JOHANNES KEPLER
UNIVERSITY LINZ

Astrid Pechstein

Institut für Technische
Mechanik

WS 2024/25

# SCIENTIFIC COMPUTING

**Lecture notes**

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

# Contents


**1 Object-oriented scientific computing** **1**

1.1 C/C++ preliminaries . . . 1

1.1.1 Why C++? . . . 1

1.1.2 Tools for compiling C/C++ code . . . 2

1.1.3 C/C++ basics . . . 2

1.2 Basic linear algebra – an introduction to C++ . . . 11

1.2.1 A basic vector class in C++ . . . 11

1.2.2 Inheritance and abstract base classes . . . 26

**2 Numerical linear algebra** **39**

2.1 Accuracy and stability . . . 43

2.1.1 Floating point representation . . . 44

2.1.2 Data stability . . . 46

2.1.3 Numeric stability of an algorithm . . . 48

2.2 Solving linear systems of equations . . . 49

2.2.1 Gaussian elimination . . . 50

2.2.2 LU decomposition . . . 53

2.2.3 Cholesky factorization for symmetric positive definite matrices . . . 55

2.2.4 QR decomposition – Householder method . . . 56

2.2.5 QR for overdetermined systems . . . 57

2.2.6 Singular value decomposition (SVD) . . . 60

2.2.7 LAPACK . . . 62

2.3 Sparse linear systems . . . 65

2.3.1 Sparse matrix storage . . . 67

2.3.2 Generating the matrix graph . . . 68

2.3.3 Data access for sparse matrices . . . 70

2.3.4 Solving sparse systems . . . 71

2.4 Iterative solution methods for linear systems . . . 75

2.4.1 Residual-based iterative methods . . . 76

2.4.2 CG for symmetric/hermitian positive definite problems . . . 80

2.4.3 Krylov spaces and Arnoldi iteration . . . 82

2.4.4 GMRES . . . 83

**3 Applied computational methods** **85**
3.1 Eigenvalues . . . . . 85
3.1.1 The QR algorithm for eigenvalue problems . . . . . 89
3.1.2 Inverse iteration for real-symmetric matrices . . . . . 93
3.1.3 Example problem . . . . . 97
3.2 Nonlinear problems . . . . . 98
3.2.1 Fixed point iteration . . . . . 98
3.2.2 Newton's method . . . . . 99
3.2.3 Nonlinear regression – the Gauss-Newton method . . . . . 101

# 1 Object-oriented scientific computing

## 1.1 C/C++ preliminaries

### 1.1.1 Why C++?

C++ is an extension of the C language that was developed by Bjarne Stroustrup. It was updated in 2011, 2014, 2017, 2020 and 2023 to C++11, C++14, C++17, C++20, C++23. We list some of its features:

- C++ is an object-oriented language, which makes it favourable for large software projects (high-level programming). Object-oriented code is generally easier to maintain and extend than purely procedural code.
- In using C++, you get a high level of control over system resources and memory. Pointer access allows one to directly manipulate storage, which makes the language suitable for low-level programming.
- C++ is a compiled language without garbage collection or dynamic typing, which makes it very fast in execution. Thus, C++ can be used to create high-performance applications.
- C++ is a cross-platform language, the executable has to be compiled for a specific platform but is then machine independent.
- There exists a huge number of libraries that can be compiled and/or linked.

Most of these features can also be viewed as drawbacks, especially as compared to interpreted languages such as python,

- C++ is a compiled language ...: a C++ program needs to be compiled before running it, which means more steps are necessary for small projects.
- ... without garbage collection ...: the programmer controls memory allocation and deallocation, which makes the code more unsafe and error-prone.
- ... or dynamic typing: C++ syntax is very strict and harder to learn than e.g. python. There is a lot of overhead in C++ code to satisfy syntax requirements.
- There exists a huge number of libraries that can be compiled and/or linked: linking more than one independent library can be challenging, also there is no "package manager" choosing correct versions as e.g. in python. Linking exernal libraries often leads to platform-dependence of the project.

Nevertheless, C++ is the most widely used language in 21st century scientific computing and computer simulation. Additionally, inter-linking with FORTRAN libraries is possible. Prominent C/C++ projects include Adobe Photoshop, Spotify, YouTube, Amazon, Windows, Microsoft Office.

The aim of this course is to discuss basic algorithms of scientific computing concerning their mathematical background, but also their efficient implementation. Theoretical strategies to solve mathematical problems commonly coming up in engineering applications are presented. Additionally, the practical implementation of these methods is discussed in detail, with emphasis on computational efficiency. As C++ is not only efficient but provides ample access to internal memory to the user, it is an optimal choice to achieve the above goals.

It is expected that students bring basic knowledge on the C language. Within the following sections, some of these basic C concepts are discussed to the extent that is crucial for the understanding of the more advanced implementations later.

### 1.1.2 Tools for compiling C/C++ code

C/C++ are *compiled languages*, meaning that in the usual context code is translated by the compiler into machine language. Through the compilation process, either *executables*, *statically linked libraries* or *dynamically linked libraries* are generated. The present course will be restricted to generating executables by using *header-only libraries*, where C/C++ code is included directly.

As all example code should run on Windows, Linux and MacOS, we will use cross-platform makefiles from cmake `cmake.org`. To simplify the interaction with makefiles/compiler, get visual language support (syntax highlighting, intellisense) and allow for easy debugging, it is proposed to use Visual Studio Code `code.visualstudio.com` as a graphical user interface. For details on what to install on different operating systems, see the Moodle page of this course.

### 1.1.3 C/C++ basics

**Variables** In C++, each variable has to be declared before usage. Common are variables of simple type, which are declared e.g. via

```
int a;
double d;
char e;
size_t len;
```

Then, when the code `double d;` is executed, some piece of RAM is appointed to `d` (exactly 64 bit, see Section 2.1.1). This process is called *allocation*. This memory allocation is deleted automatically at the end of the scope of `d` (i.e. the present block). It is not possible to change the type of a variable within its scope, i.e. one cannot declare `int d;` within the scope of `d`. However, this can be done outside the respective block.

Basic types in C/C++ include – see also `https://en.cppreference.com/w/cpp/language/types`

- `bool` integer type, capable of holding one of the two values: `true` or `false`. The internal width of `bool` is implementation dependent.

- `char` type for character representation which can be most efficiently processed on the target system (can be specified `signed` or `unsigned`; the range is -128 to 127 or 0 to 255; if not specified the signedness depends on platform and compiler). Internally, `char` is represented by (at least) 1 byte.
- `int` basic integer type. Can be specified `signed` or `unsigned`; length modifiers `long` or `short` can be used to determine its internal width. If no length modifiers are present, it's guaranteed to have a width of at least 16 bits. However, on 32/64 bit systems it is almost exclusively guaranteed to have width of at least 32 bits. The range of an unsigned 32 bit integer is $-2\,147\,483\,648$ to $2\,147\,483\,647$
- `size_t` unsigned integer type; its range is specified from 0 to the maximum size of a theoretically possible object of any type.
- `double` floating point type, width 64 bit, floating point accuracy $\simeq 10^{-16}$, for details see Section 2.1.1. The 32 bit type `float` is typically not used in scientific computing any more.

**Functions** As inherited from C, functions can be declared in C++. The declaration specifies function name, return type and parameter list. An exemplary function for summing two double inputs and returning the sum is *declared* via

```
double Sum(double a, double b);
```

The function needs to be *defined*, a code block is provided that is executed on call. The definition can be done in-line with the declaration, or (in case of more involved definitions) in some extra source file that is compiled and linked. For the `Sum` example, a definition is

```
double Sum(double a, double b)
{
    return a + b;
}
```

If the declaration of a function is missing when used, we get a *compiler error*. If the definition of a function that has been declared is missing, we get a *linker error*.

**Namespaces** In C/C++, all identifiers within the same scope are unique. This can lead to problems if including different header files – no global functions or objects may have the same name.

Namespaces offer the possibility to group objects within different scopes. Thereby, collisions can be avoided when including multiple libraries. We have already used the standard namespace `std::`, e.g. in the Hello World file

```
std::cout << "Hello World!" << std::endl;
```

Console-output stream `cout` and end-of-line `endl` are defined within the `std` namespace, and have to be identified as such. If one wants to avoid typing `std::` (for better readability etc), all identifiers in the namespace can be made visible by `using namespace std;`. E.g. the Hello-World code above can be replaced by

```
using namespace std;
...
cout << "Hello World!" << endl;
```

Individual namespaces can be introduced, for example in this lecture we use `SC::`,

```
namespace SC
{
    // put your definitions here..
}
```

In the actual code, quantities are then addressed either using `SC::` or `using namespace SC;`. If an identifier is not unique due to using statements, a compiler error is observed. Warning – *never* put `using namespace` within global scope in a header file. On including this header, all namespace identifiers are visible in *all* headers included later and the actual source code. This may lead to unexpected conflicts that can hardly be removed.

**Referencing and de-referencing** After a variable is declared,

```
double d = 5.;
```

some piece of RAM is appointed to `d`. Using the *address-of*, or *referencing operator* `&`, we can get this address:

```
cout << "value of d: " << d << endl;   // print value of d
cout << "address of d: " << &d << endl;  // print address of d
```

Output e.g.

```
value of d:   5
address of d: 000000B8C16FFC18
```

Given some memory address, the *dereferencing operator* `*` returns the value at this memory address:

```
cout << "value at address of d: " << *(&d) << endl;  // print value of double sto
```

Output e.g.

```
value at address of d: 5
```

**Pointers** A pointer directly stores a memory address.

```
double d = 5.;
double* ptr = &d;
cout << "value of pointer " << ptr << endl;
```

We can get the value of the object at the pointer address through the de-referencing operator

```
...
cout << "de-referencing the pointer gives the value " << *ptr << endl;
```

Output to both, e.g.

```
value of pointer 00000080D05AF738
de-referencing the pointer gives the value 5
```

If one declares `double * ptr;` without initialization, it contains a garbage address. De-referencing will lead to memory corruption. The null-pointer is used to indicate that some pointer does not (yet) contain a valid address. Since C++11, `nullptr` is available as a standard keyword for the nullpointer, otherwise `0` or `NULL` can be used.

```
double* ptr = nullptr;
```

**References** A reference to an existing object can be defined, thereby another name for the same object is generated.

```
double d = 5.;
double& refd = d;
cout << "address of d    " << &d << endl;
cout << "address of refd " << &refd << endl;
```

The same address is given for both objects. If we increase `d`, `refd` is affected in the same manner.

Whenever a reference is initialized, it must directly be set to refer to an exising valid object. A compiler error is generated e.g. on

```
// compiler error - ref not initialized
double& refb;
```

**Call by value/call by reference** The concept of *call by value* vs. *call by reference* is essential in C++ function calls. A simple example (more to-the-point examples are to follow in the following sections): Design a function that increases the value of the argument by one.

```
// call by value, does not work
void Increase(double d)
{
    // print memory address of d inside function
    cout << "internal address " << &d << endl;
    d++;
    cout << "internal value of d " << d << endl;
}

double d = 4.;
// print memory address of "outside" d
cout << "external address " << &d << endl;
Increase(d);
// value of d
cout << "after function call: value of d " << d << endl;
```

Output e.g.:

```
external address 000000039A4FF758
internal address 000000039A4FF710
internal value of d 5
after function call: value of d 4
```

Above, at function call, $d$ is *copied* and passed to the `Increase` function. However, we want to increase the outside `d`. Remedy: call by reference, pass the memory address of `d`

```
// call by reference
void Increase(double &d)
{
    // print memory address of d inside function
    cout << "internal address " << &d << endl;
    d++;
    cout << "internal value of d " << d << endl;
}

double d = 4.;
// print memory address of "outside" d
cout << "external address " << &d << endl;
Increase(d);
// value of d
cout << "after function call: value of d " << d << endl;
```

Output e.g.

```
external address 00000095C1EFFD08
internal address 00000095C1EFFD08
internal value of d 5
after function call: value of d 5
```

The memory address is copied, and thus virtually the same piece of memory is treated as `d` inside and outside the `Increase` function.

The `Increase` example above can also be realized using call by reference with a pointer argument, although slightly less convenient

```
// call by reference, pointer argument
void Increase(double *p)
{
    // print memory address of pointer inside function
    cout << "internal address " << p << endl;
    // increase value
    (*p)++;
    cout << "internal value " << (*p) << endl;
}

double d = 4.;
// print memory address of "outside" d
```

```
cout << "external address " << &d << endl;
// pass address of d to Increase function
Increase(&d);
// value of d
cout << "after function call: value of d " << d << endl;
```

**Memory allocation** We differ between *static* and *dynamic memory allocation*.

**Static memory allocation** Memory is allocated statically whenever the size of the object (the amount of memory needed) is known at compile time. It is also known as *compile time allocation*. Examples are objects or arrays of fixed size,

```
double d;
int a[100];
```

The memory is allocated on the stack, allocation is fast, and the memory is freed at the end of the scope of the object (end of function body etc.).

**Dynamic memory allocation** Dynamic memory allocation happens at run time, and is also known as *run time allocation*. Here, the size of the object needs not be known at compile time. The memory is allocated on the heap, allocation is (relatively) slow.

The programmer is responsible for allocation and deallocation of the memory. Allocation happens through the `new` keyword, while the memory is freed explicitly at `delete`. If the memory is not freed explicitely, it is kept through the run time – a *memory leak* occurs. If the pointer to the allocated memory is lost or overwritten, the memory becomes unreachable.

The C equivalent to `new/delete` is `malloc/free`.

```
int n;
cin >> n;
// dynamically allocated double
double *c = new double;
// dynamically allocated double array
double *d = new double[n];

// work with objects...
*c = 0;
d[0] = 3;

// free memory of new double c
delete c;
// free memory of new double[n] d
delete [] d;
```

**Classes** As an introductory example, a complex number class `Complex` is implemented in `complex.cpp`.

In C++, simple classes are defined using the `class` keyword, the definition of member objects and member functions is done within braces `{}`, a semicolon `;` concludes the definition. Member functions can be defined also outside the braces scope, in that case the class scope operator is added to the definition. For the `Complex` class, we list a very minimal description,

```
class Complex
{
    public:

    // two members: real and imaginary part
    double real;
    double imag;

    // constructor for given real and imaginary part
    Complex(double r, double i)
    {
        real = r; imag = i;
    }

    // copy constructor
    Complex(const Complex& c)
    {
        real = c.real;
        imag = c.imag;
    }

    // destructor
    ~Complex() {}

    // assignment operator
    Complex& operator=(const Complex& c)
    {
        real = c.real;
        imag = c.imag;
        return *this;
    }

    // member functions add functionality to class
    double abs();
    double arg();
};
```

If the definition of `double abs();` is added inline, *replace* the declaration above by

```
double abs() const { return sqrt(real*real + imag*imag); }
```

If the definition is added in some separate place, *add outside the class definition block*

```
double Complex::abs() const { return sqrt(real*real + imag*imag); }
```

The former variant is preferable for the sake of optimization – the function is defined *inline* and can be replaced/optimized at compile time.

**Operators** Operators such as `+`, `-` can be overloaded for classes. For a list of operators, and there precedence, see the following table from `https://en.cppreference.com/w/cpp/language/operator_precedence`

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | `::` | Scope resolution | Left-to-right → |
| 2 | `a++` `a--` | Suffix/postfix increment and decrement | |
| | *type*`()` *type*`{}` | Functional cast | |
| | `a()` | Function call | |
| | `a[]` | Subscript | |
| | `.` `->` | Member access | |
| 3 | `++a` `--a` | Prefix increment and decrement | Right-to-left ← |
| | `+a` `-a` | Unary plus and minus | |
| | `!` `~` | Logical NOT and bitwise NOT | |
| | (*type*) | C-style cast | |
| | `*a` | Indirection (dereference) | |
| | `&a` | Address-of | |
| | `sizeof` | Size-of[note 1] | |
| | `co_await` | await-expression (C++20) | |
| | `new` `new[]` | Dynamic memory allocation | |
| | `delete` `delete[]` | Dynamic memory deallocation | |
| 4 | `.*` `->*` | Pointer-to-member | Left-to-right → |
| 5 | `a*b` `a/b` `a%b` | Multiplication, division, and remainder | |
| 6 | `a+b` `a-b` | Addition and subtraction | |
| 7 | `<<` `>>` | Bitwise left shift and right shift | |
| 8 | `<=>` | Three-way comparison operator (since C++20) | |
| 9 | `<` `<=` `>` `>=` | For relational operators < and ≤ and > and ≥ respectively | |
| 10 | `==` `!=` | For equality operators = and ≠ respectively | |
| 11 | `a&b` | Bitwise AND | |
| 12 | `^` | Bitwise XOR (exclusive or) | |
| 13 | `\|` | Bitwise OR (inclusive or) | |
| 14 | `&&` | Logical AND | |
| 15 | `\|\|` | Logical OR | |
| 16 | `a?b:c` | Ternary conditional[note 2] | Right-to-left ← |
| | `throw` | throw operator | |
| | `co_yield` | yield-expression (C++20) | |
| | `=` | Direct assignment (provided by default for C++ classes) | |
| | `+=` `-=` | Compound assignment by sum and difference | |
| | `*=` `/=` `%=` | Compound assignment by product, quotient, and remainder | |
| | `<<=` `>>=` | Compound assignment by bitwise left shift and right shift | |
| | `&=` `^=` `\|=` | Compound assignment by bitwise AND, XOR, and OR | |
| 17 | `,` | Comma | Left-to-right → |

Above, the assignment operator `operator=` has already been overloaded for `Complex`. In principle, any of the above operators can be overloaded. A standard implementation of overloading `operator+` is

```
Complex operator+(const Complex& c1, const Complex& c2)
{
    return Complex(c1.real+c2.real, c1.imag+c2.imag);
}
```

Then, we can equivalently use for `Complex c, d, e`

```
e.operator=(operator+(d, c));
e = d + c;
```

It is entirely possible, though of course not advisable, to overload the `operator*` with the complex division, or any other operation. An example, where the standard meaning of an operator is "misused" frequently, is the bitshift `operator<<` and `operator>>`, see the next section on streams.

**Streams** Standard C-style in/output functionality (`printf`) works in C++, too. Additionally, *stream-based* in/output is provided in C++. We have already used the standard output stream `cout` and input stream `cin`. These are provided within the `iostream` library in namespace `std::`,

```
#include<iostream>
...
std::cout << "enter some number\n";
std::cin >> number;
...
```

`cout` is an object of type `std::ostream` (an output stream), while `cin` is of type `std::istream` (an input stream). We can use some reference `myout` like

```
std::ostream& myout = std::cout;
myout << "hello\n";
```

These streams use *operators* `operator<<` and `operator>>`, the following two lines are equivalent:

```
std::cout << "hello\n";
operator<<(std::cout, "hello\n");
```

The return type of `operator<<` is again an output stream, the operator returns `std::cout`. Thus, the return value can be passed to `operator<<` again, which enables multiple output objects consecutively,

```
operator<<(operator<<(std::cout, "hello "), "world\n");
// or, more conveniently
std::cout << "hello " << "world\n";
```

We have already used *manipulators* such as `std::endl`, which puts a newline and flushes the output. Manipulators are used like

```
std::cout "print text now!" << std::flush; // flush output (print NOW)
std::cout << "Line 1" << std::endl << "Line 2";
```

Other manipulators can be used for output formatting, such as `setprecision` or `fixed`, `scientific`. They are available in `#include <iomanip>`.

Once the operator `operator<<` is overloaded for the `Complex` class, printing via `cout << c;` is possible for `Complex c`.

*Stringstreams* `std::stringstream` are used to stream something to a string. They can e.g. be used to convert other types (`double`, `int`) to strings. They are available in `#include <sstream>`.

*Filestreams* can be used to read/write from a file on disc, include `#include <fstream>`. Relative pathnames are understood relative to the directory where the executable/application is executed! In a standard VSCode cmake build, this is the `build` folder.

## 1.2 Basic linear algebra – an introduction to C++

### 1.2.1 A basic vector class in C++

We develop a framework for linear algebra in C++. Based on the implementation of a vector (and matrix) class, we introduce concepts of object-oriented programming in C++. Performance of different implementations shall be discussed.

**Why?** Within the standard template library, there exists the `std::vector<>`, which we could use for vector operations. Moreover, there are high-performance linear algebra libraries such as Eigen `https://eigen.tuxfamily.org` that will surely be more complete, more versatile and faster than everything we can develop within this course. However, we implement both vector and matrix class from scratch. Vector and matrix class are a convenient example to discuss the basics of object oriented programming in C++ with a view to performance. Usage of templates, pointers and references, do's and dont's concerning memory allocation will pop up even for the most simple vector class implementation.

**Disclaimer** The basic linear algebra package developed within this course contains all functionality we need for the more advanced computational methods from the following chapters. However, they are

- not complete, in the sense that all routines that might be useful are implemented,
- not all-intuitive in handling, since optimal performace is a prerogative, and there is a trade-off between simplicity of implementation and intuitivity in usage,
- not parallel, in order to keep things simple, not requiring additional libraries,
- not using all the latest C++ standards (only some C++11 where it simplifies implementations).

The overall aim is to give an idea what scientific computing is all about, and enable to use and understand more advanced codes/libraries.

**Declaration of a vector class** We define the template vector class `Vector<T>` in our header file, where we intend the template argument `T` to be a simple type such as `int`, `double` or `complex<double>`. Below, the declaration of the template vector class `Vector<T>` is listed. This code snippet includes the minimal information that needs to be provided in the according header file `SCvector.h`. As we develop an all-header template class, the member function declarations will successively be replaced by definitions including the function bodies. Definitions within the header file are treated as `inline` and can be optimized at compile time.

```
template<typename T=double>
class Vector
{
private:
    T* data;
    int size;
    int alloc_size;

public:
    // default constructor
    Vector();
    // copy constructor
    Vector(const Vector& vec);
    // destructor
    ~Vector();
    // the assignment operator
    Vector& operator= (const Vector& vec);

    // constructor for vector of given size
    Vector(int size_);

    // provide read-only access to private member size
    int Size() const;
    // (re)allocate data memory and set size
    void AllocateMemory(int alloc_size_)
    // set size of vector, (re)allocate if necessary
    void SetSize(int size_);

    // direct access to data entries
    T& operator()(int i);
    const T& operator()(int i) const;
    // set all entries to val
    void SetAll(T val);
    // set all entries to random values
    void SetRandom(double min=0., double max=1., int seed=0);

    // vector addition
    void Add(const Vector &vec);
    // multiplication with scalar
    void Mult(T factor);
    // vector addition via += operator
```

```
    Vector& operator+=(const Vector &vec);
    // multiplication with scalar via *= operator
    Vector& operator*=(T factor);
    // L2-norm
    double Norm() const;
    // output routine
    void Print(std::ostream& os) const;
};
```

**Template class definition** The vector class is defined as a template class. The general idea of using templates is that we get different classes (in our case, for different vector element types, e.g. `int`, `double` or `complex<double>`) from the same code. We call such a vector via `Vector<double> a;` or `Vector<int> b;`. The default template argument (`typename T=double`) means that we can use `Vector<>` synonymously for `Vector<double>`.

