L) &= 0 \\ u'(1) + \alpha(u(1) - v_R) &= 0 \end{aligned}$$

- Let $h = \frac{1}{n-1}$, $x_i = x_0 + (i-1)h$ $i = 1 \ldots n$ be discretization points, let u_i approximations for $u(x_i)$ and $f_i = f(x_i)$
- Finite difference approximation:

$$\begin{aligned} -u'(0) + \alpha(u(0) - v_L) &\approx \frac{1}{h}(u_0 - u_1) + \alpha(u_0 - v_L) \\ -u''(x_i) - f(x_i) &\approx \frac{1}{h^2}(u_{i+1} - 2u_i - u_{i-1}) - f_i && (i = 2 \ldots n-1) \\ u'(1) + \alpha(u(1) - v_R) &\approx \frac{1}{h}(u_n - u_{n-1}) + \alpha(u_n - v_R) \end{aligned}$$

1D heat conduction: discretization matrix

- equations $2 \ldots n-1$ multiplied by h
- only nonzero entries written

$$\begin{pmatrix} \alpha + \frac{1}{h} & -\frac{1}{h} & & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} \\ & & & & -\frac{1}{h} & \frac{1}{h} + \alpha \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} \alpha v_L \\ hf_2 \\ hf_3 \\ \vdots \\ hf_{N-2} \\ hf_{N-1} \\ \alpha v_R \end{pmatrix}$$

- Each row contains ≤ 3 elements
- Only $3n-2$ of n^2 elements are non-zero

General tridiagonal matrix

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & a_n & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{pmatrix}$$

- To store matrix, it is sufficient to store only nonzero elements in three one-dimensional arrays for a_i, b_i, c_i, respectively

Gaussian elimination for tridiagonal systems

Gaussian elimination using arrays a, b, c as matrix storage ?

From what we have seen, this question arises in a quite natural way, and historically, the answer has been given several times

- TDMA (tridiagonal matrix algorithm)
- “Thomas algorithm” (Llewellyn H. Thomas, 1949 (?))
- “Progonka method” (from Russian "run through"; Gelfand, Lokutsievski, 1952, published 1960)

Progonka: derivation

- $a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i \quad (i = 1 \dots n)$; $a_1 = 0$, $c_N = 0$
- For $i = 1 \dots n-1$, assume there are coefficients α_i, β_i such that $u_i = \alpha_{i+1} u_{i+1} + \beta_{i+1}$.
- Then, we can express u_{i-1} and u_i via u_{i+1}:
  $(a_i \alpha_i \alpha_{i+1} + b_i \alpha_{i+1} + c_i) u_{i+1} + a_i \alpha_i \beta_{i+1} + a_i \beta_i + b_i \beta_{i+1} - f_i = 0$
- This is true independently of u if

$$\begin{cases} a_i \alpha_i \alpha_{i+1} + b_i \alpha_{i+1} + c_i & = 0 \\ a_i \alpha_i \beta_{i+1} + a_i \beta_i + b_i \beta_{i+1} - f_i & = 0 \end{cases}$$

- or for $i = 1 \dots n-1$:

$$\begin{cases} \alpha_{i+1} & = -\frac{c_i}{a_i \alpha_i + b_i} \\ \beta_{i+1} & = \frac{f_i - a_i \beta_i}{a_i \alpha_i + b_i} \end{cases}$$

Progonka: realization

- Forward sweep:

$$\begin{cases} \alpha_2 & = -\frac{c_1}{b_1} \\ \beta_2 & = \frac{f_i}{b_1} \end{cases}$$

  for $i = 2 \dots n-1$

$$\begin{cases} \alpha_{i+1} & = -\frac{c_i}{a_i \alpha_i + b_i} \\ \beta_{i+1} & = \frac{f_i - a_i \beta_i}{a_i \alpha_i + b_i} \end{cases}$$

- Backward sweep:

$$u_n = \frac{f_n - a_n \beta_n}{a_n \alpha_n + b_n}$$

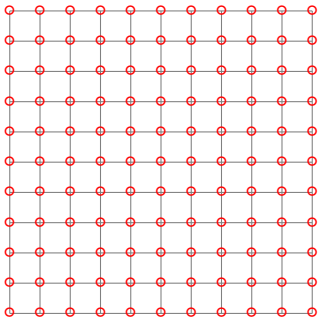
  for $n-1 \dots 1$:

$$u_i = \alpha_{i+1} u_{i+1} + \beta_{i+1}$$

Progonka: properties

- n unknowns, one forward sweep, one backward sweep
  $\Rightarrow O(n)$ operations vs. $O(n^3)$ for algorithm using full matrix
- No pivoting $\Rightarrow$ stability issues
  - Stability for diagonally dominant matrices ($|b_i| > |a_i| + |c_i|$)
  - Stability for symmetric positive definite matrices

2D finite difference grid



- Each discretization point has not more then 4 neighbours
- Matrix can be stored in five diagonals,
  LU factorization not anymore $\equiv$ "fill-in"
- Certain iterative methods can take advantage of the regular and hierachical structure (multigrid) and are able to solve system in $O(n)$ operations
- Another possibility: fast Fourier transform with $O(n \log n)$ operations

Sparse matrices

- Tridiagonal and five-diagonal matrices can be seen as special cases of *sparse matrices*
- Generally they occur in finite element, finite difference and finite volume discretizations of PDEs on structured and unstructured grids
- Definition: Regardless of number of unknowns n, the number of non-zero entries per row remains limited by n_r
- If we find a scheme which allows to store only the non-zero matrix entries, we would need $nn_r = O(n)$ storage locations instead of n^2
- The same would be true for the matrix-vector multiplication if we program it in such a way that we use every nonzero element just once: matrix-vector multiplication would use $O(n)$ instead of $O(n^2)$ operations

Sparse matrix questions

- What is a good storage format for sparse matrices?
- Is there a way to implement Gaussian elimination for general sparse matrices which allows for linear system solution with $O(n)$ operation ?
- Is there a way to implement Gaussian elimination *with pivoting* for general sparse matrices which allows for linear system solution with $O(n)$ operations?
- Is there *any algorithm* for sparse linear system solution with $O(n)$ operations?

4.3. Sparse matrix problems – direct solvers

Coordinate (triplet) format

- Store all nonzero elements along with their row and column indices
- One real, two integer arrays, length = nnz= number of nonzero elements

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	12.	9.	7.	5.	1.	2.	11.	3.	6.	4.	8.	10.
JR	5	3	3	2	1	1	4	2	3	2	3	4
JC	5	5	3	4	1	4	4	1	1	2	4	3

Y.Saad, Iterative Methods, p.92

Compressed Row Storage (CRS) format

(aka Compressed Sparse Row (CSR) or IA-JA etc.)

- real array `AA`, length nnz, containing all nonzero elements row by row
- integer array `JA`, length nnz, containing the column indices of the elements of `AA`
- integer array `IA`, length n+1, containing the start indizes of each row in the arrays `IA` and `JA` and `IA(n+1)=nnz+1`

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

Y.Saad, Iterative Methods, p.93

- Used in most sparse matrix solver packages

The big schism

- Should array indices count from zero or from one ?
- Fortran, Matlab, Julia count from one
- C/C++, python count from zero
- It matters when passing index arrays to sparse matrix packages

CRS again

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

```
AA: 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12.
JA: 0 3 0 1 3 0 2 3 4 2 3 4
IA: 0 2 4 0 11 12
```

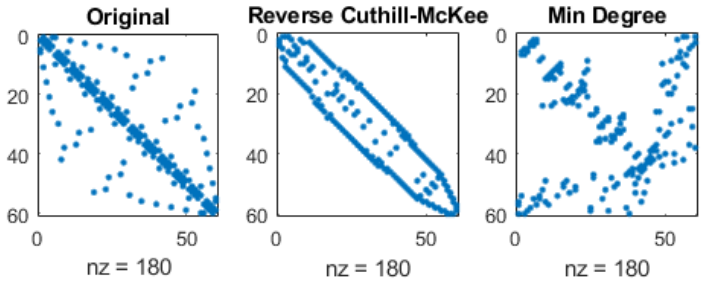
- some package APIs provide the possibility to specify array offset
- index shift is not very expensive compared to the rest of the work

Sparse direct solvers

- Sparse direct solvers implement Gaussian elimination with different pivoting strategies
  - UMFPACK
  - Pardiso (omp + MPI parallel)
  - SuperLU
  - MUMPS (MPI parallel)
  - Pastix
- Quite efficient for 1D/2D problems
- They suffer from *fill-in*:
  - Memory usage
  - Operation count

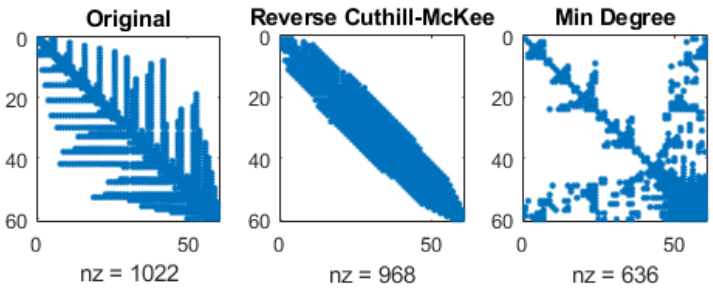
Sparse direct solvers: influence of reordering

- Sparsity patterns for original matrix with three different orderings of unknowns unknowns:



https://de.mathworks.com

- Sparsity patterns for corresponding LU factorizations unknowns:



https://de.mathworks.com

Sparse direct solvers: solution steps (Saad Ch. 3.6)

1. Pre-ordering
   - Decrease amount of non-zero elements generated by fill-in by re-ordering of the matrix
   - Several, graph theory based heuristic algorithms exist
2. Symbolic factorization
   - If pivoting is ignored, the indices of the non-zero elements are calculated and stored
   - Most expensive step wrt. computation time
3. Numerical factorization
   - Calculation of the numerical values of the nonzero entries
   - Not very expensive, once the symbolic factors are available
4. Upper/lower triangular system solution
   - Fairly quick in comparison to the other steps

- Separation of steps 2 and 3 allows to save computational costs for problems where the sparsity structure remains unchanged, e.g. time dependent problems on fixed computational grids
- With pivoting, steps 2 and 3 have to be performed together
- Instead of pivoting, *iterative refinement* may be used in order to maintain accuracy of the solution

Sparse direct solvers: Complexity

- Complexity estimates depend on storage scheme, reordering etc.
- Sparse matrix - vector multiplication has complexity $O(N)$
- Some estimates can be given for from graph theory for discretizations of heat eqauation with $N = n^d$ unknowns on close to cubic grids in space dimension d
  - sparse LU factorization:

d	work	storage
1	$O(N) \mid O(n)$	$O(N) \mid O(n)$
2	$O(N^{\frac{3}{2}}) \mid O(n^3)$	$O(N \log N) \mid O(n^2 \log n)$
3	$O(N^2) \mid O(n^6)$	$O(N^{\frac{4}{3}}) \mid O(n^4)$

  - triangular solve: work dominated by storage complexity

d	work
1	$O(N) \mid O(n)$
2	$O(N \log N) \mid O(n^2 \log n)$
3	$O(N^{\frac{4}{3}}) \mid O(n^4)$

Source: J. Poulson, PhD thesis,http://hdl.handle.net/2152/ETD-UT-2012-12-6622

5. Iterative solution of linear systems of equations

5.1. Definition and convergence criteria

Elements of iterative methods (Saad Ch.4)

Let $V = \mathbb{R}^n$ be equipped with the inner product $(\cdot,\cdot)$, let A be an $n \times n$ nonsingular matrix.

Solve $Au = b$ iteratively

- Preconditioner: a matrix $M \approx A$ "approximating" the matrix A but with the property that the system $Mv = f$ is easy to solve
- Iteration scheme: algorithmic sequence using M and A which updates the solution step by step

Simple iteration with preconditioning

Idea: $A\hat{u} = b \Rightarrow$

$$\hat{u} = \hat{u} - M^{-1}(A\hat{u} - b)$$

$\Rightarrow$ iterative scheme

$$u_{k+1} = u_k - M^{-1}(Au_k - b) \quad (k = 0, 1\ldots)$$

1. Choose initial value u_0, tolerance ε, set $k = 0$
2. Calculate *residuum* $r_k = Au_k - b$
3. Test convergence: if $||r_k|| < \varepsilon$ set $u = u_k$, finish
4. Calculate *update*: solve $Mv_k = r_k$
5. Update solution: $u_{k+1} = u_k - v_k$, set $k = i + 1$, repeat with step 2.

