Precourse to the PeC^3 Summer School on

Introduction on Numerical Modelling with Differential Equations

Ole Klein[1]

[1]Universität Heidelberg
Interdisziplinäres Zentrum für Wiss. Rechnen
Im Neuenheimer Feld 205, D-69120 Heidelberg
email: ole.klein@iwr.uni-heidelberg.de

UNIVERSIDAD NACIONAL AGRARIA
LA MOLINA

DAAD
Deutscher Akademischer Austausch Dienst
German Academic Exchange Service

BMZ
Federal Ministry
for Economic Cooperation
and Development

PeC^3
Peruvian Competence Center
of Scientific Computing

Universidad Nacional Agraria La Molina,
Lima, Perú, October 23–25, 2019

Contents I

What is C++?

C++ is a programming language that was developed for both system and application programming.

- Support for several programming paradigms: imperative, object oriented, generic (templates)
- Emphasizes efficiency, performance and flexibility
- Applications range from embedded controllers to high-performance super computers
- Allows direct management of hardware resources
- "Zero-cost abstractions", "pay only for what you use"
- Open standard with several implementations
- Most important compilers:
  - Open source: GCC and Clang (LLVM)
  - Proprietary: Microsoft and Intel

History

- 1979: Bjarne Stroustrup develops "C with Classes"
- 1985: First commercial C++ compiler
- 1989: C++ 2.0
- 1998: Standardization as ISO/IEC 14882:1998 (C++98)
- 2011: Next important version with new functionality (C++11)
- 2014: C++14 with many bug fixes and useful features
- 2017: C++17, current version
- 2020: upcoming standard

First C++ Program

```cpp
/* make parts of the standard library available */
#include <iostream>
#include <string>

// custom function that takes a string as an argument
void print(std::string msg)
{
  // write to stdout
  std::cout << msg << std::endl;
}

// main function is called at program start
int main(int argc, char** argv)
{
  // variable declaration and initialization
  std::string greeting = "Hello, world!";
  print(greeting);
  return 0;
}
```

Central Components of a C++ Program

A basic C++ program consists of two components:

Include directives to import software libraries:

- always the first lines of a program
- only one include directive per line

User-defined functions:

- like mathematical functions, with arguments and return value
- every program must implement the function

```
int main(int argc, char** argv)
{
  ...
}
```

This function is called by the operating system when the program is executed.

Comments

Rules for comments in C++ code:

- Comments can be placed anywhere in the code
- Comments starting with `//` (C-style comments) end at a line break:

```
int i = 42; // the answer
int x = 0;
```

- Multiline comments are started by `/*` and ended by `*/`:

```
/* This comment spans
   multiple lines */
int x = 0;
```

Functions

- During execution of a C++ program functions are called, starting with the special function `main(int argc, char** argv)`
- Functions can call other functions
- Function definitions consist of a function signature and a function body:

```
return-type functionName(arg-type argName, ...) // signature
{
  // function body
}
```

- The signature defines the name of the function and which arguments it needs
- In C++, a function **always** has a return type. The special type `void` is used if a function shouldn't return anything.
- The function body describes what the function does

Statements

```
int i = 0;
i = i + someFunction();
anotherFunction();
return i;
i = 2; // never executed
```

- A C++ function consists of a number of statements, executed one by one
- Statements are separated by semicolons
- The special statement `return val;` **immediately** leaves the current function and returns `val` as its return value
  - `void` functions can leave out the value or even the whole `return` statement

Variables

- Variables are used for storing intermediate values
- In C++, variables always have a fixed **type** (integer, floating-point, text, ...)
- Variables may contain upper and lower case letters, digits and underscores, but may not start with a digit
- Names are case sensitive!
- Variables have to be **declared** before they can be used
  - Normal variables are declared through a statement:

```
variable-type variableName = initial-value;
```

  - Function arguments are declared in the function signature:

```
void func(var-type1 arg1, var-type2 arg2)
```

Important Variable Types

C++ knows many variable types, here some important ones (ranges valid on 64-bit Linux):

```
// 32-bit integer, whole numbers in [−2^31, 2^31 − 1]
int i = 1;
// 64-bit integer, whole numbers in [−2^63, 2^63 − 1]
long l = 1;
// 8-bit integer, whole numbers in [−2^7, 2^7 − 1]
char c = 1;
// Boolean (truth value), true (=1) or false (=0)
bool b = true;
// Text (sequence of symbols), requires #include <string>
std::string msg = "Hello";
// Floating point with double accuracy
double d = 3.141;
// Floating point with single accuracy
float f = 3.141;
```

Integer variables restricted to positive numbers by prepending **unsigned**, with range $[0, 2^{\text{bits}} - 1]$

Scopes and Variable Lifetime

- A **block scope** is a group of statements within braces (e.g., function body)
- Scopes can be nested arbitrarily
- Variables have a limited **lifetime**:
  - the lifetime of a variable starts with its declaration
  - it ends when leaving the scope where it was declared

```
int cube(int x)
{
  // x exists everywhere in the function
  {
    int y = x * x; // y exists from here on
    x = x * y;
  } // here y doesn't exist anymore
  return x;
} // here x doesn't exist anymore
```

Scopes and Name Collisions

It is impossible to create two variables with the same name within a scope, and names in an inner scope temporarily shadow names from an outer scope

```
{
  int x = 2;
  int x = 3; // compile error!
}

int abs(int x) { ... } // absolute value

{
  int x = -2;
  {
    double x = 3.3; int abs = -2;
    std::cout << x << std::endl; // prints 3.3
    x = abs(x); // compile error, here abs is a variable!
  }
  x = abs(x); // now: x == 2
}
```

Namespaces

Scopes can also have an name, and are then called *namespaces*. Namespaces are used to group parts of a program together that share functionality or form a larger unit. They are a tool for the organization of large code bases.

```
namespace MyScientificProgram {
  namespace LinearSolvers {
    // any user-defined functions, objects, etc.,
    // that deal with linear solvers
  }

  namespace NonlinearSolvers {
    // any functions, etc., that belong to nonlinear solvers,
    // e.g., Newton's method
  }
}
```

Tools provided by the C++ standard library are in namespace `std`.

Expressions

We use expressions to calculate things in C++

- Expressions are combinations of values, variables, function calls, and mathematical operators, that produce some value that can be assigned to a variable:

```
i = 2;
j = i * j;
d = std::sqrt(2.0) + j;
```

- Composite expressions like `(a * b + c) * d` use standard mathematical precedence rules, known as **operator precedence**

Rule Overview

`https://en.cppreference.com/w/cpp/language/operator_precedence`

Operators for Numbers

- The usual binary operators +,−,∗,/
- a % b calculates the remainder of integer division of a by b:

```
13 % 5 // result: 3
```

- Division of integers always rounds to 0
- Integer division by 0 crashes the program
- = assigns its righthand side to its lefthand side, and at the same time returns this value

```
a = b = 2 * 21; // both a and b have value 42
```

- Abbreviations for frequent combinations:

```
a += b; // shortcut for a = a + b (also for -,*,/,%)
x = i++; // post-increment, shortcut for x = i; i = i + 1;
x = ++i; // pre-increment,  shortcut for i = i + 1; x = i;
```

  - of course there's also pre- and post-decrement (−−)

Comparison Operators

- Comparison operators produce truth values (`bool`):

```
4 > 3; // true
```

- Available operators

```
a < b;  // a strictly less than b
a > b;  // a strictly greater than b
a <= b; // a less than or equal to b
a >= b; // a greater than or equal to b
a == b; // a equal to b (note the double =!)
a != b; // a not equal to b
```

Combination of Truth Values

Test results can be combined using symbolic or text-based operators:

- Combination of several tests with "and" or "or":

```
a == b ||  b == c; // a equal b or b equal c
a == b or  b == c; // a equal b or b equal c

a == b &&  b == c; // a equal b and b equal c
a == b and b == c; // a equal b and b equal c
```

- Inversion of a truth value:

```
!true == false;
not true == false;
```

Texts / Strings

- Texts (strings) are stored in variables of type `std::string`
- Fixed strings are enclosed in double quotes

```
std::string msg = "Hello world!";
```

- Strings can be concatenated with `+`

```
std::string hello = "Hello, ";
std::string world = "world";
std::string msg = hello + world;
```

- They can be compared with `==` and `!=`

```
std::string a = "a";
a == "b"; // false
```

Warning

When comparing or concatenating strings, the left argument must **always** be a variable!

Programming Tasks

Task 1

Consider the "hello, world!" program. Modify the program so that it uses a function `mark` that

- reads a `std::string` from `std::cin`
- adds an exclamation mark "!" to its end
- prints the resulting string using `std::cout`

Task 1

Add a function `calculate` that

- reads in two numbers, an `int` and a `double`
- prints their sum and product on one line, separated by a space
- returns their difference as a value

I/O Streams under UNIX

- UNIX (and Linux) programs communicate with the operating system using so-called I/O (Input/Output) **streams**
- Streams are one-way streets — you can either read from them or write to them
- Every program starts with three open streams, namely

stdin Standard input reads user input from the terminal, is connected to **file descriptor** 0

stdout Standard output receives results printed by the program, is connected to **file descriptor** 1

stderr Standard error output receives diagnostic messages like errors, is connected to **file descriptor** 2

Redirecting I/O Streams I

- Normally all standard streams are connected to the terminal
- Sometimes it is useful to redirect these streams to files
- **stdout** is redirected by writing `"> fileName"` after the program name

```
[user@host ~] ls > files
[user@host ~] cat files
file1
file2
files
```

  The output file is created before the command is executed
- Error messages are still printed to the terminal

```
[user@host ~] ls missingdir > files
ls: missingdir: No such file or directory
[user@host ~] cat files
[user@host ~]
```

Redirecting I/O Streams II

- These three operators may be combined, of course
- **stdin** is read from a file by appending "< fileName"

```
[user@host ~] cat # no argument means copying stdin to stdout
terminal input^D # (CTRL+D) terminates input
terminal input
[user@host ~] cat < files
file1
file2
files
```

- **stderr** is saved to file by using "2> fileName"

```
[user@host ~] ls missingdir 2> error
[user@host ~] cat error
ls: missingdir: No such file or directory
```

- These three operators may be combined, of course

Printing to the Terminal

- A C++ program can use the three streams **stdin**, **stdout** and **stderr** to communicate with a user on the terminal (shell)
- Output uses `std::cout`. Everything we want to print is "pushed" into the standard output using `<<`

```
#include <iostream> // required for input / output
...
std::string user = "Joe";
std::cout << "Hello, " << user << std::endl;
```

- A line break is created by printing `std::endl` (**end line**)

Reading from the Terminal

- Reading user input uses `std::cin`
- The corresponding variable has to be created first
- Input is "pulled" out of standard input with `>>`

```
#include <iostream>
...
std::string user = "";
int answer = 0;
std::cout << "Enter your name: " << std::endl;
std::cin >> user;
std::cout << "Enter your answer: " << std::endl;
std::cin >> answer;
std::cout << "Hi " << user << "! Your answer was: "
          << answer << std::endl;
```

- Input on the terminal has to be committed using the return key

Control Flow

Most programs are impossible or very difficult to write as a simple sequence of statements in fixed order.

Examples:

- a function returning the absolute value of a number
- a function catching division by zero and printing an error message
- a function summing all numbers from 1 to $N \in \mathbb{N}$
- ...

Programming languages contain special statements that execute different code paths based on the value of an expression

Branches

- The **if** statement executes different code depending on whether an expression is true or false

```
int abs(int x)
{
  if (x > 0)
  {
    return x;
  }
  else
  {
    return -x;
  }
}
```

- The **else** clause is optional:

```
if (weekday == "Wednesday")
{
  cpp_lecture();
}
```

Repetition

Often, a program has to execute the same code several times, e.g., when calculating the sum $\sum_{i=1}^{n} i$

Two different approaches:

- **Recursion**: the function calls itself with different arguments
- **Iteration**: a special statement executes a list of statements several times

Recursion

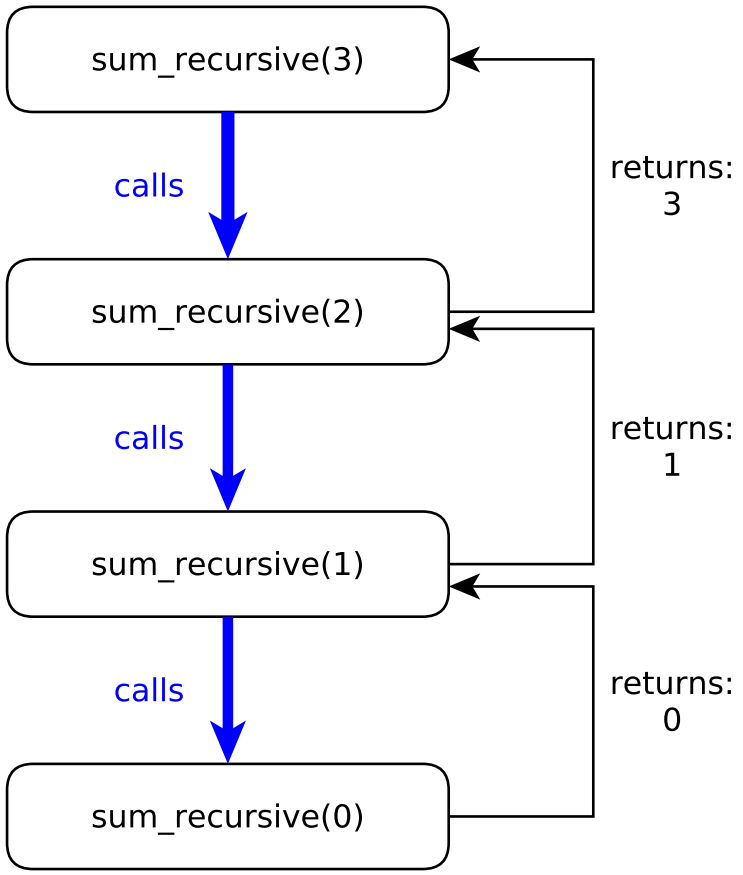
- Idea: a function calls itself again and again with changed input arguments, until some termination criterion is fulfilled:

```
int sum_recursive(int n)
{
  if (n > 0)
  {
    return sum_recursive(n - 1) + n;
  }
  else
  {
    return 0;
  }
}
```

- Requires at least one **if** statement, with exactly one of the branches calling the function again!
- Not suitable for functions that don't return anything and only have side effects (e.g., printing the first N numbers on the terminal)

Recursion: Example

- `sum_recursive(3)` calls `sum_recursive(2)` ...
- ...which calls `sum_recursive(1)` ...
- ...which calls `sum_recursive(0)` ...
- ...which ends the recursion due to the special case.
- Values on arrows are return values of corresponding function call.



Iteration Using While Loop

- A `while` loop executes the following block repeatedly, as long as its expression evaluates as true

```
int sum_iterative(int n)
{
  int result = 0;
  int i = 0;
  while (i <= n)
  {
    result += i;
    ++i;
  }
  return result;
}
```

- Often easier to understand
- Often more explicit and requires more variables

Iteration with For Loop I

- Many loops are executed several times for different values of some counting variable (index)
- C++ has a special **for** loop for these cases:

```
int sum_for(int n)
{
  int result = 0;
  for (int i = 0 ; i <= n ; ++i)
  {
    result += i;
  }
  return result;
}
```

- communicates to the reader that we sum over some index
- restricts lifetime of `i` to the loop itself
- somewhat more complicated than a **while** loop

Iteration with For Loop II

Every `for` loop can be converted into an equivalent `while` loop:

```
for (int i = 0 ; i <= n ; ++i)
{
  ...
}
```

becomes

```
{
  int i = 0;
  while (i <= n)
  {
    ... ;
    ++i;
  }
}
```

Integer Powers

Consider $q \in \mathbb{N}$ raised to the power of $n \in \mathbb{N}$.