Whenever a `Vector<T>` class is used for some specific type `T` in our actual project, the compiler generates the according template class, generating a unique and (somewhat) intelligible internal name. All occurances of `T` are replaced by the actual typename. E.g., if we use `Vector<int>`, the VisualStudio2019 compiler generates `??0?$Vector@H@SC@@QEAA@HH@Z`, if we use `Vector<double>` it generates `??0?$Vector@N@SC@@QEAA@HH@Z` and if we use `Vector<complex<double>>` it generates `??0?$Vector@V?$complex@N@std@@@SC@@QEAA@HH@Z`. C++ code is generated only as needed. In our case, if we use `Vector<int>` but not `Vector<complex<double>>`, then the compiler generates a `Vector<int>` class setting `T = int`, but not a complex vector class.

> At compile time, a template class is compiled *as needed*, i.e. when it is called, substituting the specific template argument(s).

All implemented routines have to work for those types that are actually used. Using `Vector<string>` will e.g. work only if all vector class members can be compiled if `T` is a `string`[1]. If all declarations and definitions of a template class are collected within header files that are included in the cpp source code, it is guaranteed that the relevant code is always generated and available for linking.

**Class member objects** The vector class has three member objects:

- its (logical) size `int size`,
- the size of the reserved data pointer `alloc_size`, which is always greater or equal to `size`,
- and the data pointer itself `T *data`.

It is, so to say, the container for the `size`, `alloc_size` and `*data` information, similar to a C `struct`. In our class, we define these members as `private`. They cannot be accessed from outside (via `a.size`), but only via `public` member functions (via `a.Size()` or `a.SetSize(n)`, see later).

[1] Some compilers are more generous than others, requesting that only used members can be compiled.

**Access specifiers:** Any class member object or function is declared `public`, `protected` or `private`:

- `public:` the member is accessible anywhere
- `protected:` the member is only accessible to members (and friends) of the class and to members (and friends) of derived classes
- `private:` the member is only accessible to members (and friends) of that class

**Class member functions** The first four member functions that are declared in our vector class are: default and copy constructor, destructor and assignment operator. These four members are special in the sense that, if any of these routines is missing, the compiler will add a default implementation. Additional member functions add functionality to our vector class. We define all member functions as `public`, they are accessible to the user of our vector class.

**Default constructor** The default constructor is called when the user puts `Vector<T> a;` (with `T` some type; if `T` is `double`, it can be omitted). For our vector, it should set the vector length to zero and not allocate any memory. Thus, we set the data pointer to null, where we use the C++11 `nullptr`. A simple inline definition looks like

```
Vector() { data = nullptr; size = 0; alloc_size = 0; }
```

Using *initializer lists*, the same constructor is implemented below

```
Vector() : data(nullptr), size(0), alloc_size(0) {}
```

One of these two lines replaces the respective line in the declaration above – the constructor is defined `inline`.

**Initializer list** Data members can be initialized via the initializer list or in the function body. (If the class is derived from some base class, also the base class can be initialized this way, see later). The initializer list is put after a colon `:` and before the function body within braces `{ }`. Initialization by the initializer list is executed before the function body. Initialization by the initializer list is mandatory for class members of reference type and constant class members.

**Implicit generation of the default constructor:** If no other constructor is present, the compiler will define a default constructor with empty body and empty initializer list. One can force this default constructor (if other constructors are present) by the `default` keyword, `Vector() = default;` The default constructor can be explicitly deleted by `Vector() = delete;` In this case, `Vector<> a;` will result in a compiler error. For a more complete introduction, see `https://en.cppreference.com/w/cpp/language/default_constructor`.

**Copy constructor** The general idea behind the copy constructor is that a new vector object `b` is generated, which exactly resembles some given, existing vector object `a`. Its signature is `Vector(const Vector&` it is called when the user puts `Vector<> b(a);` or `Vector<> b = a`. Then, `b` is an entity of its own, with its own data array, which contains the same values as the data array from `a`.

> **Implicit generation of the copy constructor:** If no copy constructor is present, the compiler will declare a copy constructor. The copy constructor calls the copy constructors of all member objects. The copy constructor can be explicitly deleted via `Vector(const Vector&) = delete;`. An implicitly generated copy constructor can be deleted by the compiler for several reasons, e.g. if the class has member objects that cannot be copied.

On call of the copy constructor, a new vector object shall be generated. It shall exactly resemble the original vector object. We want to allocate a new `data` array of the same size `vec.alloc_size`, and set `size` and `alloc_size` to the original vector's `vec.size` and `vec.alloc_size`. Then, we want to copy the data from the original vector, element by element, to the new vector. An inline definition is listed here

```
Vector(const Vector& vec) : size(vec.size), alloc_size(vec.alloc_size)
{
    if (alloc_size > 0)
    {
        data = new T[alloc_size];
        for (int i = 0; i < size; i++)
        {
            data[i] = vec.data[i];
        }
    }
    else
        data = nullptr;
}
```

A default copy constructor would not work well for our example; it would generate something entirely different. The default copy constructor would simply *copy the data pointer*, and not *allocate new memory for the vector copy*! Then, on `Vector<> b(a)`, `b` would have a separate integer size information `b.size` and `b.alloc_size`, but its `b.data` pointer would point to the same memory as `a.data`.

Note that, instead of copying element by element, a C-style block copy `memcpy` or `memmove` could be used. This is possible if and only if `T` is a simple type, and not e.g. a class. With the above routine, the assignment operator for `T` is called for copying each vector element.

Before we proceed to the destructor, there are some issues to discuss that arise when looking closer to the definition of the copy constructors. These issues comprise the keywords `new`, `const` and the reference operator `&`. These issues apply not only to the copy constructor, but member functions in general.

**Aspects of memory allocation** Above, memory is allocated dynamically using `new`. From a performance point of view, this may become critical. Allocating memory dynamically for a *small* number of times is of no consequence (and probably unavoidable), but should be avoided in large numbers.

**The `new` command and memory leaks** The `new` expression allocates storage for either a single object or an array of objects. The object exists until it is deleted by `delete`. If the original pointer is lost, the object becomes unreachable and cannot be deallocated: a *memory leak* occurs.

**Dynamic allocation of memory** Whenever an object of fixed size is allocated, this is done on the *stack*. When allocating e.g. an array via `new`, with its length *not* known at compile time, *heap* memory is used. Generally: dynamic memory allocation and deallocation are slow operations when compared to automatic memory allocation and deallocation on the stack, and should thus be avoided if performance is an issue.

**Call by value vs. call by reference** Note the argument of the copy constructor: `const Vector& vec`.

The ampersand `&` declares a *call by reference*, i.e. a reference to the input vector is passed to the constructor. If we change `vec` within the constructor body, the object from outside is changed. By using a `const` reference, `const Vector& vec`, it is actively prohibited to change the input vector inside the constructor (see next paragraph). The obvious alternative to call by reference is *call by value*, e.g. using `Vector(Vector vec)` without reference symbol `&`. Then, a copy of `vec` is passed to the function body. If we changed `vec` within the function, the original `vec` object outside is not changed.

**Call by value vs. call by reference**

- Call by value: realized e.g. as `void f(Vector<> v)`. On call, the input vector `v` is *copied*, the copy is available within the function body. If changed inside, the outside object `v` does not change.
- Call by reference: realized e.g. as `void f(Vector<> &v)`. On call, a reference to the original input vector `v` is provided within the function body. If changed inside, the original object `v` is changed as well. To prohibit such changes, use the `const` keyword.

**On `const` correctness** The `const` keyword states that the input vector `vec` must not be changed within the function body. Here, `const` is used as a type qualifier (the input vector is declared `const`). It can also be used to declare `const`-qualified member functions, there it marks members as "ok" to be used for `const`-qualified objects.

**`const` type qualifier.** If an object is declared const (as in `const int i = 5`) or an argument to a function is declared as `const` (as in `void f(const Vector<> &v)`, it must not be modified. Any attempt to do so (in the function body) results in a compiler error. A `const` defined object may only be passed as `const` reference or by value; in the latter case its copy constructor is called. See also `https://en.cppreference.com/w/cpp/language/cv`.

**`const`-qualified member functions.** A member function can be declared with `const` qualifier; the `const` is put after the parameter list in the function declaration. In a `const`-qualfied member function, `*this` is `const`; all member objects must remain `const`, only other `const`-qualified member functions may be called.

What is the general intention behind `const`-qualification? By defining an object as `const`, unintentional changes shall be detected by the compiler. E.g. for the `Vector<T>` copy constructor, the input vector should not be changed (or destroyed) as a new vector is constructed. The `const` keyword prohibits any such changes. In the vector class, we mark all member functions as `const`-qualified, where the `*this` object itself is not changed. They may be called also for `const` objects, they are marked as "not-changing-myself" to the compiler.

A very small example: let class `A` have two member functions, one of them `const`-qualified, one of them not:

```
class A
{
    void f();
    void f_const() const;
};
```

Above, `f()` is a member that possibly changes the object `*this` itself, while `f_const()` does not. They are qualified accordingly. For some instance of class `A`, both members can be called,

```
A a;
a.f();
a.f_const();
```

On the other hand, for some `const` object, only `f_const()` may be called, as `f()` will result in a compiler error,

```
const A a;
// a.f(); // compiler error!
a.f_const();
```

So, if we want to change something that shall stay the same "by accident" using `f()`, the compiler will tell us so.

Of course, using `const` is not at all mandatory; one can do without `const` completely. If the `const`/non-`const` qualfications are implemented consistently throughout the library, we speak of `const`-correctness. For any larger project, there will probably arise the necessity to evade `const`-correctness at some level, as *logically* constant functions are not always *physically* constant; this can be achieved e.g. through `const_cast` or the definition of `mutable` members.

**Destructor** We proceed to the next basic member function of any class.

The destructor `~` is called at the end of the object's lifetime, i.e. when an object ceases to exist at the end of its scope, or when an object allocated with `new` is deleted explicitly by `delete`. On call, the destructor shall free the resources that were allocated. If no destructor is provided, it is implicitly defined by the compiler. It calls all member object constructors in reverse order, and all base classes' destructors in reverse order.

For our vector class, the allocated data (remember: `new`!) should be deleted. All other members are integers (`int size` and `int alloc_size`) and need not be deleted explicitely.

```
~Vector() { if (data != nullptr) delete[] data; }
```

**Assignment operator** Last, the copy assignment operator (or `operator=`) is defined. It is called, whenever we use `a = b;`, which is the same as writing `a.operator=(b);`. The assignment operator does not only copy the input vector `b` to the vector itself `a`, but it also returns a reference to the same vector `a`. Thus, the expression `(a = b)` is a `Vector<T>&`, and can be e.g. assigned again to another `Vector<T> c`, reading `c = a = b`.

**Copy assignment operator:** The `operator=` will be declared by the compiler if it is not provided. It returns a reference to the object itself. It can be explicitly defaulted by `= default` or deleted (i.e. forbidden to be used) by `= delete`. For details on implicitly defined, defaulted or deleted assignment operators, see `https://en.cppreference.com/w/cpp/language/copy_assignment`.
For the sake of completeness, we mention that since C++11, there is also the **move assignment operator**, which allows to assign rvalues. `https://en.cppreference.com/w/cpp/language/move_assignment`

An inline implementation of the copy assignment for the vector class is given below. Within this implementation, the member functions `AllocateMemory` and `SetSize` are used, that are explained in detail later. For the moment, it is enough to know that `AllocateMemory` allocates some data pointer of given size and sets `alloc_size`, while `SetSize` sets the logical size `size`.

```
Vector& operator= (const Vector& vec)
{
    if (this == &vec)
        return *this;

    AllocateMemory(vec.alloc_size);
    SetSize(vec.size);

    for (int i=0; i<size; i++)
    {
        data[i] = vec.data[i];
    }
    return *this;
}
```

What happens as `if (this == &vec)`? Here, two pointer addresses are compared: `this` is the address of the vector object (what would be `self` in python), while `&vec` is the address of `vec`. If these addresses are the same, `*this` and `vec` are virtually the same object (with the same `size`, `alloc_size` and `data` array) and nothing is done. A possible example is calling `a = a;`.

**User-defined constructors** Depending on the specific application, other constructors can be defined. In the vector class example, we define a constructor that provides a vector of some given size. For positive size, the according data array is allocated.

```
Vector(int s) : size(s), alloc_size(s), data(nullptr)
{
    if (alloc_size > 0) data = new T[alloc_size];
}

Vector(int s, int as) : size(s), alloc_size(as), data(nullptr)
{
    if (alloc_size < size) alloc_size = size;
    if (alloc_size > 0) data = new T[alloc_size];
}
```

**Access to private members** Private members cannot be accessed directly from outside. E.g. the logical size of some `Vector<T> a` cannot be accessed via `a.size`, this results in a compiler error as `size` is `private`. Also the `data` array is not accessible. We discuss different ways of access.

For the vector length, we want read-only access, as re-setting the size requires also data memory reallocation. For re-setting the size or re-allocating memory, the `SetSize` and `AllocateMemory` members will be discussed below. For read-only access, we define

```
// provide read-only access to private member size
int Size() const { return size; }
```

Why is this read-only access? The return type of the `Size()` function is `int`. On call, at the `return` statement, a new and nameless `int` is allocated, copied from `len` and returned. If we write `int n = a.Size();`, this new and nameless `int` object is moved to `n`. If `n` is changed afterwards, this does not affect `a.size`, as it is another object. If we try to manipulate `a.Size()` directly, like `a.Size() += 1;`, the compiler returns an error statement: the nameless `a.Size()` return object is an rvalue, which can only form the right hand side of an assignment.

Currently, there is no read-only access to `alloc_size` implemented.

For vector elements, this read-only access will not be sufficient, as we want to modify vector entries in computational methods. This is realized through returning a reference to the data element.

```
// direct access to data entries
T& operator()(int i) { return data[i]; }
const T& operator()(int i) const { return data[i]; }
```

On `return` only the reference to the $i^{th}$ data entry, i.e. the information where to find it, is returned. Now we can access the $i^{th}$ element of `Vector<T> a` via `a(i)`. Contrary to the `a.Size()`, `a(i)` is an lvalue, which can form the left hand side of an assignment. `a(i) += 1;` will increase the $i^{th}$ vector entry by one.

Why do we need two access routines, one with `const` and one without? This is due to `const`-correctness mentioned earlier. The first operator gives read-and-write access (`T&`) to elements of `a`, but can only be called if `a` is not `const`. The second operator gives read-only access (`const T&`), and can also be called for `const Vector<T>`. The compiler chooses the `const`-qualified operator if possible, and returns an error if read-and-write access is requested for some `const Vector<T>`.

Naturally, if one prefers brackets `[]` over braces `()`, the according `operator[]` can be defined as well:

```
// access via brackets
T& operator[](int i) { return data[i]; }
const T& operator[](int i) const { return data[i]; }
```

There exist codes where e.g. parantheses `()` are used for one-based indexing and brackets `[]` for zero-based indexing. While this is perfectly possible (one operator returning `data[i-1]` and one returning `data[i]`) it is seriously not recommended!

Both the logical size of the vector `size` as well as the amount of allocated memory `alloc_size` can be changed via member functions. Of course, it is not enough just to set `size` or `alloc_size` to the given value, but also the correct amount of memory has to be allocated. We first discuss re-allocation of a given amount of memory: a new `data` pointer of given size is allocated, the original `data` is deleted, the information lost. The vector's logical size is set to `alloc_size`

```
void AllocateMemory(int alloc_size_)
{
    if (data != nullptr) delete [] data;
    if (alloc_size_ > 0)
        data = new T[alloc_size_];
    else
        data = nullptr;
    alloc_size = alloc_size_;
    size = alloc_size_;
}
```

In the `SetSize` member function, memory is re-allocated only if the new logical size exceeds the allocated memory size `alloc_size`. Otherwise, the data memory is kept. This is useful if within some iterative procedure, items are appended to a vector for a known number of times.

```
void SetSize(int size_)
{
    if (size_ > alloc_size)
        AllocateMemory(size_);
    size = size_;
}
```

Note: implementing `AllocateMemory` as above leads to loss of data, `SetSize` only does so if a `Vector<T>` grows in size to some value greater than the allocated memory. Another possibility includes *copying* the existing data into the new `data` array. As often in software development, both variants have their merit, and it's up to the developer which one to choose, or whether even both variants are supplied.

**Range checks etc.** Even for the small number of operations discussed for the `Vector<T>` class so far, one can think of many different range-checks for input data that could be useful. For example, sizes should always be non-negative, and data entry `data[i]` should only be accessed if `i<size`. Implementing many of these checks certainly helps in debugging, and makes the code more safe to use. On the other hand, many checks will lead to slower performance. A viable alternative is to implement checks in Debug mode only. When using CMake, the `NDEBUG` preprocessor flag can be used to decide between Debug and Release mode. A range check at data access could look like

```
T& operator()(int i)
{
    #ifndef NDEBUG
    if(i < 0 || i >= size)
        throw "Error in Vector::operator(): out of range";
    #endif
    return data[i];
}
```

At compile time, `NDEBUG` is evaluated, and the range check is either included or removed from the compiled code.

**Basic linear algebra member functions** Next, we concentrate on the basic usage of vectors. Essentially, we want to `multiply a vector by a scalar` and `add two vectors`. Below, the `Add` member function adds the content of a given `Vector<T> vec` to the actual `*this` vector, resembling a `+=` operation. The `Mult` member means multiplication of `*this` by a scalar `T factor`, resembling a `*=` operation. Additionally, in numerical methods we will need the possibility to add a multiple of some given vector to `*this`, which is implemented in `AddMultiple`,

```
void Add(const Vector &vec)
{
    for (int i=0; i<size; i++)
        data[i] += vec.data[i];
}
void Mult(T factor)
{
    for (int i=0; i<size; i++)
        data[i] *= factor;
}
void AddMultiple(T factor, const Vector &vec)
{
    for (int i=0; i<size; i++)
        data[i] += factor*vec.data[i];
```

```
}
```

Note that in all functions, the vector entries are changed in place, and no new vector object is generated. Also, the summand `vec` is called by const reference, and not copied. Thus, we avoid any `new` calls that decrease performance. The vectors have to be provided by the user. For debugging, it is again useful to add some check for equal length of the vectors.

The operators `+=` and `*=` implement similar functionalities.

```
Vector& operator+=(const Vector &vec2)
{
    for (int i=0; i<size; i++)
        data[i] += vec2.data[i];

    return *this;
}

Vector& operator*=(T factor)
{
    for (int i=0; i<size; i++)
        data[i] *= factor;

    return *this;
}
```

As with the assignment operator, they return references to the (augmented) vector itself.

We discuss how these linear algebra functions are to be used. As an example, we implement the statement `c = 2*a + b;`:

```
c = b;
c.MultAdd(2.,a);
```

This is definitely not as nice as writing `c = 2*a + b`. However, if we overloaded the `operator+` and `operator*`, we could use `c = 2*a + b`. Why is this not a good idea?

The reason is performance. A straightforward implementation of the `operator+` and `operator*` (outside the class) could look like

```
template <typename T>
Vector<T> operator*(const T& factor, const Vector<T>& vec)
{
    Vector<T> multvec(vec);
    multvec *= factor;
    return multvec;
}

template <typename T>
Vector<T> operator+(const Vector<T>& vec1, const Vector<T>& vec2)
{
    Vector<T> sumvec(vec1);
```

```
    sumvec += vec2;
    return sumvec;
}
```

Let us count the number of vector objects that are constructed, and the number of `new` statements, if we write `c = 2*a + b`. We find, using the VisualStudio 2022 compiler in Debug mode:

1. The first operator to be called is `operator*`, where a new vector object `Vector<> multvec` is generated.
2. On return, a temporary `Vector<>` object is generated, as the return type is `Vector`. We call this object `nameless1`. The `multvec` object is deleted.
3. Next, `operator+` is called for `nameless1` and `b`. In its first line, `Vector<> sumvec(*this);` calls the copy constructor, another vector object is generated.
4. On `return`, the return `Vector<>` object is generated as a copy of `sumvec`, while `sumvec` is deleted. We call this new temporary object `nameless2`.
5. Last, `operator=` is called, where all the data from the temporary `nameless2` is copied to `c`. Depending on whether `c` had data already allocated or not, this might include another `new` statement. The `nameless` objects are deleted.

On the other hand, `c = b; c.MultAdd(2.,a);` means only one call of the `operator=`, where data allocation happens if necessary. We can directly see that this affects performance in an exemplary C++ project. Note that in Release mode the behavior is optimized in so far as that the copy constructor is not called on `return`, in this case the objects `nameless1` and `nameless2` are never generated. Some compilers also do so in debug mode. Still, there are two additional vector objects generated (`multvec` and `sumvec`).

It is possible to implement the easy-to-use operators at optimal performance. This is done using *expression templates*. We will not cover expression templates in this lecture; the technique is used in high-performance linear algebra libraries such as Eigen `https://eigen.tuxfamily.org`. If expression templates are used, in the limit case a comparable performance to member-functions can be achieved. Depending on platform/CPU/compiler, the Eigen-Lib performs (much) better than our own implementation, using explicit vectorization (perform multiple operations at once).

**Additional nice-to-have member functions** We want to have some additional vector features, such as setting all vector elements to some given value, computing the Euclidean norm or printing the vector,

```
void SetAll(T val) { for (int i=0; i<size; i++) data[i] = val; }

double Norm() const
{
    double sum = 0;
    for (int i=0; i<size; i++)
    {
        // std::norm implements the absolute value squared!
        sum += std::norm(data[i]);
```

```
    }
    return sqrt(sum);
}

void Print(std::ostream& os) const
{
    os << "[ ";
    for (int i=0; i<size; i++)
        os << data[i] << " ";
    os << "]" << std::flush;
}
```

In using `std::norm` from the standard template library, we assume that the element type `T` is some type for which `std::norm` is defined, i.e. some numeric type such as `int`, `double` or `complex<double>`. Note that `std::norm` in fact is the square of the mathematical norm by definition, `https://en.cppreference.com/w/cpp/numeric/complex/norm`. Using `Vector<string>` or similar and calling its `Norm()` will lead to a compiler error.

Another nice-to-have function that is implemented outside the class is the outstream operator `operator<<`,

```
template <typename T>
inline std::ostream& operator<<(std::ostream& os, const Vector<T>& vec)
{
    vec.Print(os);
    return os;
}
```

Now, one can print any `Vector<> a;` like `cout << a << endl;`.

**Template specialization** For some functions, the implementation is essentially different depending on the template type `T`. For all functions discussed so far, the implementation was essentially the same for `T==int`, `T==double` and `T==complex<double>`, the compiler only replaced `T` accordingly. As an example where template specialization is necessary, we consider the `SetRandom` member function. Generating random `int`, `double` and `complex<double>` is inherently different.