The Jacobi method

- Let $A = D - E - F$, where D: main diagonal, E: negative lower triangular part F: negative upper triangular part
- Preconditioner: $M = D$, where D is the main diagonal of $A \Rightarrow$

$$u_{k+1,i} = u_{k,i} - \frac{1}{a_{ii}}\left(\sum_{j=1\ldots n} a_{ij}u_{k,j} - b_i\right) \quad (i = 1\ldots n)$$

- Equivalent to the succesive (row by row) solution of

$$a_{ii}u_{k+1,i} + \sum_{j=1\ldots n, j\neq i} a_{ij}u_{k,j} = b_i \quad (i = 1\ldots n)$$

- Already calculated results not taken into account
- Alternative formulation with $A = M - N$:

$$\begin{aligned} u_{k+1} &= D^{-1}(E+F)u_k + D^{-1}b \\ &= M^{-1}Nu_k + M^{-1}b \end{aligned}$$

- Variable ordering does not matter

The Gauss-Seidel method

- Solve for main diagonal element row by row
- Take already calculated results into account

$$a_{ii}u_{k+1,i} + \sum_{j<i} a_{ij}u_{k+1,j} + \sum_{j>i} a_{ij}u_{k,j} = b_i \qquad (i = 1 \ldots n)$$
$$(D-E)u_{k+1} - Fu_k = b$$

- May be it is faster
- Variable order probably matters
- Preconditioners: forward $M = D - E$, backward: $M = D - F$
- Splitting formulation: $A = M - N$
  forward: $N = F$, backward: $M = E$
- Forward case:

$$\begin{aligned} u_{k+1} &= (D-E)^{-1}Fu_k + (D-E)^{-1}b \\ &= M^{-1}Nu_k + M^{-1}b \end{aligned}$$

SOR and SSOR

- SOR: Successive overrelaxation: solve $\omega A = \omega B$ and use splitting

$$\omega A = (D - \omega E) - (\omega F + (1 - \omega D))$$
$$M = \frac{1}{\omega}(D - \omega E)$$

  leading to

$$(D - \omega E)u_{k+1} = (\omega F + (1 - \omega D))u_k + \omega b$$

- SSOR: Symmetric successive overrelaxation

$$(D - \omega E)u_{k+\frac{1}{2}} = (\omega F + (1 - \omega D))u_k + \omega b$$
$$(D - \omega F)u_{k+1} = (\omega E + (1 - \omega D))u_{k+\frac{1}{2}} + \omega b$$

- Preconditioner:

$$M = \frac{1}{\omega(2-\omega)}(D - \omega E)D^{-1}(D - \omega F)$$

- Gauss-Seidel and symmetric Gauss-Seidel are special cases for $\omega = 1$.

Block methods

- Jacobi, Gauss-Seidel, (S)SOR methods can as well be used block-wise, based on a partition of the system matrix into larger blocks,
- The blocks on the diagonal should be square matrices, and invertible
- Interesting variant for systems of partial differential equations, where multiple species interact with each other

Convergence

- Let $\hat{u}$ be the solution of $Au = b$.
- Let $e_k = u_j - \hat{u}$ be the error of the k-th iteration step

$$\begin{aligned} u_{k+1} &= u_k - M^{-1}(Au_k - b) \\ &= (I - M^{-1}A)u_k + M^{-1}b \\ u_{k+1} - \hat{u} &= u_k - \hat{u} - M^{-1}(Au_k - A\hat{u}) \\ &= (I - M^{-1}A)(u_k - \hat{u}) \\ &= (I - M^{-1}A)^k(u_0 - \hat{u}) \end{aligned}$$

resulting in

$$e_{k+1} = (I - M^{-1}A)^k e_0$$

- So when does $(I - M^{-1}A)^k$ converge to zero for $k \to \infty$?

Jordan canonical form of a matrix A

- λ_i $(i = 1 \dots p)$: eigenvalues of A
- $\sigma(A) = \{\lambda_1 \dots \lambda_p\}$: spectrum of A
- μ_i: algebraic multiplicity of λ_i:
  multiplicity as zero of the characteristic polynomial $\det(A - \lambda I)$
- γ_i geometric multiplicity of λ_i: dimension of $\mathrm{Ker}(A - \lambda I)$
- l_i: index of the eigenvalue: the smallest integer for which $\mathrm{Ker}(A - \lambda I)^{l_i+1} = \mathrm{Ker}(A - \lambda I)^{l_i}$
- $l_i \leq \mu_i$

Theorem (Saad, Th. 1.8) Matrix A can be transformed to a block diagonal matrix consisting of p diagonal blocks, each associated with a distinct eigenvalue λ_i.

- Each of these diagonal blocks has itself a block diagonal structure consisting of γ_i *Jordan blocks*
- Each of the Jordan blocks is an upper bidiagonal matrix of size not exceeding l_i with λ_i on the diagonal and 1 on the first upper diagonal.

Jordan canonical form of a matrix II

$$X^{-1}AX = J = \begin{pmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_p \end{pmatrix}$$

$$J_i = \begin{pmatrix} J_{i,1} & & & \\ & J_{i,2} & & \\ & & \ddots & \\ & & & J_{i,\gamma_i} \end{pmatrix}$$

$$J_{i,k} = \begin{pmatrix} \lambda_i & 1 & & \\ & \lambda_i & 1 & \\ & & \ddots & 1 \\ & & & \lambda_i \end{pmatrix}$$

Each $J_{i,k}$ is of size l_i and corresponds to a different eigenvector of A.

Spectral radius and convergence

Definition The spectral radius $\rho(A)$ is the largest absolute value of any eigenvalue of A: $\rho(A) = \max_{\lambda \in \sigma(A)} |\lambda|$.

Theorem (Saad, Th. 1.10) $\lim_{k\to\infty} A^k = 0 \Leftrightarrow \rho(A) < 1$.

Proof, $\Rightarrow$: Let u_i be a unit eigenvector associated with an eigenvalue λ_i. Then

$$\begin{aligned} Au_i &= \lambda_i u_i \\ A^2 u_i &= \lambda_i A_i u_i = \lambda^2 u_i \\ &\vdots \\ A^k u_i &= \lambda^k u_i \\ \text{therefore} \quad ||A^k u_i||_2 &= |\lambda^k| \\ \text{and} \quad \lim_{k\to\infty} |\lambda^k| &= 0 \end{aligned}$$

so we must have $\rho(A) < 1$

Spectral radius and convergence II

Proof, $\Leftarrow$: Jordan form $X^{-1}AX = J$. Then $X^{-1}A^kX = J^k$.
Sufficient to regard Jordan block $J_i = \lambda_i I + E_i$ where $|\lambda_i| < 1$ and $E_i^{l_i} = 0$.
Let $k \geq l_i$. Then

$$J_i^k = \sum_{j=0}^{l_{i-1}} \binom{k}{j} \lambda^{k-j} E_i^j$$

$$||J_i||^k \leq \sum_{j=0}^{l_{i-1}} \binom{k}{j} |\lambda|^{k-j} ||E_i||^j$$

One has $\binom{k}{j} = \frac{k!}{j!(k-j)!} = \sum_{i=0}^{j} \begin{bmatrix} j \\ i \end{bmatrix} \frac{k^i}{j!}$ is a polynomial of degree j in k
where the Stirling numbers of the first kind are given by
$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1, \quad \begin{bmatrix} j \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ j \end{bmatrix} = 0, \quad \begin{bmatrix} j+1 \\ i \end{bmatrix} = j \begin{bmatrix} j \\ i \end{bmatrix} + \begin{bmatrix} j \\ i-1 \end{bmatrix}$.
Thus, $\binom{k}{j} |\lambda|^{k-j} \to 0 \, (k \to \infty)$ as exponential decay beats polynomial growth □.

Corollary from proof

Theorem (Saad, Th. 1.12)

$$\lim_{k\to\infty} ||A^k||^{\frac{1}{k}} = \rho(A)$$

□

Back to iterative methods

Sufficient condition for convergence: $\rho(I - M^{-1}A) < 1$.

Convergence rate

Assume λ with $|\lambda| = \rho(I - M^{-1}A) < 1$ is the largest eigenvalue and has a single Jordan block of size l. Then the convergence rate is dominated by this Jordan block, and therein by the term with the lowest possible power in λ which due to $E^l = 0$ is

$$\lambda^{k-l+1}\binom{k}{l-1}E^{l-1}$$

$$||(I - M^{-1}A)^k(u_0 - \hat{u})|| = O\left(|\lambda^{k-l+1}|\binom{k}{l-1}\right)$$

and the "worst case" convergence factor ρ equals the spectral radius:

$$\begin{aligned}\rho &= \lim_{k\to\infty}\left(\max_{u_0}\frac{||(I - M^{-1}A)^k(u_0 - \hat{u})||}{||u_0 - \hat{u}||}\right)^{\frac{1}{k}} \\ &= \lim_{k\to\infty}||(I - M^{-1}A)^k||^{\frac{1}{k}} \\ &= \rho(I - M^{-1}A)\end{aligned}$$

Depending on u_0, the rate may be faster, though

Richardson iteration, sufficient criterion for convergence

Assume A has positive real eigenvalues $0 < \lambda_{min} \le \lambda_i \le \lambda_{max}$, e.g. A symmetric, positive definite (spd),

- Let $\alpha > 0$, $M = \frac{1}{\alpha}I \Rightarrow I - M^{-1}A = I - \alpha A$
- Then for the eigenvalues μ_i of $I - \alpha A$ one has:
  $1 - \alpha\lambda_{max} \le \mu_i \le 1 - \alpha\lambda_{min}$
  and $\mu_i < 1$ due to $\lambda_{min} > 0$
- We also need $1 - \alpha\lambda_{max} > -1 \Rightarrow 0 < \alpha < \frac{2}{\lambda_{max}}$.

Theorem. The Richardson iteration converges for any α with $0 < \alpha < \frac{2}{\lambda_{max}}$.

The convergence rate is $\rho = \max(|1 - \alpha\lambda_{max}|, |1 - \alpha\lambda_{min}|)$.

□

Richardson iteration, choice of optimal parameter

- We know that

$$-(1-\lambda_{max}\alpha) > -(1-\lambda_{min}\alpha)$$
$$+(1-\lambda_{min}\alpha) > +(1-\lambda_{max}\alpha)$$

- Therefore, in reality we have $\rho = \max((1-\alpha\lambda_{max}), -(1-\alpha\lambda_{min}))$.

- The first curve is monotonically decreasing, the second one increases, so the minimum must be at the intersection

$$1-\alpha\lambda_{max} = -1+\alpha\lambda_{min}$$
$$2 = \alpha(\lambda_{max}+\lambda_{min})$$

Theorem. The optimal parameter is $\alpha_{opt} = \frac{2}{\lambda_{min}+\lambda_{max}}$.
For this parameter, the convergence factor is

$$\rho_{opt} = \frac{\lambda_{max}-\lambda_{min}}{\lambda_{max}+\lambda_{min}} = \frac{\kappa-1}{\kappa+1}$$

where $\kappa = \kappa(A)\frac{\lambda_{max}}{\lambda_{min}}$ is the spectral condition number of A. □

Spectral equivalence

Theorem. M, A spd. Assume the *spectral equivalence estimate*

$$0 < \gamma_{min}(Mu,u) \le (Au,u) \le \gamma_{max}(Mu,u)$$

Then for the eigenvalues λ_i of $M^{-1}A$ we have

$$\gamma_{min} \le \lambda_{min} \le \lambda_i \le \lambda_{max} \le \gamma_{max}$$

and $\kappa(M^{-1}A) \le \frac{\gamma_{max}}{\gamma_{\min}}$

Proof. Let the inner product $(\cdot,\cdot)_M$ be defined via $(u,v)_M = (Mu,v)$. In this inner product, $C = M^{-1}A$ is self-adjoint:

$$(Cu,v)_M = (MM^{-1}Au,v) = (Au,v) = (M^{-1}Mu,Av) = (Mu,M^{-1}Av)$$
$$= (u,M^{-1}A)_M = (u,Cv)_M$$

Minimum and maximum eigenvalues can be obtained as Ritz values in the $(\cdot,\cdot)_M$ scalar product

$$\lambda_{min} = \min_{u\neq 0}\frac{(Cu,u)_M}{(u,u)_M} = \min_{u\neq 0}\frac{(Au,u)}{(Mu,u)} \ge \gamma_{min}$$
$$\lambda_{max} = \max_{u\neq 0}\frac{(Cu,u)_M}{(u,u)_M} = \max_{u\neq 0}\frac{(Au,u)}{(Mu,u)} \le \gamma_{max}$$

□

Matrix preconditioned Richardson iteration

M, A spd.

- Scaled Richardson iteration with preconditoner M

$$u_{k+1} = \alpha M^{-1}(Au_k - b)$$

- Spectral equivalence estimate

$$0 < \gamma_{min}(Mu,u) \leq (Au,u) \leq \gamma_{max}(Mu,u)$$

- $\Rightarrow \gamma_{min} \leq \lambda_i \leq \gamma_{max}$
- $\Rightarrow$ optimal parameter $\alpha = \frac{2}{\gamma_{max}+\gamma_{min}}$
- Convergence rate with optimal parameter: $\rho \leq \frac{\kappa(M^{-1}A)-1}{\kappa(M^{-1}A)+1}$
- This is one possible way for convergence analysis which at once gives convergence rates
- But … how to obtain a good spectral estimate for a particular problem ?