Recursive definition:

$$q^n := \begin{cases} q^{n-1} \cdot q & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

"Iterative definition":

$$q^n := \underbrace{q \cdot \dots \cdot q}_{n \text{ times}}$$

Task

How can this function be implemented in C++?

Recursive Power Function

```
int pow_recursive(int q, int n)
{
  if (n == 0)
    return 1;
  else
    return q * pow_recursive(q,n-1);
}
```

- if $n = 0$ nothing needs to be computed
- else compute q^{n-1} and multiply by q (compare definition)

Iterative Power Function

```
int pow_iterative(int q, int n)
{
  int out = 1;
  for (int i = 0; i < n; i++)
    out *= q;
  return out;
}
```

- start out with value 1 (case $n = 0$)
- multiply with q n times

Recursive Power Function II

```
int pow_rec_fast(int q, int n)
{
  if (n == 0)
    return 1;
  else
  {
    int t = pow_rec_fast(q,n/2);
    if (n % 2 == 0)
      return t*t;
    else
      return q * t*t;
  }
}
```

- $q^n = q^{2k} = \left(q^k\right)^2$ for n even
- $q^n = q^{2k+1} = \left(q^k\right)^2 \cdot q$ for n odd

Iterative Power Function II

```
int pow_iter_fast(int q, int n)
{
  int out = 1;
  while (n > 0)
  {
    if (n % 2 != 0)
    {
      out *= q;
      n--;
    }

    q = q*q;
    n /= 2;
  }

  return out;
}
```

Which Version is best?

Two different measures of "best":

Readability and conciseness: definitely the first two versions

Speed: measure time for 100 billion (10^8) evaluations of 2^{30}

Version	Time	Version	Time
recursive	11.113s	recursive 2	1.988s
iterative	7.663s	iterative 2	1.397s

`std::pow()`: 3.378s

Note: This is the maximum range of the user-defined versions. The built-in `std::pow()` works for a much wider range, and also works for floating-point arguments. This explains why it is more expensive.

Which Version is best? II

These results were obtained without optimization. Compilers can optimize code and produce equivalent programs that are significantly faster (e.g., compiler option -O2).

What changes when we turn on optimization?

Version	Time	Version	Time
recursive	0.001s	recursive 2	1.317s
iterative	0.001s	iterative 2	0.257s

`std::pow()`: 0.001s

The simplest versions are now the fastest! Why? Because they are simple enough to be optimized away (evaluated **at compile time**)

Therefore: don't think too much about optimal code, most of the time it is irrelevant, and if not, **measure**

Classes and Objects

Objects are representations of components of a program, i.e., a self-contained collection of data with associated functions (called methods).

Classes are blueprints for objects, i.e., they define how the objects of a certain data type are structured.

Classes provide two special types of functions, *constructors* and *destructors*, which are used to create resp. destroy objects.

Example: Matrix Class

```
  class Matrix
{
  private: // can't be accessed by other parts of program

    std::vector<std::vector<double> > entries; // data
    int numRows;                               // number of rows
    int numCols;                               // number of columns

  public: // defines parts of object that are visible / usable

    Matrix(int numRows_, int numCols_); // constructor
    double& elem(int i, int j);         // access entry
    void print();                       // print to screen
    int rows();                         // number of rows
    int cols();                         // number of columns
};
```

Encapsulation

```
class Matrix
{
  public:
    // a list of public methods
  private:
    // a list of private methods and attributes
};
```

The keyword **public**: marks the description of the interface, i.e., those methods of the class which can be accessed from the outside.

The keyword **private**: accompanies the definition of attributes and methods that are *only available to objects of the same class*. This includes the data and implementation-specific methods reqired by the class. To ensure data integrity it should *not* be possible to access stored data from outside the class.

Definition of Methods

```
class Matrix
{
  public:
    // ...
    double& elem(int i, int j)
    {
      return entries[i][j];
    }
};
```

The method definition (i.e., listing of the actual function code) can be placed directly in the class (so-called inline functions). In the case of inline functions the compiler can omit the function call and use the code directly.

Definition of Methods II

```
void Matrix::Matrix(int numRows_, int numCols_)
{
  entries.resize(numRows_);
  for (int i = 0; i < entries.size(); i++)
      entries[i].resize(numCols_);
  numRows = numRows_;
  numCols = numCols_;
}
```

If methods are defined outside the definition of a class, then the name of the method must be prefixed with the name of the class followed by two colons.

Generic Programming

Often the same algorithms are required for different types of data. Writing them again and again is tedious and error-prone.

```
int square(int x)
{
    return(x*x);
}
long square(long x)
{
    return(x*x);
}
```

```
float square(float x)
{
    return(x*x);
}
double square(double x)
{
    return(x*x);
}
```

Generic programming makes it possible to write an algorithm once and parameterize it with the data type. The language device for this is called **template** in C++ and can be used for functions, classes, and variables.

Function Templates

A function template starts with the keyword **template** and a list of template arguments, separated by commas and enclosed by angle brackets:

```
template<typename T>
T square(T x)
{
    return(x*x);
}
```

This way, the function basically has two types of arguments:

- Types, specified in angle brackets
- Variables, specified in parentheses

This becomes clearer when actually calling the function (see below).

Template Instantiation

At the first use of the function with a specific combination of data types the compiler automatically generates code for these types. This is called *template instantiation*, and has to be unambiguous.

Ambiguities can be avoided through:

- Explicit type conversion of arguments
- Explicit specification of template arguments in angle brackets:

```
std::cout << square<int>(4) << std::endl;
```

The argument types must match the declaration and the types have to provide all the necessary operations (e.g. the `operator*()`).

Example: Unary Function Template

```cpp
#include <cmath>
#include <iostream>

template<typename T>
T square(T x)
{
  return(x*x);
}

int main()
{
  std::cout << square<int>    (4)    << std::endl;
  std::cout << square<double>(M_PI) << std::endl;
  std::cout << square        (3.14) << std::endl;
}
```

Example: Binary Function Template

```cpp
#include <iostream>

template<class U>
const U& maximum(const U& a, const U& b)
{
  if (a > b)
    return a;
  else
    return b;
}

int main()
{
  std::cout << maximum(1,4)           << std::endl;
  std::cout << maximum(3.14,7.)       << std::endl;
  std::cout << maximum(6.1,4)         << std::endl; // comp. error
  std::cout << maximum<double>(6.1,4) << std::endl; // unambiguous
  std::cout << maximum(6.1,double(4)) << std::endl; // unambiguous
  std::cout << maximum<int>(6.1,4)    << std::endl; // warning
}
```

Specialization of Function Templates

It is possible to define special template functions for certain parameter values. This is called *template specialization*. It can, e.g., be used for speed optimizations:

```
template <typename V, size_t N>
double scalarProduct(const V& a, const V& b)
{
    double result = 0;
    for (size_t i = 0; i < N; ++i)
        result += a[i] * b[i];
    return result;
};

template<typename V>
double scalarProduct<V,2>(const V& a, const V& b)
{
    return a[0] * b[0] + a[1] * b[1];
};
```

Class Templates

In addition to function templates, it is often also useful to parameterize classes. Here is a simple stack as an example:

```
template<typename T>
class Stack
{
  private:
    std::vector<T> elems; // storage for elements of type T

  public:
    void push(const T&);  // put new element on top of storage
    void pop();           // retrieve uppermost element
    T top() const;        // look at uppermost element
    bool empty() const    // check if stack is empty
    {
      return elems.empty();
    }
};

// + implementations of push, pop, and top
```

Further Reading

Online Tutorials

`http://www.cplusplus.com/doc/tutorial/`

Quick Reference

`https://en.cppreference.com/w/`

Other Resources

`https://isocpp.org/get-started/`

Best Practices for Scientific Computing

G. Wilson, D.A. Aruliah, C.T. Brown, N.P.C. Hong, M. Davis, R.T. Guy, S.H.D. Haddock, K.D. Huff, I.M. Mitchell, M.D. Plumbley, B. Waugh, E.P. White, P. Wilson: *Best Practices for Scientific Computing*, PLOS Biology

1. Write programs for people, not computers
2. Let the computer do the work
3. Make incremental changes
4. Don't repeat yourself (or others)
5. Plan for mistakes
6. Optimize software only after it works correctly
7. Document design and purpose, not mechanics
8. Collaborate

Write Programs for People, not Computers

Software must be easy to read and understand by other programmers (especially your future self!)

- Programs should not require readers to hold more than a handful of facts in memory at once
  - Short-term memory: The Magical Number Seven, Plus or Minus Two
  - Chunking (psychology): binding individual pieces of information into meaningful whole
  - Good reason for encapsulation and modularity
- Make names and identifiers consistent, distinctive, and meaningful
- Make coding style and formatting consistent

Minimum Style Requirements

Programming style conventions can be inconvenient, but there is a bare minimum that should be followed in any case:

- Properly indent your code, use meaningful names for variables and functions
- Comment your code, but *don't* comment every line: main idea/purpose of class or function, hidden assumptions, critical details
- Don't just fix compilation errors, also make sure that the compiler doesn't issue warnings

This makes it easier for other people to understand your work, including:

- Colleagues you may ask for help or input
- Other researchers extending or using your work
- Yourself in a few weeks or months (!)

Let the Computer do the Work[1]

Typing the same commands over and over again is both time-consuming and error-prone

- Make the computer repeat tasks (*i.e., write shell scripts or python scripts*)
- Save recent commands in a file for re-use (*actually, that's already done for you, search for "reverse-i-search" online*)
- Use a build tool to automate workflows (*e.g., makefiles, cmake*)

But make sure you don't waste time building unnecessarily intricate automation structures!

[1] *italics: my personal remarks*

Make Incremental Changes

Requirements of scientific software typically aren't known in advance

- Work in small steps with frequent feedback and course correction (e.g., agile development)
- Use a version control system (e.g., Git, GitLab)
- Put everything that has been created manually under version control
  - The source code, of course
  - Source files for papers / documents
  - Raw data (from field experiments or benchmarks)

Large chunks of data (data from experiments, important program results, figures) can be stored efficiently using Git-LFS (large file storage) extension

Don't Repeat Yourself (or Others)

Anything that exists in two or more places is harder to maintain and may introduce inconsistencies

- Every piece of data must have a single authoritative representation in the system
- Modularize code rather than copying and pasting
- Re-use code instead of rewriting it
- Make use of external libraries (*as long as it is not too cumbersome*)

First point is actually known as DRY principle in general software design and engineering

Plan for Mistakes

Mistakes are human and happen on all levels of software development (e.g. bug, misconception, problematic design choice), even in experienced teams

- Add assertions to programs to check their operation
  - Simple form of error detection
  - Can serve as documentation (that is auto-updated)
- Use an off-the-shelf unit testing library
  - Can assist in finding unexpected side effects of code changes
  - In case of GitLab: GitLab CI (continuous integration)
- Turn bugs into test cases
- Use a symbolic debugger (e.g. GDB, DDD, Gnome Nemiver, LLDB)

Assertions should only be used to catch misconceptions and omissions by programmers, not expected runtime errors (file not found, out of memory, solver didn't converge)!

The Importance of Backups

Losing your program code means losing months or even years of hard work. There are three standard ways to guard against this possibility:

- Create automatic backups (*but know how the restore process works!*)
- Create a manual copy on an external drive or another computer (*messy!*)
- Use a source code repository to manage the files

It's a good idea to use two approaches as a precaution

Of the above options, using a repository has the largest number of benefits

The Importance of Backups

Advantages of using a repository for code management:

- The repository can be on a different computer, i.e., the repository is automatically also an old-fashioned backup
- Repositories preserve project history, making it easy to retrieve older code or compare versions
- Repositories can be shared with others for collaboration

One of the tools most often used for this is Git, which can be used in conjunction with websites like GitHub or Bitbucket. An open source alternative is GitLab, which can also be installed locally to provide repositories for the research group.

Optimize Software Only After it Works Correctly

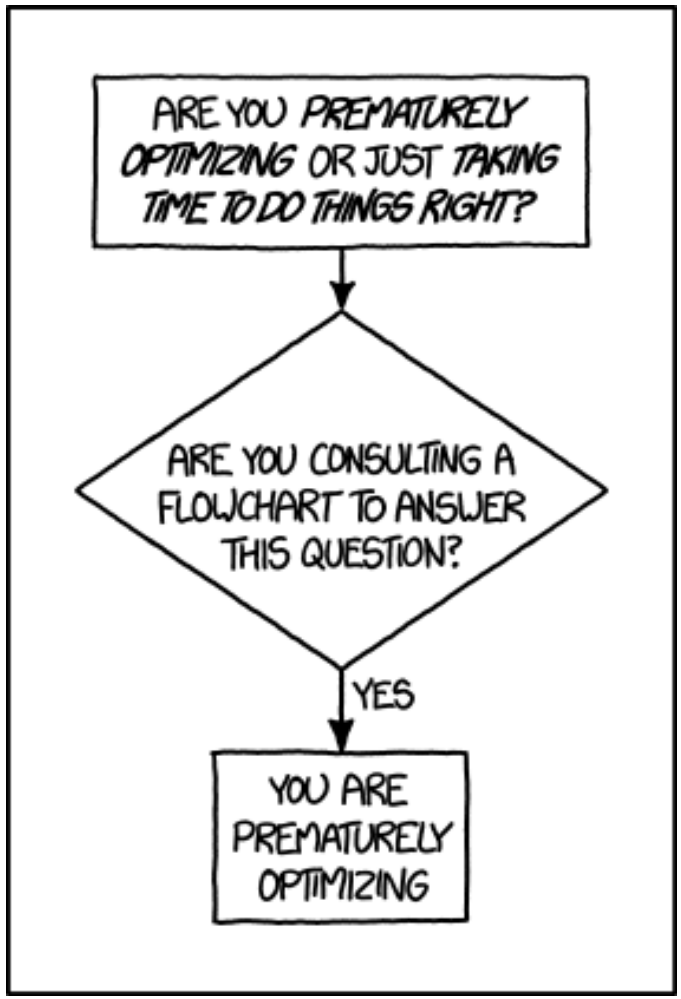
Identifying potential bottlenecks is hard, and optimization of other parts of the program is a waste of time

- Use a profiler to identify bottlenecks
- Write code in the highest-level language possible

This is an optimization with regard to development time

High-level prototypes (e.g. in Python, Matlab) can serve as oracles for low-level high-performance implementations (e.g. in C++)

Code and Coding Efficiency



Source: Randall Munroe, xkcd.com

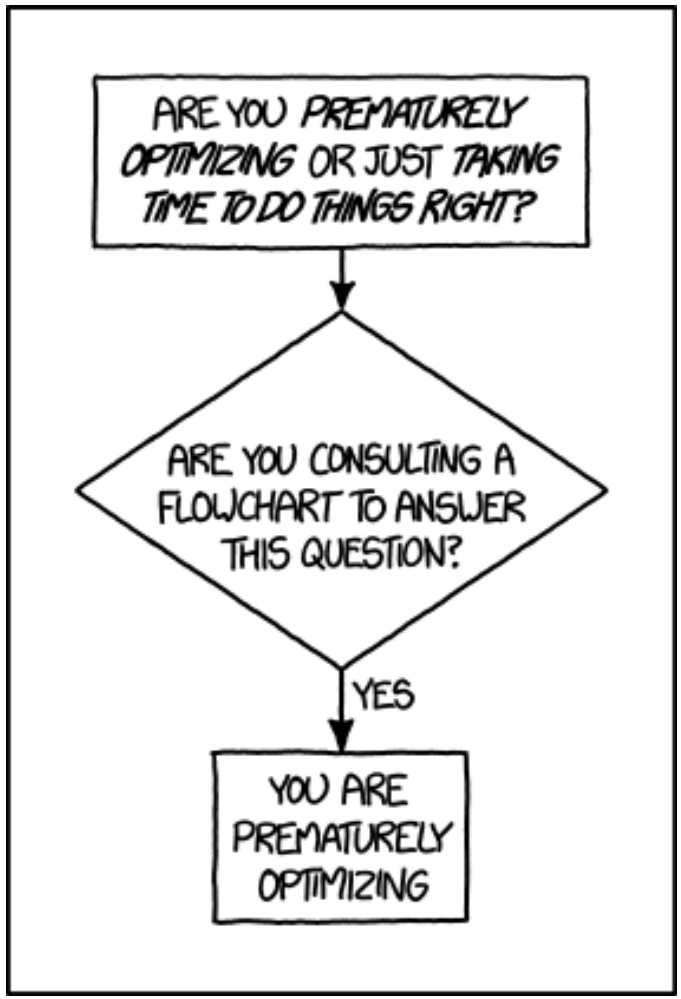
The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.
— Tom Cargill, Bell Labs