We include `random` from the C++ `std::` namespace. For `T == double`, the function body essentially contains

```
std::uniform_real_distribution<double> unif(min, max);
std::default_random_engine re;
re.seed(seed);
for (int i = 0; i < size; i++) data[i] = unif(re);
```

For complex arguments, the last line is replaced by

```
for (int i = 0; i < size; i++)
    data[i] = std::complex<double>(unif(re), unif(re));
```

For integer arguments, a different random distribution is needed, the first line is replaced by

```
int imin = min; int imax = max;
std::uniform_int_distribution<> unif(imin, imax);
```

For our template class, we can implement these different functions by *template specialization*:

```
template <>
void Vector<double>::
SetRandom(double min, double max, int seed)
{
    std::uniform_real_distribution<double> unif(min, max);
    std::default_random_engine re;
    re.seed(seed);
    for (int i = 0; i < size; i++) data[i] = unif(re);
}

template <>
void Vector<std::complex<double>>::
SetRandom(double min, double max, int seed)
{
    // ...
}

template <>
void Vector<int>::
SetRandom(double min, double max, int seed)
{
    // ...
}
```

This specialized functions are listed *outside* the class scope (after `template<typename T=double> class Ve`

In this variant, `SetRandom` is declared only in the template class, and defined for the three distinct template arguments `int, double, complex<double>`. There is no definition for other template arguments available. A *default* member can be defined either directly inside the class,

```
template<typename T=double>
class Vector {
    ...
    void SetRandom(double min, double max, int seed)
    {
        // put default code here
    }
};
```

or, equivalently, outside the class definition,

```
template <typename T>
void Vector<T>:: SetRandom(double min, double max, int seed)
{
```

```
    // put default code here
}
```

For some other template type, this default fallback is called, e.g. when calling

```
Vector<size_t> a(5); a.SetRandom(0,100);
```

The specialization (for `int`, `double`, `complex<double>`) always overrides the default definition.

### 1.2.2 Inheritance and abstract base classes

In the previous section, we discussed object-oriented programming, or rather basic class implementation, along the example of a vector class. In this section, we extend this discussion from stand-alone template classes (as the `Vector<T>` class) to defining families of objects linked through inheritance. Again, we keep things as simple as possible. Again, we choose an exemplary class (or rather, a family of classes): matrices and other linear operators.

**Matrices and linear operators** One can see a matrix as a stand-alone object, as a tabularized collection of numerical data. Let $\mathbf{A}$ be an $m \times n$ matrix,

$$\mathbf{A} = \begin{bmatrix} A_{00} & \dots & A_{0\,n-1} \\ \vdots & \vdots & \vdots \\ A_{m-1\,0} & \dots & A_{m-1\,n-1} \end{bmatrix}. \tag{1.1}$$

The transpose $\mathbf{A}^T$ of a real $m \times n$ matrix is of shape $n \times m$. For complex matrices, not only its transpose $\mathbf{A}^T$, but also its complex adjoint $\mathbf{A}^H = \overline{\mathbf{A}^T} = (\overline{\mathbf{A}})^T$ is often used,

$$\mathbf{A}^T = \begin{bmatrix} A_{00} & \dots & A_{m-1\,0} \\ \vdots & \vdots & \vdots \\ A_{0\,n-1} & \dots & A_{n-1\,m-1} \end{bmatrix}, \qquad \mathbf{A}^H = \begin{bmatrix} \overline{A_{00}} & \dots & \overline{A_{m-1\,0}} \\ \vdots & \vdots & \vdots \\ \overline{A_{0\,n-1}} & \dots & \overline{A_{n-1\,m-1}} \end{bmatrix}. \tag{1.2}$$

Matrices satisfying $\mathbf{A} = \mathbf{A}^T$ are symmetric, while $\mathbf{A} = \mathbf{A}^H$ implies that $\mathbf{A}$ is *hermitian*. Other common notations for the complex adjoint are $\mathbf{A}^*$ or $\mathbf{A}^\dagger$.

Matrices can be multiplied to vectors (or other matrices), given the dimensions fit. For a matrix $\mathbf{A}$ and a vector $\boldsymbol{x}$, the matrix-vector product is another vector, defined by

$$\boldsymbol{y} = \mathbf{A}\boldsymbol{x}, \quad \begin{bmatrix} y_0 \\ \vdots \\ y_{m-1} \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + \dots + A_{0\,n-1}x_{n-1} \\ \vdots \\ A_{m-1\,0}x_0 + \dots + A_{m-1\,n-1}x_{n-1} \end{bmatrix} \quad \text{or} \quad y_i = \sum_{j=0}^{n-1} A_{ij}x_j. \tag{1.3}$$

Matrices are most commonly used to describe linear transformations of vectors. Examples are

- the rotation matrix transforming a vector in three-dimensional space to its rotated counterpart,

- material parameters for anisotropic materials, transforming e.g. electric field to electric current (conductivity matrix), magnetic field intensity to magnetic flux density (magnetic permeability) or strains to stresses (the fourth order stiffness tensor)

Even more commonly, matrices are used to describe linear systems of equations. An important example in applications is the stiffness matrix of the finite element method, linking solution vector and vector of external forces.

By definition, a linear operator is an object that transforms a vector to another vector while maintaining linearity, i.e. transforming the sum of two vectors yields the sum of the respective transformed vectors, and transforming a multiple of a vector yields the multiple of the transformed vector. A linear operator acting on a vector space can always be represented by a matrix. On the other hand, each matrix represents a linear operator. Thus, one might argue, matrices and linear operators are virtually the same object.

We choose a different approach: *a matrix is a special type of linear operator*, namely one where the table of matrix entries is known and stored in a tabular manner. One can easily imagine linear operators, where the table of matrix entries is not known, or not stored in the traditional $m$ by $n$ form:

- a specialized matrix class for symmetric matrices,
- a sparse matrix, where the zero entries are not stored,
- solving a linear system of equations – the according matrix would be the inverse of the system matrix, but one will often not compute this matrix explicitly,
- the application of a low-rank matrix, e.g. $\boldsymbol{a}\boldsymbol{b}^T$, where storing or applying vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ consecutively is much cheaper than computing the actual matrix,
- observations of distributed quantities, which are e.g. computed through integration,
- preconditioning of linear systems, which will be discussed in *Numerik und Optimierung*.

Thus, we implement the `LinearOperator<T>` as an *abstract base class*: the abstract base class contains all information that is available, no matter what the exact implementation of a specific linear operator might be. In our case, `LinearOperator<T>` contains most essentially

- its dimensions (`height` and `width`): the `LinearOperator` can be applied to vectors of size `width` and the application yields a vector of size `height`,
- the pure virtual definition of the `Apply` member function that needs to be overloaded for each specific linear operator to implement the application to a vector,
- default implementations of member functions `ApplyT, ApplyH` that implement operations like $\boldsymbol{r} = \mathbf{A}^T\boldsymbol{x}$, $\boldsymbol{r} = \mathbf{A}^H\boldsymbol{x}$, respectively.

The `LinearOperator` class will be `virtual`, it cannot be instanciated like `LinearOperator<T> A;`.

From this abstract base class, then `Matrix<T>` is derived. It contains

- a data pointer for the matrix entries,
- some kind of `operator()` to get to the matrix entries from outside,

- it implements the `Apply` member, which is the matrix multiplication,
- if possible/needed, it also implements `ApplyT` and `ApplyH`, otherwise the default implementation `LinearOperator<T>::ApplyT`, `LinearOperator<T>::ApplyH` will be called,
- other useful members, like `Transpose`, `SetDiag`, ...

Whenever other specialized classes are derived from `LinearOperator<T>`, different useful members will be appropriate. However, each derived class will use the base classes height and width parameters, and each must implement the `Apply` routine.

Why do we need this abstract base class? Through the common abstract base class, the compiler enforces consistency of the derived classes. In our case, all matrix classes have to sport the same – exactly the same – interface to the `Apply` member function and the same `height` and `width`. Public access to `height` and `width` is ensured through the base class, and needs not be re-implemented for each derived class. More importantly still, we will implement tools that work for all kinds of `LinearOperator`, as such a tool uses only `Apply` – e.g. the conjugate gradient solver or iterative eigenvalue solvers. Then, we can use the abstract base class `LinearOperator` as a common interface in defining the respective function. When calling it later, we pass some specialized `Matrix` or other object – virtual inheritance will make sure that the correct `Apply` routine is called.

**An abstract base class `LinearOperator`** The abstract base class `LinearOperator` is a template class, where the template argument `typename T` corresponds to the vector template type (and later, the matrix entry type). The template types of `Vector` and `LinearOperator` have to coincide exactly, we do not provide automatic type promotion. In other words, some `LinearOperator<int>` cannot be applied to some `Vector<double>`, although the underlying mathematics would make sense[2]. The `LinearOperator` class is implemented within a few lines, without much functionality:

```
template <typename T = double>
class LinearOperator
{
protected:
    int height;
    int width;

    LinearOperator() : height(0), width(0) {}
    LinearOperator(int h) : height(h), width(h) {}
    LinearOperator(int h, int w) : height(h), width(w) {}

    virtual ~LinearOperator() {}

    LinearOperator(const LinearOperator &) = default;
    LinearOperator &operator=(const LinearOperator &) = default;

public:
```

[2] This kind of type promotion is possible using template traits classes in "classic" C++ or `std::common_type` in C++11

```
    int Height() const {return height;}
    int Width() const {return width;}
    bool IsSquare() const {return (width==height);}

    virtual void Apply(const Vector<T> &x,
                       Vector<T> &r,
                       T factor = 1.) const = 0;

    // overload if wanted, otherwise fallback exception called when used
    virtual void ApplyT(const Vector<T> &a,
                       Vector<T> &result,
                       T factor = 1.) const;
    virtual void ApplyH(const Vector<T> &a,
                       Vector<T> &result,
                       T factor = 1.) const;
};
```

The `Apply` member function,

```
// in abstract base class
virtual void Apply(const Vector<T> &x,
                       Vector<T> &r,
                       T factor = 1.) const = 0;
```

is a `virtual` member function. Its behavior is overridden by the implementation of the derived class, even if the derived class is not known at compile time. What does this mean? Imagine there is the `LinearOperator`, with derived classes `Matrix` and `SparseMatrix`. We define a function, where we pass `LinearOperator<T>& A`, as the function should work for `A` a `Matrix<T>` or a `SparseMatrix<T>`. As `Apply` is a virtual member, inside this function, `A.Apply` will be overriden by the derived class `Apply`. Thus, if `A` is a `Matrix<T>`, `Matrix::Apply` is called, and if it is a `SparseMatrix<T>`, `SparseMatrix::Apply` is called, although it was not known at compile time which version would be needed. If we forgot the `virtual` keyword, inside the function the `LinearOperator::Apply` would be called – not the desired effect (and indeed, leading to a compiler error, see the next paragraph). There is a small overhead for building the virtual function lookup table, but it is not considered relevant for the level of performance in this course.

**Virtual member functions** Virtual functions are member functions whose behavior can be overridden in derived classes. As opposed to non-virtual functions, the overriding behavior is preserved even if there is no compile-time information about the actual type of the class. ((c) `https://en.cppreference.com/w/cpp/language/virtual`)

The `= 0` statement after `Apply` makes the abstract base class a pure virtual class. A pure virtual class cannot be explicitly constructed. Using

```
LinearOperator<T> A;
```

leads to a compiler error, as `A` has no `A.Apply` member function implemented. In each derived class that should be instanciable, `Apply` *must* be overridden. One can (not necessarily) specify this by `override`

```
// in derived class, e.g. Matrix<T>
virtual void Apply(const Vector<T> &x,
                         Vector<T> &r,
                         T factor = 1.) const override
                         { ... implementation here ... }
```

**Pure virtual functions:** A pure virtual function is declared `= 0`. Any class with a pure virtual member is an abstract (base) class. No objects of this class can be created, it cannot be used as return type (references to the ABC can be returned). In a concrete derived class, all pure virtual members must be overridden.

The `override` specifier can be used whenever a virtual member of the base class is overridden. This base class member is not necessarily pure virtual (i.e. not necessarily `= 0`), but may be defined (i.e. have a function body within braces `{}`). The main reasons to use the `override` specifier when overriding any `virtual` member are that

- the compiler errs if you make a mistake and override something that *isn't there* in the base class,
- the compiler errs if you make a mistake and override something that *isn't virtual* in the base class,
- everyone else can see that this is an implementation overriding something from the base class.

If one wants to make sure that the overriding implementation is not again overridden by a further derived class (i.e. by the grand-child of the base class), one can put `final` instead of or additionally to `override`.

**Virtual function specifiers**

- `override` ensures that the function is virtual and is overriding a virtual function from a base class,
- `final` ensures that the function is virtual and specifies that it may not be overridden by derived classes,
- `override` and `final` can be combined.

The members `ApplyT`, `ApplyH` are virtual member functions, but not pure virtual. This means they *can* be overridden in any derived class, but it is not necessary to do so. If these functions are not overridden, but called at run-time, the default implementation from the abstract base class is executed. In our case, an exception is thrown.

```
// overload if wanted, otherwise fallback exception called when used
virtual void ApplyT(const Vector<T> &a, Vector<T> &result, T factor = 1.)
    const
{
    throw "LinearOperator::ApplyT called in base class, needs to be overridden in
}
```

```
virtual void ApplyH(const Vector<T> &a, Vector<T> &result, T factor = 1.)
    const
{
    throw "LinearOperator::ApplyT called in base class, needs to be overri
}
```

**A basic matrix class** We discuss a matrix class, which is derived from the base class `LinearOperator<T>`. It is declared as

```
template <typename T = double>
class Matrix : public LinearOperator<T>
{
protected:
    // data pointer
    T *data;
    // memory size
    int alloc_size;

public:
    // constructors
    Matrix();
    Matrix(int h);
    Matrix(int h, int w);
    Matrix(const Matrix &m);
    // destructor
    virtual ~Matrix();
    // assignment
    Matrix &operator=(const Matrix &m);

    // overridden from LinearOperator<T>
    void Apply(const Vector<T> &x, Vector<T> &r, T factor = 1.)
        const override;
    void ApplyT(const Vector<T> &x, Vector<T> &r, T factor = 1.)
        const override;
    void ApplyH(const Vector<T> &x, Vector<T> &r, T factor = 1.)
        const override;

    // matrix-specific members
    void AllocateMemory(int alloc_size_);
    void SetSize(int h, int w);

    inline T &operator()(int i, int j);
    inline const T &operator()(int i, int j) const;

    void SetAll(T val);
    void SetRandom(double min=0., double max=1., int seed=0);
```

```
    void SetDiag(T val);

    Matrix &operator+=(const Matrix &mat2);
    Matrix &operator*=(T factor);

    void Print(std::ostream& os) const override;
};
```

`Matrix<T>` is derived from `LinearOperator<T>`. This is stated in the first two lines above:

```
    template <typename T = double>
    class Matrix : public LinearOperator<T>
```

Any class can be declared as derived from one (or more) base classes; the base class may in turn be derived from its own base class. They form an inheritance hierarchy. In this course, we will not use multiple inheritance.

A base class is present as a *sub-object*. All members of the base class are also members of the derived class. In our example, `Matrix` does not need another height and width specifier, it uses `LinearOperator::height` and `LinearOperator::width`. Also the public members `LinearOperator::Height()` and `LinearOperator::Width()` are available as public members of `Matrix`.

**Public inheritance** Public inheritance is specified by the `public` keyword. In this case, all public members of the base class are accessible as public members of the derived class. Protected members of the base class are accessible as protected members in the derived class. Private members of the base class are NOT available in the derived class.
Other options are `protected` inheritance (public and protected members of the base class are accessible as protected members of the derived class) or `private` inheritance (all public and protected members of the base class are accessible as private members of the derived class).

**Member variables** `Matrix<T>` uses `height` and `width` inherited from `LinearOperator<T>`. They must not be defined repeatedly. Additional members are the data pointer `T* data` and its size `int alloc_size` (similar to the `Vector<T>` example).

**Constructors, destructor and assignment operator in derived class** Any constructor of the derived `Matrix` class has to initialize also the `LinearOperator` members. To this end, one calls the appropriate base class constructor. For the default constructor of `Matrix`, we want a matrix of zero size. We add the base class default constructor (which sets `height = width = 0`) to the initializer list, for example

```
Matrix() : LinearOperator<T>(), data(nullptr), alloc_size(0) {}
```

The data pointer is, as for the default vector constructor, the nullpointer. Similarly, we provide a constructor for an $h \times w$ `Matrix`

```
Matrix(int h, int w)
: LinearOperator<T>(h, w), data(nullptr), alloc_size(h*w)
{
    if (h > 0 && w > 0)
    {
        data = new T[h*w];
    }
}
```

If we call `Matrix<T> M(m, n);`, the following steps are taken consecutively: first `LinearOperator(m, n)` is called, which sets `M.height = m` and `M.width = n`. Then, the `data` pointer is initialized as the null-pointer `nullptr` and `alloc_size = h*w`. Afterwards, the constructor function body is executed, where a `data` array is allocated in case `m>0` and `n>0`, i.e. height and width are positive.

The destructor of a derived class acts in the following way: first, the function body (code within braces `{}`) is executed, then destructors of all class members are called in reverse order of declaration, and finally destructors of all base classes in reverse order of declaration are executed. This holds at least for our simple inheritance architectures, otherwise see `https://en.cppreference.com/w/cpp/language/destructor`. Thus, in the `Matrix` destructor, only the `data` pointer is deleted in the function body (if not `nullptr`).

```
~Matrix()
{
    if (data != nullptr)
        delete[] data;
}
```

On call, i.e. at the end of the lifetime of a `Matrix` object, first the `data` array is deleted as specified above. Then, the `data` pointer is destroyed as a member of `Matrix` (this destroys the address, does not free the memory – we had to do that beforehand). Last, the `LinearOperator` destructor is called, which essentially destroys `height` and `width`.

Depending on the compiler (as a rule of thumb, Microsoft's Visual Studio compiler is per default rather permissive, while gcc is not), when template classes are derived, members of the base class cannot be accessed directly. I.e., within some `Matrix<T>` function, using `height` or `width` will result in a compiler error. One workaround is to use `this->height` and `this->width` in the derived class. Another option is to use

```
using LinearOperator<T>::height;
using LinearOperator<T>::width;
```

in the derived `Matrix<T>` class.

Sometimes, we want to call a base class function explicitly. This can be achieved via putting `LinearOperator<T>::` in front. This is done for example in the assignment operator

```
Matrix &operator=(const Matrix &m)
{
    if (this == &m)
        return *this;
```

```
    // call the LinearOperator assignment op (for height and width)
    LinearOperator<T>::operator=(m);

    if (alloc_size < m.height*m.width)
        AllocateMemory(m.height*m.width);

    for (int i = 0; i < height * width; i++)
        data[i] = m.data[i];

    return *this;
}
```

Above, the `LinearOperator::operator=` is called inside `Matrix::operator=`. Let us recall the action of `LinearOperator::operator=`: in the relevant line, essentially `this->height` and `this->width` are set to `m.height` and `m.width`. Afterwards, `data` is allocated and copied.

**Data access** When implementing the matrix entry operator, `operator()`, that provides access to the correct data array entry, we have to make a fundamental decision: whether the one-dimensional `data` array is interpreted as containing the rows or the columns of the matrix consecutively. The first alternative, also called row-major storage, is the C-style variant[3]. Row vectors are stored consecutively, which can be visualized via

$$\begin{bmatrix} \texttt{data[0]} & \texttt{data[1]} & \dots & \texttt{data[width-1]} \\ \texttt{data[width]} & \texttt{data[width+1]} & \dots & \texttt{data[2*width-1]} \\ \texttt{data[2*width]} & \texttt{data[2*width+1]} & \dots & \texttt{data[3*width-1]} \\ \vdots & \vdots & \ddots & \vdots \\ \texttt{data[(height-1)*width]} & \texttt{data[(height-1)*width+1]} & \dots & \texttt{data[height*width-1]} \end{bmatrix} \tag{1.4}$$

The benefit for this type of storage is that, on matrix-vector-multiplication, each row is multiplied with the vector, where for each row the entries are stored consecutively.

The second alternative is column-major storage, or FORTRAN-style. The `data` array contains the *column* vectors consecutively. This type of storage is to be preferred if the same kind of operation, for example a linear operator, is applied to all the columns. This happens if the matrix contains different right hand side to some linear system, for example.

Both variants – row-major and column-major – have their benefits and drawbacks. Ideally, `data` array access is not used directly anywhere, but entries are accessed only via the `operator()`. Then, the mapping from index pair to position in the data array happens only once (or twice, thanks to const-correctness) in the code. If the storage type is changed for some reason, only one (or two) functions need to be modified.

[3] In C, defining `double m[3][4]` yields a data array of size 12, where three blocks of four doubles are aligned consecutively.

**Overloading pure virtual members** To allow a call for `Matrix<T> M`, the pure virtual member `LinearOperator<T>::Apply` has to be overridden, we have to provide an apply-function.

```
void Apply(const Vector<T> &x, Vector<T> &r, T factor = 1.) const
{
    if (x.Size() != width)
    throw "Matrix::Apply dimensions don't fit";
    r.SetSize(height);

    for (int i = 0; i < height; ++i)
    {
        r(i) = 0.0;
        for (int j = 0; j < width; ++j)
            r(i) += operator()(i, j) * x(j);
        r(i) *= factor;
    }
}
```

This function implements the matrix-vector product: the input vector `x` is multiplied by the matrix object `A` and, optionally, by a scalar `factor`. The result is written into the (non-`const`) output vector `r`.

The matrix-vector product $r_i = \sum_{j=0}^{n-1} A_{ij} x_j$ can be realized conveniently through two nested `for`-loops using the data-access `operator()`. The number of operations is of order $m \cdot n$. When it comes to efficiency, *parallelization* and *vectorization* can greatly increase the performance. We do not attempt to achieve such an optimized matrix-vector product here; however, we compare our textbook implementation to matrix-vector multiplications as offered by Eigen etc.

**Overloading virtual members** The `LinearOperator<T>::ApplyT/ApplyH` functions have been implemented in the base class. However, if called, an exception is thrown. Therefore, the base class function is *overloaded* in the derived `Matrix<T>` class in the following way

```
void ApplyT(const Vector<T> &x, Vector<T> &r, T factor = 1.)
    const override
{
    r.SetSize(width);

    for (int i = 0; i < width; ++i)
    {
        r(i) = 0.0;
        for (int j = 0; j < height; ++j)
        {
            r(i) += operator()(j, i) * x(j);
        }
        r(i) *= factor;
    }
}
```

Whether or not dimensions of matrix and vectors are checked beforehand is – again – up to the developer.

**Other `Matrix` members** We do not elaborate on the other member functions such as `SetSize`, `SetAll` etc. The only member we discuss is the `Transpose()` operation.

On `A.Transpose()`, `Matrix<T> A` shall be transposed. Ideally this is done in-place, i.e. no new memory is allocated. There are inherently different ways to achieve this:

- `height` and `width` are swapped, the `data` array is changed in such a way that it represents the transposed matrix. This is fairly simple for square matrices; for rectangular matrices, this swapping is far from trivial! It becomes simple (and not optimally efficient) if a new `data` array is appointed. Probably it is best to abstain from a `Transpose()` member, but implement multiplication with the transpose matrix (`ApplyT`) instead.

- derive a class `TransposeMatrix<T> : public Matrix<T>` where only the `operator()` needs to be redefined. The `data` entry is untouched, `Transpose()` returns a `TransposeMatrix` that evaluates `operator(i,j)` differently. Main disadvantage: the `operator()` needs to be `virtual`, which makes it slower, which is in turn not optimally efficient – `operator(i,j)` is a small operation that is called for a large number of times!

- add another member `storage_type`, which can be set to indicate whether row-major or col-major storage is used. In `operator()`, the `storage_type` is evaluated, it returns either `data[i*width+j]` or `data[j*height+i]`. On `Transpose`, `storage_type` is changed. Having this kind of control over the storage is also useful for interfacing with LAPACK routines, which rely on FORTRAN-style column-major storage. Compared to the virtual function table lookup, evaluating `storage_type` is fast.