Richardson for 1D heat conduction

- Regard the $n \times n$ 1D heat conduction matrix with $h = \frac{1}{n-1}$ and $\alpha = \frac{1}{h}$ (easier to analyze).

$$A = \begin{pmatrix} \frac{2}{h} & -\frac{1}{h} & & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} \\ & & & & -\frac{1}{h} & \frac{2}{h} \end{pmatrix}$$

- Eigenvalues (tri-diagonal Toeplitz matrix):

$$\lambda_i = \frac{2}{h}\left(1 + \cos\left(\frac{i\pi}{n+1}\right)\right) \quad (i = 1 \ldots n)$$

Source: A. Böttcher, S. Grudsky: Spectral Properties of Banded Toeplitz Matrices. SIAM,2005

- Express them in h: $n+1 = \frac{1}{h} + 2 = \frac{1+2h}{h} \Rightarrow$

$$\lambda_i = \frac{2}{h}\left(1 + \cos\left(\frac{ih\pi}{1+2h}\right)\right) \quad (i = 1 \ldots n)$$

Richardson for 1D heat conduction: spectral bounds

- For $i = 1 \ldots n$, the argument of cos is in $(0, \pi)$
- cos is monotonically decreasing in $(0, \pi)$, so we get λ_{max} for $i = 1$ and λ_{min} for $i = n = \frac{1+h}{h}$
- Therefore:

$$\lambda_{max} = \frac{2}{h}\left(1 + \cos\left(\pi \frac{h}{1+2h}\right)\right) \approx \frac{2}{h}\left(2 - \frac{\pi^2 h^2}{2(1+2h)^2}\right)$$
$$\lambda_{min} = \frac{2}{h}\left(1 + \cos\left(\pi \frac{1+h}{1+2h}\right)\right) \approx \frac{2}{h}\left(\frac{\pi^2 h^2}{2(1+2h)^2}\right)$$

  Here, we used the Taylor expansion

$$cos(\delta) = 1 - \frac{\delta^2}{2} + O(\delta^4) \quad (\delta \to 0)$$
$$cos(\pi - \delta) = -1 + \frac{\delta^2}{2} + O(\delta^4) \quad (\delta \to 0)$$

  and $\frac{1+h}{1+2h} = \frac{1+2h}{1+2h} - \frac{h}{1+2h} = 1 - \frac{h}{1+2h}$

Richardson for 1D heat conduction: Jacobi

- The Jacobi preconditioner just multiplies by $\frac{h}{2}$, therefore for $M^{-1}A$:

$$\lambda_{max} \approx 2 - \frac{\pi^2 h^2}{2(1+2h)^2}$$
$$\lambda_{min} \approx \frac{\pi^2 h^2}{2(1+2h)^2}$$

- Optimal parameter: $\alpha = \frac{2}{\lambda_{max} + \lambda_{min}} \approx 1 \; (h \to 0)$
- Good news: this is independent of h resp. n
- No need for spectral estimate in order to work with optimal parameter
- Is this true beyond this special case ?

Richardson for 1D heat conduction: Convergence factor

- Condition number + spectral radius

$$\kappa(M^{-1}A) = \kappa(A) = \frac{4(1+2h)^2}{\pi^2 h^2} - 1$$
$$\rho(I - M^{-1}A) = \frac{\kappa - 1}{\kappa + 1} = 1 - \frac{\pi^2 h^2}{2(1+2h)^2}$$

- Bad news: $\rho \to 1 \quad (h \to 0)$
- Typical situation with second order PDEs:

$$\kappa(A) = O(h^{-2}) \quad (h \to 0)$$
$$\rho(I - D^{-1}A) = 1 - O(h^2) \quad (h \to 0)$$

Iterative solver complexity I

- Solve linear system iteratively until $||e_k|| = ||(I - M^{-1}A)^k e_0|| \leq \epsilon$

$$\begin{aligned} \rho^k e_0 &\leq \epsilon \\ k \ln \rho &< \ln \epsilon - \ln e_0 \\ k \geq k_\rho &= \left\lceil \frac{\ln e_0 - \ln \epsilon}{\ln \rho} \right\rceil \end{aligned}$$

- Assume $\rho < \rho_0 < 1$ independent of h resp. N, A sparse and solution of $Mv = r$ has complexity $O(N)$.
  $\Rightarrow$ Number of iteration steps k_ρ independent of N
  $\Rightarrow$ Overall complexity $O(N)$.

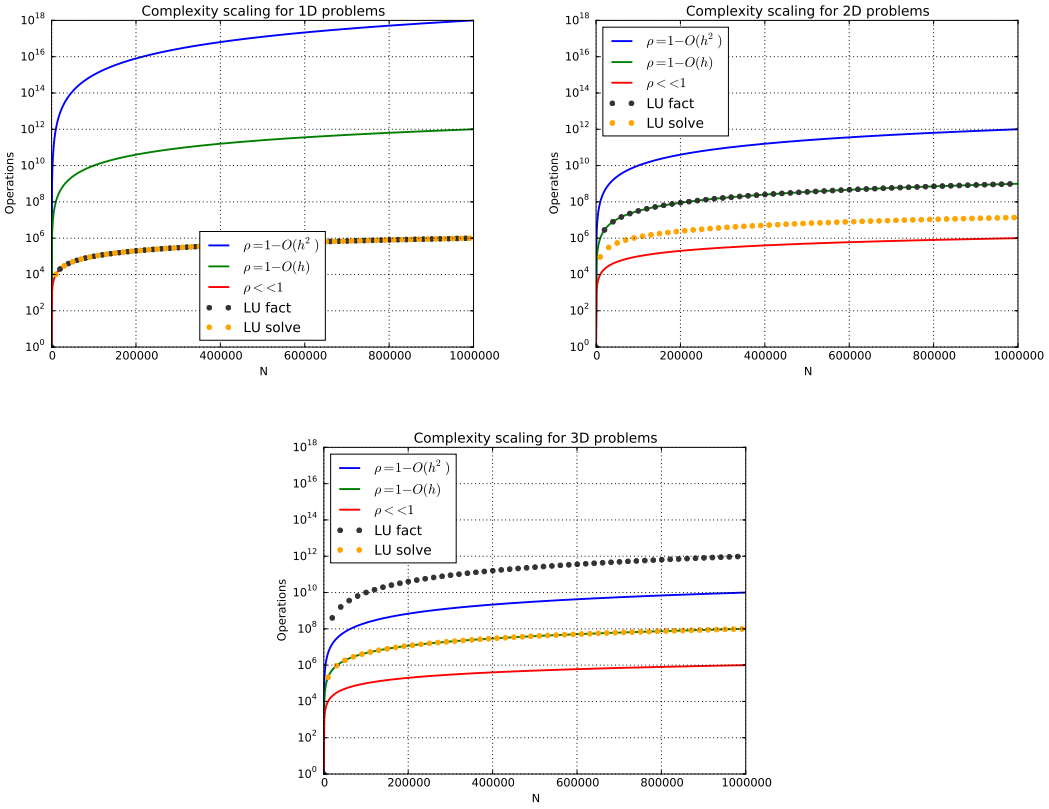
Iterative solver complexity II

- Assume $\rho = 1 - h^\delta \Rightarrow \ln \rho \approx -h^\delta$
- $k = O(h^{-\delta})$
- d: space dimension, then $h \approx N^{-\frac{1}{d}} \Rightarrow k = O(N^{\frac{\delta}{d}})$
- Assume $O(N)$ complexity of one iteration step
  $\Rightarrow$ Overall complexity $O(N^{\frac{d+\delta}{d}})$
- Jacobi: $\delta = 2$, something better with at least $\delta = 1$?

dim	$\rho = 1 - O(h^2)$	$\rho = 1 - O(h)$	LU fact.	LU solve
1	$O(N^3)$	$O(N^2)$	$O(N)$	$O(N)$
2	$O(N^2)$	$O(N^{\frac{3}{2}})$	$O(N^{\frac{3}{2}})$	$O(N \log N)$
3	$O(N^{\frac{5}{3}})$	$O(N^{\frac{4}{3}})$	$O(N^2)$	$O(N^{\frac{4}{3}})$

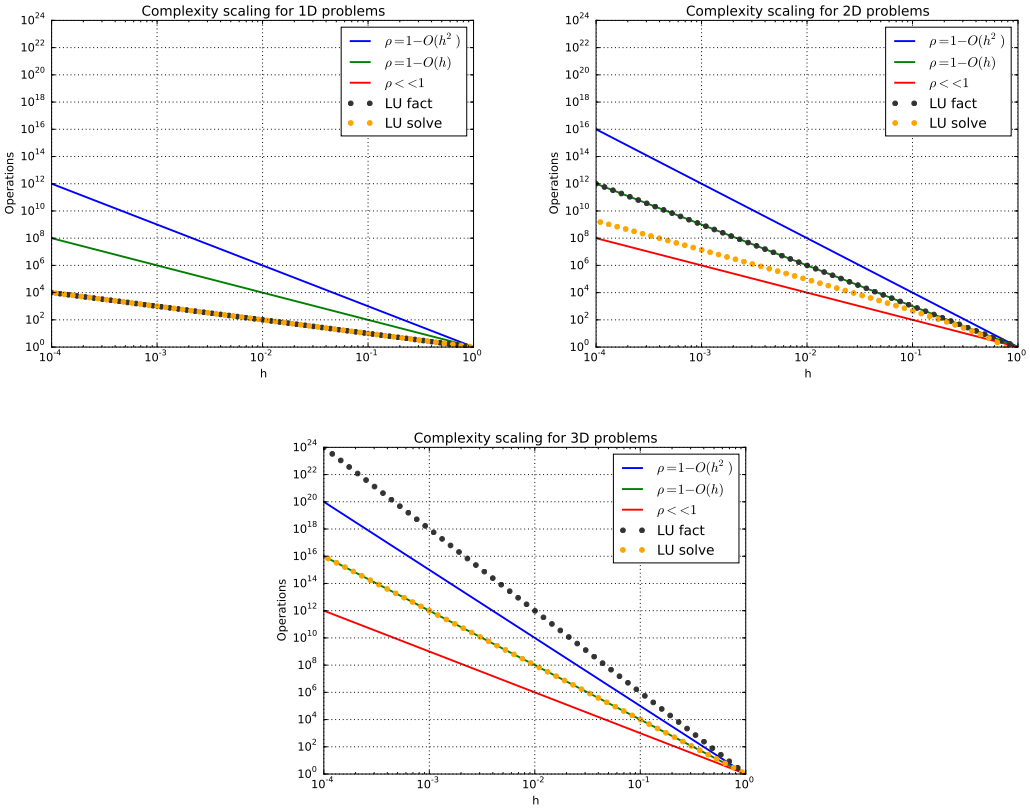
- In 1D, iteration makes not much sense
- In 2D, we can hope for parity
- In 3D, beat sparse matrix solvers with $\rho = 1 - O(h)$?

Solver complexity: scaling with problem size



Scaling with problem size.

Solver complexity: scaling with accuracy



- Accuracy of numerial solutions is proportional to some power of h.
- Amount of operations for to reach a given accuracy.

What could be done ?

- Find a better preconditioner with $\kappa(M^{-1}A) = O(h^{-1})$ or independent of h
- Find a better iterative scheme:
  Assume e.g. $\rho = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$. Let $\kappa = X^2 - 1$ where $X = \frac{2(1+2h)}{\pi h} = O(h^{-1})$.

$$\begin{aligned}
\rho &= 1 + \frac{\sqrt{X^2-1}-1}{\sqrt{X^2-1}+1} - 1 \\
&= 1 + \frac{\sqrt{X^2-1}-1-\sqrt{X^2-1}-1}{\sqrt{X^2-1}+1} \\
&= 1 - \frac{1}{\sqrt{X^2-1}+1} \\
&= 1 - \frac{1}{X\left(\sqrt{1-\frac{1}{X^2}}+\frac{1}{X}\right)} \\
&= 1 - O(h)
\end{aligned}$$

- Here, we would have $\delta = 1$. Together with a good preconditioner ...

5.2. Iterative methods for diagonally dominant and M-Matrices

Eigenvalue analysis for more general matrices

- For 1D heat conduction we used a very special regular structure of the matrix which allowed exact eigenvalue calculations
- Generalizations to tensor product is possible
- Generalization to varying coefficients, unstructured grids ...
  ⇒ what can be done for general matrices ?