Even experts are often unable to accurately predict the time requirements for developing a piece of software

This means:

- Project plans should be conservative
- Don't underestimate the work needed for the hard parts of coding

Code and Coding Efficiency



Source: Randall Munroe, xkcd.com

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.
— Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid

As a result, there are two measures of efficiency, the time your program needs to run, and the time you need to write it

- Don't optimize too soon, get a working prototype first (you will likely need to change optimized parts along the way)
- Look for bottlenecks and only optimize those (your program will spend most of its time in roughly 5% of your code)

Document Design and Purpose, not Mechanics

The main purpose of documentation is assistance of readers who are trying to use an implementation, choosing between different implementations, or planning to extend one

- Document interfaces and reasons, *not* implementations
  - Function and method signatures
  - Public members of classes
- Refactor code in preference to explaining how it works
- Embed documentation for a piece of software in that software

Refactored code that needs less documentation will often save time in the long run, and documentation that is bundled with code has much higher chance to stay relevant

Collaborate

Scientific programs are often produced by research teams, and this provides both unique opportunities and potential sources of conflict

- Use pre-merge code reviews
- Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems
- Use an issue tracking tool

Pre-merge reviews are the only way to guarantee that code has been checked by another person

Pair programming is very efficient but intrusive, and should be used with care

Contents I

Positional Notation

Definition 1 (Positional Notation)

System representing numbers $x \in \mathbb{R}$ using:

$$x = \pm \ldots m_2\beta^2 + m_1\beta + m_0 + m_{-1}\beta^{-1} + m_{-2}\beta^{-2} \ldots$$
$$= \sum_{i \in \mathbb{Z}} m_i \beta^i$$

$\beta \in \mathbb{N}, \beta \geq 2$, is called *base*,
$m_i \in \{0, 1, 2, \ldots, \beta - 1\}$ are called *digits*

History:

- Babylonians (≈ -1750), $\beta = 60$
- Base 10 from ~ 1580
- Pascal: all values $\beta \geq 2$ may be used

Fixed-Point Numbers

Fixed-point numbers: truncate series after finite number of terms

$$x = \pm \sum_{i=-k}^{n} m_i \beta^i$$

Problem: scientific applications use numbers of very different orders of magnitude

Planck constant:	$6.626093 \cdot 10^{-34}$ Js
Avogadro constant:	$6.021415 \cdot 10^{23}$ $\frac{1}{\text{mol}}$
Electron mass:	$9.109384 \cdot 10^{-31}$ kg
Speed of light:	$2.997925 \cdot 10^{8}$ $\frac{\text{m}}{\text{s}}$

Floating-point numbers can represent all such numbers with acceptable accuracy

Floating-Point Numbers

Definition 2 (Floating-Point Numbers)

Let $\beta, r, s \in \mathbb{N}$ and $\beta \geq 2$. The *set of floating-point numbers* $\mathbb{F}(\beta, r, s) \subset \mathbb{R}$ consists of all numbers with the following properties:

1. $\forall x \in \mathbb{F}(\beta, r, s) \colon x = m(x) \cdot \beta^{e(x)}$ with

$$m(x) = \pm \sum_{i=1}^{r} m_i \beta^{-i}, \quad e(x) = \pm \sum_{j=0}^{s-1} e_j \beta^j$$

   with digits m_i and e_j.
   m is called *mantissa*, e is called *exponent*.

2. $\forall x \in \mathbb{F}(\beta, r, s) \colon x = 0 \vee m_1 \neq 0$. This is called *normalization* and makes the representation unique.

Example

Example 3

1. $\mathbb{F}(10, 3, 1)$ consists of the numbers

$$x = \pm(m_1 \cdot 0.1 + m_2 \cdot 0.01 + m_3 \cdot 0.001) \cdot 10^{\pm e_0}$$

with $m_1 \neq 0 \vee (m_1 = m_2 = m_3 = 0)$, e.g., 0, $0.999 \cdot 10^4$, and $0.123 \cdot 10^{-1}$, but *not* $0.140 \cdot 10^{-10}$ (exponent to small)

2. $\mathbb{F}(2, 2, 1)$ consists of the numbers

$$x = \pm(m_1 \cdot 0.5 + m_2 \cdot 0.25) \cdot 2^{\pm e_0}$$

$$\implies \mathbb{F}(2, 2, 1) = \left\{ -\frac{3}{2}, -1, -\frac{3}{4}, -\frac{1}{2}, -\frac{3}{8}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{3}{8}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2} \right\}$$

Standard: IEEE 754 / IEC 559

Goal: portability of programs with floating-point arithmetics
Finalized 1985

$\beta = 2$, with four levels of accuracy and normalized representation:

	single	single-ext	double	double-ext
e_{max}	127	$\geq$ 1024	1023	$\geq$ 16384
e_{min}	-126	$\leq$ -1021	-1022	$\leq$ -16381
Bits expon.	8	$\leq$ 11	11	$\geq$ 15
Bits total	32	$\geq$ 43	64	$\geq$ 79

The standard defines four kinds of rounding:
to $-\infty$, to $+\infty$, to 0, and to nearest

Since 2008: additionally half precision and quadruple precision

Double Precision

Let's have a closer look at double precision:

- 64 bit in total
- 11 bit for exponent, stored without sign as $c \in [1, 2046]$
- Let $e := c - 1023 \implies e \in [-1022, 1023]$, no sign necessary
- The values $c \in \{0, 2047\}$ are special:
  - $c = 0 \wedge m = 0$ encodes zero
  - $c = 0 \wedge m \neq 0$ encodes denormalized representation
  - $c = 2047 \wedge m = 0$ encodes ∞ (overflow)
  - $c = 2047 \wedge m \neq 0$ encodes NaN = "not a number", e.g., when dividing by zero

Double Precision

- $64 - 11 = 53$ bit for mantissa, one for sign, 52 bit remaining for mantissa digits
- $\beta = 2$ implies $m_1 = 1$
- This digit is called *hidden bit* and is never stored
- Therefore $r = 53$ in the sense of our definition of floating-point numbers

Double precision corresponds to $\mathbb{F}(2, 53, 10)$ + additional special codes.

Rounding Function

To approximate $x \in \mathbb{R}$ in $\mathbb{F}(\beta, r, s)$, we need a map

$$\mathrm{rd}\colon D(\beta, r, s) \to \mathbb{F}(\beta, r, s), \tag{1}$$

where $D(\beta, r, s) \subset \mathbb{R}$ is the domain containing $\mathbb{F}(\beta, r, s)$:

$$D := [X_-, x_-] \cup \{0\} \cup [x_+, X_+]$$

with $X_{+/-}$ being the numbers in $\mathbb{F}(\beta, r, s)$ with largest absolute value, and $x_{+/-}$ those with the smallest (apart from zero).

Note: this implies that x lies within the representable domain!

A reasonable demand is:

$$\forall x \in D\colon\ |x - \mathrm{rd}(x)| = \min_{y \in \mathbb{F}} |x - y|$$

(known as *best approximation property*)

Rounding Function

With $l(x) := \max\{y \in \mathbb{F} | y \leq x\}$ and $r(x) := \min\{y \in \mathbb{F} | y \geq x\}$ we have:

$$\mathrm{rd}(x) = \begin{cases} x & l(x) = r(x), x \in \mathbb{F} \\ l(x) & |x - l(x)| < |x - r(x)| \\ r(x) & |x - l(x)| > |x - r(x)| \\ ? & |x - l(x)| = |x - r(x)| \end{cases}$$

The last case requires further considerations. There are several possible choices.

Natural Rounding

Definition 4 (Natural Rounding)

Let $x = \text{sign}(x) \cdot \left(\sum_{i=1}^{\infty} m_i \beta^{-i}\right) \beta^e$ the *normalized* representation of $x \in D$. Define

$$\text{rd}(x) := \begin{cases} l(x) = \text{sign}(x) \cdot \left(\sum_{i=1}^{r} m_i \beta^{-i}\right) \beta^e & \text{if } 0 \leq m_{r+1} < \beta/2 \\ r(x) = l(x) + \beta^{e-r} \,(\text{last digit}) & \text{if } \beta/2 \leq m_{r+1} < \beta \end{cases}$$

This is the usual rounding everyone knows from school. It has the undesirable property of introducing bias, since rounding up is slightly more likely.
This is irrelevant in everyday life, but becomes important for small β, e.g., $\beta = 2$, and/or many operations (as in scientific computing).

Even Rounding

Definition 5 (Even Rounding)

Let (with notation as before)

$$\mathrm{rd}(x) := \begin{cases} l(x) & \text{if } |x - l(x)| < |x - r(x)| \\ l(x) & \text{if } |x - l(x)| = |x - r(x)| \wedge m_r \text{ even} \\ r(x) & \text{else} \end{cases}$$

This ensures that m_r in $\mathrm{rd}(x)$ is always even after rounding.

- For $\mathrm{rd}(x) = l(x)$ this is by definition.
- Else $\mathrm{rd}(x) = r(x) = l(x) + \beta^{e-r}$, m_r in $l(x)$ is odd, and addition of β^{e-r} changes the last digit by one.

This choice of rounding avoids systematic drift when rounding up, and corresponds to "round to nearest" in the standard.

Absolute and Relative Error

Definition 6 (Absolute and Relative Error)

Let $x' \in \mathbb{R}$ an approximation of $x \in \mathbb{R}$. Then we call

$$\Delta x := x' - x \quad \text{absolute error}$$

and for $x \neq 0$

$$\epsilon_{x'} := \frac{\Delta x}{x} \quad \text{relative error}$$

Rearranging leads to:

$$x' = x + \Delta x = x \cdot \left(1 + \frac{\Delta x}{x}\right) = x \cdot (1 + \epsilon_{x'})$$

Motivation

Motivation:

Let $\Delta x = x' - x = 100\,\text{km}$.
For $x =$ Distance Earth—Sun $\approx 1.5 \cdot 10^8\,\text{km}$,

$$\epsilon_{x'} = \frac{10^2\,\text{km}}{1.5 \cdot 10^8\,\text{km}} \approx 6.6 \cdot 10^{-7}$$

is relatively small.
But for $x =$ Distance Heidelberg—Paris $\approx 460\,\text{km}$,

$$\epsilon_{x'} = \frac{10^2\,\text{km}}{4.6 \cdot 10^2\,\text{km}} \approx 0.22 \quad (22\%)$$

is relatively large.

Error Estimation

Lemma 7 (Rounding Error)

When rounding in $\mathbb{F}(\beta, r, 2)$ the absolute error fulfills

$$|x - \mathrm{rd}(x)| \leq \frac{1}{2}\beta^{e(x)-r} \tag{2}$$

and the relative error (for $x \neq 0$)

$$\frac{|x - \mathrm{rd}(x)|}{|x|} \leq \frac{1}{2}\beta^{1-r}.$$

This estimate is sharp (i.e., the case "=" exists).
The number $\mathrm{eps} := \frac{1}{2}\beta^{1-r}$ is called *machine precision*.
$\mathrm{eps} = 2^{-24} \approx 6 \cdot 10^{-8}$ for single precision, and
$\mathrm{eps} = 2^{-53} \approx 1 \cdot 10^{-16}$ for double precision.

Floating-Point Arithmetics

We need arithmetics on $\mathbb{F}$:

$$\circledast : \mathbb{F} \times \mathbb{F} \to \mathbb{F} \quad \text{with} \quad \circledast \in \{\oplus, \ominus, \odot, \oslash\}$$

corresponding to the well-known operations $* \in \{+, -, \cdot, /\}$ on $\mathbb{R}$.

Problem: typically $x, y \in \mathbb{F} \not\Longrightarrow x * y \in \mathbb{F}$

Therefore the result has to be rounded. We *define*

$$\forall x, y \in \mathbb{F} : x \circledast y := \mathrm{rd}(x * y) \tag{3}$$

This guarantees "exact rounding". The implementation of such a mapping is nontrivial!

Guard Digit

Example 8 (Guard Digit)

Let $\mathbb{F} = \mathbb{F}(10, 3, 1), x = 0.215 \cdot 10^8, y = 0.125 \cdot 10^{-5}$. We consider the subtraction $x \ominus y = \mathrm{rd}(x - y)$.

1. Subtraction followed by rounding requires an extreme number of mantissa digits $\mathcal{O}(\beta^s)$!
2. Rounding before subtraction seems to produce same result. Good idea?
3. But: consider, e.g., $x = 0.101 \cdot 10^1, y = 0.993 \cdot 10^0$
   $\Longrightarrow$ relative error 18% $\approx$ 35 eps
4. One, two additional digits are enough to achieve exact rounding!
5. These digits are called *guard digits* and are also used in practice (CPU), e.g., performing internal computations in 80 bit precision.

Table Maker Dilemma

Algebraic functions:
e.g., polynomials, $1/x$, $\sqrt{x}$, rational functions, ...
more or less: finite combination of basic arithmetic operations and roots

Transcendent functions:
everything else, e.g., $\exp(x), \ln(x), \sin(x), x^y, \ldots$

Table Maker Dilemma:
One cannot decide a priori how many guard digits are reqired to achieve exact rounding for a given combination of transcendent function f and argument x.

IEEE754 guarantees exact rounding for $\oplus, \ominus, \odot, \oslash$, and $\sqrt{x}$.

Further Problems / Properties

The following has to be considered:

- Floating-point arithmetics don't have the associative and distributive properties, i.e., the order of operations matters!
- There is $y \in \mathbb{F}, y \neq 0$, so that $x \oplus y = x$
- Example: $(\epsilon \oplus 1) \ominus 1 = 1 \ominus 1 = 0 \neq \epsilon = \epsilon \oplus 0 = \epsilon \oplus (1 \ominus 1)$
- But the commutative property holds:
  $x \circledast y = y \circledast x$ for $\circledast \in \{\oplus, \odot\}$
- Some further simple rules that are valid:
  - $(-x) \odot y = -(x \odot y)$
  - $1 \odot x = x \oplus 0 = x$
  - $x \odot y = 0 \implies x = 0 \vee y = 0$
  - $x \odot z \leq y \odot z \quad \text{if} \quad x \leq y \wedge z > 0$

Contents I

Error Analysis

Rounding errors are propagated by computations.