**Matrix and vector products** Various products such as the vector inner product $\boldsymbol{a} \cdot \boldsymbol{b}$, the tensor product of two vectors $\boldsymbol{a} \otimes \boldsymbol{b}$, the matrix inner product $\mathbf{A} : \mathbf{B}$, the matrix vector product $\mathbf{A}\boldsymbol{b}$, or the matrix product $\mathbf{AB}$ can be defined in additional routines. For complex numbers, one should take care to distinguish between the complex inner product $\boldsymbol{a}^H \boldsymbol{b}$ and the matrix product $\boldsymbol{a}^T \boldsymbol{b}$, probably both implementations can be useful. As always, we avoid generation of new objects. Exemplarily, we provide the matrix vector product as a standalone function (not a member of either matrix or vector class). Similar to `numpy`, we refer to it as `DotProduct`

```
// c = A . b
template <typename T>
void DotProduct(const Matrix<T>& A, const Vector<T>& b, Vector<T>& c)
{
    c.SetSize(A.Height());
    c.SetAll(0);
    for (int i=0; i<A.Height(); i++)
        for (int j=0; j<A.Width(); j++)
            c(i) += A(i,j)*b(j);
}
```

**Efficiency of implementation** We compare the efficiency of our implementation to the Eigen-lib. We observe (at least on my MacBook Pro, using the native clang compiler)

- when using member functions (`Apply` function, `operator+=`) etc, the performance is comparable for small-sized matrices and vectors, small meaning height/width up to 10,
- for larger matrices, Eigen behaves better than our implementation, probably using storage layout; in debug-mode, Eigen is slower than all our own implementations,
- when the `operator*` is implemented explicitly, performance drops significantly, with and without debug information; the drop is less significant for larger matrices, where the `new` and `delete` statements become less significant.

There are different options to get forward-operations like matrix-vector multiplication more efficient, but are not treated in this course. Note that these options can lead to platform dependence in the linear algebra code.

- **the BLAS package** (Basic Linear Algebra Subprograms) is a library implementing variants of matrix-vector multiplications. Different implementations for different platforms/architectures are available. The usage is similar to the LAPACK library discussed in Section 2.2.7. C-style function calls are used instead of `for`-loops in our implementations,
- **shared-memory parallelization** using OpenMP means that the loops can be parallelized with a rather small amount of preprocessor commands (`#pragma omp ...`) to use multi-core processors on shared memory,
- **vectorization** by the Single Input Multiple Data (SIMD) concept, which is applicable for structured processes such as matrix-vector multiplication, needs to be supported by the processor, which is usually the case nowadays; this concept is used in the Eigen lib,
- **GPU programming** using the massively parallel layout of dedicated computing GPUs (conventional GPUs were designed to compute `float` only, a large drawback for scientific computing applications), using packages like NVIDIA's CUDA toolkit.

# 2 Numerical linear algebra

A major task in scientific computing is solving linear systems of equations. Such a linear system is of the form

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \tag{2.1}$$

for some given right hand side $\boldsymbol{b}$. We distinguish the following cases:

1. **Invertible matrices** Matrix $\mathbf{A}$ is invertible if its inverse $\mathbf{A}^{-1}$ exists. This implies that $\mathbf{A}$ is square ($n = m$). We consider three different cases of application:

   - Solving the linear system once (one right hand side vector $\boldsymbol{b}$, or also a matrix containing several right hand sides). We discuss Gaussian elimination and iterative solution techniques.

   - Solving the linear system repeatedly (consecutive application, e.g. in an iterative scheme), which means an efficient implementation of the operator application of $\mathbf{A}^{-1}$. We discuss LU and Householder (QR) factorizations as well as singular value decompositions.

   - Actually computing $\mathbf{A}^{-1}$.

2. **Overdetermined systems** Matrix is not square, there are more equations than degrees of freedom in $\boldsymbol{x}$ ($m > n$). We discuss least-squares fits.

3. **Underdetermined systems** Matrix is not square, there are less equations than degrees of freedom in $\boldsymbol{x}$ ($m < n$), or the equations are degenerate. We discuss pseudo inverses and singular value decompositions.

4. **Large sparse systems** For very large $n, m$, but with few non-zero matrix entries per line, special techniques have been developed. A major field of application are systems generated through the finite element, finite volume, or finite difference method. We discuss implementation of a sparse matrix class as well as appropriate solution techniques.

5. **Iterative solution techniques** Are mostly used for large sparse systems.

This chapter heavily relies on [4], for background information on the mathematics we refer also to [2]. Within this course, preconditioning techniques are not addressed in a universal sense, as preconditioning depends mostly on the mathematical structure of the underlying physical problem.

**Basic matrix properties** We recall the essential properties of matrices (and vectors), as far as they are necessary to specify concepts of solvability, accuracy of solutions and stability of algorithms.

Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ or $\mathbb{C}^{m\times n}$ be a real or complex valued matrix. As all concepts work for real and complex numbers, we use complex matrices in the following.

We define rank, image and nullspace (or kernel) of $\mathbf{A}$ as follows.

The *image* of $\mathbf{A}$ is the space of all possible right hand sides,

$$\mathrm{im}(\mathbf{A}) = \{\boldsymbol{b} = \mathbf{A}\boldsymbol{x} \text{ for } \boldsymbol{x} \in \mathbb{C}^m\}\,. \tag{2.2}$$

The *nullspace* or *kernel* of $\mathbf{A}$ is the space of all vectors mapped to zero,

$$\ker(\mathbf{A}) = \{\boldsymbol{x} \in \mathbb{C}^n \text{ such that } \mathbf{A}\boldsymbol{x} = \mathbf{0}\}\,. \tag{2.3}$$

The *rank* of a matrix is the number of linear independent lines. It holds

$$\mathrm{rank}(\mathbf{A}) = \dim(\mathrm{im}(\mathbf{A})) = n - \dim(\ker(\mathbf{A})). \tag{2.4}$$

A square matrix is regular, the linear system is uniquely solveable if and only if $\mathrm{rank}(A) = m = n$. Then, the nullspace is zero $\ker(\mathbf{A}) = \{\mathbf{0}\}$ and the image is the whole space $\mathrm{im}(\mathbf{A}) = \mathbb{C}^n$.

We will see that eigenvalues and – more importantly – singular values characterize numerical properties of matrices.

For hermitian (real: symmetric) square matrices $\mathbf{A} \in \mathbb{C}^{n\times n}$ with $\mathbf{A}^H = \mathbf{A}$ it is possible to find $n$ real-valued eigenvalues $\lambda_i$ and corresponding eigenvectors $\boldsymbol{v}_i$ such that

$$\mathbf{A}\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i. \tag{2.5}$$

The number of zero eigenvalues corresponds to the dimension of the nullspace, the number of non-zero eigenvalues corresponds tot he dimension of the image.

For non-square, non-hermitian (non-symmetric) matrices, the *singular value decomposition* is probably of higher interest.

A matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$ can be decomposed

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H \tag{2.6}$$

with

- $\mathbf{U} \in \mathbb{C}^{m\times m}$ unitary (orthogonal),
- $\mathbf{V} \in \mathbb{C}^{n\times n}$ unitary (orthogonal),
- $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ diagonal with real-valued diagonal entries $\Sigma_{ii} = \sigma_i \in \mathbb{R}$ the *singular values*.

Again, the number of non-zero singular values corresponds to the dimension of the image. The singular values of $\mathbf{A}$ equal the square root of the eigenvalues of $\mathbf{A}^H\mathbf{A}$,

$$\sigma_i(\mathbf{A}) = \sqrt{\lambda_i(\mathbf{A}^H\mathbf{A})}. \tag{2.7}$$

Note that, for any matrix $\mathbf{A}$, the product $\mathbf{A}^H\mathbf{A}$ is hermitian (symmetric) and positive semi-definite, i.e. all eigenvalues are real and non-negative.

**Vector norms** In the following, we will want to measure errors. Assume the exact solution is some vector $\boldsymbol{y}$, and $\tilde{\boldsymbol{y}}$ is the (approximate) solution vector computed by some numerical algorithm. Then, we need to quantify the "size" of the error vector

$$\boldsymbol{e} = \tilde{\boldsymbol{y}} - \boldsymbol{y}. \tag{2.8}$$

In theory, any *vector norm* can be used to measure the error,

$$e = \|\boldsymbol{e}\| = \|\tilde{\boldsymbol{y}} - \boldsymbol{y}\|. \tag{2.9}$$

Recall the essential properties any norm must satisfy such that we can call it a norm

- *positive definiteness* $\|\boldsymbol{a}\| \geq 0$ and $\|\boldsymbol{a}\| = 0$ implies $\boldsymbol{a} = \mathbf{0}$,
- *homogeneity* $\|\alpha\boldsymbol{a}\| = \alpha\|\boldsymbol{a}\|$ for $\alpha \geq 0$,
- *triangle inequality* $\|\boldsymbol{a} + \boldsymbol{b}\| \leq \|\boldsymbol{a}\| + \|\boldsymbol{b}\|$.

Additionally, recall the properties of an *inner product* $(\cdot, \cdot)$

- *conjugate symmetry* $(\boldsymbol{a}, \boldsymbol{b}) = \overline{(\boldsymbol{b}, \boldsymbol{a})}$,
- *linearity in the second component* $(\boldsymbol{a}, \lambda_1\boldsymbol{b}_1 + \lambda_2\boldsymbol{b}_2) = \lambda_1(\boldsymbol{a}, \boldsymbol{b}_1) + \lambda_2(\boldsymbol{a}, \boldsymbol{b}_2)$
- *positive definiteness* $(\boldsymbol{a}, \boldsymbol{a}) > 0$ for $\boldsymbol{a} \neq \mathbf{0}$.

Different norms are appropriate to indicate different error measures, for example

- the *Euclidean norm* gives the (geometric) length of a vector,

$$\|\boldsymbol{e}\|_2 = \sqrt{\sum_{i=0}^{n-1} |e_i|^2}. \tag{2.10}$$

  The Euclidean norm is induced by the (real or complex) Euclidean inner product,

$$(\boldsymbol{a}, \boldsymbol{b})_2 = \boldsymbol{a}^H\boldsymbol{b} = \overline{\boldsymbol{a}}^T\boldsymbol{b}, \qquad \|\boldsymbol{e}\|_2 = \sqrt{(\boldsymbol{e}, \boldsymbol{e})_2}. \tag{2.11}$$

  As Euclidean norm and inner product are most common, we will use $\|\cdot\| \equiv \|\cdot\|_2$ and $(\cdot, \cdot) \equiv (\cdot, \cdot)_2$ equivalently,

- the maximum absolute error of all components is given by the *maximum norm*,

$$\|\boldsymbol{e}\|_\infty = \max_{i=0\ldots n-1} |e_i|. \tag{2.12}$$

- the sum of all absolute error components is given by

$$\|\boldsymbol{e}\|_1 = \sum_{i=0}^{n-1} |e_i|. \tag{2.13}$$

These norms are special cases of the $\ell_p$ norm

$$\|\boldsymbol{e}\|_p = \left( \sum_{i=0}^{n-1} |e_i|^p \right)^{1/p}. \tag{2.14}$$

Sometimes, it is useful to scale vector entries differently, e.g. when they correspond to different physical quantities. Given scaling factors $\rho_i > 0$, a valid norm is

$$\|\boldsymbol{e}\|_\rho = \sqrt{\sum_{i=0}^{n-1} \rho_i |e_i|^2}. \tag{2.15}$$

Another important case is that of *matrix-induced norms*: let $\mathbf{M}$ be a *hermitian/symmetric positive definite* matrix, then $\|\cdot\|_{\mathbf{M}}$ defined as below is indeed a norm,

$$\|\boldsymbol{e}\|_{\mathbf{M}} = \left(\boldsymbol{e}^H \mathbf{M} \boldsymbol{e}\right)^{1/2}. \tag{2.16}$$

Indeed, one can also define a matrix-induced inner product,

$$(\boldsymbol{a}, \boldsymbol{b})_{\mathbf{M}} := \boldsymbol{a}^H \mathbf{M} \boldsymbol{b} \tag{2.17}$$

One can check easily that $(\cdot,\cdot)_{\mathbf{M}}$ is an inner product verifying conjugate symmetry, linearity and definiteness.

The scaled norm $\|\cdot\|_\rho$ is a special case of the matrix induced norms, where $\mathbf{M}$ is the diagonal matrix $\mathrm{diag}(\rho_i)$. Other use-cases of matrix-induced norms will be met in context of the finite element method: there, one obtains the *energy-norm*, if $\mathbf{M}$ is chosen as the stiffness matrix, or the $L^2$ norm if $\mathbf{M}$ is chosen as the mass matrix.

**Matrix norms** For stability estimates, specifically of linear problems, we also need matrix norms. Naturally, any matrix norm must satisfy the defining properties of a norm, i.e. be positive definite, homogeneous and satisfy the triangle inequality. However, there are additional features one may need from a matrix norm. Also, when using both matrix and vector norms, they need to match in some sense. Let $\|\cdot\|_V$ denote the vector norm, and $\|\cdot\|_M$ the matrix norm. Then,

- $\|\cdot\|_V$ and $\|\cdot\|_M$ are *compatible* (or *consistent*) if for all vectors $\boldsymbol{x}$ and matrices $\mathbf{A}$

$$\|\mathbf{A}\boldsymbol{x}\|_V \leq \|\mathbf{A}\|_M \|\boldsymbol{x}\|_V, \tag{2.18}$$

- a matrix norm is *sub-multiplicative* if for all $\mathbf{A}$, $\mathbf{B}$ we have $\|\mathbf{AB}\|_M \leq \|\mathbf{A}\|_M \|\mathbf{B}\|_M$.

For a given vector norm, a compatible matrix norm can be defined. It is called *induced by the vector norm*, and defined by

$$\|\mathbf{A}\|_M := \sup_{\boldsymbol{x}\neq\mathbf{0}} \frac{\|\mathbf{A}\boldsymbol{x}\|_V}{\|\boldsymbol{x}\|_V}. \tag{2.19}$$

We list some common matrix norms that are compatible with certain vector norms:

- *the Frobenius norm* is sub-multiplicative but not compatible to the standard vector norms. It is easy to compute,

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} |A_{ij}|^2} \tag{2.20}$$

- *the spectral norm* is induced by the $\ell_2$ norm. It is given as the largest singular value of $\mathbf{A}$, (or the square root of the largest eigenvalue of $\mathbf{A}^H\mathbf{A}$),

$$\|\mathbf{A}\|_2 = \sigma_{max}(\mathbf{A}) = \sqrt{\lambda_{max}(\mathbf{A}^H\mathbf{A})}. \tag{2.21}$$

- *the maximum norm* is indeed a norm, but not sub-multiplicative,

$$\|\mathbf{A}\|_{max} = \max_{i,j} |A_{ij}|. \tag{2.22}$$

- *the row-sum norm* is induced by the $\ell_\infty$ norm,

$$\|\mathbf{A}\|_\infty = \max_i \sum_j |A_{ij}|. \tag{2.23}$$

- *the column-sum norm* is induced by the $\ell_1$ norm,

$$\|\mathbf{A}\|_1 = \max_j \sum_i |A_{ij}|. \tag{2.24}$$

## 2.1 Accuracy and stability

The notions of accuracy and stability will be covered in different aspects during this course. *Accuracy* and *stability* issues include

- at least when using C arithmetics or numerical software products such as python/numpy, computers do not work with *real* numbers (i.e. $\frac{1}{3}$, $\sqrt{2}$ or $\pi$) but with floating point approximations, cutting of after a finite amount of information,
- iterative solution methods do not provide the exact (or floating-point exact) solution to some problem, but *converge* towards the solution, where we might or might not know about the size of the error.

- given erroneous input data, can we estimate to what amount the solution is changed?

Throughout the following, an *algorithm* is a set of rules that transforms some *input data* collected in the *input vector* $\boldsymbol{x}$ to the *output vector* collected in the *output vector* $\boldsymbol{y}$. Note that $\boldsymbol{x}$ and $\boldsymbol{y}$ may consist of a single scalar quantity, or also of several vectors (ordered consecutively to form $\boldsymbol{x}$ or $\boldsymbol{y}$). For given input data, the output shall be unique, which makes the algorithm a *function* mapping input to output,

$$\Phi : \boldsymbol{x} \mapsto \boldsymbol{y}. \tag{2.25}$$

We will analyze the size of the error of an algorithm, where the error is the difference between the *exact* and the *computed* solution. Different sources of error accumulate, where we usually have

- errors due to noisy input data,
- errors due to roundoff effects,
- errors due to the algorithm itself.

*Errors due to data noise* can be classified by *stability* properties of the problem itself, and will be treated in Section 2.1.2. This error part is due to the problem, and not due to the specific algorithm for solving it.

*Roundoff errors* are due to limited floating-point accuracy in all computations within the algorithm. This error part is different for different algorithms solving the same problem. One speaks of *stability of the algorithm*.

*Algorithm-inherent errors* appear e.g. when a solution is computed iteratively, e.g. when a nonlinear equation is solved by Netwon's method. Depending on the number of iterations that are conducted, on given error criteria, etc, the computed solution differs from the exact solution. Convergence estimates will allow to bound the inherent error for different algorithms, these estimates will be discussed together with the actual algorithm.

### 2.1.1 Floating point representation

While integer representations are exact, floating point (or `double`) numbers are cut off after finitely many bits. As this may affect computational methods discussed at a later point, we discuss different internal representations of "real" numbers. Internally, each `double` floating point number is stored using 64 bit. The representation of some `double d` includes three parts,

$$\texttt{d} = S \times M \times b^{E-e}, \qquad \text{with} \begin{cases} S & \text{the sign} & 1 \text{ bit} \\ M & \text{the mantissa} & 52 \text{ bit} \\ E & \text{the exponent} & 11 \text{ bit.} \end{cases} \tag{2.26}$$

Above, $b = 2$ is the *basis* and $e = 1023$ is the *bias of the exponent*.

The sign bit states whether the number is positive ($S = 0$) or negative ($S = 1$).

The length of the mantissa $M$ specifies the accuracy of the floating point arithmetics. For 52 bit of mantissa, we get $2^{52} \simeq 4.5 \times 10^{15}$, which corresponds to the maximum of 16 relevant (decimal) digits, see also the end of this section.

Mantissa and exponent are not unique. For some representation, one can multiply $M$ by two and decrease $E$ by one, and get the same number. In standard architectures, the mantissa is stored in such a way that its leading bit is one, i.e. $M \in [1, 2)$. Then, the leading bit needs not be stored, but only the *faction* $F = M - 1$ is stored. Thus one additional bit of accuracy is gained.

Why is the bias of the exponent $e = 1023$? If an integer exponent is represented by an 11 bit binary number, it is in between 0 and $2^{11} - 1 = 2047$. Thus, the shift $e$ is chosen such that exponents are symmetric w.r.t. zero, and $b^{E-e}$ lies in $[2^{-1023}, 2^{1024}] \simeq [10^{-308}, 10^{308}]$.

Numbers between (approximately) $\pm 10^{-308}$ and zero (with the exception of zero itself) will result in an underflow, while numbers higher than (approximately) $10^{308}$ result in an overflow.

An example: how are floating point numbers 1.0 and $-6.5$ stored? We represent both as `d` $= S \times M \times 2^{E-e}$,

$$\texttt{d} = S \times M \times 2^{E-e}, \qquad 1.0 = 1 \times 1 \times 2^0, \qquad -6.5 = -1 \times 1.625 \times 2^2. \tag{2.27}$$

In both cases, the mantissa satisfies $1 \leq M < 2$. The mantissas are represented in binary form:

$$1 = 1_2, \qquad 1.625 = 1 + \tfrac{1}{2} + \tfrac{1}{8} = 1.101_2. \tag{2.28}$$

The exponents are represented by

$$0 = 1023 - 1023 = E - e, \qquad E = 1023 = 1111111111_2, \tag{2.29}$$
$$2 = 1025 - 1023 = E - e, \qquad E = 1025 = 10000000001_2. \tag{2.30}$$

Thus we find the internal representations

$$\begin{array}{cccccccc} s & e_0 \dots e_{10} & f_0 \dots f_{51} & = & (-1)^s \times & 2^{m-1023} \times & 1.f_0 f_1 \dots f_{51}, & \\ 0 & 01111111111 & 0000\,0 \dots 0 & = & 1 \times & 2^{1023-1023} \times & 1.0000_2 & = 1, \\ 1 & 10000000001 & 1010\,0 \dots 0 & = & -1 \times & 2^{1025-1023} \times & 1.1010_2 & = -6.5, \end{array} \tag{2.31}$$

A small workaround lets us print the internal representation of the 64bit `double` numbers. The code snippet below *reinterprets* the 8 byte of `double d` as 8 one-byte `unsigned char`. These one-byte characters each represent a number between 0 and 255, which can be printed in hexadecimal format, i.e. using two hex numbers using `%02x`. There, `02` stands for exactly two digits and `x` means hexadecimal. We print the internal represenation of `double d = -6.5;` in hexadecimal notation via

```
double d = -6.5;// 64 bit double
// reinterpret as 8 1bit char
unsigned char *c = reinterpret_cast<unsigned char*>(&d);
for (int i=0; i<8; i++)
    printf("%02x ", c[i]);
```

We expect

$$\underbrace{1100}_{4+8=c}\ \underbrace{0000}_{0}\ \underbrace{0001}_{1}\ \underbrace{1010}_{2+8=a}\ \underbrace{0\ldots0}_{12\times0} \qquad c0\,1a\,00\,00\,00\,00\,00\,00. \tag{2.32}$$

If your processor is little-endian, you will get exactly the representation above. Otherwise, if it is big-endian, you get exactly the bit-wise reverse order $00\,00\,00\,00\,00\,00\,1a\,c0$.

**Floating point accuracy** By floating point accuracy we define the smallest number that, if added to 1.0, yields a different result. The internal representation of 1.0 and $1.0+\epsilon_m$ is put below, we see $\epsilon_m = 2^{-52} \simeq 2.22\times10^{-16}$.

$$\begin{array}{cccccccl} s & e_0\ldots e_{10} & f_0\ldots f_{51} & = & (-1)^s\times & 2^{m-1023}\times & 1.f_0f_1\ldots f_{51}, & \\ 0 & 01111111111 & 000\ldots0000 & = & 1\times & 2^{1023-1023}\times & 1.0\ldots00_2 & = 1.0, \\ 0 & 01111111111 & 0000\ldots0001 & = & 1\times & 2^{1023-1023}\times & 1.0\ldots01_2 & = 1+2^{-52} = 1+\epsilon_m, \end{array} \tag{2.33}$$

Floating point accuracy $\epsilon_m$ is a bound for the maximum overall accuracy of all numerical methods discussed in this course. We will see that, however, the accuracy of an algorithm can be much less not only due to repeated roundoff errors, but also due to lacking numerical stability.