The Gershgorin Circle Theorem (Semyon Gershgorin,1931)

(everywhere, we assume $n \geq 2$)

Theorem (Varga, Th. 1.11) Let A be an $n \times n$ (real or complex) matrix. Let

$$\Lambda_i = \sum_{\substack{j=1\ldots n \\ j \neq i}} |a_{ij}|$$

If λ is an eigenvalue of A then there exists r, $1 \leq r \leq n$ such that

$$|\lambda - a_{rr}| \leq \Lambda_r$$

Proof Assume λ is eigenvalue, $\mathbf{x}$ a corresponding eigenvector, normalized such that $\max_{i=1\ldots n} |x_i| = |x_r| = 1$. From $A\mathbf{x} = \lambda\mathbf{x}$ it follows that

$$(\lambda - a_{ii})x_i = \sum_{\substack{j=1\ldots n \\ j \neq i}} a_{ij}x_j$$

$$|\lambda - a_{rr}| = |\sum_{\substack{j=1\ldots n \\ j \neq r}} a_{rj}x_j| \leq \sum_{\substack{j=1\ldots n \\ j \neq r}} |a_{rj}||x_j| \leq \sum_{\substack{j=1\ldots n \\ j \neq r}} |a_{rj}| = \Lambda_r$$

□

Gershgorin Circle Corollaries

Corollary: Any eigenvalue of A lies in the union of the disks defined by the Gershgorin circles

$$\lambda \in \bigcup_{i=1\ldots n} \{\mu \in \mathbb{V} : |\mu - a_{ii}| \leq \Lambda_i\}$$

Corollary:

$$\rho(A) \leq \max_{i=1\ldots n} \sum_{j=1}^{n} |a_{ij}| = ||A||_\infty$$

$$\rho(A) \leq \max_{j=1\ldots n} \sum_{i=1}^{n} |a_{ij}| = ||A||_1$$

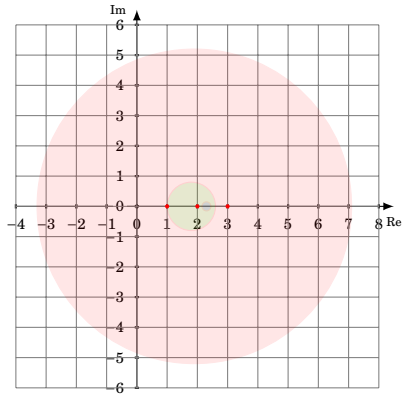
Proof

$$|\mu - a_{ii}| \leq \Lambda_i \quad \Rightarrow \quad |\mu| \leq \Lambda_i + |a_{ii}| = \sum_{j=1}^{n} |a_{ij}|$$

Furthermore, $\sigma(A) = \sigma(A^T)$. □

Gershgorin circles: example

$$A = \begin{pmatrix} 1.9 & 1.8 & 3.4 \\ 0.4 & 1.8 & 0.4 \\ 0.05 & 0.1 & 2.3 \end{pmatrix}, \lambda_1 = 1, \lambda_2 = 2, \lambda_3 = 3, \Lambda_1 = 5.2, \Lambda_2 = 0.8, \lambda_3 = 0.15$$



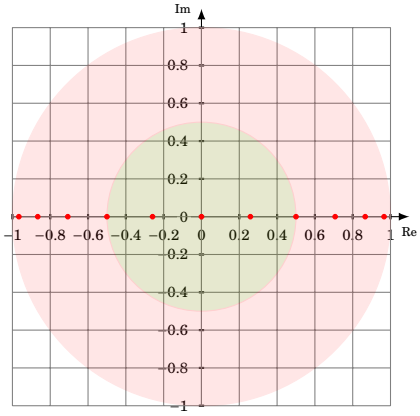
Gershgorin circles: heat example I

$$A = \begin{pmatrix} \frac{2}{h} & -\frac{1}{h} & & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} \\ & & & & -\frac{1}{h} & \frac{2}{h} \end{pmatrix}$$

$$B = (I - D^{-1}A) = \begin{pmatrix} 0 & \frac{1}{2} & & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & & \\ & \frac{1}{2} & 0 & \frac{1}{2} & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & \frac{1}{2} & 0 & \frac{1}{2} & \\ & & & \frac{1}{2} & 0 & \frac{1}{2} \\ & & & & \frac{1}{2} & 0 \end{pmatrix}$$

We have $b_{ii} = 0$, $\Lambda_i = \begin{cases} \frac{1}{2}, & i = 1, n \\ 1 & i = 2 \dots n-1 \end{cases} \Rightarrow$ estimate $|\lambda_i| \le 1$

Gershgorin circles: heat example II



n=11, h=0.1

$$\lambda_i = \cos\left(\frac{ih\pi}{1+2h}\right) \quad (i = 1\ldots n)$$

Reducible and irreducible matrices

Definition A is *reducible* if there exists a permutation matrix P such that

$$PAP^T = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}$$

A is *irreducible* if it is not reducible.

Directed matrix graph:

- Nodes: $\mathcal{N} = \{N_i\}_{i=1\ldots n}$
- Directed edges: $\mathcal{E} = \{\overrightarrow{N_k N_l} | a_{kl} \neq 0\}$

Theorem (Varga, Th. 1.17): A is irreducible $\Leftrightarrow$ the matrix graph is connected, i.e. for each *ordered* pair (N_i, N_j) there is a path consisting of directed edges, connecting them.

Equivalently, for each i, j there is a sequence of nonzero matrix entries $a_{ik_1}, a_{k_1k_2}, \ldots, a_{k_rj}$.

□

Taussky theorem (Olga Taussky, 1948)

Theorem (Varga, Th. 1.18) Let A be irreducible. Assume that the eigenvalue λ is a boundary point of the union of all the disks

$$\lambda \in \partial \bigcup_{i=1\ldots n} \{\mu \in \mathbb{C} : |\mu - a_{ii}| \leq \Lambda_i\}$$

Then, all n Gershgorin circles pass through λ, i.e. for $i = 1\ldots n$,

$$|\lambda - a_{ii}| = \Lambda_i$$

Taussky theorem proof

Proof Assume λ is eigenvalue, $\mathbf{x}$ a corresponding eigenvector, normalized such that $\max_{i=1\ldots n} |x_i| = |x_r| = 1$. From $A\mathbf{x} = \lambda\mathbf{x}$ it follows that

$$|\lambda - a_{rr}| \leq \sum_{\substack{j=1\ldots n \\ j\neq r}} |a_{rj}| \cdot |x_j| \leq \sum_{\substack{j=1\ldots n \\ j\neq r}} |a_{rj}| = \Lambda_r \qquad (*)$$

Boundary point $\Rightarrow |\lambda - a_{rr}| = \Lambda_r$

$\Rightarrow$ For all $l \neq r$ with $a_{r,p} \neq 0$, $|x_p| = 1$.

Due to irreducibility there is at least one such p. For this p, equation $(*)$ is valid (with p in place of r) $\Rightarrow$ $|\lambda - a_{pp}| = \Lambda_p$

Due to irreducibility, this is true for all $p = 1\ldots n$. □

Consequences for heat example from Taussky

$B = I - D^{-1}A$

We had $b_{ii} = 0$, $\Lambda_i = \begin{cases} \frac{1}{2}, & i = 1, n \\ 1 & i = 2\ldots n-1 \end{cases}$ $\Rightarrow$ estimate $|\lambda_i| \leq 1$

Assume $|\lambda_i| = 1$. Then λ_i lies on the boundary of the union of the Gershgorin circles. But then it must lie on the boundary of both circles with radius $\frac{1}{2}$ and 1 around 0.

Contradiction $\Rightarrow |\lambda_i| < 1$, $\rho(B) < 1$!

Diagonally dominant matrices

Definition

- A is *diagonally dominant* if
  - (i) for $i = 1\ldots n$, $|a_{ii}| \geq \sum_{\substack{j=1\ldots n \\ j\neq i}} |a_{ij}|$
- A is *strictly diagonally dominant* (sdd) if
  - (i) for $i = 1\ldots n$, $|a_{ii}| > \sum_{\substack{j=1\ldots n \\ j\neq i}} |a_{ij}|$
- A is *irreducibly diagonally dominant* (idd) if
  - (i) A is irreducible
  - (ii) A is diagonally dominant – for $i = 1\ldots n$, $|a_{ii}| \geq \sum_{\substack{j=1\ldots n \\ j\neq i}} |a_{ij}|$
  - (iii) for at least one r, $1 \leq r \leq n$, $|a_{rr}| > \sum_{\substack{j=1\ldots n \\ j\neq r}} |a_{rj}|$

A very practical nonsingularity criterion

Theorem (Varga, Th. 1.21): Let A be strictly diagonally dominant or irreducibly diagonally dominant. Then A is nonsingular.

If in addition, $a_{ii} > 0$ for $i = 1\ldots n$, then all real parts of the eigenvalues of A are positive:

$$\mathrm{Re}\lambda_i > 0, \quad i = 1\ldots n$$

A very practical nonsingularity criterion, proof

Proof:

- Assume A sdd. Then the union of the Gershgorin disks does not contain 0 and $\lambda = 0$ cannot be an eigenvalue.

  As for the real parts, the union of the disks is

  $$\bigcup_{i=1\ldots n} \{\mu \in \mathbb{C} : |\mu - a_{ii}| \leq \Lambda_i\}$$

  and Reμ must be larger than zero if μ should be contained.

- Assume A idd. Then, if 0 is an eigenvalue, it sits on the boundary of one of the Gershgorin disks. By the Taussky theorem, we have $|a_{ii}| = \Lambda_i$ for all $i = 1 \ldots n$. This is a contradiction as by definition there is at least one i such that $|a_{ii}| > \Lambda_i$

  Assume $a_{ii} > 0$, real. All real parts of the eigenvalues must be ≥ 0. Therefore, if a real part is 0, it lies on the boundary of one disk. So by Taussky it must be contained at the same time in the boundary of all the disks and the imaginary axis. This contradicts the fact that there is at least one disk which does not touch the imaginary axis.
  □

Corollary

Theorem: If A is complex hermitian or real symmetric, sdd or idd, with positive diagonal entries, it is positive definite.

Proof: All eigenvalues of A are real, and due to the nonsingularity criterion, they must be positive, so A is positive definite.

□

Heat conduction matrix

$$A = \begin{pmatrix} \alpha + \frac{1}{h} & -\frac{1}{h} & & & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & & & -\frac{1}{h} & \frac{1}{h} + \alpha \end{pmatrix}$$

- A is idd $\Rightarrow$ A is nonsingular
- diag$A$ is positive real $\Rightarrow$ eigenvalues of A have positive real parts
- A is real, symmetric $\Rightarrow$ A is positive definite

Perron-Frobenius Theorem (1912/1907)

Definition: A real n-vector $\mathbf{x}$ is

- positive ($\mathbf{x} > 0$) if all entries of $\mathbf{x}$ are positive
- nonnegative ($\mathbf{x} \geq 0$) if all entries of $\mathbf{x}$ are nonnegative

Definition: A real $n \times n$ matrix A is

- positive ($A > 0$) if all entries of A are positive
- nonnegative ($A \geq 0$) if all entries of A are nonnegative

Theorem(Varga, Th. 2.7) Let $A \geq 0$ be an irreducible $n \times n$ matrix. Then

(i) A has a positive real eigenvalue equal to its spectral radius $\rho(A)$.
(ii) To $\rho(A)$ there corresponds a positive eigenvector $\mathbf{x} > 0$.
(iii) $\rho(A)$ increases when any entry of A increases.
(iv) $\rho(A)$ is a simple eigenvalue of A.

Proof: See Varga. □

Perron-Frobenius for general nonnegative matrices

Each $n \times n$ matrix can be brought to the normal form

$$PAP^T = \begin{pmatrix} R_{11} & R_{12} & \dots & R_{1m} \\ 0 & R_{22} & \dots & R_{2m} \\ \vdots & & \ddots & \\ 0 & 0 & \dots & R_{mm} \end{pmatrix}$$

where for $j = 1 \dots m$, either R_{jj} irreducible or $R_{jj} = (0)$.

Theorem(Varga, Th. 2.20) Let $A \geq 0$ be an $n \times n$ matrix. Then

(i) A has a nonnegative eigenvalue equal to its spectral radius $\rho(A)$. This eigenvalue is positive unless A is reducible and its normal form is strictly upper triangular
(ii) To $\rho(A)$ there corresponds a nonzero eigenvector $\mathbf{x} \geq 0$.
(iii) $\rho(A)$ does not decrease when any entry of A increases.

Proof: See Varga; $\sigma(A) = \bigcup_{j=1}^{m} \sigma(R_{jj})$, apply irreducible Perron-Frobenius to R_{jj}. □

Theorem on Jacobi matrix

Theorem: Let A be sdd or idd, and D its diagonal. Then

$$\rho(|I - D^{-1}A|) < 1$$

Proof: Let $B = (b_{ij}) = I - D^{-1}A$. Then

$$b_{ij} = \begin{cases} 0, & i = j \\ -\frac{a_{ij}}{a_{ii}}, & i \neq j \end{cases}$$

If A is sdd, then for $i = 1 \dots n$,

$$\sum_{j=1\dots n} |b_{ij}| = \sum_{\substack{j=1\dots n \\ j \neq i}} |\frac{a_{ij}}{a_{ii}}| = \frac{\Lambda_i}{|a_{ii}|} < 1$$

Therefore, $\rho(|B|) < 1$.

Theorem on Jacobi matrix II

If A is idd, then for $i = 1 \dots n$,

$$\sum_{j=1\dots n} |b_{ij}| = \sum_{\substack{j=1\dots n \\ j\neq i}} |\frac{a_{ij}}{a_{ii}}| = \frac{\Lambda_i}{|a_{ii}|} \leq 1$$

$$\sum_{j=1\dots n} |b_{rj}| = \frac{\Lambda_r}{|a_{rr}|} < 1 \text{ for at least one } r$$

Therefore, $\rho(|B|) <= 1$. Assume $\rho(|B|) = 1$. By Perron-Frobenius, 1 is an eigenvalue. As it is in the union of the Gershgorin disks, for some i,

$$|\lambda| = 1 \leq \frac{\Lambda_i}{|a_{ii}|} \leq 1$$

and it must lie on the boundary of this union. By Taussky then one has for all i

$$|\lambda| = 1 \leq \frac{\Lambda_i}{|a_{ii}|} = 1$$

which contradicts the idd condition. □

Jacobi method convergence

Corollary: Let A be sdd or idd, and D its diagonal. Assume that $a_{ii} > 0$ and $a_{ij} \leq 0$ for $i \neq j$. Then $\rho(I - D^{-1}A) < 1$, i.e. the Jacobi method converges.

Proof In this case, $|B| = B$ □.

Regular splittings

- $A = M - N$ is a regular splitting if
  - M is nonsingular
  - M^{-1}, N are nonnegative, i.e. have nonnegative entries
- Regard the iteration $u_{k+1} = M^{-1}Nu_k + M^{-1}b$.
- We have $I - M^{-1}A = M^{-1}N$.

Convergence theorem for regular splitting

Theorem: Assume A is nonsingular, $A^{-1} \geq 0$, and $A = M - N$ is a regular splitting. Then $\rho(M^{-1}N) < 1$.

Proof: Let $G = M^{-1}N$. Then $A = M(I - G)$, therefore $I - G$ is nonsingular.