- Let $F\colon \mathbb{R}^m \to \mathbb{R}^n$, in components $F(x) = \begin{pmatrix} F_1(x_1, \ldots, x_m) \\ \vdots \\ F_n(x_1, \ldots, x_m) \end{pmatrix}$
- Compute F in a computer using *numerical realization* $F'\colon \mathbb{F}^m \to \mathbb{F}^n$.
  F' is an *algorithm*, i.e., consists of
  - finitely many (= termination)
  - elementary (= known, i.e., $\oplus, \ominus, \odot, \oslash$)

  operations:

$$F'(x) = \varphi_l(\ldots \varphi_2(\varphi_1(x)) \ldots)$$

Error Analysis

Important:

1. A given F typically has many different realizations, because of different *orders of computation*

$$a + b + c \approx (a \oplus b) \oplus c \neq a \oplus (b \oplus c)!$$

2. Every step φ_i contributes some (unknown) error.
3. In principle, the computational accuracy can be improved arbitrarily, i.e., we have a sequence $(F')^{(k)} : (\mathbb{F}^{(k)})^m \to (\mathbb{F}^{(k)})^n$. But in the following we consider only a given fixed finite precision.

Error Analysis

$$F(x) - F'(\mathrm{rd}(x)) = \underbrace{F(x) - F(\mathrm{rd}(x))}_{\text{conditional analysis}} + \underbrace{F(\mathrm{rd}(x)) - F'(\mathrm{rd}(x))}_{\text{rounding error analysis}} \quad (4)$$

Where:

- $F(x)$: exact result
- $F'(\mathrm{rd}(x))$: numerical evaluation
- $F(\mathrm{rd}(x))$: exact result for $\mathrm{rd}(x) \approx x$

From now on:

- "first order" analysis
- absolute / relative errors

Differential Condition Analysis

We assume that $F\colon \mathbb{R}^m \to \mathbb{R}^n$ is twice continuously differentiable. Taylor's theorem holds for the components F_i:

$$F_i(x+\Delta x) = F_i(x) + \sum_{j=1}^{m} \frac{\partial F_i}{\partial x_j}(x)\Delta x_j + R_i^F(x;\Delta x) \quad i = 1,\ldots,n.$$

The remainder is

$$R_i^F(x;\Delta x) = \mathcal{O}\left(\|\Delta x\|^2\right),$$

i.e., the approximation error is quadratic in Δx.

Differential Condition Analysis

Therefore, we can rearrange Taylor's formula:

$$F_i(x + \Delta x) - F_i(x) = \underbrace{\sum_{j=1}^{m} \frac{\partial F_i}{\partial x_j}(x)\Delta x_j}_{\text{leading (first) order}} + \underbrace{R_i^F(x; \Delta x)}_{\text{higher orders}}$$

One often omits higher order terms and writes "$\doteq$" instead of "$=$".

Differential Condition Analysis

Then we have:

$$\frac{F_i(x+\Delta x)-F_i(x)}{F_i(x)} \doteq \sum_{j=1}^{m} \frac{\partial F_i}{\partial x_j}(x)\frac{\Delta x_j}{F_i(x)} \tag{5}$$

$$\doteq \sum_{j=1}^{m} \underbrace{\left(\frac{\partial F_i}{\partial x_j}(x)\frac{x_j}{F_i(x)}\right)}_{\text{amplification factor } k_{ij}(x)} \cdot \underbrace{\left(\frac{\Delta x_j}{x_j}\right)}_{\leq \text{eps}},$$

i.e., the amplification factors $k_{ij}(x)$ specify how (relative) input errors $\frac{\Delta x_j}{x_j}$ contribute to (relative) errors in the i-th comp. of F!

Condition

Definition 9 (Condition)

We call the evaluation $y = F(x)$ "ill-conditioned" in point x, iff $|k_{ij}(x)| \gg 1$, else "well-conditioned".
$|k_{ij}(x)| < 1$ is error dampening, $|k_{ij}(x)| > 1$ is error amplification.

The symbol "$\gg$" means "much larger than". Normally this means one number is several orders of magnitude larger than another (e.g., 1 million $\gg$ 1).

This definition is a continuum: there is no sharp separation between "well-conditioned" and "ill-conditioned"!

Example I

Example 10

1. Addition: $F(x_1, x_2) = x_1 + x_2$, $\frac{\partial F}{\partial x_1} = \frac{\partial F}{\partial x_2} = 1$.
   According to our formula:

$$\frac{F(x_1 + \Delta x_1, x_2 + \Delta x_2) - F(x_1, x_2)}{F(x_1, x_2)}$$

$$\doteq \underbrace{1 \cdot \frac{x_1}{x_1 + x_2}}_{=k_1} \frac{\Delta x_1}{x_1} + \underbrace{1 \cdot \frac{x_2}{x_1 + x_2}}_{=k_2} \frac{\Delta x_2}{x_2}$$

   Ill-conditioned for $x_1 \to -x_2$!

Example II

Example 10

❷ $F(x_1, x_2) = x_1^2 - x_2^2$, $\frac{\partial F}{\partial x_1} = 2x_1$, $\frac{\partial F}{\partial x_2} = -2x_2$.

$$\frac{F(x_1 + \Delta x_1, x_2 + \Delta x_2) - F(x_1, x_2)}{F(x_1, x_2)}$$

$$\doteq \underbrace{2x_1 \cdot \frac{x_1}{x_1^2 - x_2^2}}_{=k_1} \frac{\Delta x_1}{x_1} + \underbrace{(-2x_2) \cdot \frac{x_2}{x_1^2 - x_2^2}}_{=k_2} \frac{\Delta x_2}{x_2}$$

$$\Longrightarrow \quad k_1 = \frac{2x_1^2}{x_1^2 - x_2^2}, k_2 = -\frac{2x_2^2}{x_1^2 - x_2^2},$$

Ill-conditioned for $|x_1| \approx |x_2|$.

Contents I

Rounding Error Analysis

Also known as "forward rounding error analysis", there are other variants.

After error decomposition, Eq. (4):
consider $F(x) - F'(x)$ with $x \in \mathbb{F}^m$, F' "composed" from single operations $\circledast \in \{\oplus, \ominus, \odot, \oslash\}$

Eq. (3) (exactly rounded arithmetics) and Lemma 7 (rounding error) imply

$$\frac{(x \circledast y) - (x * y)}{(x * y)} = \epsilon \quad \text{with } |\epsilon| \leq \text{eps}$$

Careful, ϵ depends on x and y, and therefore is different for each individual operation!

$$\implies x \circledast y = (x * y) \cdot (1 + \epsilon) \quad \text{for an } |\epsilon(x, y)| \leq \text{eps}$$

Example I

Example 11

$F(x_1, x_2) = x_1^2 - x_2^2$ with two different realizations:

1. $F_a(x_1, x_2) = (x_1 \odot x_1) \ominus (x_2 \odot x_2)$
2. $F_b(x_1, x_2) = (x_1 \ominus x_2) \odot (x_1 \oplus x_2)$

First realization:

$$u = x_1 \odot x_1 = (x_1 \cdot x_1) \cdot (1 + \epsilon_1)$$
$$v = x_2 \odot x_2 = (x_2 \cdot x_2) \cdot (1 + \epsilon_2)$$
$$F_a(x_1, x_2) = u \ominus v = (u - v) \cdot (1 + \epsilon_3)$$
$$\frac{F_a(x_1, x_2) - F(x_1, x_2)}{F(x_1, x_2)} \doteq \frac{x_1^2}{x_1^2 - x_2^2}(\epsilon_1 + \epsilon_3) + \frac{x_2^2}{x_2^2 - x_1^2}(\epsilon_2 + \epsilon_3)$$

Example II

Example 11

Second realization:

$$u = x_1 \ominus x_2 = (x_1 - x_2) \cdot (1 + \epsilon_1)$$
$$v = x_1 \oplus x_2 = (x_1 + x_2) \cdot (1 + \epsilon_2)$$
$$F_b(x_1, x_2) = u \odot v = (u \cdot v) \cdot (1 + \epsilon_3)$$
$$\frac{F_b(x_1, x_2) - F(x_1, x_2)}{F(x_1, x_2)} \doteq \frac{x_1^2 - x_2^2}{x_1^2 - x_2^2}(\epsilon_1 + \epsilon_2 + \epsilon_3) = \epsilon_1 + \epsilon_2 + \epsilon_3$$

$\Longrightarrow$ second realization is better than first realization.

Numerical Stability

Definition 12 (Numerical Stability)

We call a numerical algorithm "numerically stable", if the rounding errors accumulated during computation have the same order of magnitude as the unavoidable problem error from condition analysis.

In other words:
Amplification factors from rounding analysis $\leq$ those from condition analysis $\implies$ "numerically stable"

Both realizations a, b from Ex. 11 are numerically stable.

Quadratic Equation

Let $p^2/4 > q \neq 0$, then the equation

$$y^2 - py + q = 0$$

has two real and separate solutions

$$y_{1,2} = f_{\pm}(p,q) = \frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}. \quad \text{(defines two } f\text{!)}$$

Condition analysis with $D := \sqrt{\frac{p^2}{4} - q}$:

$$\frac{f(p+\Delta p, q+\Delta q) - f(p,q)}{f(p,q)}$$
$$\doteq \left(1 \pm \frac{p}{2D}\right) \frac{p}{p \pm 2D} \frac{\Delta p}{p} - \frac{q}{D\,(p \pm 2D)} \frac{\Delta q}{q}$$

Quadratic Equation

This means:

- For $\frac{p^2}{4} \gg q$ and $p < 0$

$$f_-(p,q) = \frac{p}{2} - \sqrt{\frac{p^2}{4} - q}$$

  is well-conditioned.
- For $\frac{p^2}{4} \gg q$ and $p > 0$

$$f_+(p,q) = \frac{p}{2} + \sqrt{\frac{p^2}{4} - q}$$

  is well-conditioned.
- For $\frac{p^2}{4} \approx q$ both f_+ and f_- are ill-conditioned, this cannot be avoided.

Quadratic Equation

Numerically handy evaluation for the case $\frac{p^2}{4} \gg q$:

$p < 0$:
Compute $y_2 = \frac{p}{2} - \sqrt{\frac{p^2}{4} - q}$, then $y_1 = \frac{q}{y_2}$ using Vieta's Theorem ($q = y_1 \cdot y_2$).

$p > 0$:
Compute $y_1 = \frac{p}{2} + \sqrt{\frac{p^2}{4} - q}$, then $y_2 = \frac{q}{y_1}$.

$\Longrightarrow$ every problem has to be considered individually!

Cancellation

The discussed examples contain the phenomenon of *cancellation*. It appears during

- addition $x_1 + x_2$ with $x_1 \approx -x_2$
- subtraction $x_1 - x_2$ with $x_1 \approx x_2$

Remark 13

Cancellation means extreme amplification of errors introduced *before* the addition or subtraction.

If $x_1, x_2 \in \mathbb{F}$ are *machine numbers*, then

$$\left| \frac{(x_1 \ominus x_2) - (x_1 - x_2)}{(x_1 - x_2)} \right| \leq \text{eps}$$

holds, so this is not problematic. The problem of cancellation only occurs if x_1 and x_2 already contain errors.

Example

Example 14

Consider $\mathbb{F} = \mathbb{F}(10, 4, 1)$.

$x_1 = 0.11258762 \cdot 10^2, x_2 = 0.11244891 \cdot 10^2$
$\implies \mathrm{rd}(x_1) = 0.1126 \cdot 10^2, \mathrm{rd}(x_2) = 0.1124 \cdot 10^2$

$x_1 - x_2 = 0.13871 \cdot 10^{-1}$, but $\mathrm{rd}(x_1) - \mathrm{rd}(x_2) = 0.2 \cdot 10^{-1}$

The result has not a single valid digit! Relative error:

$$\frac{0.2 \cdot 10^{-1} - 0.13871 \cdot 10^{-1}}{0.13871 \cdot 10^{-1}} \approx 0.44 \approx 883 \cdot \underbrace{\frac{1}{2} \cdot 10^{-3}}_{=\mathrm{eps}}!$$

Basic Rule

In the given example: error caused by rounding of arguments.

Source of errors is irrelevant, this also happens if x_1, x_2 contain errors from previous computation steps.

Rule 15

Employ potentially dangerous operations as soon as possible in algorithms, when the least possible amount of errors has been accumulated (compare Ex. 11).

Exponential Function

The function $\exp(x) = e^x$ can be written as a power series for all $x \in \mathbb{R}$:

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

Obvious approach: truncate calculation after n terms, $\exp(x) \approx \sum_{k=0}^{n} \frac{x^k}{k!}$.

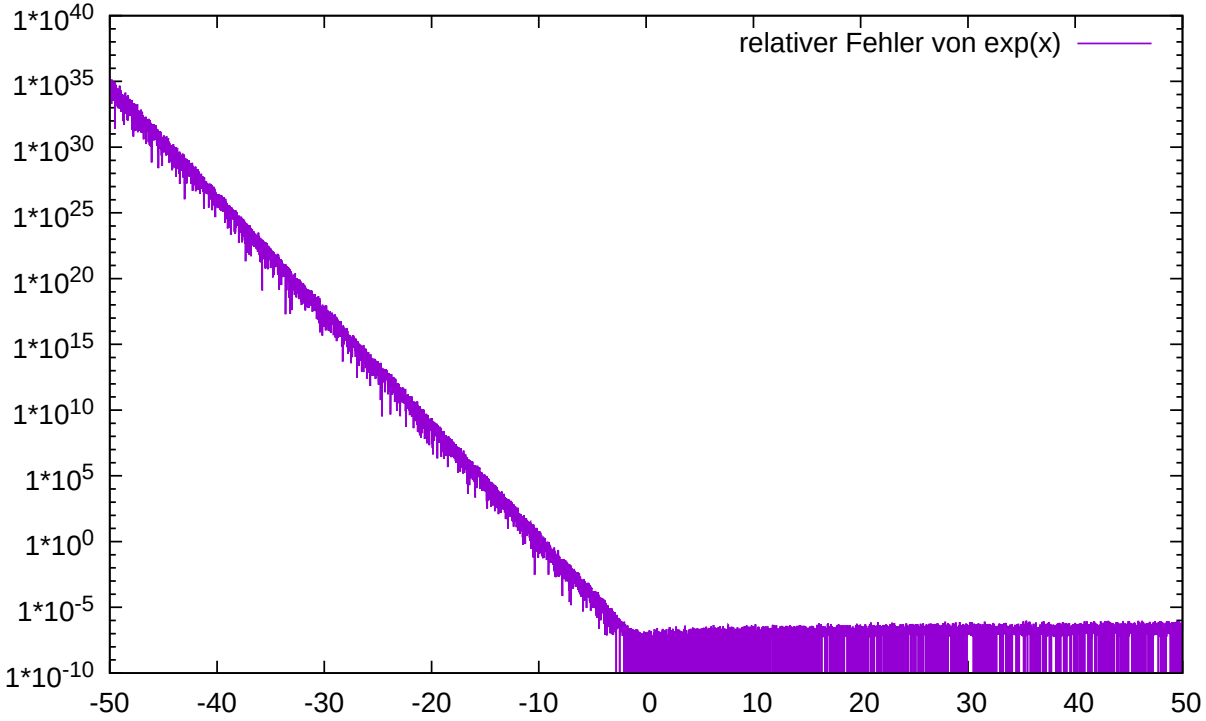
Use recursion:

$$y_0 := 1, \quad S_0 := y_0 = 1,$$
$$\forall k > 0: \quad y_k := \frac{x}{k} \cdot y_{k-1}, \quad S_k := S_{k-1} + y_k$$

y_n: terms of series, S_n: partial sums

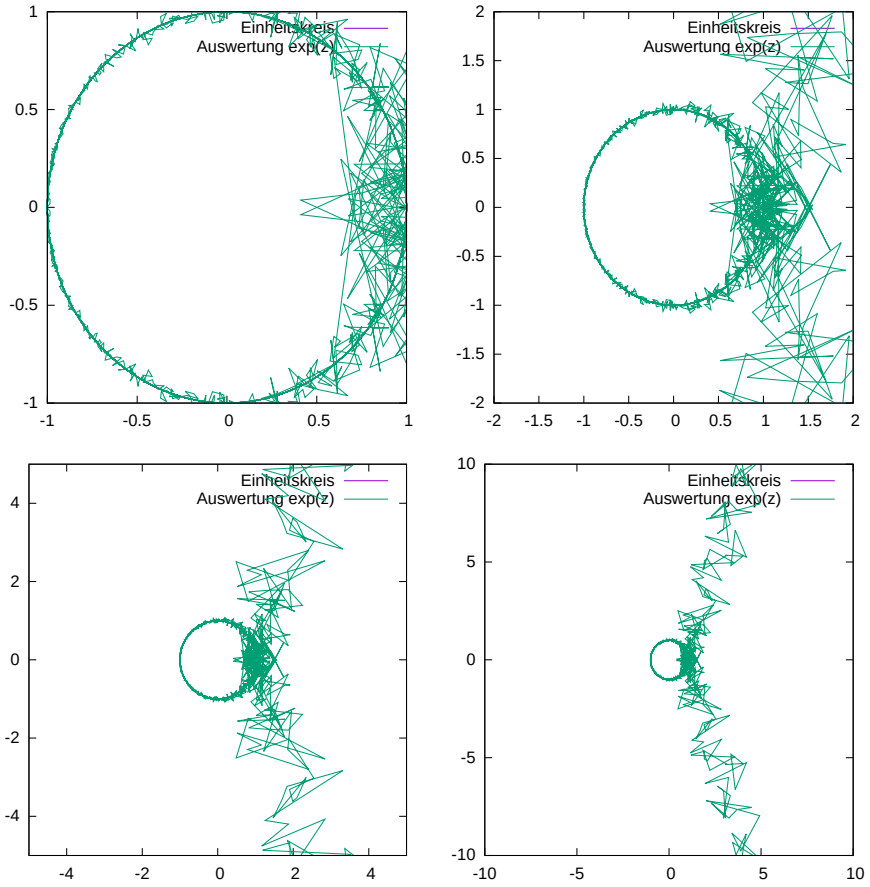
Error for Different Values of x

Results for recursion formula with **`float`**, $n = 100$:



- Negative values of x lead to arbitrarily large errors
- This effect is *not* caused by the truncation of the series!