### 2.1.2 Data stability

In this section, we quantify the error due to noisy input data. If small errors in the input result in small errors in the output, the problem is data-stable; thus, we also speak of stability analysis. We assume that the algorithm itself is exact, such that for exact input data $\boldsymbol{x}$ it returns the exact solution $\Phi(\boldsymbol{x}) = \boldsymbol{y}$. Let now the input data be perturbed, $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \Delta\boldsymbol{x}$, which results in an output $\tilde{\boldsymbol{y}} = \Phi(\tilde{\boldsymbol{x}})$. The error is given by

$$\boldsymbol{e} = \Phi(\boldsymbol{x}+\Delta\boldsymbol{x}) - \Phi(\boldsymbol{x}). \tag{2.34}$$

When using component notation and approximating the difference according to Taylor, we get

$$e_i = \Phi_i(\boldsymbol{x}+\Delta\boldsymbol{x}) - \Phi_i(\boldsymbol{x}) \simeq \sum_{j=0}^{n-1} \frac{\partial\Phi_i}{\partial x_j}(\boldsymbol{x})\,\Delta x_j. \tag{2.35}$$

For the relative error (if $\Phi_i(\boldsymbol{x}) = y_i \neq 0$), we see

$$\frac{e_i}{y_i} \simeq \sum_{j=0}^{n-1} \underbrace{\frac{\partial\Phi_i}{\partial x_j}(\boldsymbol{x})\frac{x_j}{\Phi_i(\boldsymbol{x})}}_{k_{ij}} \frac{\Delta x_j}{x_j} \tag{2.36}$$

$$\tag{2.37}$$

The relative input error in component $j$, $\Delta x_j/x_j$, is amplified by factors $k_{ij}$. The $k_{ij}$ are called *condition numbers* of the problem. Small condition numbers $k_{ij} \simeq 1$ mean well-conditioned problems. Ill-conditioned problems are characterized through condition numbers $k_{ij} >> 1$. When solving an

ill-conditioned problem, small relative errors in the input can (and mostly will) result in large output errors, independent of the chosen numerical algorithm.

**Example** As a very simple example of data stabilty we analyze the addition operation,

$$\Phi(x_0, x_1) = x_0 + x_1 = y. \tag{2.38}$$

The relative error is given approximately according to (2.36)

$$\frac{e}{y} = \sum_{j=0}^{1} \frac{\partial \Phi}{\partial x_j}(\boldsymbol{x}) \frac{x_j}{\Phi(\boldsymbol{x})} \frac{\Delta x_j}{x_j} = 1 \frac{x_0}{x_0 + x_1} \frac{\Delta x_0}{x_0} + 1 \frac{x_1}{x_0 + x_1} \frac{\Delta x_1}{x_1}. \tag{2.39}$$

We have two condition numbers, $k_0 = \frac{x_0}{x_0+x_1}$ and $k_1 = \frac{x_1}{x_0+x_1}$. These numbers can grow arbitrarily large for $x_0 \simeq -x_1$. This effect is known as cancellation (Auslöschung). Small roundoff errors in the input data lead to large relative errors in the result, although the addition operation of the rounded numbers in itself is exact.

Advanced examples of ill-conditioned problems are numeric differentiation, parameter estimation in partial differential equations, and the backwards-in-time solution of diffusion processes (relevant in image reconstruction).

**Stability of solving linear problems – condition number** Solving linear problems form a large and highly relevant sub-class of numeric problems, therefore we consider stability of linear problems separately. Such a linear system is defined by a system matrix $\mathbf{A}$, such that the inverse $\mathbf{A}^{-1}$ defines the solution algorithm,

$$\mathbf{A}\,\boldsymbol{y} = \boldsymbol{x}, \qquad \Phi(\boldsymbol{x}) = \mathbf{A}^{-1}\,\boldsymbol{x} = \boldsymbol{y}. \tag{2.40}$$

Note that, in this section, $\boldsymbol{x}$ is the input vector, which is the right hand side of the linear system (and not its solution, which is the usual notation).

Due to linearity of the problem, an error in the input data $\Delta\boldsymbol{x}$ results in an output error that depends linearly on the input error,

$$\boldsymbol{e} = \mathbf{A}^{-1}\,\Delta\boldsymbol{x}. \tag{2.41}$$

Let now $\|\cdot\|_V$ be a vector norm of interest, and $\|\cdot\|_M$ a compatible matrix norm (i.e., if we consider the $\ell_2$ norm, the matrix norm is the spectral norm).

We can estimate the norm of the error through the compatibility of the matrix norm,

$$\|\boldsymbol{e}\|_V = \|\mathbf{A}^{-1}\,\Delta\boldsymbol{x}\|_V \leq \|\mathbf{A}^{-1}\|_M \|\Delta\boldsymbol{x}\|_V = \|\mathbf{A}^{-1}\|_M \|\boldsymbol{x}\|_V \frac{\|\Delta\boldsymbol{x}\|_V}{\|\boldsymbol{x}\|_V}. \tag{2.42}$$

Now, we insert $\boldsymbol{x} = \mathbf{A}\boldsymbol{y}$, and again use the compatibility

$$\|\boldsymbol{e}\|_V \leq \|\mathbf{A}^{-1}\|_M \|\mathbf{A}\boldsymbol{y}\|_V \frac{\|\Delta\boldsymbol{x}\|_V}{\|\boldsymbol{x}\|_V} \leq \|\mathbf{A}^{-1}\|_M \|\mathbf{A}\|_M \|\boldsymbol{y}\|_V \frac{\|\Delta\boldsymbol{x}\|_V}{\|\boldsymbol{x}\|_V}. \tag{2.43}$$

Divide by $\|\boldsymbol{y}\|_V$, and we get an upper bound on the relative error in the output in terms of the relative error in the input,

$$\frac{\|\boldsymbol{e}\|_V}{\|\boldsymbol{y}\|_V} \leq \underbrace{\|\mathbf{A}^{-1}\|_M \|\mathbf{A}\|_M}_{=:\kappa(\mathbf{A})} \frac{\|\Delta\boldsymbol{x}\|_V}{\|\boldsymbol{x}\|_V}. \tag{2.44}$$

The factor $\kappa(\mathbf{A}) = \|\mathbf{A}^{-1}\|_M \|\mathbf{A}\|_M$ is called the *condition number* of the matrix. The condition number tells whether errors in the data is amplified when solving the linear system exactly. One can show (more difficult) that it also indicates the amplification of errors in the matrix $\mathbf{A}$ itself.

If we use the $\ell_2$ norm (the Euclidean norm) for the vectors, the induced matrix norm is the spectral norm, $\|\mathbf{A}\|_2 = \sigma_{max}(\mathbf{A}) = \sqrt{\lambda_{max}(\mathbf{A}^H\mathbf{A})}$. Then the condition number is the relation between largest and smallest singular value of $\mathbf{A}$,

$$\kappa(\mathbf{A}) = \frac{\sigma_{max}(\mathbf{A})}{\sigma_{min}(\mathbf{A})}. \tag{2.45}$$

For symmetric positive definite matrices, the singular values can be replaced by the Eigenvalues,

$$\kappa(\mathbf{A}) = \frac{\lambda_{max}(\mathbf{A})}{\lambda_{min}(\mathbf{A})}. \tag{2.46}$$

### 2.1.3 Numeric stability of an algorithm

In the following, we consider a different aspect of stability, namely the stability of an algorithm with respect to roundoff errors happening throughout the process. If one wants to analyze an algorithm, it is necessary to subdivide it into its simple sub-steps. Each sub-step is then analyzed for its data stability, where the input error consists of the output error estimate of the previous step and an additional round-off error.

We do not provide details here, stability analysis is e.g. discussed in [2]. Instead, we consider the following example of solving a 3 by 3 linear system,

$$\begin{bmatrix} \alpha & 8 & 1 \\ 3 & 7 & 5 \\ 8 & 5 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 9+\alpha \\ 15 \\ 14 \end{bmatrix}. \tag{2.47}$$

One easily sees that $x_0 = x_1 = x_2 = 1$ solves the system exactly. Moreover, the condition number of the matrix is, for $\alpha \in [0, 1]$, lower than 5, which means that the problem is not ill-conditioned in itself.

We solve the problem using Gaussian elimination. The Gauss algorithm is discussed in detail in Section 2.2.1, we assume the general procedure is known. For $\alpha = 1$, the standard algorithm yields step by step, using output precision 16 digits,

$$\left[\begin{array}{ccc|c} 1 & 8 & 1 & 10 \\ 3 & 7 & 5 & 15 \\ 8 & 5 & 1 & 14 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & 8 & 1 & 10 \\ 0 & -17 & 2 & -15 \\ 0 & -59 & -7 & -66 \end{array}\right] \to \tag{2.48}$$

$$\left[\begin{array}{ccc|c} 1 & 0 & 1.941176470588235 & 2.941176470588236 \\ 0 & 1 & -0.1176470588235294 & 0.8823529411764706 \\ 0 & 0 & -13.94117647058824 & -13.94117647058824 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0.9999999999999998 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{array}\right] \tag{2.49}$$

The (absolute and relative) error in the components of the right hand side is $2 \times 10^{-16}$, which is the expected floating point accuracy.

For $\alpha = 0$, the first pivot element is zero, lines need to be reordered. However, for $\alpha$ small, we can in principle do the same steps as above. For example, let $\alpha = 10^{-12}$. We get,

$$\left[\begin{array}{ccc|c} 0.000000000001 & 8 & 1 & 9.000000000001 \\ 3 & 7 & 5 & 15 \\ 8 & 5 & 1 & 14 \end{array}\right] \to \tag{2.50}$$

$$\left[\begin{array}{ccc|c} 1 & 8000000000000 & 1000000000000 & 9000000000001 \\ 0 & -23999999999993 & -2999999999995 & -26999999999988 \\ 0 & -63999999999995 & -7999999999999 & -71999999999994 \end{array}\right] \to \tag{2.51}$$

$$\left[\begin{array}{ccc|c} 1 & 0.0009765625 & 1.3751220703125 & 2.375 \\ 0 & 0.9999999999999999 & 0.1249999999998281 & 1.124999999999828 \\ 0 & -0.0078125 & -10.6259765625 & -10.625 \end{array}\right] \to \tag{2.52}$$

$$\left[\begin{array}{ccc|c} 1 & -3.446374414116339e-05 & 0 & 1.000004307968018 \\ 0 & 0.9999080966822902 & 0 & 1.000011487914714 \\ 0 & 0.0007352265416781545 & 1 & 0.9999080966822902 \end{array}\right] \tag{2.53}$$

The error in the components of the output vector is of size $10^{-5}$, which is 10 orders of magnitude higher than floating point accuracy! Roundoff errors in the second step are amplified in the subsequent addition (adding two very large numbers differing by only 1). However, if one interchanges lines 0 and 1, the same algorithm works fine and yields an error of size $10^{-16}$.

## 2.2 Solving linear systems of equations

In this section, we are concerned with solving regular systems of equations of moderate size, or inverting regular matrices. Thus, matrix $\mathbf{A}$ shall be a square $n \times n$ matrix that is invertible.

We discuss several approaches to solving linear systems. The first approach, discussed in Section 2.2.1, is Gaussian elimination. Given some right hand side vector $\boldsymbol{b}$, the solution $\boldsymbol{x}$ is computed directy. If later one wants to solve the same system with another right hand side $\boldsymbol{b}^*$, the whole procedure of the elimination process is repeated in computing $\boldsymbol{x}^*$.

Another possibility is the actual computation of the inverse matrix, which can be done as an extension of Gaussian elimination at essentially the same (or not more than double) cost. While this is often done for very small systems, computing the inverse has some severe drawbacks for large systems. We will see that, for a large sparse system of equations, the inverse matrix is fully populated. *For large sparse systems, computing the inverse is much more expensive than solving the linear system!*

Most popular is probably the last approach of matrix factorization. In matrix factorization, the matrix is split multiplicatively into two (or more) factors, where for each factor the inverse can be computed in a simple and efficient manner,

$$\mathbf{A} = \mathbf{BC}, \qquad \text{then} \qquad \mathbf{A}^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}. \tag{2.54}$$

Thus, applying the inverse of $\mathbf{A}$ means applying the inverses of the factors consecutively. These factors are e.g. triangular matrices, where applying the inverse means solving a staggered system of equations, or rotation matrices, where the inverse is the (conjugate) transpose of the matrix itself.

Prominent examples are the LU and QR decompositions, and also the singular value decomposition. For the LU decompositions, $\mathbf{A} = \mathbf{LU}$ with a lower left triangular matrix $\mathbf{L}$ and an upper right triangular matrix $\mathbf{U}$. For the QR decomposition, factors are an orthogonal matrix $\mathbf{Q}$ and an upper right triangular matrix $\mathbf{R}$. A special case of the LU factorization is the Cholesky factorization for symmetric positive definite matrices with $\mathbf{A} = \mathbf{LL}^T$. The singular value decomposition (SVD) is closely linked to eigenvalue computations. It provides an ideal estimate over stability properties of the matrix. We will discuss its properties and applications in Section 2.2.6.

### 2.2.1 Gaussian elimination

We treat Gaussian elimination (also known as Gauss-Jordan elimination) as a first method, as it is well-known and rather straightforward. Using Gaussian elimination, an equation with right hand side $\boldsymbol{b}$, or also several equations can be solved at once. However, all $k$ right hand sides have to be known at the same time, and are collected in a column matrix $\mathbf{B} = [\boldsymbol{b}_0\ \boldsymbol{b}_1\ \ldots\ \boldsymbol{b}_k]$. The solution vectors are analogously collected in $\mathbf{X} = [\boldsymbol{x}_0\ \boldsymbol{x}_1\ \ldots\ \boldsymbol{x}_k]$, such that the system reads

$$\mathbf{AX} = \mathbf{B}. \tag{2.55}$$

When using the identity matrix $\mathbf{I}$ as right hand side, the result matrix $\mathbf{X}$ contains the matrix inverse, $\mathbf{X} = \mathbf{A}^{-1}$. Thus, Gaussian elimination can also be used to compute the matrix inverse.

Gaussian elimination can be applied for all types of regular matrices.

**The algorithm** Gaussian elimination relies on the fact that, in a system of equations, each equation (i.e. each row of the matrix $\mathbf{A}$ and the right hand side $\mathbf{B}$) can be replaced by a linear combination of itself and any other rows.

The general idea of Gaussian elimination is the following: in the first step, the first row is divided by $A_{00}$, such that the new diagonal entry is one. From all other rows $1 \leq i < n$, the $A_{i0}$ multiple of the

new first row is subtracted, such that the new $A_{i0}$ entries are equal to zero. The same manipulations have to be exacted for the right hand side.

In the next step, we proceed to the second line (with index $i = 1$). The same procedure is applied, i.e. all entries of the current line are divided by $A_{11}$, and from all other lines (including the first line above) an according multiple $A_{i1}$ is subtracted. Note that the new "one" diagonal entry of the first line is not corrupted by this procedure.

Once all lines are treated, the matrix $\mathbf{A}$ has transformed into the unit matrix, and $\boldsymbol{b}$ contains the solution $\mathbf{x}$. The kernel of the elimination loop (overwriting $\mathbf{a}$ and $\boldsymbol{b}$) can be implemented as below

```
for (int i = 0; i < height; i++)
{
    if (a(i, i) == 0.0)
        throw("Zero pivot element");

    pivinv = 1.0 / a(i,i);
    // multiply line i by 1/a(i,i)
    for (int l=0; l<height; l++)
    {
        a(i,l) *= pivinv;
    }
    b(i) *= pivinv;

    // subtract multiple of line i from line ll
    for (int ll = 0; ll < height; ll++)
        if (ll != i)
        {
            fac = a(ll,i);
            for (int l = 0; l < height; l++)
                a(ll,l) -= a(i,l) * fac;
            b(ll) -= fac * b(i);
        }
}
```

**Pivoting** Obviously, the above procedure will crash if any of the diagonal elements $A_{ii}$ is zero on division. This can of course happen for perfectly regular matrices, as e.g.

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{2.56}$$

To remedy this breakdown, a pivoting strategy can be employed. When pivoting, it is used that any two rows can be interchanged without changing the system solution. Additionally, two colums can be interchanged, which means that the respective solution degrees of freedom, rows of $\mathbf{X}$ must be changed also.

We refer to *partial pivoting* when interchanging rows only. In this case, in the first step of the Gauss elimination process, we do not automatically divide by the $A_{00}$ entry, but choose one of all entries

column 0, $A_{j0}$. There is no definite answer on which element to pick best. Choosing the row with $A_{j0}$ of maximal absolute value is generally considered a good strategy, although it may depend on the relative scaling of equations. This pivoting strategy requires a bit of additional book-keeping, as we have to store which rows have already been treated (namely the pivot choices) and which are left to be treated, where the next pivot element can be taken from.

*Full pivoting* means that not only elements of the corresponding row are considered as possible pivot elements, but all elements from the remaining part of the matrix. This means that additionally a switching of columns is performed, which induces a little more book-keeping of pivot indices.

**Implementation** Two different variants of Gaussian elminiation are implemented in `SCGaussSolver.h`:

- `SimpleGaussSolver<T>` is a rather straightforward implementation of the elimination procedure. No pivoting strategy is implemented. On initialization, a reference `const Matrix<T> &mat` to matrix $\mathbf{A}$ is stored. On `Apply`, the elimination procedure is carried out, using a temporary matrix object where the data from `mat` is copied to.

- `GaussSolver<T>` uses a different approach. On initialization, the inverse of $\mathbf{A}$ is computed and stored as a class member `Matrix<T> invmat`. On `Apply`, multiplication with `invmat` is carried out. This approach enables instant implementations of `ApplyT` and `ApplyH`, and can also be used to compute the inverse of a given matrix. The computational complexity is about twice the complexity of the simple Gauss solver. In this implementation, a pivoting strategy is used.

**Operation count and memory requirements** Gaussian elimination for a single input vector results in $O(n^3)$ operations, which is of the same order as matrix factorization methods discussed in the next section, but still slower by a factor three. Additionally, if one needs to solve for a second right hand side at some later time, the algorithm has to be re-started all over, while for the factorization methods matrix factorizations can be stored and reused, thereby largely reducing computational time.

When one wants to compute the matrix inverse explicitely, Gaussian elimination is as good as any other of the factorization methods detailed below. Matrix factorization can be achieved within $O(n^3)$ operations.

Concerning memory requirements, Gaussian elimination can be performed *in-place*, meaning it can be done using the memory allocated for storing matrix $\mathbf{A}$ and right hand side $\mathbf{B}$, such that $\mathbf{B}$ contains the solution $\mathbf{X}$ on exit. This implies, of course, that the original input matrix and right hand side are destroyed. Thus, it may be feasible to generate a copy of matrix and vector, which is then passed to the routine, or to generate these copies within the solver function. Also when computing the inverse $\mathbf{A}^{-1}$, an in-place conversion is possible, where $\mathbf{A}$ is replaced by $\mathbf{A}^{-1}$. Only integer vectors for book-keeping of pivot elements need to be stored additionally.

**Numerical stability** Without pivoting, Gaussian elimination is not stable, independent of the conditioning of the matrix (see the introductory example). This means, even for well-conditioned matrices, small input errors (e.g. roundoff errors due to floating point accuracy) may result in arbitrarily large errors in the solution. Thus, Gaussian elimination without pivoting should only be used if additional knowledge on the matrix structure ensures that no very small pivot elements can occur during elimination.

With full or partial pivoting, always choosing the pivot element of largest absolute value, Gaussian elimination is stable, where full pivoting is even more desirable.

In the example provided in `gaussian_stability.cpp`, we test the behavior of Gaussian elimination with and without pivoting for two test cases. The first test case is a symmetric positive definite matrix that arises – in similar structure – e.g. in the discretization of one-dimensional second order boundary value problems with finite differences, see *Numerik und Optimierung*. The condition number of this matrix grows with the number of unknowns squared. The second test case is a matrix with random entries in the range $[0, 100]$, which is (with high probability) not ill-conditioned, but pivot elements can be expected to get very small in comparison to non-pivot elements (off-diagonal entries). Additionally, we can set the upper-leftmost entry $A_{00}$ to some small value, which will not affect the conditioning of the random matrix much, but leads to further instabilities when using Gaussian elimination without pivoting.

### 2.2.2 LU decomposition

The LU decomposition can be interpreted as a variant of Gauss-Jordan elimination, where the system of equations is simplified to triangular form (stored in $\mathbf{U}$), and then the result is computed through substitution. All the necessary transformations are stored in form of two triangular matrices. The original matrix $\mathbf{A}$ is the product of a lower left triangular matrix $\mathbf{L}$ and an upper right triangular matrix $\mathbf{U}$,

$$\begin{pmatrix} l_{00} & 0 & 0 \\ l_{i0} & l_{ii} & 0 \\ l_{n0} & l_{nj} & l_{nn} \end{pmatrix} \begin{pmatrix} u_{00} & u_{0j} & u_{0n} \\ 0 & u_{ii} & u_{in} \\ 0 & 0 & u_{nn} \end{pmatrix} = \mathbf{A}. \tag{2.57}$$

**Generating the factorization** The procedure to generate the $\mathbf{L}$ and $\mathbf{U}$ factors is quite similar to Gaussian elimination. On the diagonal, the matrix $\mathbf{L}$ shall have $l_{ii} = 1$ as entries. Then, the factorization has exactly the same number of unknown coefficients as the original matrix $\mathbf{A}$. We will see that the factorization can again be computed in-place, storing

$$\begin{pmatrix} u_{00} & u_{0j} & u_{0n} \\ l_{i0} & u_{ii} & u_{in} \\ l_{n0} & l_{nj} & u_{nn} \end{pmatrix}. \tag{2.58}$$

To compute the factors, we pose $N^2$ equations, one for each entry of $\mathbf{A}$,

$$A_{ij} = \sum_{k=0}^{n-1} l_{ik} u_{kj} = \sum_{k=0}^{\min(i,j)} l_{ik} u_{kj}. \tag{2.59}$$

Above, we used that $l_{i,k} = 0$ for $k > i$, (i.e. in the upper right part), and $u_{k,j} = 0$ for $k > j$ (i.e. in the lower left part). The number of unknowns is $n^2$ if the diagonal entries of $\mathbf{L}$ are chosen as one, $l_{ii} = 1$. Then, the system of equations above is a staggered system:

- in column $j$, if $i \leq j$:
$$A_{ij} = \sum_{k=0}^{i} l_{ik} u_{kj} = \underbrace{l_{ii}}_{=1} u_{ij} + \sum_{k=0}^{i-1} l_{ik} u_{kj},$$
we can compute $u_{ij}$ from $A_{ij}$ and all $l_{ik}$ and $u_{kj}$ for $k < i$.
- in column $j$, if $i > j$:
$$A_{ij} = \sum_{k=0}^{j} l_{ik} u_{kj} = l_{ij} u_{jj} + \sum_{k=0}^{j-1} l_{ik} u_{kj},$$
we can compute $l_{ij}$ from $A_{ij}$, $u_{jj}$ and all $l_{ik}$ and $u_{kj}$ for $k < j$.

The computation of the factors, without pivoting, is as follows

```
for j=0 to n-1:
    for i=0 to j:
        u(i,j) = a(i,j)
        for k=0 to i-1:
            u(i,j) -= l_(i,k)*u(k,j)
    for i=j+1 to n:
        l(i,j) = a(i,j)/u(j,j)
        for k=0 to j-1:
            l(i,j) -= l(i,k)*u(k,j)/u(j,j)
```

Reviewing carefully, we see that all computations can be done in-place as indicated in (2.58).

Without pivoting, computation of the factors is not necessarily possible, divisions by zero may occur. As for the Gaussian elimination, searching for the pivot element through all lines ensures stability of the algorithm.