In addition

$$A^{-1}N = (M(I - M^{-1}N))^{-1}N = (I - M^{-1}N)^{-1}M^{-1}N = (I - G)^{-1}G$$

By Perron-Frobenius (for general matrices), $\rho(G)$ is an eigenvalue with a nonnegative eigenvector $\mathbf{x}$. Thus,

$$0 \leq A^{-1}N\mathbf{x} = \frac{\rho(G)}{1 - \rho(G)}\mathbf{x}$$

Therefore $0 \leq \rho(G) \leq 1$.
As $I - G$ is nonsingular, $\rho(G) < 1$. □

Convergence rate comparison

Corollary: $\rho(M^{-1}N) = \frac{\tau}{1+\tau}$ where $\tau = \rho(A^{-1}N)$.

Proof: Rearrange $\tau = \frac{\rho(G)}{1-\rho(G)}$ □

Corollary: Let $A \geq 0$, $A = M_1 - N_1$ and $A = M_2 - N_2$ be regular splittings. If $N_2 \geq N_1 \geq 0$, then $1 > \rho(M_2^{-1}N_2) \geq \rho(M_1^{-1}N_1)$.

Proof: $\tau_2 = \rho(A^{-1}N_2) \geq \rho(A^{-1}N_1) = \tau_1$, $\frac{\tau}{1+\tau}$ is strictly increasing.

M-Matrix definition

Definition Let A be an $n \times n$ real matrix. A is called M-Matrix if

(i) $a_{ij} \leq 0$ for $i \neq j$

(ii) A is nonsingular

(iii) $A^{-1} \geq 0$

Corollary: If A is an M-Matrix, then $A^{-1} > 0 \Leftrightarrow A$ is irreducible.

Proof: See Varga. □

Main practical M-Matrix criterion

Corollary: Let A be sdd or idd. Assume that $a_{ii} > 0$ and $a_{ij} \leq 0$ for $i \neq j$. Then A is an M-Matrix.

Proof:

- Let $B = I - D^{-1}A$. Then $\rho(B) < 1$, therefore $I - B$ is nonsingular.
- We have for $k > 0$:

$$I - B^{k+1} = (I - B)(I + B + B^2 + \cdots + B^k)$$
$$(I - B)^{-1}(I - B^{k+1}) = (I + B + B^2 + \cdots + B^k)$$

The left hand side for $k \to \infty$ converges to $(I - B)^{-1}$, therefore

$$(I - B)^{-1} = \sum_{k=0}^{\infty} B^k$$

As $B \geq 0$, we have $(I - B)^{-1} = A^{-1}D \geq 0$. As $D > 0$ we must have $A^{-1} \geq 0$. □

Application

Let A be an M-Matrix. Assume $A = D - E - F$.

- Jacobi method: $M = D$ is nonsingular, $M^{-1} \geq 0$. $N = E + F$ nonnegative $\Rightarrow$ convergence
- Gauss-Seidel: $M = D - E$ is an M-Matrix as $A \leq M$ and M has non-positive off-digonal entries. $N = F \geq 0$. $\Rightarrow$ convergence
- Comparison: $N_J \geq N_{GS} \Rightarrow$ Gauss-Seidel converges faster.
- More general: Block Jacobi, Block Gauss Seidel etc.

Intermediate Summary

- Given some matrix, we now have some nice recipies to establish nonsingularity and iterative method convergence:
- **Check if the matrix is irreducible.**
  This is mostly the case for elliptic and parabolic PDEs.
- **Check if the matrix is strictly or irreducibly diagonally dominant**.
  If yes, it is in addition nonsingular.
- **Check if main diagonal entries are positive and off-diagonal entries are nonpositive.**
  If yes, in addition, the matrix is an M-Matrix, its inverse is nonnegative, and elementary iterative methods converge.

Example: 1D finite volume matrix:

$$Au = \begin{pmatrix} \alpha+\frac{1}{h} & -\frac{1}{h} & & & & \\ -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\ & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\ & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} \\ & & & & -\frac{1}{h} & \frac{1}{h}+\alpha \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix} = f = \begin{pmatrix} \alpha v_1 \\ hf_2 \\ hf_3 \\ \vdots \\ hf_{N-2} \\ hf_{N-1} \\ \alpha v_n \end{pmatrix}$$

- idd
- positive main diagonal entries, nonpositive off-diagonal entries

$\Rightarrow A$ is nonsingular, has the M-property, and we can e.g. apply the Jacobi and Gauss-Seidel iterative method to solve it.

$\Rightarrow$ for $f \geq 0$ and $v \geq 0$ it follows that $u \geq 0$.
$\equiv$ heating and positive environment temperatures cannot lead to negative temperatures in the interior.

Incomplete LU factorizations (ILU)

Idea (Varga, Buleev, 1960):

- fix a predefined zero pattern
- apply the standard LU factorization method, but calculate only those elements, which do not correspond to the given zero pattern
- Result: incomplete LU factors L, U, remainder R:

$$A = LU - R$$

- Problem: with complete LU factorization procedure, for any nonsingular matrix, the method is stable, i.e. zero pivots never occur. Is this true for the incomplete LU Factorization as well ?

Comparison of M-Matrices

Theorem(Saad, Th. 1.33): Let A, B $n \times n$ matrices such that

(i) $A \leq B$

(ii) $b_{ij} \leq 0$ for $i \neq j$.

Then, if A is an M-Matrix, so is B.

Proof: For the diagonal parts, one has $D_B \geq D_A > 0$, $D_A - A \geq D_B - B \geq 0$ Therefore

$$I - D_A^{-1}A \geq D_A^{-1}(D_B - B) \geq D_B^{-1}(D_B - B) = I - D_B^{-1}B =: G \geq 0.$$

Perron-Frobenius $\Rightarrow \rho(G) = \rho(I - D_B^{-1}B) \leq \rho(I - D_A^{-1}A) < 1$
$\Rightarrow I - G$ is nonsingular. From the proof of the M-matrix criterion, $D_B^{-1}B = (I-G)^{-1} = \sum_{k=0}^{\infty} G^k \geq 0$. As $D_B > 0$, we get $B \geq 0$.

□

M-Property propagation in Gaussian Elimination

Theorem:(Ky Fan; Saad Th 1.10) Let A be an M-matrix. Then the matrix A_1 obtained from the first step of Gaussian elimination is an M-matrix.

Proof: One has $a_{ij}^1 = a_{ij} - \frac{a_{i1}a_{1j}}{a_{11}}$,
$a_{ij}, a_{i1}, a_{1j} \le 0$, $a_{11} > 0$
$\Rightarrow a_{ij}^1 \le 0$ for $i \neq j$

$A = L_1 A_1$ with $L_1 = \begin{pmatrix} 1 & 0 & \dots & 0 \\ \frac{-a_{12}}{a_{11}} & 1 & \dots & 0 \\ \vdots & & \ddots & 0 \\ \frac{-a_{1n}}{a_{11}} & 0 & \dots & 1 \end{pmatrix}$ nonsingular, nonnegative

$\Rightarrow A_1$ nonsingular

Let $e_1 \dots e_n$ be the unit vectors. Then $A_1^{-1} e_1 = \frac{1}{a_{11}} e_1 \ge 0$. For $j > 1$, $A_1^{-1} e_j = A^{-1} L^{-1} e_j = A^{-1} e_j \ge 0$.
$\Rightarrow A_1^{-1} \ge 0$

□

Stability of ILU

Theorem (Saad, Th. 10.2): If A is an M-Matrix, then the algorithm to compute the incomplete LU factorization with a given nonzero pattern

$$A = LU - R$$

is stable. Moreover, $A = LU - R$ is a regular splitting.

Stability of ILU decomposition II

Proof

Let $\tilde{A}_1 = A_1 + R_1 = L_1 A + R_1$ where R_1 is a nonnegative matrix which occurs from dropping some off diagonal entries from A_1. Thus, $\tilde{A}_1 \ge A_1$ and $\tilde{A}_1$ is an M-matrix. We can repeat this recursively

$$\begin{aligned} \tilde{A}_k = A_k + R_k &= L_k A_{k-1} + R_k \\ &= L_k L_{k-1} A_{k-2} + L_k R_{k-1} + R_k \\ &= L_k L_{k-1} \cdot \ldots \cdot L_1 A + L_k L_{k-1} \cdot \ldots \cdot L_2 R_1 + \cdots + R_k \end{aligned}$$

Let $L = (L_{n-1} \cdot \ldots \cdot L_1)^{-1}$, $U = \tilde{A}_{n-1}$. Then $U = L^{-1} A + S$ with

$$S = L_{n-1} L_{n-2} \cdot \ldots \cdot L_2 R_1 + \cdots + R_{n-1} = L_{n-1} L_{n-2} \cdot \ldots \cdot L_2 (R_1 + R_2 + \ldots R_{n-1})$$

Let $R = R_1 + R_2 + \ldots R_{n-1}$, then $A = LU - R$ where $U^{-1} L^{-1}$, R are nonnegative.

□

ILU(0)

- Special case of ILU: ignore any fill-in.
- Representation:

$$M = (\tilde{D} - E)\tilde{D}^{-1}(\tilde{D} - F)$$

- $\tilde{D}$ is a diagonal matrix (wich can be stored in one vector) which is calculated by the incomplete factorization algorithm.
- Setup:

```
for(int i=0;i<n;i++)
  d(i)=a(i,i)

for(int i=0;i<n;i++)
{
  d(i)=1.0/d(i)
  for (int j=i+1;j<n;j++)
  d(j)=d(j)-a(i,j)*d(i)*a(j,i)
}
```

ILU(0)

Solve $Mu = v$

```
for(int i=0;i<n;i++)
{
  double x=0.0;
  for (int j=0;j<i;i++)
      x=x+a(i,j)*u(j)
  u(i)=d(i)*(v(i)-x)

}

for(int i=n-1;i>=0;i--)
{
   double x=0.0
   for(int j=i+1;j<n;j++)
       x=x+a(i,j)*u(j)
   u(i)=u(i)-d(i)*x
}
```

ILU(0)

- Generally better convergence properties than Jacobi, Gauss-Seidel
- One can develop block variants
- Alternatives:
  - ILUM: (“modified”): add ignored off-diagonal entries to $\tilde{D}$
  - ILUT: zero pattern calculated dynamically based on drop tolerance
- Dependence on ordering
- Can be parallelized using graph coloring
- Not much theory: experiment for particular systems
- I recommend it as the default initial guess for a sensible preconditioner
- Incomplete Cholesky: symmetric variant of ILU

5.3. Orthogonalization methods

Generalization of iteration schemes

- Simple iterations converge slowly
- For most practical purposes, Krylov subspace methods are used.
- We will introduce one special case and give hints on practically useful more general cases
- Material after J. Shewchuk: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain“

Solution of SPD system as a minimization procedure

Regard $Au = f$,where A is symmetric, positive definite. Then it defines a bilinear form $a : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$

$$a(u,v) = (Au,v) = v^T Au = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} v_i u_j$$

As A is SPD, for all $u \neq 0$ we have $(Au,u) > 0$.

For a given vector b, regard the function

$$f(u) = \frac{1}{2} a(u,u) - b^T u$$

What is the minimizer of f ?

$$f'(u) = Au - b = 0$$

- Solution of SPD system ≡ minimization of f.

Method of steepest descent

- Given some vector u_i, look for a new iterate u_{i+1}.
- The direction of steepest descend is given by $-f'(u_i)$.
- So look for u_{i+1} in the direction of $-f'(u_i) = r_i = b - Au_i$ such that it minimizes f in this direction, i.e. set $u_{i+1} = u_i + \alpha r_i$ with α choosen from

$$\begin{aligned} 0 &= \frac{d}{d\alpha} f(u_i + \alpha r_i) = f'(u_i + \alpha r_i) \cdot r_i \\ &= (b - A(u_i + \alpha r_i), r_i) \\ &= (b - Au_i, r_i) - \alpha (Ar_i, r_i) \\ &= (r_i, r_i) - \alpha (Ar_i, r_i) \\ \alpha &= \frac{(r_i, r_i)}{(Ar_i, r_i)} \end{aligned}$$

Method of steepest descent: iteration scheme

$$r_i = b - Au_i$$
$$\alpha_i = \frac{(r_i, r_i)}{(Ar_i, r_i)}$$
$$u_{i+1} = u_i + \alpha_i r_i$$

Let $\hat{u}$ the exact solution. Define $e_i = u_i - \hat{u}$, then $r_i = -Ae_i$

Let $||u||_A = (Au, u)^{\frac{1}{2}}$ be the *energy norm* wrt. A.

Theorem The convergence rate of the method is

$$||e_i||_A \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^i ||e_0||_A$$

where $\kappa = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$ is the spectral condition number.

Method of steepest descent: advantages

- Simple Richardson iteration $u_{k+1} = u_k - \alpha(Au_k - f)$ needs good eigenvalue estimate to be optimal with $\alpha = \frac{2}{\lambda_{max} + \lambda_{min}}$
- In this case, asymptotic convergence rate is $\rho = \frac{\kappa - 1}{\kappa + 1}$
- Steepest descent has the same rate without need for spectral estimate

Conjugate directions

For steepest descent, there is no guarantee that a search direction $d_i = r_i = -Ae_i$ is not used several times. If all search directions would be orthogonal, or, indeed, A-orthogonal, one could control this situation.