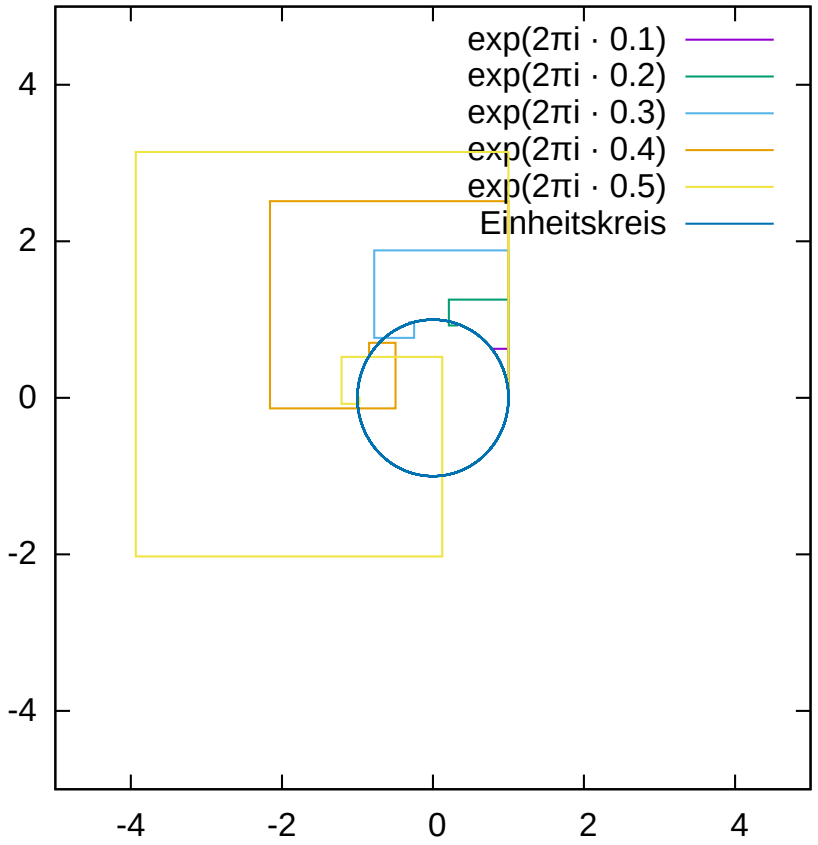
Deviations for Imaginary Arguments



Results for the imaginary interval $[-50, 50] \cdot i$

- For $|z| \leq \pi$ the result is somewhat acceptable
- For $|z| \to 2\pi$ the error continues to grow
- Then the values leave the circle (the trajectory approaches a straight line and won't return)

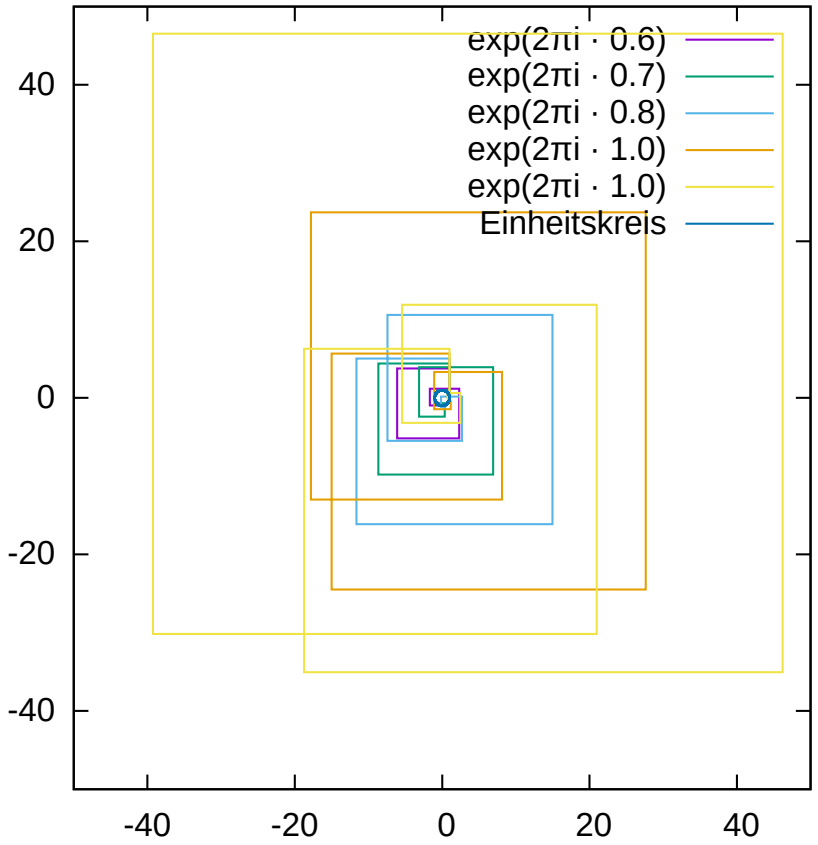
Visualization of Convergence Behavior



- even powers contribute to the real part of $\exp(2\pi i \cdot x)$
- odd powers contribute to the imaginary part

$\Longrightarrow$ addition of terms alternates between changes to real and imaginary part

Visualization of Convergence Behavior



- Absolute value of intermediate results grows exponentially in x
- Shape of trajectory looks more and more like a square

$\Longrightarrow$ cancellation

Condition Analysis for $\exp(x)$

For the function exp we have $\exp' = \exp$, and therefore:

$$\frac{\exp(x+\Delta x)-\exp(x)}{\exp(x)} \doteq \left(\exp'(x)\frac{x}{\exp(x)}\right)\cdot\left(\frac{\Delta x}{x}\right) = \Delta x$$

$\Longrightarrow$ absolute error of x becomes relative error of $\exp(x)$
(compare: exp is isomorphism between $(\mathbb{R},+)$ and $(\mathbb{R}_+,\cdot)$.)

$k = x$ means exp is well-conditioned if x is not too large
$\Longrightarrow$ considered algorithm is unstable for $x < 0$

Is there a more stable algorithm? $\rightsquigarrow$ exercise

Recursion Formula for Integrals

Integrals of the form

$$I_k = \int_0^1 x^k \exp(x)\, dx$$

can be solved using a recursion formula:

$$I_0 = e - 1, \quad \forall k > 0 \colon I_k = e - k \cdot I_{k-1}$$

We have a primitive integral for the first term in the sequence, because $\exp'(x) = \exp(x)$, other terms can be computed using the formula above.

How well does this work in practice?

Recursion Formula for Integrals

The first 26 values of $\{I_k\}_k$, computed with finite precision:

k	computed I_k	error $\lvert\Delta I_k\rvert$	k	computed I_k	error $\lvert\Delta I_k\rvert$
0	1.718281828459050	$2.6 \cdot 10^{-15}$	13	0.181983054536145	$3.3 \cdot 10^{-7}$
1	1	(zero)	14	0.170519064953013	$4.6 \cdot 10^{-6}$
2	0.718281828459045	$1.5 \cdot 10^{-15}$	15	0.160495854163853	$7.0 \cdot 10^{-5}$
3	0.563436343081910	$5.5 \cdot 10^{-16}$	16	0.150348161837404	$1.1 \cdot 10^{-3}$
4	0.464536456131406	$1.0 \cdot 10^{-15}$	17	0.162363077223183	$1.9 \cdot 10^{-2}$
5	0.395599547802016	$6.0 \cdot 10^{-15}$	18	-0.204253561558257	$3.4 \cdot 10^{-1}$
6	0.344684541646949	$3.8 \cdot 10^{-14}$	19	6.59909949806592	$6.7 \cdot 10^{0}$
7	0.305490036930402	$2.7 \cdot 10^{-13}$	20	-129.263708132859	$1.3 \cdot 10^{1}$
8	0.274361533015832	$2.1 \cdot 10^{-12}$	21	2717.25615261851	$2.7 \cdot 10^{3}$
9	0.249028031316559	$1.9 \cdot 10^{-11}$	22	-59776.9170757787	$6.0 \cdot 10^{4}$
10	0.228001515293454	$1.9 \cdot 10^{-10}$	23	1374871.81102474	$1.4 \cdot 10^{6}$
11	0.210265160231056	$2.1 \cdot 10^{-9}$	24	-32996920.7463119	$3.3 \cdot 10^{7}$
12	0.195099905686377	$2.5 \cdot 10^{-8}$	25	824923021.376079	$8.2 \cdot 10^{8}$

Recursion formula $I_k = e - k \cdot I_{k-1}$ leads to error amplification by a factor of k in k-th step!

Better Options

1. All I_k are of the form $a \cdot e + b$, where $a, b \in \mathbb{Z}$. Compute these numbers using the recursion formula, and use floating-point numbers only in the last step of computation.
2. Flip the recursion formula: if $I_k \to I_{k+1}$ amplifies the error by k, then $I_{k+1} \to I_k$ reduces it by k!

Because of $0 \leq x^k \leq 1$ and $0 \leq \exp(x) \leq 3$ on $[0, 1]$, $0 \leq I_k \leq 3$ must hold. If we more or less arbitrarily set, e.g., $I_{50} := 1.5$, then the error can be at most 1.5.

Use inverted recursion formula

$$I_k = (k+1)^{-1} \cdot (e - I_{k+1}).$$

Recursion Formula for Integrals

The values for I_k between $k = 25$ and 50, calculated backwards:

k	computed I_k	error $\lvert\Delta I_k\rvert$	k	berechnetes I_k	Fehler $\lvert\Delta I_k\rvert$
50	1.5	$1.4 \cdot 10^{0}$	37	0.0697442966294832	$2.8 \cdot 10^{-16}$
49	0.0243656365691809	$2.9 \cdot 10^{-2}$	36	0.0715820954548530	$1.9 \cdot 10^{-16}$
48	0.0549778814671401	$5.9 \cdot 10^{-4}$	35	0.0735194370278942	$2.8 \cdot 10^{-17}$
47	0.0554854988956647	$1.2 \cdot 10^{-5}$	34	0.0755646397551757	$2.2 \cdot 10^{-16}$
46	0.0566552410545400	$2.6 \cdot 10^{-7}$	33	0.0777269761383491	$2.7 \cdot 10^{-18}$
45	0.0578614475522718	$5.7 \cdot 10^{-9}$	32	0.0800168137066878	$1.8 \cdot 10^{-16}$
44	0.0591204529090394	$1.3 \cdot 10^{-10}$	31	0.0824457817110112	$2.6 \cdot 10^{-16}$
43	0.0604354858079547	$2.9 \cdot 10^{-12}$	30	0.0850269692499366	$2.9 \cdot 10^{-16}$
42	0.0618103800616533	$6.7 \cdot 10^{-14}$	29	0.0877751619736370	$4.5 \cdot 10^{-17}$
41	0.0632493201999379	$1.8 \cdot 10^{-15}$	28	0.0907071264305313	$4.3 \cdot 10^{-16}$
40	0.0647568904453441	$1.4 \cdot 10^{-16}$	27	0.0938419536438755	$4.7 \cdot 10^{-16}$
39	0.0663381234503425	$2.1 \cdot 10^{-16}$	26	0.0972014768450063	$1.3 \cdot 10^{-17}$
38	0.0679985565386847	$1.5 \cdot 10^{-16}$	25	0.1008107827543860	$1.1 \cdot 10^{-16}$

Despite a completely unusable estimate for the initial value I_{50}, the new recursion formula $I_k = (k+1)^{-1} \cdot (e - I_{k+1})$ quickly leads to very good results!

Idea of Error Estimates

Error analysis for initial value $I_{k+m}, m \geq 1$:

$$|\Delta I_k| \approx \frac{k!}{(k+m)!}|\Delta I_{k+m}| \leq \frac{k!}{(k+m)!} \cdot 1.5 \leq (k+1)^{-m} \cdot 1.5$$

Idea: compute required number of steps m from desired accuracy $|\Delta I_k| < \text{tol}$.

$$(k+1)^{-m} \cdot 1.5 < \text{tol} \implies \exp(-m \cdot \ln(k+1)) < \frac{\text{tol}}{1.5}$$

$$\implies -m \cdot \ln(k+1) < \ln\left(\frac{\text{tol}}{1.5}\right) \implies m > \left|\frac{\ln(\text{tol}) - \ln(1.5)}{\ln(k+1)}\right|$$

Example: $k = 25$, $\text{tol} = 10^{-8} \implies m > 5.7$

Idea of Error Estimates

Result for $m = 6$:

k	computed I_k	error $\lvert\Delta I_k\rvert$
31	1.5	$1.4 \cdot 10^{0}$
30	0.0392994138212595	$4.6 \cdot 10^{-2}$
29	0.0892994138212595	$1.5 \cdot 10^{-3}$
28	0.0906545660219926	$5.3 \cdot 10^{-5}$
27	0.0938438308013233	$1.9 \cdot 10^{-6}$
26	0.0972014073206564	$7.0 \cdot 10^{-8}$
25	0.1008107854284000	$2.9 \cdot 10^{-9}$

- Inverted recursion formula is numerically stable, in contrast to naive approach
- Error estimate minimizes effort for prescribed accuracy

$\Longrightarrow$ stable and efficient

Contents I

Introduction

Goal: representation and evaluation of functions on a computer.