**Application of the solver** Once the $\mathbf{L}$ and $\mathbf{U}$ factors are computed, the inverse matrix $\mathbf{A}^{-1}$ can be applied through forward and backward substitution, solving the two staggered systems for $\mathbf{U}$ and $\mathbf{L}$ consecutively,

$$\mathbf{L}\boldsymbol{y} = \boldsymbol{b}, \qquad \mathbf{U}\boldsymbol{x} = \boldsymbol{y}. \tag{2.60}$$

The intermediate vector $\boldsymbol{y}$ is computed through forward substitution (recall $l_{ii} = 1$),

$$y_0 = b_0, \qquad y_i = b_i - \sum_{j=0}^{i-1} l_{ij} y_j. \tag{2.61}$$

The solution is then computed through backward substitution,

$$x_{n-1} = y_{n-1}/u_{n-1,n-1}, \qquad x_i = \frac{1}{u_{ii}}\big(y_i - \sum_{j=i+1}^{n-1} u_{ij} x_j\big). \tag{2.62}$$

**Computation of matrix determinant** Computing the determinant of a matrix is (apart from very small matrices) not as simple as one might assume. Indeed, a straightforward implementation of the Leibnitz formula is of exponential complexity $O(n!)$!

Once the LU factorization is performed, the matrix determinant can be computed easily. We recall that $\det \mathbf{A} = \det \mathbf{L} \det \mathbf{U}$, and that the determinant of a triangular matrix equals the product of all diagonal entries, i.e.

$$\det \mathbf{L} = \Pi_{i=0}^{n-1} l_{ii} = 1, \qquad \det \mathbf{A} = \det \mathbf{U} = \Pi_{i=0}^{n-1} u_{ii}. \tag{2.63}$$

For factorizations with pivoting, we have to recall the number of times rows were interchanged. Each change of rows means a factor of $-1$ in the determinant!

**LU decomposition vs. computation of the inverse matrix** We have seen a method (Gauss-Jordan) to compute the matrix inverse, as well LU factorization. Which one is to prefer? This depends on the application. LU factorization is cheaper than computing the inverse (but of the same complexity $O(n^3)$). On application, multiplication with the inverse is slightly cheaper than solving two staggered systems (but of the same, smaller complexity $O(n^2)$). If the inverse is needed explicitely in matrix form, there is no way around computing it.

LU factorization is definitely preferred for large sparse systems, where most of the matrix entries equal zero. Such systems, mostly stemming from finite element, finite volume or finite difference schemes, can be decomposed into LU form quite efficiently. For a sparse system from a two-dimensional physical problem on a regular mesh, $O(n \log n)$ operations are necessary for efficient LU solvers, whereas for three-dimensional problems $O(n^2)$ operations are counted. This is not optimal (which would be $O(n)$), but perfectly satisfying for many applications. In comparison, computing the inverse in matrix form would result in a fully populated matrix and always neccesitate order of $O(n^3)$ operations. For most applications, even storing this inverse matrix would exceed the available memory by far.

### 2.2.3 Cholesky factorization for symmetric positive definite matrices

For positive definite hermitian/symmetric matrices $\mathbf{A}$ (i.e. $\mathbf{A} = \mathbf{A}^H$), the LU factorization can be altered in such a way that $\mathbf{A} = \mathbf{L}\mathbf{L}^H$. Then, the diagonal entries of $\mathbf{L}$ are no longer one, but contain the square root of the pivot element. Thus, it is necessary that the diagonal entry is positive throughout the factorization, which is the case for positive definite matrices. Thus, the Cholesky factorization is always possible for SPD matrices. If pivoting is considered, it is essential that rows *and* columns must be interchanged, such that the pivot element is found among the diagonal elements of $\mathbf{A}$.

We do not provide details on the factorization here. Note that, in making use of the conjugate symmetry of the system matrix $\mathbf{A}$, the number of operations is approximately half of the number of operations for the LU factorization. Also the necessary memory is cut down to one half.

### 2.2.4 QR decomposition – Householder method

When performing a QR decomposition, the matrix is factorized, yielding the product of an orthogonal matrix $\mathbf{Q}$ and an upper right triangular matrix $\mathbf{R}$. The algorithm is also knwon as *Householder method*, and is in similar form widely used also in eigenvalue computations. A main benefit of the method is its unconditional numerical stability: as $\mathbf{Q}$ is orthogonal (unitary), its condition number is one $\kappa(\mathbf{Q}) = 1$. Therefore, the condition number of the original matrix $\mathbf{A}$ and of the triangular matrix $\mathbf{R}$ are equal,

$$\kappa(\mathbf{A}) = \kappa(\mathbf{Q})\kappa(\mathbf{R}) = \kappa(\mathbf{R}). \tag{2.64}$$

Thus, solving for $\mathbf{R}$ will not introduce any stability drawbacks.

The orthogonal matrix $\mathbf{Q}$ is computed as a product of Householder matrices, which are each defined through a unit vector $\boldsymbol{w}_i$,

$$\mathbf{Q} = \mathbf{P}_1\mathbf{P}_2 \dots \mathbf{P}_{n-1}, \qquad \mathbf{P}_i = \mathbf{I} - 2\boldsymbol{w}_i\boldsymbol{w}_i^H. \tag{2.65}$$

In case of real numbers, all "hermitian" operations $(\cdot)^H = (\bar{\cdot})^T$ are replaced by "transpose" $(\cdot)^T$. Obviously, the Householder matrices $\mathbf{P}_i$ are symmetric/hermitian. One easily verifies that for a unit vector $\boldsymbol{w}$ with $\|\boldsymbol{w}\| = 1$, $\mathbf{P} = \mathbf{I} - 2\boldsymbol{w}\boldsymbol{w}^H$ is indeed orthogonal,

$$\mathbf{P}\mathbf{P}^H = (\mathbf{I} - 2\boldsymbol{w}\boldsymbol{w}^H)(\mathbf{I} - 2\boldsymbol{w}\boldsymbol{w}^H) = \mathbf{I} - 2(2\boldsymbol{w}\boldsymbol{w}^H) + 4\boldsymbol{w}\underbrace{\boldsymbol{w}^H\boldsymbol{w}}_{=1}\boldsymbol{w}^H = \mathbf{I}. \tag{2.66}$$

Therefore, we see $\mathbf{P}^{-1} = \mathbf{P}^H = \mathbf{P}$.

**Computing the factorization** We choose the first Householder matrix such that it eliminates the first column in $\mathbf{A}$, similar to

$$\mathbf{P}_1\mathbf{A} = \left(\begin{array}{c|ccc} * & * & \dots & * \\ 0 & & & \\ \vdots & & \mathbf{A}_1 & \\ 0 & & & \end{array}\right) \tag{2.67}$$

To this end, we denote by $\boldsymbol{a}_0$ the first column of $\mathbf{A}$, and $\boldsymbol{e}_0$ the first unit vector. Choose

$$\boldsymbol{v}_1 = \boldsymbol{a}_0 + \frac{a_{00}}{|a_{00}|}\|\boldsymbol{a}_0\|\boldsymbol{e}_0, \qquad \boldsymbol{w}_1 = \frac{\boldsymbol{v}_1}{\|\boldsymbol{v}_1\|}. \tag{2.68}$$

Note that the fraction $a_{00}/|a_{00}|$ equals the signum operation for real numbers and the complex unit director for complex numbers. For small $a_{00}$, the fraction is still of size one (and not arbitrarily large). In case of $a_{00} = 0$ exactly, we can replace e.g. $a_{00}/|a_{00}|$ by 1.

We see that the norm of $\boldsymbol{v}_1$ computes as

$$\|\boldsymbol{v}_1\|^2 = 2\|\boldsymbol{a}_0\|^2 + 2\|\boldsymbol{a}_0\||a_{00}|. \tag{2.69}$$

Then, one easily computes

$$\mathbf{P}_1\boldsymbol{a}_0 = (\mathbf{I} + 2\boldsymbol{w}_1\boldsymbol{w}_1^H)\boldsymbol{a}_0 = \boldsymbol{a}_0 - \frac{2}{\|\boldsymbol{v}_1\|^2}\boldsymbol{v}_1(\boldsymbol{a}_0 - \frac{a_{00}}{|a_{00}|}\|\boldsymbol{a}_0\|\boldsymbol{e}_0)^H\boldsymbol{a}_0 \tag{2.70}$$

$$= \boldsymbol{a}_0 - \frac{1}{\|\boldsymbol{a}_0\|^2 + \|\boldsymbol{a}_0\||a_{00}|}(\|\boldsymbol{a}_0\|^2 + \|\boldsymbol{a}_0\||a_{00}|)\boldsymbol{v}_1 \tag{2.71}$$

$$= \boldsymbol{a}_0 - \boldsymbol{v}_1 = -\frac{a_{00}}{|a_{00}|}\|\boldsymbol{a}_0\|\boldsymbol{e}_0. \tag{2.72}$$

Thus, the first column of $\mathbf{P}_1\mathbf{A}$ contains a multiple of the first unit vector, $-\frac{a_{00}}{|a_{00}|}\|\boldsymbol{a}_0\|\boldsymbol{e}_0$, and we get the structure as in (2.67). Once $\mathbf{A}_1$ is computed, the same procedure can be applied to this submatrix, choosing $\mathbf{P}_2$ of the form

$$\mathbf{P}_2 = \left(\begin{array}{c|ccc} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & \tilde{\mathbf{P}}_2 & \\ 0 & & & \end{array}\right), \quad \text{with} \quad \tilde{\mathbf{P}}_2 = \mathbf{I}_{n-1} - 2\boldsymbol{w}_2\boldsymbol{w}_2^H, \quad \boldsymbol{w}_2 = \frac{\boldsymbol{v}_2}{\|\boldsymbol{v}_2\|}, \quad \boldsymbol{v}_2 \text{ according to } \mathbf{A}_1. \tag{2.73}$$

Iteratively, the orthogonal tensor $\mathbf{Q}$ is build in $n-1$ steps. It is not necessary to compute $\mathbf{Q}$ in matrix form, but it is sufficient to store the vectors $\boldsymbol{w}_i$ necessary for the application of $\mathbf{P}_i$ to some vector $\boldsymbol{y}$:

$$\mathbf{P}_i\boldsymbol{y} = (\mathbf{I} - 2\boldsymbol{w}_i\boldsymbol{w}_i^H)\boldsymbol{y} = \boldsymbol{y} - 2(\boldsymbol{w}_i, \boldsymbol{y})\boldsymbol{w}_i. \tag{2.74}$$

The upper right triangular matrix $\mathbf{R}$ is computed successively through transforming $\mathbf{A}$,

$$\mathbf{R} = \mathbf{P}_{n-1}\dots\mathbf{P}_1\mathbf{A}. \tag{2.75}$$

**Application of the inverse** When the factorization is applied, first $\mathbf{Q}^H = \mathbf{P}_{n-1}\dots\mathbf{P}_1$ is applied. To this end, only the vectors $\boldsymbol{w}_i$ are necessary; indeed, applying $\mathbf{I}-2\boldsymbol{w}\boldsymbol{w}^H$ is cheaper than full matrix multiplication as it involves only the computation of an inner product and vector addition, which can be done in order $O(n)$ operations instead of $O(n^2)$ (compare eq. 2.74).

In the second step, the staggered system $\mathbf{R}$ is solved via backward substitution.

**Computation of the determinant** As for the LU factorization, the QR factorization provides the determinant of a matrix in a simple manner. As $\det\mathbf{Q} = 1$, the determinant is given as the product of the diagonal entries of $\mathbf{R}$,

$$\det\mathbf{R} = \Pi_{i=0}^{n-1} r_{ii}. \tag{2.76}$$

### 2.2.5 QR for overdetermined systems

The general interpretation of overdetermined systems is "there are more equations than unknowns". So, if $\mathbf{A} \in \mathbb{C}^{m\times n}$ is an $m \times n$ matrix, we expect $m > n$. In this case, for most right hand sides there

is no solution to the equation $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. Overdetermined systems come up in least-squares problems, such as linear regression.

In the following, we further restrict $\mathbf{A}$ to be of full rank, which means the set of unknowns is minimal,

$$\mathbf{A} \in \mathbb{C}^{m\times n}, \qquad m > n, \qquad \operatorname{rank}(A) = n. \tag{2.77}$$

As $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ probably does not have a solution, we replace the equation by setting $\boldsymbol{x}$ such that the residual of the equation is minimized,

$$\frac{1}{2}\|\boldsymbol{b} - \mathbf{A}\boldsymbol{x}\|_2^2 \to \min. \tag{2.78}$$

Using the (complex or real) inner product $(\cdot,\cdot)$, this minimization problem reads

$$W(\boldsymbol{x}) = \frac{1}{2}(\boldsymbol{b} - \mathbf{A}\boldsymbol{x}, \boldsymbol{b} - \mathbf{A}\boldsymbol{x}) = \frac{1}{2}(\mathbf{A}^H\mathbf{A}\boldsymbol{x}, \boldsymbol{x}) - (\mathbf{A}^H\boldsymbol{b}, \boldsymbol{x}) + \frac{1}{2}(\boldsymbol{b}, \boldsymbol{b}) \to \min \tag{2.79}$$

This is a quadratic optimization problem, for real $\boldsymbol{x}$ the according sufficient and necessary condition is

$$\frac{\partial W}{\partial \boldsymbol{x}} = \mathbf{A}^H\mathbf{A}\boldsymbol{x} - \mathbf{A}^H\boldsymbol{b} = \mathbf{0}. \tag{2.80}$$

For complex matrices, the proof does not use complex differentiation, but differentiation with respect to real and imaginary part of $\boldsymbol{x}$ (Wirtinger derivatives), to the same effect. We call

$$\mathbf{A}^H\mathbf{A}\boldsymbol{x} = \mathbf{A}^H\boldsymbol{b} \tag{2.81}$$

*normal equation* for the overdetermined system.

Feeding the normal equation, with system matrix $\mathbf{A}^H\mathbf{A}$ (or $\mathbf{A}^T\mathbf{A}$ for real matrices) directly into some linear solver is not a good idea! The condition number of $\mathbf{A}^H\mathbf{A}$ is the square of the original condition number,

$$\kappa(\mathbf{A}^H\mathbf{A}) = \big(\kappa(\mathbf{A})\big)^2. \tag{2.82}$$

Numerical stability for the normal equation will be much worse than necessary.

Instead, consider the QR factorization of $\mathbf{A}$. If we review the QR algorithm step by step, computing the factors $\mathbf{Q}$ and $\mathbf{R}$ is perfectly possible for $\mathbf{A}$ being an $m \times n$ matrix. Given there are only $n < m$ columns, $n-1$ Householder vectors are computed, which yields a factorization

$$\mathbf{A} = \mathbf{Q}\mathbf{R}, \tag{2.83}$$

$$\mathbf{Q} = \mathbf{P}_1\mathbf{P}_2\ldots\mathbf{P}_n \text{ an } m \times m \text{ matrix computable from } n \text{ vectors}, \tag{2.84}$$

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix} \text{ an } m \times n \text{ matrix, with } \mathbf{R}_1 \text{ an } n \times n \text{ matrix}. \tag{2.85}$$

As $\mathbf{A}$ is of full rank, the triangular matrix $\mathbf{R}_1$ is also of full rank, all diagonal entries are different from zero.

Rewriting the normal equation leads to

$$\mathbf{R}^H \underbrace{\mathbf{Q}^H \mathbf{Q}}_{=\mathbf{I}} \mathbf{R}\boldsymbol{x} = \mathbf{R}^H \mathbf{Q}^H \boldsymbol{b}. \tag{2.86}$$

Let now the right hand side $\boldsymbol{d} := \mathbf{Q}^H \boldsymbol{b}$, then this equation reads

$$\begin{bmatrix} \mathbf{R}_1^H & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} \mathbf{R}_1^H & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \end{bmatrix}. \tag{2.87}$$

It is satisfied if the $n \times n$ system is solved,

$$\mathbf{R}_1 \boldsymbol{x} = \boldsymbol{d}_1. \tag{2.88}$$

The original problem $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ corresponds to the stronger condition

$$\begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \end{bmatrix}. \tag{2.89}$$

If $\boldsymbol{d}_2 = \mathbf{0}$, there exists indeed a unique solution (to $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$), and it can be computed fast by solving the triangular $n \times n$ system $\mathbf{R}_1 \boldsymbol{x} = \boldsymbol{d}_1$. Otherwise, the solution to $\mathbf{R}_1 \boldsymbol{x} = \boldsymbol{d}_1$ still solves the normal equation, and minimizes (2.78):

$$\mathbf{A}^H \mathbf{A}\boldsymbol{x} = \mathbf{R}^H \mathbf{R}\boldsymbol{x} = \begin{bmatrix} \mathbf{R}_1^H & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix} \boldsymbol{x} \tag{2.90}$$

$$= \mathbf{R}_1^H \mathbf{R}_1 \boldsymbol{x} = \mathbf{R}_1^H \boldsymbol{d}_1 = \begin{bmatrix} \mathbf{R}_1^H & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{d}_1 \\ * \end{bmatrix} = \mathbf{R}^H \mathbf{Q}^H \boldsymbol{b} = \mathbf{A}^H \boldsymbol{b}. \tag{2.91}$$

The norm of $\boldsymbol{d}_2$ corresponds to the norm of the residual of the original overdetermined system of equations:

$$\|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2 = \|\mathbf{Q}^H(\mathbf{A}\boldsymbol{x} - \boldsymbol{b})\|_2 = \|\mathbf{R}_1 \boldsymbol{x} - \boldsymbol{d}\|_2 = \|\boldsymbol{d}_2\|_2. \tag{2.92}$$

Above, we used that the unitary matrix $\mathbf{Q}$ does not change the Euclidean norm of a vector, and that $\mathbf{R}_1 \boldsymbol{x} = \boldsymbol{d}_1$.

**Example problem polynomial regression** As an example problem for linear overdetermined systems, we use the problem of polynomial (or linear) regression. Given a set of $m$ data points $(x_i, y_i)$, we want to find the "best" approximation by a polynomial function $f(x) = c_0 + c_1 x + \cdots + c_n x^n = \sum_{j=0}^{n} c_j x^j$ of given order $n$. If $f(x)$ is a linear function, we speak of linear regression.

We pose the overdetermined system of equations for the coefficients $c_j$,

$$f(x_i) = \sum_{j=0}^{n} c_j x_i^j = y_i \qquad \text{for } i = 0 \ldots m-1, j = 0 \ldots n. \tag{2.93}$$

The above system translates into matrix-vector form

$$\mathbf{A}\boldsymbol{c} = \boldsymbol{y}, \quad \mathbf{A} = \begin{bmatrix} 1 & x_0 & \dots & x_0^n \\ 1 & x_1 & \dots & x_1^n \\ 1 & x_{m-1} & \dots & x_{m-1}^n \end{bmatrix}, \quad \boldsymbol{c} = [c_0, c_1, \dots c_n]^T, \quad \boldsymbol{y} = [y_0, y_1, \dots y_{m-1}]^T. \tag{2.94}$$

In case the given data points fit the polynomial function exactly (e.g. they lie on a straight line), the remaining part $\boldsymbol{d}_2$ from $\boldsymbol{d} = \mathbf{Q}^H\boldsymbol{y}$ is exactly zero. Otherwise, the function $f(x) = \sum_{j=0}^{n} c_j x_i^j$ is the best possible choice in terms of the quadratic error

$$\|\mathbf{A}\boldsymbol{c} - \boldsymbol{y}\|_2^2 = \sum_{i=0}^{m-1} (f(x_i) - y_i)^2 \to \min. \tag{2.95}$$

### 2.2.6 Singular value decomposition (SVD)

The singular value decomposition is another factorization of a matrix into three easy-to-invert matrices. Differently from most of the factorizations above, it works directly also for non-square matrices and for singular matrices.

Let now $\mathbf{A}$ be an $m \times n$ matrix, then it can be decomposed in the form $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$ with

- $\mathbf{U} \in \mathbb{C}^{m\times m}$ unitary (orthogonal),
- $\mathbf{V} \in \mathbb{C}^{n\times n}$ unitary (orthogonal),
- $\mathbf{\Sigma} = \mathrm{diag}(\sigma_i)$ diagonal $m \times n$, with real-valued singular values $\sigma_i \geq 0, i = 0 \dots \min(m, n) - 1$. Singular values are usually ordered such that $\sigma_0 \geq \sigma_1 \geq \dots \geq \sigma_{\min(m,n)-1}$.

For a regular matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$, all singular values $\sigma_i$ are positive. If $\mathbf{A}$ is rectangular, but of full rank, all singular values are positive. In this case, the diagonal matrix $\Sigma$ is an $m \times n$ matrix, where $\min(m, n)$ singular values $\sigma_i$ are positioned at the diagonal $\Sigma_{ii}$.

If $\mathbf{A}$ is not of full rank, there are singular values equal to zero. Indeed, if $\mathrm{rank}(\mathbf{A}) = k \leq \min(m, n)$, then the last $\min(m, n) - k$ singular values are equal to zero.

**SVD and image** By the SVD, we get an orthonormal basis for the image of a matrix. The columns of $\mathbf{U}$ form a basis for $\mathbb{C}^m$ (even an orthonormal one). The range of $\mathbf{A}$ is a subspace:

| The range of $\mathbf{A}$ is the sub-space spanned by the column vectors of $\mathbf{U}$ that correspond to non-zero singular values, $\mathrm{im}(\mathbf{A}) = \mathrm{lin}(\{\boldsymbol{u}_j, j = 0 \dots \mathrm{rank}(\mathbf{A})\})$. |
|---|

Why? Let $\boldsymbol{x} \in \mathbb{R}^n$, the image of $\mathbf{A}$ consists of all vectors of the form $\mathbf{A}\boldsymbol{x}$. Let now $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H$ be the singular value decomposition of $\mathbf{A}$, and let $\boldsymbol{u}_i$ and $\boldsymbol{v}_j$ denote the columns of $\mathbf{U}$ and $\mathbf{V}$, respectively. Then,

$$\begin{aligned}\mathbf{A}\boldsymbol{x} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H\boldsymbol{x} &= \mathbf{U}\boldsymbol{\Sigma}\sum_{j=0}^{n-1}(\boldsymbol{v}_j, \boldsymbol{x})\boldsymbol{e}_j \\ &= \mathbf{U}\sum_{j=0}^{\min(m,n)-1}(\boldsymbol{v}_j, \boldsymbol{x})\sigma_j\,\boldsymbol{e}_j = \sum_{j=0}^{k-1}(\boldsymbol{v}_j, \boldsymbol{x})\sigma_j\mathbf{U}\boldsymbol{e}_j = \sum_{j=0}^{k-1}(\boldsymbol{v}_j, \boldsymbol{x})\sigma_j\boldsymbol{u}_j.\end{aligned} \tag{2.96}$$

Thus, $\mathbf{A}\boldsymbol{x}$ can be expressed in the basis $\boldsymbol{u}_j$, $j = 0 \ldots k-1$.

**SVD and nullspace** The SVD also generates an orthonormal basis of the nullspace of a matrix. The columns of $\mathbf{V}$ form a basis of $\mathbb{C}^n$, those columns associated to zero singular values generate $\ker(\mathbf{A})$.

The nullspace of $\mathbf{A}$ is the sub-space spanned by the column vectors of $\mathbf{V}$ that correspond to zero singular values or to indices $j \geq m$, $\ker(\mathbf{A}) = \mathrm{lin}(\{\boldsymbol{v}_j, j = \mathrm{rank}(\mathbf{A}) \ldots n\})$.