So, let $d_0, d_1 \ldots d_{n-1}$ be a series of A-orthogonal (or conjugate) search directions, i.e. $(Ad_i, d_j) = 0$, $i \neq j$.

- Look for u_{i+1} in the direction of d_i such that it minimizes f in this direction, i.e. set $u_{i+1} = u_i + \alpha_i d_i$ with α choosen from

$$\begin{aligned} 0 = \frac{d}{d\alpha} f(u_i + \alpha d_i) &= f'(u_i + \alpha d_i) \cdot d_i \\ &= (b - A(u_i + \alpha d_i), d_i) \\ &= (b - Au_i, d_i) - \alpha(Ad_i, d_i) \\ &= (r_i, d_i) - \alpha(Ad_i, d_i) \\ \alpha_i &= \frac{(r_i, d_i)}{(Ad_i, d_i)} \end{aligned}$$

Conjugate directions II

$e_0 = u_0 - \hat{u}$ (such that $Ae_0 = -r_0$) can be represented in the basis of the search directions:

$$e_0 = \sum_{i=0}^{n-1} \delta_j d_j$$

Projecting onto d_k in the A scalar product gives

$$\begin{aligned}(Ae_0, d_k) &= \sum_{i=0}^{n-1} \delta_j (Ad_j, d_k) \\ &= \delta_k (Ad_k, d_k) \\ \delta_k &= \frac{(Ae_0, d_k)}{(Ad_k, d_k)} = \frac{(Ae_0 + \sum_{i<k} \alpha_i d_i, d_k)}{(Ad_k, d_k)} = \frac{(Ae_k, d_k)}{(Ad_k, d_k)} \\ &= \frac{(r_k, d_k)}{(Ad_k, d_k)} \\ &= -\alpha_k\end{aligned}$$

Conjugate directions III

Then,

$$\begin{aligned}e_i &= e_0 + \sum_{j=0}^{i-1} \alpha_j d_j = -\sum_{j=0}^{n-1} \alpha_j d_j + \sum_{j=0}^{i-1} \alpha_j d_j \\ &= -\sum_{j=i}^{n-1} \alpha_j d_j\end{aligned}$$

So, the iteration consists in component-wise suppression of the error, and it must converge after n steps. Let $k \le i$. A-projection on d_k gives

$$(Ae_i, d_k) = -\sum_{j=i}^{n-1} \alpha_j (Ad_j, d_k) = 0$$

Therefore, $r_i = Ae_i$ is orthogonal to $d_0 \dots d_{i-1}$.

Conjugate directions IV

Looking at the error norm $||e_i||_A$, the method yields the element with the minimum energy norm from all elements of the affine space $e_0 + \mathcal{K}_i$ where $\mathcal{K}_i = \text{span}\{d_0, d_1 \dots d_{i-1}\}$

$$\begin{aligned}(Ae_i, e_i) &= \left(\sum_{j=i}^{n-1} \delta_j d_j, \sum_{j=i}^{n-1} \delta_j d_j \right) = \sum_{j=i}^{n-1} \sum_{k=i}^{n-1} \delta_j \delta_k (d_j, d_k) \\ &= \sum_{j=i}^{n-1} \delta_j^2 (d_j, d_j) = \min_{e \in e_0 + \mathcal{K}_i} ||e||_A\end{aligned}$$

Furthermore, we have

$$\begin{aligned}u_{i+1} &= u_i + \alpha_i d_i \\ e_{i+1} &= e_i + \alpha_i d_i \\ Ae_{i+1} &= Ae_i + \alpha_i Ad_i \\ r_{i+1} &= r_i - \alpha_i Ad_i\end{aligned}$$

By what magic we can obtain these d_i?

Gram-Schmidt Orthogonalization

- Assume we have been given some linearly independent vectors $v_0, v_1 \dots v_{n-1}$.
- Set $d_0 = v_0$
- Define

$$d_i = v_i + \sum_{k=0}^{i-1} \beta_{ik} d_k$$

- For $j < i$, A-project onto d_j and require orthogonality:

$$\begin{aligned}(Ad_i, d_j) &= (Av_i, d_j) + \sum_{k=0}^{i-1} \beta_{ik} (Ad_k, d_j) \\ 0 &= (Av_i, d_j) + \beta_{ij} (Ad_j, d_j) \\ \beta_{ij} &= -\frac{(Av_i, d_j)}{(Ad_j, d_j)}\end{aligned}$$

- If v_i are the coordinate unit vectors, this is Gaussian elimination!
- If v_i are arbitrary, they all must be kept in the memory

Conjugate gradients (Hestenes, Stiefel, 1952)

As Gram-Schmidt builds up d_i from d_j, $j < i$, we can choose $v_i = r_i$, i.e. the residuals built up during the conjugate direction process.

Let $\mathcal{K}_i = \text{span}\{d_0 \dots d_{i-1}\}$. Then, $r_i \perp \mathcal{K}_i$

But d_i are built by Gram-Schmidt from the residuals, so we also have $\mathcal{K}_i = \text{span}\{r_0 \dots r_{i-1}\}$ and $(r_i, r_j) = 0$ for $j < i$.

From $r_i = r_{i-1} - \alpha_{i-1} A d_{i-1}$ we obtain

$\mathcal{K}_i = \mathcal{K}_{i-1} \cup \text{span}\{Ad_{i-1}\}$

This gives two other representations of $\mathcal{K}_i$:

$$\begin{aligned}\mathcal{K}_i &= \text{span}\{d_0, Ad_0, A^2 d_0, \dots, A^{i-1} d_0\} \\ &= \text{span}\{r_0, Ar_0, A^2 r_0, \dots, A^{i-1} r_0\}\end{aligned}$$

Such type of subspace of $\mathbb{R}^n$ is called *Krylov subspace*, and orthogonalization methods are more often called *Krylov subspace methods*.

Conjugate gradients II

Look at Gram-Schmidt under these conditions. The essential data are (setting $v_i = r_i$ and using $j < i$) $\beta_{ij} = -\frac{(Ar_i,d_j)}{(Ad_j,d_j)} = -\frac{(Ad_j,r_i)}{(Ad_j,d_j)}$.

Then, for $j \leq i$:

$$\begin{aligned}
r_{j+1} &= r_j - \alpha_j A d_j \\
(r_{j+1}, r_i) &= (r_j, r_i) - \alpha_j (A d_j, r_i) \\
\alpha_j (A d_j, r_i) &= (r_j, r_i) - (r_{j+1}, r_i) \\
(A d_j, r_i) &= \begin{cases} -\frac{1}{\alpha_j}(r_{j+1}, r_i), & j+1 = i \\ \frac{1}{\alpha_j}(r_j, r_i), & j = i \\ 0, & \text{else} \end{cases} = \begin{cases} -\frac{1}{\alpha_{i-1}}(r_i, r_i), & j+1 = i \\ \frac{1}{\alpha_i}(r_i, r_i), & j = i \\ 0, & \text{else} \end{cases}
\end{aligned}$$

For $j < i$:

$$\beta_{ij} = \begin{cases} \frac{1}{\alpha_{i-1}} \frac{(r_i, r_i)}{(A d_{i-1}, d_{i-1})}, & j+1 = i \\ 0, & \text{else} \end{cases}$$

Conjugate gradients III

For Gram-Schmidt we defined (replacing v_i by r_i):

$$\begin{aligned}
d_i &= r_i + \sum_{k=0}^{i-1} \beta_{ik} d_k \\
&= r_i + \beta_{i,i-1} d_{i-1}
\end{aligned}$$

So, the new orthogonal direction depends only on the previous orthogonal direction and the current residual. We don't have to store old residuals or search directions. In the sequel, set $\beta_i := \beta_{i,i-1}$.

We have

$$\begin{aligned}
d_{i-1} &= r_{i-1} + \beta_{i-1} d_{i-2} \\
(d_{i-1}, r_{i-1}) &= (r_{i-1}, r_{i-1}) + \beta_{i-1} (d_{i-2}, r_{i-1}) \\
&= (r_{i-1}, r_{i-1}) \\
\beta_i &= \frac{1}{\alpha_{i-1}} \frac{(r_i, r_i)}{(A d_{i-1}, d_{i-1})} = \frac{(r_i, r_i)}{(d_{i-1}, r_{i-1})} \\
&= \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})}
\end{aligned}$$

Conjugate gradients IV - The algorithm

Given initial value u_0, spd matrix A, right hand side b.

$$\begin{aligned} d_0 = r_0 &= b - Au_0 \\ \alpha_i &= \frac{(r_i, r_i)}{(Ad_i, d_i)} \\ u_{i+1} &= u_i + \alpha_i d_i \\ r_{i+1} &= r_i - \alpha_i A d_i \\ \beta_{i+1} &= \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)} \\ d_{i+1} &= r_{i+1} + \beta_{i+1} d_i \end{aligned}$$

At the i-th step, the algorithm yields the element from $e_0 + \mathcal{K}_i$ with the minimum energy error.

Theorem The convergence rate of the method is

$$||e_i||_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i ||e_0||_A$$

where $\kappa = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$ is the spectral condition number.

Preconditioning

Let M be spd, and spectrally equivalent to A, and assume that $\kappa(M^{-1}A) << \kappa(A)$.

Let E be such that $M = EE^T$, e.g. its Cholesky factorization. Then, $\sigma(M^{-1}A) = \sigma(E^{-1}AE^{-T})$:

Assume $M^{-1}Au = \lambda u$. We have

$$(E^{-1}AE^{-T})(E^T u) = (E^T E^{-T})E^{-1}Au = E^T M^{-1}Au = \lambda E^T u$$

$\Leftrightarrow E^T u$ is an eigenvector of $E^{-1}AE^{-T}$ with eigenvalue λ.

Preconditioned CG I

Now we can use the CG algorithm for the preconditioned system

$$E^{-1}AE^{-T}\tilde{x} = E^{-1}b$$

with $\tilde{u} = E^T u$

$$\begin{aligned} \tilde{d}_0 = \tilde{r}_0 &= E^{-1}b - E^{-1}AE^{-T}u_0 \\ \alpha_i &= \frac{(\tilde{r}_i, \tilde{r}_i)}{(E^{-1}AE^{-T}\tilde{d}_i, \tilde{d}_i)} \\ \tilde{u}_{i+1} &= \tilde{u}_i + \alpha_i \tilde{d}_i \\ \tilde{r}_{i+1} &= \tilde{r}_i - \alpha_i E^{-1}AE^{-T}\tilde{d}_i \\ \beta_{i+1} &= \frac{(\tilde{r}_{i+1}, \tilde{r}_{i+1})}{(\tilde{r}_i, \tilde{r}_i)} \\ \tilde{d}_{i+1} &= \tilde{r}_{i+1} + \beta_{i+1}\tilde{d}_i \end{aligned}$$

Not very practical as we need E

Preconditioned CG II

Assume $\tilde{r}_i = E^{-1} r_i$, $\tilde{d}_i = E^T d_i$, we get the equivalent algorithm

$$
\begin{aligned}
r_0 &= b - Au_0 \\
d_0 &= M^{-1} r_0 \\
\alpha_i &= \frac{(M^{-1} r_i, r_i)}{(Ad_i, d_i)} \\
u_{i+1} &= u_i + \alpha_i d_i \\
r_{i+1} &= r_i - \alpha_i A d_i \\
\beta_{i+1} &= \frac{(M^{-1} r_{i+1}, r_{i+1})}{(r_i, r_i)} \\
d_{i+1} &= M^{-1} r_{i+1} + \beta_{i+1} d_i
\end{aligned}
$$

It relies on the solution of the preconditioning system, the calculation of the matrix vector product and the calculation of the scalar product.

A few issues

Usually we stop the iteration when the residual r becomes small. However during the iteration, floating point errors occur which distort the calculations and lead to the fact that the accumulated residuals

$$r_{i+1} = r_i - \alpha_i A d_i$$

give a much more optimistic picture on the state of the iteration than the real residual

$$r_{i+1} = b - Au_{i+1}$$

C++ implementation

```
template < class Matrix, class Vector, class Preconditioner, class Real >
int  CG(const Matrix &A, Vector &x, const Vector &b,
const Preconditioner &M, int &max_iter, Real &tol)
{ Real resid;
  Vector p, z, q;
  Vector alpha(1), beta(1), rho(1), rho_1(1);
  Real normb = norm(b);
  Vector r = b - A*x;
  if (normb == 0.0)   normb = 1;
  if ((resid = norm(r) / normb) <= tol) {
    tol = resid;
    max_iter = 0;
    return 0;
  }
  for (int i = 1; i <= max_iter; i++) {
    z = M.solve(r);
    rho(0) = dot(r, z);
    if (i == 1)
    p = z;
    else {
      beta(0) = rho(0) / rho_1(0);
      p = z + beta(0) * p;
    }
    q = A*p;
    alpha(0) = rho(0) / dot(p, q);
    x += alpha(0) * p;
    r -= alpha(0) * q;
    if ((resid = norm(r) / normb) <= tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }
    rho_1(0) = rho(0);
  }
  tol = resid;   return 1;
}
```

C++ implementation II

- Available from http://www.netlib.org/templates/cpp//cg.h
- Slightly adapted for numcxx
- Available in numxx in the namespace netlib.