Typical applications:

- Reconstruction of a functional relationship between "measured function values", evaluation for additional arguments
- More efficient evaluation of very expensive functions
- Representation of fonts (2D), structures (3D) in a computer
- Data compression
- Solving differential and integral equations

Introduction

We restrict ourselves to functions of *one* variable, e.g.:

$$f \in C^r[a, b]$$

This is an *infinite dimensional* function space. Computers operate on function classes which are determined through *finitely many* parameters (not necessarily linear subspaces), e.g.:

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n \qquad \text{(polynomals)}$$

$$r(x) = \frac{a_0 + a_1 x + \cdots + a_n x^n}{b_0 + b_1 x + \cdots + b_m x^m} \qquad \text{(rational functions)}$$

$$t(x) = \frac{1}{2} a_0 + \sum_{k=1}^{n} (a_k \cos(kx) + b_k \sin(kx)) \qquad \text{(trigonom. polynomials)}$$

$$e(x) = \sum_{k=1}^{n} a_k \exp(b_k x) \qquad \text{(exponential sum)}$$

Approximation

Basic task of *approximation:*

Given a set of functions P (polynomials, rational functions, ...) and a function f (e.g., $f \in C[a,b]$), find $g \in P$, so that the error $f - g$ is minimized in a suitable fashion.

Examples:

$$\left(\int_a^b (f-g)^2 \, dx \right)^{1/2} \to \min \qquad \text{(2-norm)}$$

$$\max_{a \le x \le b} |f(x) - g(x)| \to \min \qquad (\infty\text{-norm})$$

$$\max_{i \in \{0,\ldots,n\}} |f(x_i) - g(x_i)| \to \min \qquad \text{for } a \le x_i \le b,\ i = 0, \ldots, n$$

Interpolation

Interpolation is a special case of approximation, where g is determined by

$$g(x_i) = y_i := f(x_i) \quad i = 0, \ldots, n$$

Special properties of interpolation:

- The error $f - g$ is only considered on a finite set of *nodes* $x_i, i = 0, \ldots, n$.
- In these finitely many points the deviation must be zero, not just minimal in some weaker sense.

Polynomial Interpolation

Let P_n the set of polynomials on $\mathbb{R}$ of degree smaller or equal $n \in \mathbb{N}_0$:

$$P_n := \{p(x) = \sum_{i=0}^{n} a_i x^i \mid a_i \in \mathbb{R}\}$$

P_n is an $n+1$-dimensional vector space.
The *monomials* $1, x, x^2, \ldots, x^n$ are a basis of P_n.

For given $n+1$ (distinct) nodes $x_0, x_1, \ldots, x_n$ the task of interpolation is

$$\text{Find } p \in P_n: \quad p(x_i) = y_i := f(x_i), \quad i = 0, \ldots, n$$

Polynomial Interpolation

This is equivalent to the linear system

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}}_{=:V[x_0,\ldots,x_n]} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

The matrix $V[x_0, \ldots, x_n]$

- is called *Vandermonde matrix*
- is regular iff all x_i are distinct
- leads to a very ill-conditioned map from values y_i to coefficients a_i
- requires computational effort in $\mathcal{O}(n^3)$ when solving the system

Polynomial Interpolation

Problem:
Assembling the Vandermonde matrix $V[x_0, \ldots, x_n]$ and then solving the linear system is not a good approach due to severe ill-conditioning and high associated cost.

Are there better and simpler approaches?

The problem is caused by the monomial basis $1, x, x^2, \ldots, x^n$, which leads to a particularly unfortunate formulation of the interpolation task. We are going to consider possible alternatives.

Langrange Interpolation

Definition 16 (Lagrange Polynomials)

For $n+1$ distinct nodes $x_i, i = 0, \ldots, n$, define the so-called *Lagrange polynomials*:

$$L_i^{(n)}(x) := \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}, \quad i = 0, \ldots, n \quad (n+1 \text{ polynomials})$$

The $L_i^{(n)}$ have degree n,

$$L_i^{(n)}(x_k) = \delta_{ik} = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

holds, and the $L_i^{(n)}$ are a basis of P_n.

Existence and Uniqueness

Using the Lagrange basis, the coefficients a_i are simply the prescribed values of the nodes: $a_i = y_i$. Solving the interpolation problem is therefore trivial in this basis.

Theorem 17 (Uniqueness of Interpolating Polynomial)

For $n+1$ distinct nodes $x_0, \ldots, x_n$ there is exactly one polynomial p of degree n with

$$p(x_i) = y_i \quad i = 0, \ldots, n,\ y_i \in \mathbb{R}$$

Therefore, the interpolation problem is solvable, and its solution is unique.

Newton Representation

Disadvantage of Lagrange polynomials:
Adding a node changes all previous basis polynomials, making this approach unsuitable for "incremental" construction of interpolation polynomials.

In this context, the *Newton representation* with basis polynomials

$$N_0(x) = 1; \quad i = 1, \ldots, n\colon N_i(x) = \prod_{j=0}^{i-1}(x - x_j)$$

is a better choice.

N_i always has degree i,

$$\forall k < i\colon N_i(x_k) = 0$$

holds, and the polynomials $N_0, \ldots, N_n$ are a basis of P_n.

Staggered Computations

In x_0 all the Newton basis polynomials but N_0 are zero, in x_1 all but N_0 and N_1, and so on.

The interpolation task

$$p(x_k) = \sum_{i=0}^{n} a_i N_i(x_k) = \sum_{i=0}^{k} a_i N_i(x_k) = y_k \quad k = 0, \ldots, n$$

leads to the following staggered computations:

$$a_0 = y_0; \quad k = 1, \ldots, n\colon\ a_k = \left[y_k - \sum_{i=0}^{k-1} a_i N_i(x_k) \right] / N_k(x_k)$$

Background

Polynomial interpolation in the language of linear algebra:

- The monomial basis $x^i, i = 0, \ldots, n$ is trivial to construct, but it leads to a dense and very ill-conditioned matrix $V[x_0, \ldots, x_n]$ (Vandermonde matrix).
- The Lagrange basis $L_i^{(n)}, i = 0, \ldots, n$, instead leads to an identity matrix, and therefore the solution is trivial. But an extension of the set of nodes changes all basis functions.
- The Newton basis $N_i, i = 0, \ldots, n$, in turn, results in a lower triangular matrix. The scheme of staggered computations corresponds to forward substitution. Solving requires more effort than with the Langrange basis, but additional nodes simply add additional rows to the matrix.

Divided Differences

Theorem 18 (Divided Differences)

The divided differences are recursively defined as

$$\forall i = 0, \ldots, n: \quad y[x_i] := y_i \quad \text{(values in nodes)}$$

$$\forall k = 1, \ldots, n - i:$$

$$y[x_i, \ldots, x_{i+k}] := \frac{y[x_{i+1}, \ldots, x_{i+k}] - y[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}$$

Then

$$p(x) = \sum_{i=0}^{n} y[x_0, \ldots, x_i] N_i(x)$$

holds.

Divided Differences

The divided differences are usually arranged in a tableau:

$$\begin{array}{lllll}
y_0 = y[x_0] & \to & y[x_0, x_1] & \to & y[x_0, x_1, x_2] \quad \to \quad y[x_0, x_1, x_2, x_3] \\
 & \nearrow & & \nearrow & \nearrow \\
y_1 = y[x_1] & \to & y[x_1, x_2] & \to & y[x_1, x_2, x_3] \\
 & \nearrow & & \nearrow & \\
y_2 = y[x_2] & \to & y[x_2, x_3] & & \\
 & \nearrow & & & \\
y_3 = y[x_3] & & & &
\end{array}$$

Then, the first row contains the desired coefficients $a_i, i = 0, \ldots, n$ of the basis polynomials. This form of computation doesn't need the values of the $N_i(x_k)$ and additionally is more stable.

Example I

Example 19

Consider the following pairs of nodes and values:

$$(x_0 = 0, y_0 = 1), \quad (x_1 = 1, y_1 = 4), \quad (x_2 = 2, y_2 = 3)$$

The monomial basis results in the system of equations

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$$

with solution $[1, 5, -2]^T$, therefore

$$p(x) = 1 + 5 \cdot x - 2 \cdot x^2$$

Example II

Example 19

The Lagrange basis leads to

$$\begin{aligned} p(x) &= 1 \cdot L_0^{(2)}(x) + 4 \cdot L_1^{(2)}(x) + 3 \cdot L_2^{(2)}(x) \\ &= 1 \cdot \frac{x-1}{0-1} \cdot \frac{x-2}{0-2} + 4 \cdot \frac{x-0}{1-0} \cdot \frac{x-2}{1-2} + 3 \cdot \frac{x-0}{2-0} \cdot \frac{x-1}{2-1} \\ &= \frac{1}{2} \cdot (x-1) \cdot (x-2) - 4 \cdot x \cdot (x-2) + \frac{3}{2} \cdot x \cdot (x-1) \end{aligned}$$

The individual basis polynomials can be precomputed for a given set of nodes $x_0, \ldots, x_n$, afterwards additional interpolation tasks on the same set of nodes are trivial to solve.

Example III

Example 19

For the Newton basis we obtain the tableau

$$\begin{array}{ccccc} y_0 = a_0 = 1 & \to & a_1 = \frac{4-1}{1-0} = 3 & \to & a_2 = \frac{(-1)-3}{2-0} = -2 \\ & \nearrow & & \nearrow & \\ y_1 = 4 & \to & \frac{3-4}{2-1} = -1 & & \\ & \nearrow & & & \\ y_2 = 3 & & & & \end{array}$$

and therefore

$$\begin{aligned} p(x) &= 1 \cdot N_0(x) + 3 \cdot N_1(x) - 2 \cdot N_2(x) \\ &= 1 + 3 \cdot x - 2 \cdot x \cdot (x-1) \end{aligned}$$

Here it is easy to add an additional node if required.

Evaluating Polynomials

The standard algorithm for the evaluation of a polymial of the form

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + \cdots + a_n x^n$$

at a point x is the *Horner scheme*

$$b_n := a_n; \quad k = n-1, \ldots, 0 \colon b_k := a_k + x \cdot b_{k+1}; \quad p(x) = b_0 \,,$$

since it is particularly efficient and stable.

For a polynomial in Newton representation

$$p(x) = \sum_{i=0}^{n} a_i N_i(x) = a_0 + a_1 N_1(x) + \cdots + a_n N_n(x)$$

this scheme leads to the recursion

$$b_n := a_n; \quad k = n-1, \ldots, 0 \colon b_k := a_k + (x - x_k) \cdot b_{k+1}; \quad p(x) = b_0$$

Interpolation Error

Let $y_i = f(x_i), i = 0, \ldots, n$, the evaluation of a function f in $n+1$ distinct nodes, and $p(x)$ the resulting interpolation polynomial of degree n.

By construction, the difference

$$e(x) := f(x) - p(x)$$

fulfills the condition

$$e(x_i) = 0 \quad \text{für} \quad i = 0, \ldots, n$$

Question: How large can this difference become at other locations?

Interpolation Error

Theorem 20 (Interpolation Error)

Let $f(x)$ $n+1$-times continuously differentiable on $[a,b]$ and

$$a \leq x_0 < x_1 < \cdots < x_n \leq b.$$

Then there is for each $x \in [a,b]$ an $\xi_x \in \overline{(x_0,\ldots,x_n,x)}$ (smallest interval containing all nodes), so that

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!} \prod_{j=0}^{n} (x - x_j)$$

Remarks

For the special case of *equidistant* nodes, i.e.,

$$x_{k+1} - x_k = h \quad \text{für} \quad k = 0, \ldots, n-1,$$

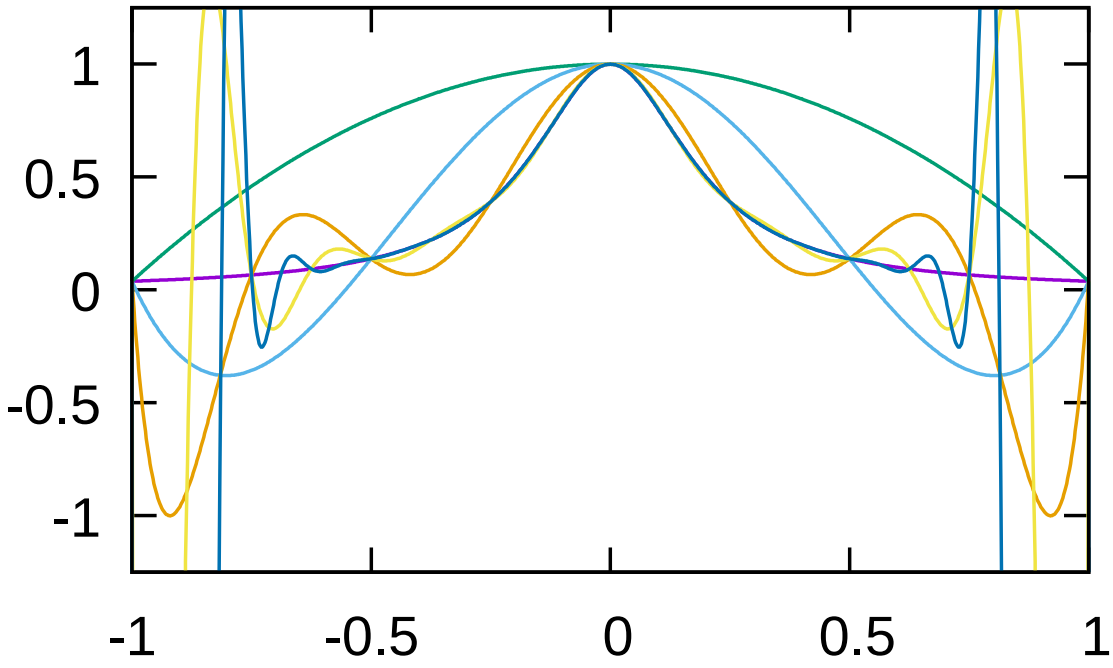
we therefore have

$$|f(x) - p(x)| \leq |f^{(n+1)}(\xi_x)| \cdot h^{n+1},$$

and for $|f^{(n+1)}|$ bounded and $n \to \infty$ it follows $|f(x) - p(x)| \to 0$.

Unfortunately, the higher derivatives of functions, even of simple ones, are often *not* bounded for $n \to \infty$, but grow very fast instead.

Runge's Counter Example



Polynomial interpolation of Runge's function $f(x) = (1 + 25x^2)^{-1}$ with equidistant nodes (3, 5, 9, 17 resp. 33 node/value pairs).
The minima / maxima of the last two polynomials are $-14.35/1.40$ resp. $-5059/2.05$ (!).

Remarks

Remark 21

According to the *Weierstraß Approximation Theorem*, any function in $C^0([a, b])$ can be approximated uniformly by polynomials.
The phenomena we observe are no contradiction, since:

- The approximation need not be based on interpolation (the proof uses Bernstein polynomials).
- Using non-equidistant nodes one can already achieve significantly improved results (if one knows how to choose these non-equidistant nodes...).

Remark 22

In general "methods of higher (polynomial) order" require sufficient differentiability.

Condition Analysis

With $p(x; y)$ the interpolation polynomial to the values $(y_0, \ldots, y_n)^T$ at *fixed* nodes $(x_0, \ldots, x_n)^T$, we have

$$\begin{aligned} p(x; y + \Delta y) - p(x; y) &= \sum_{i=0}^{n} (y_i + \Delta y_i) L_i^{(n)}(x) - \sum_{i=0}^{n} y_i L_i^{(n)}(x) \\ &= \sum_{i=0}^{n} \Delta y_i L_i^{(n)}(x) \end{aligned}$$

This implies

$$\frac{p(x; y + \Delta y) - p(x; y)}{p(x; y)} = \sum_{i=0}^{n} \frac{L_i^{(n)}(x) y_i}{p(x; y)} \cdot \frac{\Delta y_i}{y_i}$$

For large n, $L_i^{(n)}$ can become very large, then the interpolation task is ill-conditioned!

Contents I

Numerical Differentiation

Problem:
Compute the derivative (of some order n) of a function that is given as a table or implemented as a function (in the computer science sense of the word).

Idea:
Assemble interpolation polynomial for certain nodes, differentiate it and evaluate result to obtain (approximation of) derivative.

We assume order of derivative = degree of polynomial.