**Application of the inverse matrix** For a regular matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$, all singular values $\sigma_i$ are positive. The inverse of $\mathbf{A}$ is easily computed through

$$\mathbf{A}^{-1} = \mathbf{V}\boldsymbol{\Sigma}^{-1}\mathbf{U}^H, \tag{2.97}$$

where $\boldsymbol{\Sigma}^{-1} = \mathrm{diag}(1/\sigma_i)$ is the diagonal matrix containing the inverses of the singular values.

**Pseudo inverse** The SVD allows a uniform definition ofa pseudo inverse. Using this pseudo inverse, overdetermined, underdetermined and singular systems of equations can be solved alike. The pseudo inverse is defined as

$$\mathbf{A}^\dagger = \mathbf{V}\boldsymbol{\Sigma}^\dagger\mathbf{U}^H, \qquad \text{with } \boldsymbol{\Sigma}^\dagger = \mathrm{diag}(\sigma_i^\dagger),\ \sigma_i^\dagger = \begin{cases} 1/\sigma_i & \text{if } \sigma_i > 0, \\ 0 & \text{if } \sigma_i = 0. \end{cases}. \tag{2.98}$$

Using this pseudo inverse means

- for **overdetermined systems:** the computed solution $\boldsymbol{x} = \mathbf{A}^\dagger\boldsymbol{b}$ solves the normal equation, compare Section 2.2.5,
- for **underdetermined or singular systems:** the computed solution $\boldsymbol{x} = \mathbf{A}^\dagger\boldsymbol{b}$ solves $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, and it is orthogonal to the nullspace of $\mathbf{A}$.

Overdetermined systems have already been discussed to some length in Section 2.2.5. For underdetermined or singular systems, we have in general *an infinite number of solutions*. For any solution $\boldsymbol{x}$, and any element of the nullspace $\boldsymbol{x}_0 \in \ker(\mathbf{A})$, $\boldsymbol{x} + \lambda\boldsymbol{x}_0$ forms another valid solution. The singular

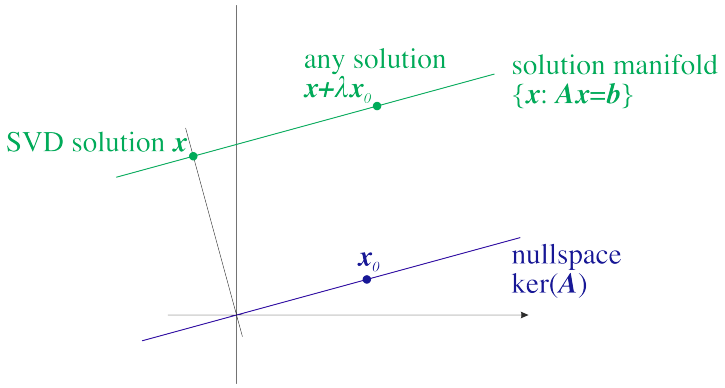


Figure 2.1: Orthogonality relation between SVD solution and nullspace of matrix $\mathbf{A}$

value decomposition, and the pseudo inverse given above, now choose the unique solution that has no component in nullspace direction:

$$(\mathbf{A}^\dagger \boldsymbol{b}, \boldsymbol{x}_0) = 0 \qquad \text{for all } \boldsymbol{x}_0 \in \ker(\mathbf{A}). \tag{2.99}$$

Graphically one can visualize this orthogonality relation as in Figure 2.1.

**Computing the SVD** Computing all singular values and vectors of a matrix is of the same overall computational complexity as other decompositions, namely $O(n^3)$. The algorithm relies on tridiagonalization by Householder's method, and is stable – we will not discuss it here but together with algorithms for eigenvalue computation. We try two ways of linking an external tool for computing the SVD. One is via interfacing to the Eigen lib, while the other works by linking the according LAPACK routine. The second method using LAPACK is computationally more efficient and requires less (only the essential amount of) copying data.

### 2.2.7 LAPACK

The *Linear Algebra Package (LAPACK)* is a standard library which includes efficient implementations of basic linear algebra routines (solving linear systems, matrix inversion, eigenvalue and singular value computations, ...). It was originally written in FORTRAN, but it can be linked against C and C++ code. It includes real and complex solvers for real and complex problems. LAPACK is strictly procedural, which means its interface consists of C (or FORTRAN) subroutines using `int*`, `float*` and `double*` pointers only.

**Subroutine names** Subroutines are named according to the following convention (see `https://en.wikipedia.org/wiki/LAPACK`):

- each C-type subroutine is of the form `pmmaaa_`
- p is a one-letter code denoting the type of numerical constants used. `s`, `d` stand for real floating-point arithmetic `float` and `double`, while `c` and `z` stand for complex arithmetic with `complex<float>` or `complex<double>`.

- `mm` is a two-letter code denoting the kind of matrix expected by the algorithm. The actual matrix data are stored in a different format depending on the specific kind of matrix. For example, `ge` indicates general matrix which is stored column-major, `sp` or `hp` indicate symmetric or hermitian packed storage (where only one half of the actual matrix is stored), `gb` indicates general band matrix and many more.
- `aaa` is a one- to three-letter code describing the actual algorithm implemented in the subroutine,
- the underscore is added in the C interface.

Examples:

- `dgesvd_` means (`d`) `double` arithmetics, (`ge`) input general matrix, (`svd`) implementing the singular value decomposition,
- `zhpsv_` means (`z`) `complex<double>` arithmetics, (`hp`) hermitian packed matrix storing only the upper right triangle, (`sv`) implementing a linear solve.

Note that LAPACK expects all matrices in column-major format, i.e. entries are stored column-wise! This is the standard ordering for FORTRAN arrays.

**Installing LAPACK** Different implementations of LAPACK can be found, the "official" LAPACK is from Netlib (`https://netlib.org/lapack/`). All implementations from this course were tried using one of the following two alternatives:

- Intel Math Kernel Library (MKL): the MKL includes not only LAPACK routines, but also sparse matrices and solvers (see next section). It can be included and linked comfortably. Probably the easiest way to install it is via python's pip:
  `> pip install mkl mkl-devel mkl-include mkl-static`
  Alternatively oneMKL can be downloaded from `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`. The Intel MKL is – up to my best knowledge – not available for Apple Silicon chips. It is available, but not at optimal performance, for AMD processors.
- Apple Accelerate includes tuned BLAS and LAPACK versions, and is accessed in the same way as other standard LAPACK implementations such as Netlib or libflame.

**Calling LAPACK routines from C++** As mentioned above, LAPACK is a FORTRAN library that can be linked against C or C++ code. No header file needs to be included, any LAPACK routine that is used in the code has to be *declared* as an external routine, e.g.

```
extern "C" void dgesv_(const int* n, const int* nrhs, double* a,
        const int* lda, int* ipiv, double* b, const int* ldb,
        int* info );
extern "C" void  zgesv_(const int* n, const int* nrhs,
        std::complex<double>* a,
        const int* lda, int* ipiv, std::complex<double>* b,
        const int* ldb, int* info);
```

The `const` specifiers may also be omitted (but no additional `const` may be added). As Fortran is case-insensitive, declaring `DGESV_` instead of `dgesv_` is equally valid. As C/C++ is case-sensitive, you must use the same variant in the code as you declared before.

LAPACK mostly does computations *in-place*, which means e.g. for the linear solve routines `dgesv` and `zgesv`, the data pointers `a` and `b` point to matrix and right hand side vector(s) on input. On output, the matrix `a` is replaced by the LU factorization and the right hand side vector `b` is replaced by the solution. Thus, it may be advisable to pass *copies* of matrix/vector to the LAPACK routines.

LAPACK expects general matrices (routines marked `*GE***`) as *column-major*. Thus it may be necessary to re-allocate matrix memory in column-major format, or use a routine for transposed matrices, or transpose the outcome (depending on the application).

For our linear algebra library, calling LAPACK linear solve for a complex linear system is done e.g. like

```
int linsolve_lapack(const Matrix<complex<double>>& a,
        const Vector<complex<double>>& b,
        Vector<complex<double>> &x)
{
    int n = a.Height();
    // make sure the copy is column-major
    Matrix<complex<double>> copya(n, n, TMatrixStorage::COL_MAJOR);
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            copya(i,j) = a(i,j);
    // on input, x = b
    x = b;

    int nrhs = 1;
    Vector<int> ipiv(n); // vector contains pivoting info
    int info;

    zgesv_(&n, &nrhs, &copya(0,0), &n, &ipiv(0), &x(0), &n, &info);

    // return LAPACK info
    return info;
}
```

**Linking LAPACK libraries** When *linking*, the correct library has to be linked. Simplest way to do so is to let cmake find and link LAPACK. To detect LAPACK and link it with project `MY_PROJECT_NAME`, add the following to CMakeLists.txt:

```
## detect LAPACK automatically
enable_language(CXX)
find_package(LAPACK)
# ..
```

```
## link libraries
target_link_libraries(MY_PROJECT_NAME ${LAPACK_LIBRARIES})
```

Alternatively, LAPACK libs can be linked manually specifying the full path to the library. We differ between *static* and *dynamic* linking.

*Static linking* uses static libraries with extension `*.a` on Linux/Macos or `*.lib` on Windows. The Linker *copies* the linked LAPACK routines into the executable. The size of the executable grows, but it can be distributed to systems where LAPACK is not pre-installed.

*Dynamic linking* uses libraries such as `*.so` on Linux or `*.dylib` on Macos. On Windows, again some `*.lib` is linked, although the library name often contains `dll`. The LAPACK routines are not copied by the linker, only the information on which library to link at runtime (on Windows, this would then be some `*.dll`). The size of the executable does not grow (much). When distributed, the correct dynamic library has to be found at runtime, it is searched for in the system `PATH` (Windows) or `LD_LIBRARY_PATH` (Linux/Macos). If not found, an error is raised.

**Calling and linking MKL-LAPACK routines from C++** Simply include the MKL header file

```
#include<mkl.h>
```

In there, `"mkl_lapack.h"` is included, where all LAPACK routines are declared in upper- and lower-case letters. They can directly be used in the source code. To use `std::complex<double>` as complex type, define `MKL_COMPLEX16` *before* including `mkl.h`,

```
#define MKL_Complex16 std::complex<double>
#include <mkl.h>
```

MKL can be detected by cmake:

```
find_package(MKL REQUIRED)
```

If found, among others `MKL_ROOT` is defined as path to the MKL installation. For MKL installed via pip, this is usually somewhere like `/Library/Frameworks/Python.framework/Versions/3.10` or `C:/Program\ Files/python310/Library`. For correct usage of MKL-cmake on different systems, including static and dynamic linking, sequential and parallel realizations, we refer to the MKL page.

## 2.3 Sparse linear systems

In technical applications, we are often concerned with large linear systems representing the mechanical, electrical or multi-physics behavior of some structure. These systems may consist of millions of unknowns, each unknown representing one degree of freedom (displacements in nodes of the finite element mesh, velocities for each finite volume, temperature distributions on a grid...). Storing a full matrix of size one million is still way beyond the capacity of most computers. However, for matrices generated through the finite element, finite volume or finite difference methods, a large number of matrix entries is zero. Matrices with a small number of non-zero elements, whose position is known, are called *sparse matrices*.

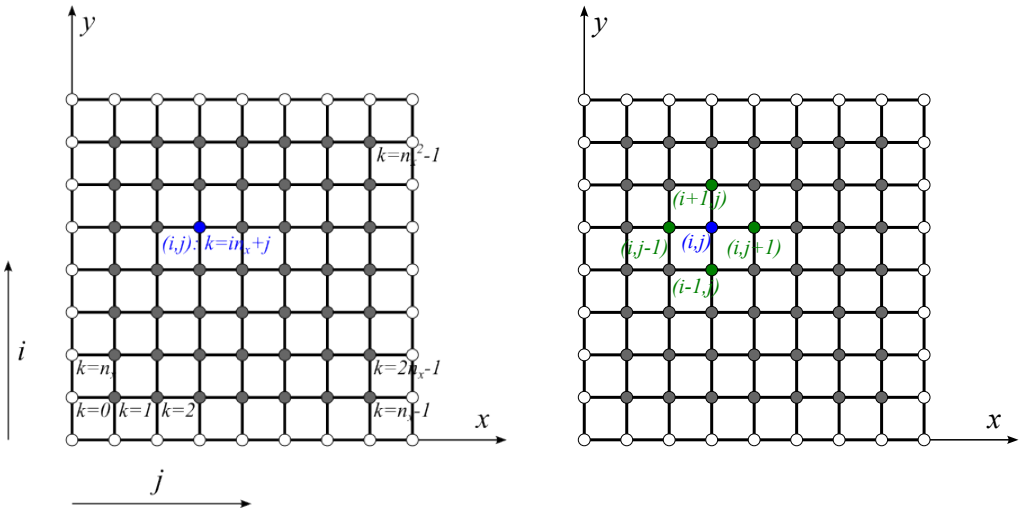


Figure 2.2: Mesh grid for the unit square. Left: consecutive numbering $k$ of nodes $(i, j)$ for heat conduction problem. Right: finite difference star.

**A simple example** A simple example where a sparse system matrix is generated is discussed below. We consider the steady-state limit problem for heat conduction on the unit square $[0, 1]^2$. Assume we are interested in temperature $u$, where we are given a (constant) thermal conductivity coefficient $\kappa$ and a heat source on the whole domain of $q(x, y)$. In steady state, the temperature is assumed to solve the heat equation,

$$-\kappa\Delta u = -\kappa\frac{\partial^2 u}{\partial x^2} - \kappa\frac{\partial^2 u}{\partial y^2} = q. \tag{2.100}$$

On the boundaries of the square, we further assume the temperature is given as zero,

$$u(x, y) = 0 \qquad \text{for } x \in \{0, 1\} \text{ or } y \in \{0, 1\}. \tag{2.101}$$

To solve this problem numerically, we apply a *finite difference scheme*, discretizing the domain by an aequidistant grid. We enumerate the *interior* grid points as indicated in Figure 2.2. Temperature values are computed in each grid point. As the temperature on the boundary nodes is known to be zero, only interior points are of interest and numbered. When using $n_x$ grid points in one coordinate direction, we get a total of $n = n_x^2$ grid points, $n = n_x^2$ unknown temperature values $u_k$. The distance between two neighboring points is the *mesh size* $h = 1/(n_x + 1)$.

Consider a mesh point $(i, j)$ at position $(x_j, y_i)$ with $x_j = (j + 1)h$, $y_i = (i + 1)h$. When numbering mesh points consecutively, it corresponds to index $k = in_x + j$. The second derivatives in mesh point $(i, j)$ can be approximated by the finite difference formula

$$\frac{\partial^2 u}{\partial x^2}(x_j, y_i) = \frac{u(x_{j-1}, y_i) - 2u(x_j, y_i) + u(x_{j+1}, y_i)}{h^2}, \tag{2.102}$$

$$\frac{\partial^2 u}{\partial y^2}(x_j, y_i) = \frac{u(x_j, y_{i-1}) - 2u(x_j, y_i) + u(x_j, y_{i+1})}{h^2}. \tag{2.103}$$

Next to the boundary, where $i-1, j-1 = -1$ or $i+j, j+1 = n_x$, the according temperatures $u$ are given as zero and can be omitted. Inserting (2.103) into the heat equation, we get $n_x^2$ equations (for each grid point) for $n_x^2$ unknown values $u_k = u(x_j, y_i)$,

$$\begin{aligned}&\text{in point } (x_j, y_i) \text{ corresp. to equation } k = in_x + j,\\ &\frac{\kappa}{h^2}\left(4u(x_j, y_i) - u(x_{j+1}, y_i) - u(x_{j-1}, y_i) - u(x_j, y_{i-1}) - u(x_j, y_{i+1})\right) = q(x_j, y_i).\end{aligned} \tag{2.104}$$

These unknowns are form a star in the finite difference grid, see Figure 2.2. This star is often referred to as *difference star* or *stencil* in finite difference methods.

We generate a set of equations, which can be described by a sparse matrix:

- each equation contains at most 5 unknowns, each line of the matrix contains at most 5 non-zero elements,
- line $k = in_x + j$ contains non-zero elements $in_x + j$, $(i-1)n_x + j$, $(i+1)n_x + j$, $in_x + (j-1)$, $in_x + (j+1)$, entries with $i-1, j-1 < 0$ or $i+1, j+1 = n_x$ are omitted (they correspond to zero boundary values).

We will build the according sparse matrix and solve the linear system. Matrices of this form are well analyzed (cf. Numerik und Optimierung); here we only state that the condition number of the matrix scales as $1/h^2$, such that $\kappa(\mathbf{A}) = O(n^4)$.

### 2.3.1 Sparse matrix storage

We develop a class for storing sparse matrices within the `LinearOperator` framework. Thus, the sparse matrix can at least be generated and applied (`Apply`).

A classical format for sparse matrices is *compressed row storage* (CRS, also known as *compressed sparse row* (CSR)). Slight differences between implementations exist, we choose the following format, which is also adopted in the Eigen lib:

- the `data` pointer (essentially `T*`) contains a list containing all *non-zero* matrix entries, ordered consecutively row-wise,
- the `colind` pointer (`int*`) contains the *column index* of the non-zero matrix entries stored in `data`,
- the `rowptr` pointer (`int*`) contains the indices of `data` and `colind` where a new row starts.

For better understandability we give an example for storing the following $3\times4$ sparse matrix,

$$\mathbf{A} = \begin{pmatrix} 10 & 0 & 0 & 22 \\ 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix}. \tag{2.105}$$

The matrix $\mathbf{A}$ is not square, this is not required for the storage class. The number of non-zero entries is `nze = 5`. Data and column index pointer are both of size `nze = 5`, while the row index pointer is one entry longer than the number of rows `height+1` which equals 4. The additional entry is not strictly necessary, but makes implementations easier.

The data pointer contains the non-zero data in consecutive order,

```
data = {10., 22., 4., 5., 3.};
```

The column index contains the column number of these data entries,

```
colind = {0, 3, 1, 2, 3};
```

The row index pointer tells for which entries a new row is started. For the example, the first row starts at entry `0`, the second row at entry `2`, the second row at `4`. The last, additional entry of `rowptr` contains the number of non-zero entries `nzw=5`,

```
rowptr = {0, 2, 4, 5};
```

The structure of the sparse matrix is determined by the two integer pointers `colind` and `rowptr`, as well as `height` and `width` of the matrix. This structure is also referred to as `matrix graph`. In finite element, finite difference or finite volume problems, the matrix graph is determined through the grid or mesh information. This is usually the first non-trivial task in generating the system matrix, filling the data pointer via system assembly being the second.

In the following, we assume a sparse matrix class containing the following members as described above,

```
template <typename T = double>
class SparseMatrix : public LinearOperator<T>
{
protected:
    T *data;
    int *colind;
    int *rowptr;

    ...

};
```

An exemplary construction of the matrix graph for the finite difference problem is provided in the source code.

### 2.3.2 Generating the matrix graph

When a sparse matrix is initialized, its graph information is necessary. Usually, `colind` and `rowptr` are not directly available, but only some kind of list providing row and column indices of the non-zero entries. The Eigen lib e.g. provides functionality that needs a list of `triplet`s as input, where each triplet consists of two indices (row and column) and the data value at this position.

For the `SparseMatrix<T>` from the current course, an interface consisting of three vectors (`std::vector<int> row`, `std::vector<int> col` and `std::vector<T> val`) where row and column index and the according data value are stored consecutively. Why `std::vector` and not `SC:Vector`? Essentially because

`std::vector` provides `push_back` to generate the input vectors by appending elements incrementally, and we do not need to think about reallocation for once.

The matrix graph is essentially built via the following code

```
SparseMatrix(int h, int w,
    std::vector<int> row,
    std::vector<int> col,
    std::vector<T>& val) : LinearOperator<T>(h,w)
{
    int nze = row.size();
    data = new T[nze];
    colind = new int[nze];
    rowptr = new int[height+1];

    Vector<int> nze_per_row(height);
    nze_per_row.SetAll(0);

    for (int i=0; i<nze; i++)
    {
        //initialize all column indices by -1
        colind[i] = -1;
        // count the number of non-zero elements per row
        nze_per_row(row[i])++;
    }

    // Step 1
    // generate the rowprt array, using count for non-zero elements
    rowptr[0] = 0;
    for (int i=0; i<height; i++)
        rowptr[i+1] = rowptr[i] + nze_per_row(i);

    // Step 2
    // generate the colind array
    // unsorted, no check for double entries
    for (int k=0; k<nze; k++)
    {
        // the first available column index
        int* colind_p = &(colind[rowptr[row[k]]]);
        // if -1, it has not been used yet - set to col[k]
        while (*colind_p != -1)
        {
            colind_p++;
            if (colind_p >= &(colind[rowptr[row[k]+1]]))
                throw "SparseMatrix: error, not enough nze in row";
            if (*colind_p == col[k])
                throw "SparseMatrix: double entry";
        }
        *colind_p = col[k];
```

```
    }

    // Step 3
    // set the data values, using Set
    for (int k=0; k<nze; k++)
    {
        this->Set(row[k], col[k], val[k]);
    }

}
```

### 2.3.3 Data access for sparse matrices

Given a `SparseMatrix<T> A`, how can an entry `A(i,j)` be accessed? The access is certainly not as direct as for dense matrices. In short, when given a row number `i` and a column number `j`, the following procedure is applied:

1. get the first index corresponding to row `i`, i.e. `rowptr[i]`,
2. for all indices from `k = rowptr[i]` up to `k < rowptr[i+1]`, check if the column index equals `j`, `colind[k] == j`,
3. if `colind[k] == j`, return the according data entry `data[k]`, if none matches the entry is zero, return `0`.

One might now want to define `operator()` accordingly, with and without `const` qualifiers, like

```
// DOES NOT WORK!
inline /*const*/ T& operator()(int i, int j) /*const*/
{
    for (int k=rowptr[i]; k<rowptr[i+1]; k++)
        {
            if (colind[k] == j) return data[k];
        }
    return 0;
}
```

As indicated, this implementation does not work. Why? While returning a reference to the `k`th data entry `data[k]` is valid, returning a reference to constant zero `0` is not. Setting values through this `operator()` cannot be done, as position `(i,j)` could be a zero entry. We resolve this problem by omitting the `T& operator()(int i, int j)`, and providing get and set functionality instead,

```
inline T Get(int i, int j) const
{
    for (int k=rowptr[i]; k<rowptr[i+1]; k++)
        {
            if (colind[k] == j) return data[k];
        }
```

```
    return 0;
}

inline void Set(int i, int j, T value)
{
    for (int k=rowptr[i]; k<rowptr[i+1]; k++)
        {
            if (colind[k] == j) data[k] = value;
            return;
        }
    // handle case of (i,j) being zero element
}
```

The above code works well as long as one does not try to set some `(i,j)` element that is not present in the matrix graph. We still have to handle this case in some way. There are essentially two different concepts:

- being user-friendly: recompute `rowptr`, `colind` and `data` arrays such that a non-zero entry at position `(i,j)` is generated, and put the value there. This operation is extremely costly (order of the number of non-zero elements)!

- being efficient: throw an exception.

Application of the sparse matrix to a vector `SparseMatrix<T>::Apply` is implemented iterating over non-zero matrix entries only. Then, if the number of matrix entries per line is bounded, the sparse matrix vector multiplication is performed in $O(n)$ operations.