Unsymmetric problems

- By definition, CG is only applicable to symmetric problems.
- The biconjugate gradient (BICG) method provides a generalization:

Choose initial guess x_0, perform

$$
\begin{aligned}
r_0 &= b - A x_0 & \hat{r}_0 &= \hat{b} - \hat{x}_0 A^T \\
p_0 &= r_0 & \hat{p}_0 &= \hat{r}_0 \\
\alpha_i &= \frac{(\hat{r}_i, r_i)}{(\hat{p}_i, A p_i)} & & \\
x_{i+1} &= x_i + \alpha_i p_i & \hat{x}_{i+1} &= \hat{x}_i + \alpha_i \hat{p}_i \\
r_{i+1} &= r_i - \alpha_i A p_i & \hat{r}_{i+1} &= \hat{r}_i - \alpha_i \hat{p}_i A^T \\
\beta_i &= \frac{(\hat{r}_{i+1}, r_{i+1})}{(\hat{r}_i, r_i)} & & \\
p_{i+1} &= r_{i+1} + \beta_i p_i & \hat{p}_{i+1} &= \hat{r}_{i+1} + \beta_i \hat{p}_i
\end{aligned}
$$

The two sequences produced by the algorithm are biorthogonal, i.e., $(\hat{p}_i, A p_j) = (\hat{r}_i, r_j) = 0$ for $i \neq j$.

Unsymmetric problems II

- BiCG is very unstable and additionally needs the transposed matrix vector product, it is seldomly used in practice
- There is as well a preconditioned variant of BiCG which also needs the transposed preconditioner.
- Main practical approaches to fix the situation:
  - "Stabilize" BiCG → BiCGstab (H. Van der Vorst, 1992)
  - tweak CG → "Conjugate gradients squared" (CGS, Sonneveld, 1989)
  - Error minimization in Krylov subspace → "Generalized Minimum Residual" (GMRES, Saad/Schulz, 1986)
- Both CGS and BiCGstab can show rather erratic convergence behavior
- For GMRES one has to keep the full Krylov subspace, which is not possible in practice ⇒ restart strategy.
- From my experience, BiCGstab is a good first guess

6. Mesh generation

Meshes

- Regard boundary value problems for PDEs in a finite domain $\Omega \subset \mathbb{R}^n$
- Assume the domain is polygonal, its boundary $\partial\Omega$ is the union of a finite number of subsets of hyperplanes in $\mathbb{R}^n$ (line segments for $d = 2$, planar polygons for $d = 3$)
- A mesh (grid) is a subdivision Ω into a finite number of elementary closed (polygonal) subsets $T_1 \dots T_M$.
- Mostly, the elementary shapes are triangles or quadrilaterals ($d = 2$) or tetrahedra or cuboids ($d = 3$)
- During this course: focus on $d = 2$, triangles
  $\Rightarrow$ mesh = grid = triangulation

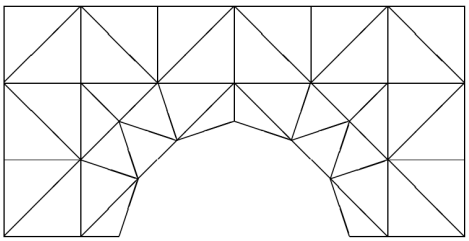
(FEM)-Admissible meshes

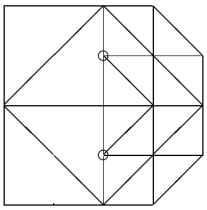
Definition: A grid is FEM-admissible if

(i) $\bar{\Omega} = \cup_{m=1}^{M} T_m$

(ii) If $T_m \cap T_n$ consists of exactly one point, then this point is a common vertex of T_m and T_n.

(iii) If for $m \neq n$, $T_m \cap T_n$ consists of more than one point, then $T_m \cap T_n$ is a common edge (or a common facet for $d = 3$) of T_m and T_n.





Source: Braess, FEM

Left: admissible mesh. Right: mesh with hanging nodes

Acute + weakly acute triangulations

Definition A triangulation of a domain Ω is

- acute, if all interior angles of all triangles are less than $\frac{\pi}{2}$,
- weakly acute, if all interior angles of all triangles are less than or equal to $\frac{\pi}{2}$.

Triangulation methods

- Geometrically most flexible
- Basis als for more general methods of subdivision into quadrilaterals
- Problem seems to be simple only at the first glance ...
- Here, we will discuss Delaunay triangulations, which have a number of interestin properties when it comes to PDE discretizations

Voronoi diagrams

Definition Let $\mathbf{p}, \mathbf{q} \in \mathbb{R}^d$. The set of points $H_{\mathbf{pq}} = \{\mathbf{x} \in \mathbb{R}^d : ||\mathbf{x} - \mathbf{p}|| \leq ||\mathbf{x} - \mathbf{q}||\}$ is the *half space* of points $\mathbf{x}$ closer to $\mathbf{p}$ than to $\mathbf{q}$.

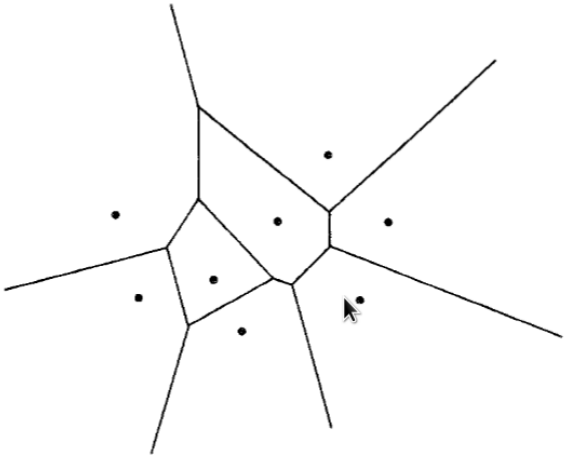
Definition Given a finite set of points $S \subset \mathbb{R}^d$, the *Voronoi region (Voronoi cell)* of a point $\mathbf{p} \in S$ is the set of points $\mathbf{x}$ closer to $\mathbf{p}$ than to any other point $\mathbf{q} \in S$:

$$V_{\mathbf{p}} = \left\{\mathbf{x} \in \mathbb{R}^d : ||\mathbf{x} - \mathbf{p}|| \leq ||\mathbf{x} - \mathbf{q}|| \, \forall \mathbf{q} \in S\right\}$$

The *Voronoi diagram* of S is the collection of the Voronoi regions of the points of S.

Voronoi diagrams II

- the Voronoi diagram subdivides the whole space into “nearest neigbor” regions
- Being intersections of half planes, the Voronoi regions are convex sets

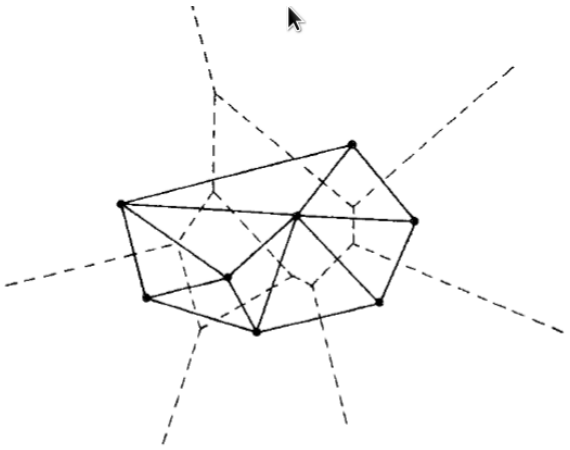


Voronoi diagram of 8 points in the plane

(H. Si)

Delaunay triangulation

- Assume that the points of S are in *general position*, i.e. no $d+2$ points of S are on one sphere (in 2D: no 4 points on one circle)
- Connect each pair of points whose Voronoi regions share a common edge with a line
- $\Rightarrow$ *Delaunay triangulation* of the convex hull of S



Delaunay triangulation of the convex hull of 8 points in the plane

(H. Si)

Delaunay triangulation II

- The circumsphere (circumcircle in 2D) of a d-dimensional simplex is the unique sphere containing all vertices of the simplex
- The circumball (circumdisc in 2D) of a simplex is the unique (open) ball which has the circumsphere of the simplex as boundary

Definition A triangulation of the convex hull of a point set S has the *Delaunay property* if each simplex (triangle) of the triangulation is Delaunay, i.e. it is empty wrt. S, i.e. it does not contain any points of S.

- The Delaunay triangulation of a point set S, where all points are in general position is unique
- Otherwise there is an ambiguity - if e.g. 4 points are one circle, there are two ways to connect them resulting in Delaunay triangles

Edge flips and locally Delaunay edges (2D only)

- For any two triangles **abc** and **adb** sharing a common edge **ab**, there is the *edge flip* operation which reconnects the points in such a way that two new triangles emerge: **adc** and **cdb**.
- An edge of a triangulation is locally Delaunay if it either belongs to exactly one triangle, or if it belongs to two triangles, and that their respective circumdisks do not contain the pointe opposite wrt. the edge
- If an edge is locally Delaunay and belongs to two triangles, the sum of the angles opposite to this edge is less or equal to π.
- If all edges of a triangulation of the convex hull of S are locally Delaunay, then the triangulation is the Delaunay triangulation
- If an edge is not locally Delaunay and belongs to two triangles, the edge emerging from the corresponding edge flip will be locally Delaunay

Edge flip algorithm (Lawson)

Input: A stack L of edges of a given triangulation of S;
while $L \neq \emptyset$ **do**
    pop an edge **ab** from L;
    if ab *is not locally Delaunay* **then**
        flip **ab** to **cd**;
        push edges **ac**, **cb**, **db**, **da** onto L;
    end
end

- This algorithm is known to terminate. After termination, all edges will be locally Delaunay, so the output is the Delaunay triangulation of S.
- Among all triangulations of a finite point set S, the Delaunay triangulation maximises the minimum angle
- All triangulations of S are connected via a flip graph

Radomized incremental flip algorithm (2D only)

- Create Delaunay triangulation of point set S by inserting points one after another, and creating the Delaunay triangulation of the emerging subset of S using the flip algorithm
- Estimated complexity: $O(n \log n)$
- In 3D, there is no simple flip algorithm, generalizations are active research subject

Triangulations of finite domains

- So far, we discussed triangulations of point sets, but in practice, we need triangulations of domains
- Create Delaunay triangulation of point set, “Intersect” with domain

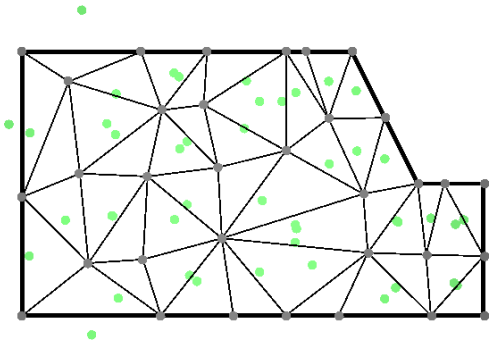
Boundary conforming Delaunay triangulations

Definition: An admissible triangulation of a polygonal Domain $\Omega \subset \mathbb{R}^d$ has the boundary conforming Delaunay property if

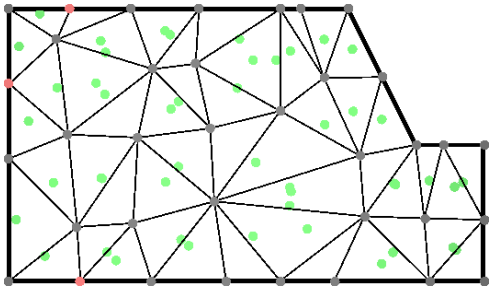
(i) All simplices are Delaunay

(ii) All boundary simplices (edges in 2D, facets in 3d) have the Gabriel property, i.e. their minimal circumdisks are empty

- Equivalent definition in 2D: sum of angles opposite to interior edges $\leq \pi$, angle opposite to boundary edge $\leq \frac{\pi}{2}$
- Creation of boundary conforming Delaunay triangulation description may involve insertion of Steiner points at the boundary



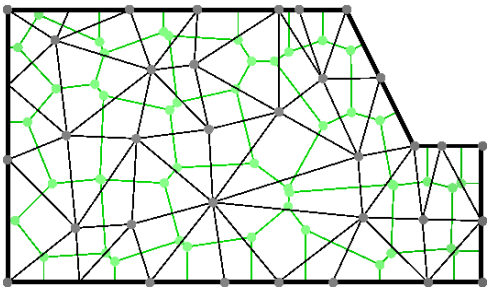
Delaunay grid of Ω



Boundary conforming Delaunay grid of Ω

Domain blendend Voronoi cells

- For Boundary conforming Delaunay triangulations, the intersection of the Voronoi diagram with the domain yields a well defined dual subdivision which can be used for finite volume discretizations



Boundary conforming Delaunay triangulations II

- Weakly acute triangulations are boundary conforming Delaunay, but not vice versa!
- Working with weakly acute triangulations for general polygonal domains is unrealistic, especially in 3D
- For boundary conforming Delaunay triangulations of polygonal domains there are algoritms with mathematical termination proofs valid in many relevant cases
  - Software in 2D: Triangle by J.R.Shewchuk https://www.cs.cmu.edu/~quake/triangle.html
  - Software in 3D: TetGen by H. Si http://tetgen.org
  - polygonal geometry description
  - automatic insertion of points according to given mesh size criteria
  - accounting for interior boundaries
  - local mesh size control for a priori refinement
  - quality control
  - standalone executable & library

Further mesh generation approaches

(Most of them lose Delaunay property)

- Advancing front: create mesh of boundary, "grow" triangles from boundary to interior (rather heuristical, implemented e.g. in netgen by J. Schöberl https://sourceforge.net/projects/netgen-mesher/
- Quadtree/octree: place points on quadtree/octree hierachy an triangulate
- Mesh improvement: equilibrate element sizes + quality by iteratively modifying point locations
- … active research topic with many open questions, unfortunately not exactly mainstream …