Numerical Differentiation

The Lagrange polynomials are

$$L_i^{(n)}(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} = \underbrace{\prod_{j \neq i} (x_i - x_j)^{-1}}_{=:\lambda_i \in \mathbb{R}} \cdot x^n + \alpha_{n-1} x^{n-1} + \cdots + \alpha_0,$$

therefore taking the n-th derivative produces

$$\frac{d^n}{dx^n} L_i^{(n)}(x) = n! \cdot \lambda_i$$

which gives us the n-th derivative of an interpolation polynomial of degree n:

$$\frac{d^n}{dx^n} \left(\sum_{i=0}^{n} y_i L_i^{(n)}(x) \right) = n! \cdot \sum_{i=0}^{n} y_i \lambda_i \quad \text{(independent of } x\text{)}$$

Numerical Differentiation

We have the following statement about the resulting error:

Theorem 23
Let $f \in C^n([a,b])$ and $a = x_0 < x_1 < \cdots < x_n = b$. Then there is $\xi \in (a,b)$, so that

$$f^{(n)}(\xi) = n! \cdot \sum_{i=0}^{n} y_i \lambda_i$$

Therefore, the derivative from the interpolation polynomial coincides in at least one point with the true derivative of f.

Numerical Differentiation

For equidistant nodes, $x_i = x_0 + ih, 0 \le i \le n$, there is an explicit formula based on the node values y_i:

$$f^{(n)}(x) \approx h^{-n} \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} y_i, \quad \text{e.g.,} \quad f^{(1)}(x) \approx \frac{y_1 - y_0}{h},$$

$$f^{(2)}(x) \approx \frac{y_2 - 2y_1 + y_0}{h^2}, \quad f^{(3)}(x) \approx \frac{y_3 - 3y_2 + 3y_1 - y_0}{h^3}$$

Based on Taylor expansion one can show

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \qquad \text{for} \quad f \in C^3,$$

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \mathcal{O}(h^2) \quad \text{for} \quad f \in C^4$$

Numerical Differentiation

These approximations of function derivatives are called *centered difference quotients*.

One can also place the nodes off-center to obtain the *forward difference quotient*

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \qquad \text{for} \quad f \in C^2$$

and *backward difference quotient*

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h) \qquad \text{for} \quad f \in C^2$$

Note that forward and backward difference quotients have lower approximation order than centered variants.

Difference quotients play an important role in the derivation of methods for differential equations.

Extrapolation to the Limit I

Let some quantity $a(h)$ be computable for $h > 0$, but not for $h = 0$. We are interested in computing

$$a(0) = \lim_{h \to 0} a(h)$$

with good accuracy.

Example 24

Possible applications:

1. L'Hospital's rule:

$$a(0) = \lim_{h \to 0} \frac{\cos(x) - 1}{\sin(x)} (= 0)$$

Extrapolation to the Limit II

Example 24

2. Numerical Differentiation:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

(small h cause cancellation)

3. Numerical Integration:

$$\int_a^b f(x) = \lim_{N \to \infty} \sum_{i=1}^{n} N^{-1} f\left(a + (i - \frac{1}{2}) \frac{b-a}{N}\right)$$

(set $h := N^{-1}$)

Extrapolation to the Limit III

Example 24

4. Numerical solution of initial value problem

$$y'(t) = f(t, y(t)) \quad \text{on} \quad [0, T]; \quad y(0) = y_0$$

Set

$$h = N^{-1}; \quad y_n = y_{n-1} + h \cdot f(t, y_n - 1); \quad y(T) \approx y_N$$

Here $h \to 0$ is equivalent to $N \to \infty$ and therefore increasing computational cost.

Central Idea of Extrapolation

Idea of extrapolation:

For $h_0 > h_1 > \cdots > h_n > 0$ construct interpolation polynomial

$$p(h_i) = a(h_i) \quad i = 0, \ldots, n$$

and compute

$$a(0) \approx p(0)$$

(Extrapolation instead of interpolation, since $0 \notin [h_n, \ldots, h_0]$)

Example I

Example 25

For $a(h) = (\cos(h) - 1) \cdot (\sin(h))^{-1}$ we have

$$\begin{aligned} h_0 = 1/8: &\quad a(h_0) = -6.258151 \cdot 10^{-2} \\ h_1 = 1/16: &\quad a(h_1) = -3.126018 \cdot 10^{-2} \\ h_2 = 1/32: &\quad a(h_2) = -1.562627 \cdot 10^{-2} \end{aligned}$$

(i.e., $a(h)$ is directly proptional to h), and with extrapolation using p_2 of degree 2:

$$a(0) \approx p_2(0) = -1.02 \cdot 10^{-5}$$

which is significantly better than the initial approximations or a possible direct evaluation for $h \ll 1$ (cancellation)!

Example II

Example 25

Why does this work so well?

Let $h_i = h \cdot r^i$ with $r < 1$ (geometric distribution), e.g., $r = 1/2$, and let p the interpolation polynomial of a to the nodes h_i. Then we have

$$|p(0) - a(0)| \leq \|V^{-T}\|_\infty |a^{(n+1)}(\xi)| \frac{h^{n+1}}{(n+1)!}(1 + r^{n+1})$$

for the extrapolation error, with Vandermonde matrix V and $\xi \in (0, h)$, as long as a is sufficiently differentiable.

Extrapolation in the Case of Derivatives

The Taylor expansion of a at zero (Mclaurin expansion) is crucial. For the usual difference quotient for the second derivative we obtain

$$\begin{aligned} a(h) &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \\ &= f''(x) + \frac{h^2}{2 \cdot 4!} f^{(4)}(x) + \cdots + \frac{h^{2n}}{2 \cdot (2n+2)!} f^{(2n+2)}(x) \\ &\qquad + \frac{h^{2n+2}}{2 \cdot (2n+4)!} [f^{(2n+4)}(\xi_+) + f^{(2n+4)}(\xi_-)] \\ &= p_x(h^2) + \mathcal{O}(h^{2(n+1)}) \end{aligned}$$

This means one gains two powers of h per evaluation (if f is sufficiently smooth)!

Newton-Cotes Formulas

The *Newton-Cotes formulas* are interpolatory quadrature (integration) formulas.

Idea: Construct the interpolation polynomial p of a function f for certain nodes and evaluate the integral of p exactly to approximate the integral of f.

Formally: nodes and values $(x_i, f(x_i)), i = 0, \ldots, n$, Lagrange representation:

$$p_n(x) = \sum_{i=0}^{n} f(x_i) L_i^{(n)}(x), \quad L_i^{(n)}(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

and therefore

$$I_{[a,b]}(f) \approx I_{[a,b]}^{(n)}(f) = \int_a^b p_n(x)\, dx = \sum_{i=0}^{n} f(x_i) \int_a^b L_i^{(n)}(x)\, dx$$

Order of Quadrature

Definition 26 (Order of Quadrature)

A quadrature formula $I^{(n)}(f)$ has at least order m, if it is able to integrate polynomials of degree $m-1$ exactly.

For example, a second order formula integrates linear functions exactly.

The Newton-Cotes formulas use polynomial interpolation and therefore they have at least order $n+1$ for $n+1$ nodes. But there are other formulas that can achieve even higher orders with the same number of nodes.

Closed and Open Formulas

The Newton-Cotes formulas use *equidistant* nodes. There are two variants:

Closed formulas:
The interval bounds a and b are nodes, i.e.,

$$x_i = a + iH, \quad i = 0, \ldots, n, \quad \text{with } H = \frac{b-a}{n}$$

Open formulas:
The bounds a and b aren't nodes, i.e.,

$$x_i = a + (i+1)H, \quad i = 0, \ldots, n, \quad \text{with } H = \frac{b-a}{n+2}$$

Examples

Closed formulas for $n = 1, 2, 3$ and $H = (b - a)/n$:

The trapezoidal rule:

$$I^{(1)}(f) = \frac{b-a}{2} \cdot [f(a) + f(b)]$$

The Simpson rule resp. Kepler's barrel rule:

$$I^{(2)}(f) = \frac{b-a}{6} \cdot [f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)]$$

The 3/8 rule resp. "pulcherrima":

$$I^{(3)}(f) = \frac{b-a}{8} \cdot [f(a) + 3f(a+H) + 3f(b-H) + f(b)]$$

Examples

Open formulas for $n = 0, 1, 2$ and $H = (b - a)/(n + 2)$:

The midpoint rule:

$$I^{(0)}(f) = (b - a) \cdot f\left(\frac{a + b}{2}\right)$$

The second open rule (no special name):

$$I^{(1)}(f) = \frac{b - a}{2} \cdot [f(a + H) + f(b - H)]$$

The third open rule (also no special name):

$$I^{(2)}(f) = \frac{b - a}{3} \cdot \left[2f(a) - f\left(\frac{a + b}{2}\right) + 2f(b)\right]$$

Remarks

Remark 27

From $n = 7$ on for closed rules, resp. from $n = 2$ on for open rules, negative weights appear in the sums. This is detrimental, because:

- Strictly non-negative functions f can have $I^{(n)}(f) < 0$ (solute concentration, mass conservation,...).
- There is increased risk of cancellation.
- Condition can become worse, while it is bounded for strictly positive weights.

Estimates for Remainder Terms I

Theorem 28 (Remainder Terms)

The resulting error can be estimated as follows:

1. Trapezoidal rule: $n = 1$, order 2, we have

$$I(f) - \frac{b-a}{2} \cdot [f(a) + f(b)] = -\frac{(b-a)^3}{12} f''(\xi), \quad \xi \in [a, b]$$

for $f \in C^2([a, b])$. This means polynomials up to degree 1 are integrated exactly, because for those $f''(x) = 0$ holds on $[a, b]$.

In general: the order of the odd formulas is the number of nodes, while the order of the even formulas is one higher.

Estimates for Remainder Terms II

Theorem 28 (Remainder Terms)

2. Simpson rule: n = 2, order 4, for $f \in C^4([a,b])$ we have

$$I(f) - \frac{b-a}{6} \cdot \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] = -\frac{(b-a)^5}{2880} f^{(4)}(\xi)$$

3. Midpoint rule: n = 0, order 2, for $f \in C^2([a,b])$ we have

$$I(f) - (b-a) \cdot f\left(\frac{a+b}{2}\right) = \frac{(b-a)^3}{24} f''(\xi)$$

so "half the error of trapezoidal rule" at just one function evaluation!

Summed Quadrature Rules

Increasing the polynomial degree doesn't make much sense, since

- negative weights appear early
- Lagrange interpolation with equidistant nodes doesn't converge pointwise
- f has to be sufficiently regular for the estimates to hold

Idea of *summed quadrature rules*:

- Subdivide interval $[a, b]$ into N smaller intervals

$$[x_i, x_{i+1}], \quad x_i = a + ih, \ i = 0, \ldots, N-1, \ h = \frac{b-a}{N}$$

- Apply one of the above formulas on each subinterval and sum the results.

Examples

For N subintervals of stepsize h we arrive at:

The summed trapezoidal rule:

$$I_h^{(1)}(f) = \sum_{i=0}^{N-1} \frac{h}{2}[f(x_i) + f(x_{i+1})] = h \cdot \left[\frac{f(a)}{2} + \sum_{i=1}^{N-1} f(x_i) + \frac{f(b)}{2} \right]$$

The summed Simpson rule:

$$I_h^{(2)}(f) = \frac{h}{3} \cdot \left[\frac{f(a)}{2} + \sum_{i=1}^{N-1} f(x_i) + 2 \sum_{i=1}^{N-1} f\left(\frac{x_i + x_{i+1}}{2} \right) + \frac{f(b)}{2} \right]$$

The summed midpoint rule:

$$I_h^{(0)}(f) = h \cdot \sum_{i=0}^{N-1} f\left(\frac{x_i + x_{i+1}}{2} \right)$$

Contents I

Vectors and Matrices I

A *vector* $v \in \mathbb{R}^n$ is a finite sequence of real numbers:

$$v = [v_1, v_2, \ldots, v_n]^T, \quad v_i \in \mathbb{R}$$

A *matrix* $A \in \mathbb{R}^{n \times m}$ is defined similarly, but uses two independent indices i and j, one for columns and one for rows:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}, \quad a_{ij} \in \mathbb{R}$$

Each matrix $A \in \mathbb{R}^{n \times m}$ corresponds to a linear mapping $\varphi_A \colon \mathbb{R}^m \to \mathbb{R}^n$ given by

$$\varphi_A(v) = A \cdot v, \quad v \in \mathbb{R}^m$$

Vectors and Matrices II

For a sufficiently regular scalar function $f(x), x = [x_1, \ldots, x_m]$, we have the following special vectors and matrices:

$$\nabla f := \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_m}\right]^T \qquad (\textit{gradient})$$

$$\mathrm{H}_f = \nabla^2 f := \left[\frac{\partial^2 f}{\partial x_i \partial x_j}\right]_{ij} \qquad (\textit{Hessian})$$

For vector-valued functions $f(x) = [f_1(x), \ldots, f_n(x)]^T$, the gradient generalizes to the *Jacobian*:

$$\mathrm{J}_f = \left[\frac{\partial f_i}{\partial x_j}\right]_{ij}$$

If $f = \varphi_A$ is a linear function, then $\mathrm{J}_f = A$, i.e., the constituent matrix A and the Jacobian coincide.

Natural Matrix Norms

Definition 29 (Associated Matrix Norm)

Let $\|\cdot\|$ an arbitrary vector norm on $\mathbb{R}^n$. Then

$$\|A\| := \sup_{x\in\mathbb{R}^n\setminus\{0\}} \frac{\|Ax\|}{\|x\|} = \sup_{x\in\mathbb{R}^n,\|x\|=1} \|Ax\|$$

is called the *matrix norm associated with* $\|\cdot\|$, or *natural matrix norm*. It is compatible with the matrix norm, i.e.,

$$\|Ax\| \leq \|A\| \cdot \|x\| \quad A \in \mathbb{R}^{n\times n}, x \in \mathbb{R}^n,$$

and submultiplicative, i.e.,

$$\|AB\| \leq \|A\| \cdot \|B\| \quad A, B \in \mathbb{R}^{n\times n}$$

(compare with triangle inequality / subadditivity).

Linear and Nonlinear Systems of Equations

Many important problems, e.g., the solution of ordinary and partial differential equations, can be framed as solving a given linear system of equations:

$$A \cdot x = b \quad \text{resp.} \quad A \cdot x - b = 0$$

or nonlinear system of equations:

$$F(x) = 0$$

where F is a possibly vector-valued function, A is a matrix, and x and b are vectors of the right dimensions.

Perturbation Theorem

Theorem 30 (Perturbation Theorem)

Let $A \in \mathbb{R}^{n\times n}$ regular and $\|\Delta A\| \leq \|A^{-1}\|^{-1}$. Then $\widetilde{A} = A + \Delta A$ is also regular, and for the relative error of the perturbed system

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

the equation

$$\frac{\|\Delta x\|}{\|x\|} \leq \frac{\operatorname{cond}(A)}{1 - \operatorname{cond}(A) \cdot \frac{\|\Delta A\|}{\|A\|}} \cdot \left\{ \frac{\|\Delta b\|}{\|b\|} + \frac{\|\Delta A\|}{\|A\|} \right\}$$

holds, where $\operatorname{cond}(A) := \|A\| \cdot \|A^{-1}\|$ is the *condition number* of the matrix A. For the special case $\Delta A = 0$ we have

$$\frac{\|\Delta x\|}{\|x\|} \leq \operatorname{cond}(A) \cdot \frac{\|\Delta b\|}{\|b\|}.$$

Triangular Systems

Let $A \in \mathbb{K}^{n \times n}$ an upper triangular matrix:

$$\begin{aligned} a_{11} \cdot x_1 + a_{12} \cdot x_2 + \cdots + a_{1n} \cdot x_n &= b_1 \\ a_{22} \cdot x_2 + \cdots + a_{2n} \cdot x_n &= b_2 \\ \ddots \quad \vdots \quad & \vdots \\ a_{nn} \cdot x_n &= b_n \end{aligned}$$

This system permits a unique solution iff $a_{ii} \neq 0, i = 1, \ldots, n$.