A. Working with compilers and source code

Command line instructions to control compiler

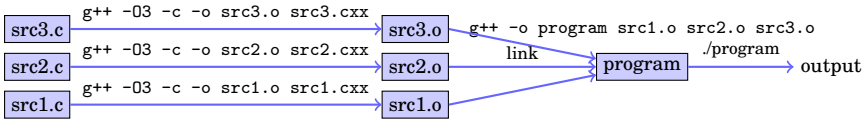
- By default, the compiler command performs the linking process as well
- Compiler command (Linux)

```
g++      GNU C++ compiler
g++-5    GNU C++ 5.x
clang++  CLANG compiler from LLVM project
icpc     Intel compiler
```

- Options (common to all of those named above, but not standardized)

```
-o name             Name of output file
-g                  Generate debugging instructions
-O0, -O1, -O2, -O3  Optimization levels
-c                  Avoid linking
-I<path>            Add <path> to include search path
-D<symbol>          Define preprocessor symbol
-std=c++11          Use C++11 standard
```

Compiling...




```
$ g++ -O3 -c -o src3.o src3.cxx
$ g++ -O3 -c -o src2.o src2.cxx
$ g++ -O3 -c -o src1.o src1.cxx
$ g++ -o program src1.o src2.o src3.o
$ ./program
```

Shortcut: invoke compiler and linker at once

```
$ g++ -O3 -o program src1.cxx src2.cxx src3.cxx
$ ./program
```

Some shell commands in the terminal window

```
ls -l           list files in directory
                subdirectories are marked with 'd'
                in the first column of permission list
cd  dir         change directory to dir
cd  ..          change directory one level up in directory hierachy
cp  file1 file2 copy file1 to file2
cp  file1 dir   copy file1 to directory
mv  file1 file2 rename file1 to file2
mv  file1 dir   move file1 to directory
rm  file        delete file
[cmd] *.o       perform command on all files with name ending with .o
```

Editors & IDEs

- Source code is written with text editors
  (as compared to word processors like `MS Word` or `libreoffice`)
- Editors installed are
  - `gedit` - text editor of gnome desktop (recommended)
  - `emacs` - comprehensive, powerful, a bit unusual GUI (my preferred choice)
  - `nedit` - quick and simple
  - `vi`, `vim` - the UNIX purist's crowbar
    (which I avoid as much as possible)
- Integrated development environments (IDE)
  - Integrated editor/debugger/compiler
  - eclipse (need to get myself used to it before teaching)

Working with source code

- Copy the code:

```
$ cp /net/wir/examples/part1/example.cxx .
```

- Editing:

```
$ gedit  example.cxx
```

- Compiling: (`-o` gives the name of the output file)

```
$ g++ -std=c++11  example.cxx -o  example
```

- Running: (`./` means file from current directory)

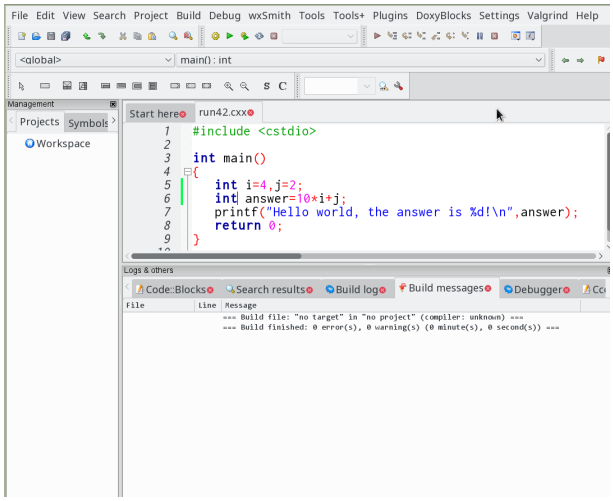
```
$ ./example
```

Alternative:Code::Blocks IDE

- `http://www.codeblocks.org/`
- Open example

```
$ codeblocks example.cxx
```

- Compile + run example: `Build/"Build and Run"` or F9
- Switching on C++11 standard: tick `Settings/Compiler/"Have g++ follow the C++11..."`



B. The numcxx library

numcxx

numcxx is a small C++ library developed for and during this course which implements the concepts introduced

- Shared smart pointers vs. references
- 1D/2D Array class
- Matrix class with LAPACK interface
- Expression templates
- Interface to triangulations
- Sparse matrices + UMFPACK interface
- Iterative solvers
- Python interface

numcxx classes

- `TArray1`: templated 1D array class
  `DArray1`: 1D double array class
- `TArray2`: templated 2D array class
  `DArray2`: 2D double array class
- `TMatrix`: templated dense matrix class
  `DMatrix`: double dense matrix class
- `TSolverLapackLU`: LU factorization based on LAPACK
  `DSolverLapackLU`

Obtaining and compiling the examples

- Copy files, creating subdirectory part2
  - the `.` denotes the current directory

```
$ ls /net/wir/numxx/examples/10-numcxx-basicx/*.cxx
$ cp -r /net/wir/examples/10-numcxx-basicx/numcxx-expressions.cxx .
```

- Compile sources (for each of the .cxx files) (integrates with codeblocks)

```
$ numcxx-build -o example numcxx-expressions.cxx
$ ./example
```

CMake

What is behind `numcxx-build`?

- CMake - the current best way to build code
- Describe project in a file called `CMakeLists.txt`

```
cmake_minimum_required(VERSION 2.8.12)
PROJECT(example C CXX)
find_package(NUMCXX REQUIRED)
include_directories("${NUMCXX_INCLUDE_DIRS}")
link_libraries("${NUMCXX_LIBRARIES}")
add_executable(example example.cxx)
```

- Set up project (only once)

```
$ mkdir builddir
$ cd buildir
$ cmake ..
$ cd ..
```

- build code

```
$ cmake --build builddir
```

- run code

```
$ ./builddir/example
```

Let's have some naming conventions

- lowercase letters: scalar values
  - `i,j,k,l,m,n` standalone or as prefixes: integers, indices
  - others: floating point
- Upper_case_letters: class objects/references

```
std::vector<double> X(n);
numcxx::DArray1<double> Y(n);
```

- pUpper_case_letters: smart pointers to objects

```
auto pX=std::make_shared<std::vector<double>>(n);
auto pY=numcxx::TArray1<double>::create(n);
auto pZ=numcxx::TArray1<double>::create({1,2,3,4});

// getting references from smart pointers
auto &X=*pX;
auto &Y=*pY;
auto &Z=*pZ;

auto W=std::make_shared<std::vector<double>>({1,2,3,4}); // doesn't work...
```

C++ code using numcxx with references

File `/net/wir/examples/10-numcxx-basicx/numcxx-ref.cxx`

```
#include <cstdio>
#include <numcxx/numcxx.hxx>
void initialize(numcxx::DArray1 &X)
{   const int n=X.size();
  for (int i=0;i<n;i++) X[i]= 1.0/(double)(1+n-i);
}
double sum_elements(numcxx::DArray1 & X)
{   double sum=0;
  for (int i=0;i<X.size();i++)sum+=X[i];
  return sum;
}
int main()
{   const int n=12345678;
  numcxx::TArray1<double> X(n);
  initialize(X);
  double s=sum_elements(X);
  printf("sum=%e\n",s);
}
```

C++ code using numcxx with smart pointers

File `/net/wir/examples/10-numcxx-basics/numcxx-sharedptr.cxx`

```
#include <cstdio>
#include <memory>
#include <numcxx/numcxx.hxx>
void initialize(numcxx::DArray1 &X)
{   const int n=X.size();
  for (int i=0;i<n;i++) X[i]= 1.0/(double)(1+n-i);
}
double sum_elements(numcxx::DArray1 & X)
{   double sum=0;
  for (int i=0;i<X.size();i++)sum+=X[i];
  return sum;
}
int main()
{   const int n=12345678;
  // call constructor and wrap pointer into smart pointer
  auto pX=numcxx::TArray1<double>::create(n);
  initialize(*pX);
  double s=sum_elements(*pX);
  printf("sum=%e\n",s);
}
```

numcxx Sparse matrix class

`numcxx::TSparseMatrix<T>`

- Class characterized by IA/JA/AA arrays
- How to create these arrays ?
- Common way (e.g. Eigen) : from a list triples i, j, a_{ij}. In practice, this can be expensive because in FEM assembly we will have many triplets repeating with the same i, j but different a_{ij}
- Remedy:
  - Internally create and update an intermediate datas structure which maintains a list of already available entries
  - Hide this behind the facade $A(i, j) = x$

Numcxx with CodeBlocks

- CodeBlocks support has been added to numcxx-build:
  - `numcxx-build --codeblocks hello.cxx` creates a subdirectory `hello.codeblocks` which contains the codeblocks project file `hello.cbp`
  - Configure and then start codeblocks:

```
$ numcxx-build --codeblocks hello.cxx
$ codeblocks hello.codeblocks/hello.cbp
```

  - Or start codeblocks immediately after configuring

```
$ numcxx-build --codeblocks --execute hello.cxx
```

  - In Codeblocks, instead of "all" select target "hello" or "hello/fast", then Build & Run as usual.

C. Visualization tools

Visualization in Scientific Computing

- Human perception much better adapted to visual representation than to numbers
- Visualization of computational results necessary for the development of understanding
- Basic needs: curve plots etc
  - python/matplotlib
- Advanced needs: Visualize discretization grids, geometry descriptions, solutions of PDEs
  - Visualization in Scientific Computing: paraview
  - Graphics hardware: GPU
  - How to program GPU: OpenGL
  - vtk

Python

- Scripting language with huge impact in Scientific Computing
- Open Source, exhaustive documentation online
  - https://docs.python.org/3/
  - https://www.python.org/about/gettingstarted/
- Possibility to call C/C++ code from python
  - Library API
  - swig - simple wrapper and interface generator (not only python)
  - pybind11 - C++11 specific
- Glue language for projects from different sources
- Huge number of libraries
- numpy/scipy
  - Array + linear algebra library implemented in C
- matplotlib: graphics library
  https://matplotlib.org/users/pyplot_tutorial.html

C++/matplotlib workflow

- Run C++ program
- Write data generated during computations to disk
- Use python/matplotlib for to visualize results
- Advantages:
  - Rich possibilities to create publication ready plots
  - Easy to handle installation (write your code, install python+matplotlib)
  - Python/numpy to postprocess calculated data
- Disadvantages
  - Long way to in-depth understanding of API
  - Slow for large datasets
  - Not immediately available for creating graphics directly from C++

Matplotlib: Alternative tools

- Similar workflow
  - gnuplot
  - Latex/tikz
- Call graphics from C++ ?
  - ???
  - Best shot: call C++ from python, return data directly to python
  - Send data to python through UNIX pipes
  - Link pyton interpreter into C++ code
- Faster graphics ?

Processing steps in visualization

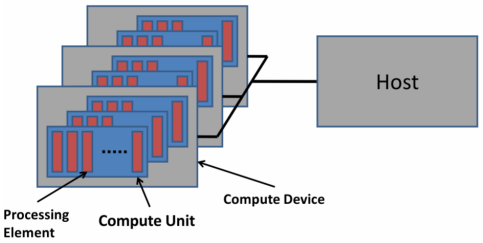
- Representation of data using elementary primitives: points, lines, triangles, …
- Coordinate transformation form world coordinates to screen coordinates
- Transformation 3D → 2D - what is visible ?
- Rasterization: smooth data into pixels
- Coloring, lighting, transparency
- Similar tasks in CAD, gaming etc.
- Huge number of very similar operations

GPU aka “Graphics Card”

- SIMD parallelism “Single instruction, multiple data” inherent to processing steps in visualization
- Mostly `float` (32bit) accuracy is sufficient
- ⇒ Create specialized coprocessors devoted to this task, free CPU from it
- Pionieering: Silicon Graphics (SGI)
- Today: nvidia, AMD
- Multiple parallel pipelines, fast memory for intermediate results

GPU Programming

- As GPU is a different processor, one needs to write extra programs to handle data on it – “shaders”
- Typical use:
  - Include shaders as strings in C++ (or load then from extra source file)
  - Compile shaders
  - Send compiled shaders to GPU
  - Send data to GPU – critical step for performance
  - Run shaders with data
- OpenGL, Vulkan



GPU Programming as it used to be

- Specify transformations
- Specify parameters for lighting etc.
- Specify points, lines etc. via API calls
- Graphics library sends data and manages processing on GPU
- No shaders - “fixed functions”
- Iris GL (SGI), OpenGL 1.x, now deprecated
- No simple, standardized API for 3D graphics with equivalent functionality
- Hunt for performance (gaming)

vtk

https://www.vtk.org/

- Visualization primitives in scientific computing
  - Datasets on rectangular and unstructured discretization grids
  - Scalar data
  - Vector data
- The *Visualization Toolkit* vtk provides an API with these primitives and uses up-to data graphics API (OpenGL) to render these data
- Well maintained, “working horse” in high performance computing
- Open Source
- Paraview, VisIt: GUI programs around vtk

Working with paraview

https://www.paraview.org/

- Write data into files using vtk specific data format
- Call paraview, load data

- Using “paraview catalyst”
  - Send data via network from simulation server to desktop running paraview
- Call vtk API directly
  - vtkfig: small library for graphics primitives compatible with numcxx

D. List of slides