Because of the simple structure this can be solved using "backward substitution".

Triangular Systems

Solution using backward substitution:

$$\begin{aligned} x_n &= b_n / a_{nn} \\ x_{n-1} &= \left(b_{n-1} - a_{(n-1)n} \cdot x_n\right) / a_{(n-1)(n-1)} \\ &\vdots \\ x_i &= \left(b_i - \sum_{k=i+1}^{n} a_{ik} \cdot x_k\right) / a_{ii} \end{aligned}$$

Required number of operations:

$$N_\Delta(n) = \sum_{i=0}^{n-1} (2i+1) = n^2$$

(Of course there is an analogous "forward substitution" for upper triangular matrices.)

Direct Methods for Linear Systems

Let $A \in \mathbb{K}^{n \times n}$ regular, but with arbitrary structure.

Goal: Transform A into (upper) triangular form, then use backward substitution.

This can be done using:

- exchange of two equations / rows
- addition of a multiple of one equation to another

This is a standard technique known as *Gauss elimination*.

Gauss Elimination

Perform the following steps until an upper triangular matrix is obtained, starting with $k = 1$:

1. For $i > k$, define $l_{ik} = a_{ik} \cdot a_{kk}^{-1}$.
2. For $i > k$, set
$$a_{ij} \longleftarrow a_{ij} - l_{ik} a_{kj}$$
(subtract a multiple of the k-th row to eliminate the first k entries of the i-th row).
3. Increase k by one: $k \longleftarrow k + 1$.
4. Repeat.

After (at most) $n - 1$ loop iterations the matrix has become upper right triangular.

Cost of Gauss Elimination

Lemma 31

The cost of transforming A into an upper right triangular matrix by Gauss elimination is

$$N_{\text{Gauß}}(n) = \frac{2}{3}n^3 + \mathcal{O}(n^2)$$

Since the cost for backward substitution is negligible ($N_\Delta(n) = n^2$), this is also the cost for solving a linear equation system using Gauss elimination.

A Note on Stability

The classic Gauss elimination is *unstable* for general matrices (because we divide by diagonal elements of the original matrix, which can become arbitrarily small).

The algorithm can be made significantly more stable through a process called *row pivotisation*. In each iteration, we search for the largest subdiagonal element in the k-th column and swap its row with the k-th row, remembering resulting row permutations.

Total pivotisation instead searches for the largest element in the lower right corner of the matrix, and employs both row and column permutations. This is more expensive, but leads to further improvements in terms of stability.

LU Decomposition I

Storing the factors l_{ik} of the elimination steps is a good idea:

Theorem 32 (LU Decomposition)

Let $A \in \mathbb{R}^{n\times n}$ regular, then there exists a decomposition $PA = LU$, where

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{11} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ l_{n1} & \cdots & l_{n(n-1)} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \ddots & u_{2n} \\ \vdots & \ddots & \ddots & u_{(n-1)n} \\ 0 & \cdots & 0 & u_{nn} \end{bmatrix},$$

and P is a permutation matrix. For $P = I$ this decomposition is unique.

LU Decomposition II

Solving a linear system via LU decomposition:

1. For given A, compute LU decomposition and matrix P
2. For given b, calculate $b' = Pb$
3. Solve triangular system $Ly = b'$ (forward substitution)
4. Solve triangular system $Rx = y$ (backward substitution)

LU decomposition is equivalent to Gauss elimination, and therefore has the same cost $N_{LU}(n) = \frac{2}{3}n^3 + \mathcal{O}(n)$.

Important difference:
The LU decomposition can be reused for new righthand sides $A\widetilde{x} = \widetilde{b}$, while Gauss elimination has to start over from the beginning!

Symmetric Positive Definite Matrices

Theorem 33

For a symmetric positive definite (s.p.d.) matrix $A \in \mathbb{R}^{n\times n}$ the LU decomposition is always stable, even *without* pivotisation. The equation

$$a_{ii}^{(k)} \geq \lambda_{\min}(A), \quad k \leq i \leq n$$

holds for the diagonal elements, where $\lambda_{\min}(A)$ is the smallest eigenvalue of A.

The symmetric structure of the matrix can be used to reduce the cost of LU decomposition.

Cholesky Decomposition

With $D = \mathrm{diag}(R)$ we have

$$A = LD(D^{-1}R) = LDR \quad \text{with} \quad R := D^{-1}U,$$

and because of symmetry $R = L^T$, therefore $A = LDL^T$. Since all diagonal elements of D are positive, the matrix $D^{1/2}$ with

$$(D^{1/2})_{ii} = d_{ii}^{1/2}, \quad (D^{1/2})_{ij} = 0 \text{ for } i \neq j$$

is well-defined, and

$$A = LD^{1/2} \cdot D^{1/2}L^T = \widetilde{L}\widetilde{L}^T \quad \text{with} \quad \widetilde{L} := LD^{1/2}$$

holds.

This special form of the decomposition is called *Cholesky decomposition*, it has half the cost of the general version.

Iterative Methods for Linear Systems I

We consider a second approach for the solution of

$$Ax = b, \quad A \in \mathbb{R}^{n\times n} \text{ regular}, \quad b \in \mathbb{R}^n.$$

Definition 34 (Sparse Matrices)

A sequence of matrices $\{A^{(n)} \mid n \in \mathbb{N}\}$ is called *sparse*, iff

$$|\{a_{ij}^{(n)} \mid a_{ij}^{(n)} \neq 0\}| =: \mathrm{nnz}(A^{(n)}) = \mathcal{O}(n)$$

(nnz = "number of non-zeros").

Because of "fill in" Gauss elimination is often unsuited for sparse matrices, and: for large systems the cost in $\mathcal{O}(n^3)$ makes the solution intractable.

Iterative Methods for Linear Systems II

We have

$$\text{Solving } Ax = b \iff \text{“root search” for } f(x) := b - Ax = 0.$$

With given matrix C, define the iteration

$$\begin{aligned} x^{(t+1)} &= g(x^{(t)}) = x^{(t)} + C^{-1} f(x^{(t)}) \\ &= x^{(t)} + C^{-1}(b - Ax^{(t)}) \\ &= \underbrace{(I - C^{-1}A)}_{=:B} x^{(t)} + C^{-1} b \end{aligned}$$

with “iteration matrix” B. The choice $C = A$ would be optimal in theory, but that requires solving the problem itself.

$\Longrightarrow$ look for easily invertible C “similar” to A

Iterative Methods for Linear Systems III

For the solution $x := A^{-1}b$,

$$g(x) := (I - C^{-1}A)x + C^{-1}b = x - (C^{-1}A) \cdot (A^{-1}b) + C^{-1}b = x$$

holds, therefore x is a fixpoint of g.

The Lipschitz constant of the function g fulfills

$$\|g(x) - g(y)\| = \|B(x - y)\| \leq \|B\| \cdot \|x - y\|,$$

i.e., if $\|B\| < 1$ (for suitable matrix norm $\|\cdot\|$), then g is a contraction on $\mathbb{R}^n$, and repeated application of g defines a sequence that converges to the solution (consequence of Banach's fixpoint theorem).

Examples for Iterative Methods

Decompose $A = L + D + U$ with L strict lower triangular matrix, D diagonal matrix and U strict upper triangular matrix.

Jacobi method:
Set $C = D$, i.e.,

$$x^{(t+1)} = x^{(t)} + D^{-1}(b - Ax^{(t)})$$

Gauß–Seidel method:
Set $C = L + D$, i.e., (forward substitution)

$$x^{(t+1)} = x^{(t)} + (L + D)^{-1}(b - Ax^{(t)})$$

Such iterative methods typically converge only for special classes of matrices (since we need $\|B\| < 1$).

Convergence of Jacobi Method

A matrix is called *strictly diagonally dominant*, iff

$$\sum_{j \neq i} |a_{ij}| < |a_{ii}| \quad \forall\, i = 1, \ldots, n.$$

Theorem 35
The Jacobi method converges for strictly diagonally dominant matrices.

There are many similar statements for symmetric positive definite matrices, weakly diagonally dominant matrices, so-called M-matrices, ...

Costs

Let $\alpha(n)$ be the cost for one iteration, typically with $\alpha(n) = \mathcal{O}(n)$. Since

$$\|x^{(t)} - x\| \leq \|B\|^t \|x^{(0)} - x\|,$$

a total of $t \geq \frac{\log(\epsilon)}{\log(\|B\|)}$ iterations are necessary for a reduction of the error by a factor $\epsilon \ll 1$, leading to a total cost of

$$T_{\text{fix}}(n) = \frac{\log(\epsilon)}{\log(\|B\|)} \alpha(n).$$

Problem: high costs if $\|B\|$ is close to one, $\|B\|$ is problem dependent and often grows with n.

Conjugate Gradients

For symmetric positive definite matrices A one can instead use the method of Conjugate Gradients. It uses an initial guess x_0, the initial residuum $r_0 := b - Ax_0$, and $d_0 := r_0$ to iteratively compute

$$\alpha_t = \frac{r_t^T r_t}{d_t^T A d_t}$$
$$x_{t+1} = x_t + \alpha_t d_t$$
$$r_{t+1} = r_t - \alpha_t A d_t$$
$$\beta_t = \frac{r_{t+1}^T r_{t+1}}{r_t^T r_t}$$
$$d_{t+1} = r_{t+1} + \beta_t d_t$$

- The CG method converges in at most n steps in exact arithmetic.
- For $n \gg 1$ it can be used as an iterative method, and often displays good convergence properties after the first few steps.

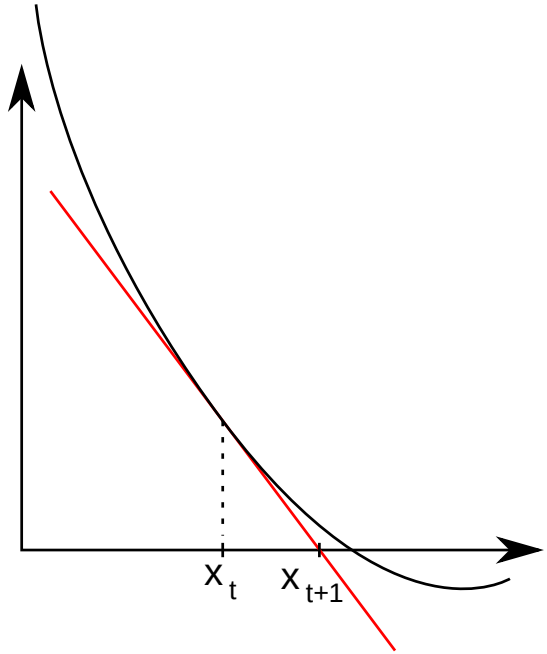
Newton's Method

Let f a differentiable function in one variable. For given x_t we have the "tangent"

$$T_t(x) = f'(x_t)(x - x_t) + f(x_t)$$

with root

$$T_t(x) = 0 \iff x = x_t - \frac{f(x_t)}{f'(x_t)}.$$



Using this root as an estimate for the root of f leads to the iteration

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}.$$

Obviously we need $|f'(x_t)| > 0$, i.e., we assume that the root of f is a *simple root*.

Newton's Method in Multiple Dimensions

Newton's method can be extended to systems $f : \mathbb{R}^n \to \mathbb{R}^n$:

Assume that the Taylor expansion of f exists:

$$f_i(x) = f_i(x_t+\Delta x) = f_i(x_t)+\sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j}(x_t)\Delta x_i+R_i(x_t,\Delta x) \quad i = 1,\ldots,n$$

or in vector notation

$$f(x_t + \Delta x) = f(x_t) + J_f(x_t)\Delta x + R(x_t, \Delta x)$$

with "Jacobian" matrix

$$[J_f(x_t)]_{ij} = \frac{\partial f_i}{\partial x_j}(x_t)$$

Newton's Method in Multiple Dimensions

Ignoring the remainder term is equivalent to "linearization of f".
Find approximate root of f:

$$f(x) \approx f(x_t) + J_f(x_t)\Delta x = 0 \iff \Delta x = -J_f^{-1}(x_t)f(x_t)$$

This leads to the iteration

$$x_{t+1} = x_t - J_f^{-1}(x_t)f(x_t)$$

Every single step requires the solution of a linear system based on the local Jacobian!

Convergence of Newton's Method I

Theorem 36 (Newton's Method)

Let $f \in C^2([a,b])$ have a root z in (a,b) (interior!), and let

$$m := \min_{a \le x \le b} |f'(x)| > 0, \quad M := \max_{a \le x \le b} |f''(x)|.$$

Let $\rho > 0$ chosen thus, that

$$q := \frac{M}{2m}\rho < 1, \quad K_\rho(z) := \{x \in \mathbb{R} \mid |x - z| \le \rho\} \subset [a,b]$$

Then for every initial value $x_0 \in K_\rho(z)$ the Newton iterations $x_t \in K_\rho(z)$ are defined and converge to the root z.

Convergence of Newton's Method II

Theorem 36 (Newton's Method)

Additionally, the a priori error erstimate

$$|x_t - z| \leq \frac{2m}{M} q^{(2^t)}, \quad t \in \mathbb{N}$$

and the a posteriori error estimate

$$|x_t - z| \leq m^{-1}|f(x_t)| \leq \frac{M}{2m}|x_t - x_{t-1}|^2, \quad t \in \mathbb{N}.$$

hold (a priori: only uses prerequisites, a posteriori: also uses iterations that were computed up to that point)

Example: Roots of Real Numbers

Example 37 (Computing Roots with Newton's Method)

Let $a > 0$ and $n \geq 1$. Solve $x^n = a$, i.e.,

$$f(x) = x^n - a = 0, \quad f'(x) = n \cdot x^{n-1}.$$

This leads to iteration

$$x_{t+1} = n^{-1} \cdot [(n-1) \cdot x_t + a \cdot x_t^{1-n}].$$

According to Thm. 36 this converges, if x_0 is close enough to $a^{1/n}$. However, in this special case it converges *globally*, i.e., for all $x_0 > 0$ (but not necessarily quadratically in the beginning).

Remarks I

Remark 38

- Newton's method converges only *locally*, i.e., when $|x_0 - z| \leq \rho$ ("basin of attraction"). Here ρ is normally not known and potentially very small.
- Newton's method exhibits quadratic convegence,

$$|x_t - z| \leq c \cdot |x_{t-1} - z|^2,$$

in contrast to alternatives like, e.g., bisection, which converges only linearly.

Remarks II

Remark 38

- Damped Newton's method:
  Convergence *outside* of the basin of attraction can be improved by setting

$$x_{t+1} = x_t - \lambda_t \frac{f(x_t)}{f'(x_t)}$$

  with the choice of some sequence $\lambda_t \in (0, 1]$ as "dampening strategy".

Remarks III

Remark 38

- Multiple roots:
  If z is a p-fold root, with $p > 1$, Newton's method will still converge, but only linearly. One can show that the modified iteration

$$x_{t+1} = x_t - p \cdot \frac{f(x_t)}{f'(x_t)}$$

  reestablishes quadratic convergence if p is known a priori.

Summary

In the last few days, we have discussed the fundamentals of the following topics:

1. Numerical programming in C++
2. Numbers and calculations with finite precision
3. Condition analysis of numerical problems and tasks
4. Error propagation and stability of numerical algorithms
5. Numerical differentiation and integration
6. Numerical solution of linear and nonlinear equation systems

The introduced concepts will form the basis of the lectures and exercises next week.