### 2.3.4 Solving sparse systems

There are two generally different approaches to solving sparse linear systems,

- generate a factorization (LU or Cholesky) in such a way that the factors are *as sparse as possible*

- use an iterative method, where only application of the matrix is necessary.

Computing the inverse explicitely is definitely not a good idea, as the inverse of a sparse matrix is dense! All advantages of the sparse matrix concerning memory and operation count are lost! Even for a tridiagonal system, the inverse is fully populated.

While iterative methods are discussed in Section 2.4, we shortly review on factorization methods below.

**Factorizations for banded systems** The LU decomposition, as discussed in Section **??**, does preserve sparsity up to a certain degree. For a simple example, we see that a tri-diagonal matrix yields factors $\mathbf{L}$ and $\mathbf{U}$ that have each one off-diagonal above or below the diagonal, the tridiagonal structure is preserved:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}, \qquad \mathbf{A}^{-1} = \begin{bmatrix} 5/6 & 2/3 & 1/2 & 1/3 & 1/6 \\ 2/3 & 4/3 & 1 & 2/3 & 1/3 \\ 1/2 & 1 & 3/2 & 1 & 1/2 \\ 1/3 & 2/3 & 1 & 4/3 & 2/3 \\ 1/6 & 1/3 & 1/2 & 2/3 & 5/6 \end{bmatrix}, \tag{2.106}$$

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1/2 & 1 & 0 & 0 & 0 \\ 0 & -2/3 & 1 & 0 & 0 \\ 0 & 0 & -3/4 & 1 & 0 \\ 0 & 0 & 0 & -4/5 & 1 \end{bmatrix}, \qquad \mathbf{U} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ 0 & 3/2 & -1 & 0 & 0 \\ 0 & 0 & 4/3 & -1 & 0 \\ 0 & 0 & 0 & 5/4 & -1 \\ 0 & 0 & 0 & 0 & 6/5 \end{bmatrix}. \tag{2.107}$$

For a tridiagonal matrix, $\mathbf{L}$ and $\mathbf{U}$ factors can be computed in $O(n)$ operations (compared to $O(n^3)$ for fully populated matrices). Also the application of the factorization is of order $O(n)$ operations. Compared to computation of the inverse $\mathbf{A}^{-1}$ (which is $O(n^3)$) and application of the inverse (which is $O(n^2)$) using a tridiagonal LU factorization means an immense gain in efficiency!

This behavior is kept for banded systems: We call $\mathbf{A}$ a banded matrix with bandwidth $b < n$ if all entries $A_{ij}$ with $i > j + b$ or $j > i + b$ equal zero,

$$A_{ij} = 0 \qquad \text{for } i > j + b \text{ or } j > i + b. \tag{2.108}$$

For such a banded matrix $\mathbf{A}$ with bandwidth $b$, also $\mathbf{L}$ and $\mathbf{U}$ are banded with bandwidth $b$. Operation costs for factorization are of order $O(b^2 n)$, memory requirements are of order $O(bn)$. Unfortunately, sparsity within the band does not reduce the cost further. Empty bands (where only the $b^{th}$ off-diagonal and the diagonal are non-zero) lead to the same structure of the LU factorization as fully populated bands. Reordering of equations and unknowns can improve the band structure of a matrix.

In how far does using the band structure help with matrices stemming from typical 2D or 3D finite element/volume/difference applications? Let $n$ denote the system size (the number of nodes or cells in the object), and $n_x$ the number of nodes or cells in one coordinate direction. For 2D, we have $n \simeq n_x^2$, while for 3D we see $n \simeq n_x^3$. For the sparse matrix bandwidth and solver costs, this means

- for 2D: bandwidth is $b \simeq n_x \simeq \sqrt{n}$, leading to $O(b^2 n) = O(n^2)$ operations for solving,
- for 3D: bandwidth is $b \simeq n_x^2 \simeq n^{2/3}$, leading to $O(b^2 n) = O(n^{7/3})$ operations for solving.

The above complexities are better than the costs for full LU solve $O(n^3)$, but still not optimal. One can also see that 3D problems are essentially harder to tackle with direct solution methods than 2D problems.

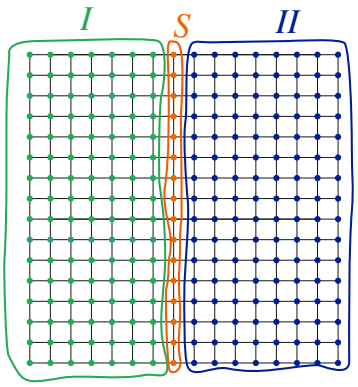


Figure 2.3: The separator $S$ divides the graph into two parts of essentially equal size $I$ and $II$.

**Nested dissection** Nested dissection was first proposed by Alan George [3], where a model problem very similar to the 2D finite difference problem described in the beginning of this section is considered. Nested dissection provides an ordering of the equations and unknowns to be solved, such that the LU factors are "as sparse as possible". The basic technique is widely used in sparse direct solvers that are commercially or freely available nowadays, including Pardiso (part of Intel's Math Kernel library), MUMPS or UMFPACK.

In this course, we do not provide a detailed algorithm. We rather discuss the general idea, from which we see why nested dissection is efficient, and why it is more efficient for matrices stemming from 2D than from 3D problems.

Recall the finite difference problem for solving the heat equation on a square domain. When building the matrix graph, we have seen that position $A_{ij}$ is non-zero if and only if node $i$ and node $j$ are direct neighbors, i.e. if they are connected by a common edge of the grid. If nodes $i$ and $j$ are not connected directly, $A_{ij}$ will be zero.

Now choose an interface $S$ that splits the grid into two (essentially) equal-sized parts $I$ and $II$. In graph theory, this interface is referred to as *separator*. These sets are visualized in Figure 2.3.

The unknowns can be reordered according to which part they belong to, the separator being much smaller than the two distinct parts.

$$\boldsymbol{x} \rightarrow \left[\begin{array}{c} \boldsymbol{x}_I \\ \hline \boldsymbol{x}_{II} \\ \hline \boldsymbol{x}_S \end{array}\right]. \tag{2.109}$$

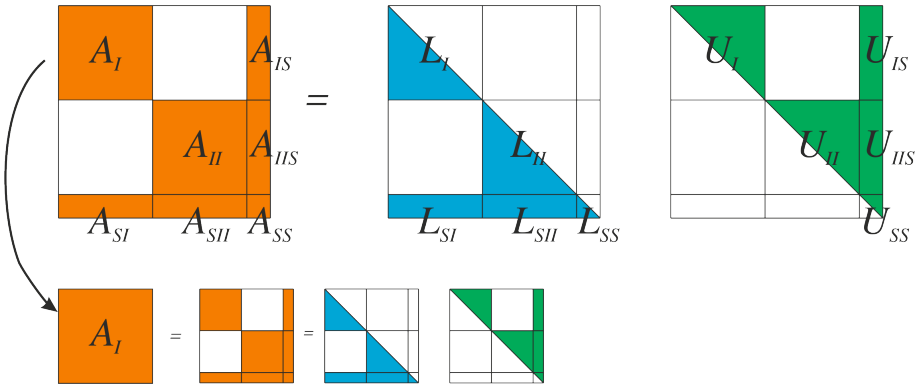


Figure 2.4: Nested dissection: recursive treatment through separators

If the matrix rows and columns are reordered in the same manner. From our knowledge on matrix zero entries, we know it is of the following structure, and also the LU decomposition of $\mathbf{A}$ respects this structure,

$$\mathbf{A} = \left[\begin{array}{c|c|c} \mathbf{A}_I & \mathbf{0} & \mathbf{A}_{IS} \\ \hline \mathbf{0} & \mathbf{A}_{II} & \mathbf{A}_{IIS} \\ \hline \mathbf{A}_{SI} & \mathbf{A}_{SII} & \mathbf{A}_{SS} \end{array}\right] = \left[\begin{array}{c|c|c} \mathbf{L}_I & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{L}_{II} & \mathbf{0} \\ \hline \mathbf{L}_{SI} & \mathbf{L}_{SII} & \mathbf{L}_{SS} \end{array}\right] \left[\begin{array}{c|c|c} \mathbf{U}_I & \mathbf{0} & \mathbf{U}_{IS} \\ \hline \mathbf{0} & \mathbf{U}_{II} & \mathbf{U}_{IIS} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{U}_{SS} \end{array}\right]. \tag{2.110}$$

In the above system, $\mathbf{L}_{IS}, \mathbf{L}_{IIS}, \mathbf{L}_{SS}$ and $\mathbf{U}_{IS}, \mathbf{U}_{IIS}, \mathbf{U}_{SS}$ are much smaller than the original matrix, as the separator contains only few unknowns, and computed in full. The diagonal entries $\mathbf{A}_I = \mathbf{L}_I\mathbf{U}_I$ and $\mathbf{A}_{II} = \mathbf{L}_{II}\mathbf{U}_{II}$ are half the size of the original problem, but still comparably large.

The $\mathbf{A}_I$ and $\mathbf{A}_{II}$ parts are treated in a recursive manner – thus the name *nested dissection*, see also Figure 2.4. For each, another separator is found, which further subdivides the sub-matrices. These separators are again of smaller size than the original separator. The number of these subdivisions grows only logarithmically in the system size, it is $O(\log n)$, which is much smaller than $O(n)$.

The estimate for this type of nested dissection is that two-dimensional problems can be treated within $O(n_x^3) = O(n^{3/2})$ operation, while three-dimensional problems need $O(n_x^6) = O(n^2)$. Once the LU factorizations are generated, application of the inverse can be done in $O(n \log n)$ for 2D and $O(n^{4/3})$ in 3D.

Efficient parallel direct solver packages such as PARDISO can handle 2D problems of moderate size ($10^6$ to $10^7$ unknowns) quite satisfactory, while for 3D the higher complexity provides fast solutions only up to approx. $10^6$ unknowns (given there is enough memory available, from my experience 32 GB for $10^6$ unknowns in 3D are needed).

## 2.4 Iterative solution methods for linear systems

In the previous sections, we have seen different methods of solving the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ exactly. We found the overall complexity of solving a linear system to be $O(n^3)$, where $n$ is the size of $\mathbf{A}$. Iterative solvers work differently; instead of computing the (up to round-off errors) exact solution, a sequence of solutions $\boldsymbol{x}^{(n)}$ is generated that converges towards the exact solution, $\boldsymbol{x}^{(n)} \to \boldsymbol{x}$. The better the iterative solution method is, the smaller the number of iterations is to generate an iterate of sufficient accuracy.

We collect some characteristics of iterative solution methods.

- Many iterative methods do not need a matrix representation of $\mathbf{A}$, just its application to some given vector $\boldsymbol{r} = \mathbf{A}\boldsymbol{x}$ is necessary. Therefore, iterative methods are often used for blackbox operators, for large sparse linear systems, and when computing the application of $\mathbf{A}$ is much cheaper than computing $\mathbf{A}$ (e.g. in boundary element methods).

- Even when the system matrix $\mathbf{A}$ is sparse, containing almost only zero elements, its inverse is fully populated. Then, an application $\mathbf{A}\boldsymbol{x}$ can be computed in $O(n)$ operations, while application of the (explicitely pre-computed) inverse needs at least $n^2$ operations. Sparse LU factorization using nested dissection needs $O(n^{3/2})$ (in 2D) or $O(n^2)$ (in 3D) operations. A good iterative solver converges within a small number of iterative steps where $\mathbf{A}$ is applied; it can produce an (approximate) solution within $O(n)$ operations, which is optimal!

- Iterative solvers often use knowledge of the structure of the underlying system of equations, such as symmetry or positive definiteness of the matrix. These characteristics are often known for matrices stemming from the finite element or finite volume method.

- Iterative solvers require less memory than direct solvers based on factorization. For the implementation of a direct solver, only application of $\mathbf{A}$ and some additional memory for storing data vectors is necessary.

- Convergence of iterative solvers depends on the condition number of the matrix. The better the matrix is conditioned, the closer the condition number is to one, the faster the solver converges. For ill-conditioned matrices, *preconditioning techniques* are developed to increase conditioning.

**Notation** Consider the linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. In each iterative method, we use

- the $k^{th}$ iterate $\boldsymbol{x}^{(k)}$,

- the $k^{th}$ residual $\boldsymbol{r}^{(k)} = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k)}$,

- the $k^{th}$ search direction $\boldsymbol{p}^{(k)}$ and update vector $\boldsymbol{w}^{(k)}$, such that $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha\boldsymbol{p}^{(k)} = \boldsymbol{x}^{(k)} + \boldsymbol{w}^{(k)}$,

- the $k^{th}$ error vector or defect $\boldsymbol{e}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}$.

**Implementations** From an implementational point of view, iterative solution methods can be designed as `LinearOperator<T>`. Provided with (a reference to) another `LinearOperator<T>` representing the system matrix $\mathbf{A}$, the solution algorithms can be implemented independent of the actual matrix representation of the system. Only application of the system matrix via `Apply` is necessary, the algorithm is the same for dense and sparse, symmetric and non-symmetric, tridiagonal etc. matrices, or also if $\mathbf{A}$ is not known in matrix form at all, but only through its `Apply` routine. This is a major difference to the direct solvers discussed in the previous sections, where density or sparsity patterns were of immediate interest to the solver routine.

### 2.4.1 Residual-based iterative methods

Four examples of iterative methods are presented, where in each iterative step $k$ the residual vector $\boldsymbol{r}^{(k)}$ is used to determine the next update vector $\boldsymbol{w}^{(k)}$.

**Richardson's method** We can reformulate the original equation $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ for some positive factor $\theta$ as

$$\boldsymbol{x} = \boldsymbol{x} + \theta(\boldsymbol{b} - \mathbf{A}\boldsymbol{x}). \tag{2.111}$$

Based on the formulation above, the update rule for Richardson's method reads

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \underbrace{\theta(\underbrace{\boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k)}}_{\boldsymbol{r}^{(k)}})}_{\boldsymbol{w}^{(k)}} = \boldsymbol{x}^{(k)} + \theta\boldsymbol{r}^{(k)}. \tag{2.112}$$

Convergence of the error vector norm can be shown if the corresponding matrix norm $\|\mathbf{I} - \theta\mathbf{A}\| < 1$. This is e.g. satisfied if $\mathbf{A}$ is *positive definite* and $\theta \leq 2/\lambda_{max}(\mathbf{A})$ for $\lambda_{max}$ the eigenvalue of maximum absolute value. An optimal choice of $\theta$ is crucial for convergence: choosing $\theta$ too large will lead to divergence of the $\boldsymbol{x}^{(k)}$, while choosing $\theta$ too small means slow convergence due to small updates. For *hermitian positive definite* matrices the optimal parameter is $\theta = 2/(\lambda_{min}(\mathbf{A}) + \lambda_{max}(\mathbf{A}))$.

For hermitian positive definite matrices $\mathbf{A}$, Richardson's method is also known as *method of steepest descent*: solving $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ is then equivalent to the minimization problem

$$W(\boldsymbol{x}) := \frac{1}{2}\boldsymbol{x}^H\mathbf{A}\boldsymbol{x} - \boldsymbol{b}^H\boldsymbol{x} \to \min_{\boldsymbol{x}}. \tag{2.113}$$

The search direction, which is the residual, equals the direction of the negative gradient, $\boldsymbol{r} = -\nabla W$, which is the direction of steepest descent.

**Jacobi's method** A very similar idea leads to Jacobi's method: instead of using the same value $\theta$ for all equations, we choose a different factor for each component of $\boldsymbol{x}^{(k+1)}$. To be concise, we set

$$x_i^{(k+1)} = x_i^{(k)} + \frac{1}{A_{ii}}r_i^{(k)} = x_i^{(k)} + \frac{1}{A_{ii}}\left(b_i - \sum_j A_{ij}x_j^{(k)}\right) \tag{2.114}$$

This is equivalent to using the inverse diagonal of $\mathbf{A}$ as a scaling matrix,

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \operatorname{diag}(\mathbf{A})^{-1}(\boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k)}) = \boldsymbol{x}^{(k)} + \operatorname{diag}(\mathbf{A})^{-1}\boldsymbol{r}^{(k)}. \tag{2.115}$$

In comparison to Richardson's method, it is not necessary to find an optimal value for $\theta$. Jacobi's method is perferrable if the equations are scaled differently, e.g. if coefficients (such as $\kappa$ in the heat equation example) or the grid size $h$ varies over the domain, or if different physical entities with different units are computed.

**Gauss-Seidel method** Another variant is the Gauss-Seidel iteration. Assuming we do the update of the components $x_i^{(k-1)}$ successively for $i = 0$ to $n-1$, at index $i$ all $x_j^{(k+1)}$ for $j < i$ have already be computed. We can use these values in the update,

$$x_i^{(k+1)} = x_i^{(k)} + \frac{1}{A_{ii}}\left(b_i - \sum_{j<i} A_{ij}x_j^{(k+1)} - \sum_{j\geq i} A_{ij}x_j^{(k)}\right). \tag{2.116}$$

This update rule can be expressed in a similar manner to (2.112) and (2.115). We reorder all terms above,

$$\begin{aligned} A_{ii}x_i^{(k+1)} + \sum_{j<i} A_{ij}x_j^{(k+1)} &= A_{ii}x_i^{(k)} + \left(b_i - \sum_{j\geq i} A_{ij}x_j^{(k)}\right) \\ &= A_{ii}x_i^{(k)} + \sum_{j<i} A_{ij}x_j^{(k)} + \left(b_i - \sum_{j} A_{ij}x_j^{(k)}\right), \end{aligned} \tag{2.117}$$

and further

$$\sum_{j\leq i} A_{ij} \underbrace{(x_j^{(k+1)} - x_j^{(k)})}_{=w_j^{(k)}} = b_i - \sum_j A_{ij}x_j^{(k)}. \tag{2.118}$$

The above system of equation can be rewritten in matrix-vector form. To this end, we need the lower left triangular block of matrix $\mathbf{A}$,

$$\mathbf{A}_L = \begin{bmatrix} A_{00} & 0 & \dots & 0 \\ A_{10} & A_{11} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ A_{n-1\,0} & A_{n-1\,1} & \dots & A_{n-1\,n-1} \end{bmatrix}. \tag{2.119}$$

Then, we get

$$\mathbf{A}_L\boldsymbol{w}^{(k)} = \boldsymbol{r}^{(k)}, \tag{2.120}$$

and the update equation is

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \mathbf{A}_L^{-1}(\boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k)}). \tag{2.121}$$

**Successive over-relaxation SOR** Successive over-relaxation is again a variant of the Gauss-Seidel method, where the update vector is additionally scaled by a factor $\omega$,

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{A_{ii}} \left( b_i - \sum_{j<i} A_{ij} x_j^{(k+1)} - \sum_{j \geq i} A_{ij} x_j^{(k)} \right). \tag{2.122}$$

Typically, one chooses $\omega \in (1, 2)$, which justifies the term *over*-relaxation.

One can again put the update equation in matrix-vector form,

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \mathbf{A}_{L,\omega}^{-1}(\boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k)}) \qquad \text{with } \mathbf{A}_{L,\omega} = \begin{bmatrix} A_{00}/\omega & 0 & \dots & 0 \\ A_{10} & A_{11}/\omega & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ A_{n-1\,0} & A_{n-1\,1} & \dots & A_{n-1\,n-1}/\omega \end{bmatrix}. \tag{2.123}$$

**General class** The general class of all these methods has the following form,

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \mathbf{P}^{-1}(\boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k)}) = \boldsymbol{x}^{(k)} + \mathbf{P}^{-1}\boldsymbol{r}^{(k)}. \tag{2.124}$$

Different choices for $\mathbf{P}$ lead to different methods.

Choosing $\mathbf{P} = \mathbf{A}$ leads to an optimal method in the sense that convergence is obtained in one single step. However, this single step includes solving the full set of equations, which was the initial problem to be solved. In general, $\mathbf{P}$ should meet the following requirements

- $\mathbf{P}$ should – in some sense – be close to $\mathbf{A}$,
- solving for $\mathbf{P}$ should be possible at optimal cost, ideally $O(n)$,
- it is not necessary to have $\mathbf{P}$ explicitely, but the application of its inverse $\mathbf{P}^{-1}$.

We call $\mathbf{P}$ (or rather $\mathbf{P}^{-1}$) a preconditioner. Above preconditioners for Jacobi's method, Gauss-Seidel or SOR methods do not lead to optimal convergence. The construction of preconditioners depends highly on the properties of the system to be solved. Nowadays, optimal preconditioners for symmetric positive definite systems (e.g. stemming from FE ov FV discretizations) exist, such as multilevel, multigrid, algebraic multigrid (AMG) or domain decomposition (DD) solvers. Ideally, these preconditioners work indepently of changing mesh size, jumping coefficients .... For indefinite systems, the construction of preconditioners is even harder.

**Convergence analysis** We define the *iteration matrix* such that

$$\boldsymbol{x}^{(k+1)} = \mathbf{M}\boldsymbol{x}^{(k)} + \mathbf{P}^{-1}\boldsymbol{b}, \qquad \text{i.e. } \mathbf{M} = \mathbf{I} - \mathbf{P}^{-1}\,\mathbf{A}. \tag{2.125}$$

One can show that the algorithm converges *independently of the starting value* if all eigenvalues of the iteration matrix $\mathbf{M} = \mathbf{I} - \mathbf{P}^{-1}\mathbf{A}$ are of absolute value smaller than one, using the spectral radius $\rho(\mathbf{M})$ this can be put as

$$\rho(\mathbf{M}) = \max |\lambda_i(\mathbf{M})| < 1. \tag{2.126}$$

For a compatible matrix norm $\|\cdot\|_M$, it is sufficient that $\|\mathbf{M}\|_M < 1$. As we usually do not have a closed representation of $\mathbf{M}$, this is usually not much easier to show.

The iteration matrix controls the evolution of the error vectors $\boldsymbol{e}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}$,

$$\boldsymbol{e}^{(k+1)} = \mathbf{M}\boldsymbol{e}^{(k)}. \tag{2.127}$$

Can we estimate the number of necessary steps, if the starting error $\|\boldsymbol{e}^{(0)}\|$ should be reduced by a factor $\varepsilon$? Then $k$ is such that

$$\varepsilon\|\boldsymbol{e}^{(0)}\| \geq \|\boldsymbol{e}^{(k)}\| = \|\mathbf{M}^k\boldsymbol{e}^{(0)}\|. \tag{2.128}$$

Thus we need

$$\varepsilon \geq \frac{\|\mathbf{M}^k\boldsymbol{e}^{(0)}\|}{\|\boldsymbol{e}^{(0)}\|}. \tag{2.129}$$

Due to the sub-multiplicativity of compatible matrix norms, it is sufficient if

$$\varepsilon \geq \|\mathbf{M}^k\|_M \qquad \text{or} \qquad \log\varepsilon \geq \log\|\mathbf{M}^k\|_M = k\log\left(\|\mathbf{M}^k\|_M^{1/k}\right). \tag{2.130}$$

It holds that $\|\mathbf{M}^k\|_M^{1/k} \to \rho(\mathbf{M})$. Recall that we need $\rho(\mathbf{M}) < 1$ for a convergent method, then $\log(\rho(\mathbf{M})) < 0$ is negative. This implies

$$k \geq \frac{-\log\varepsilon}{-\log\rho(\mathbf{M})}. \tag{2.131}$$

As a rule of thumb, $-\log\varepsilon$ corresponds to the number of digits of the error reduction. The number of iterations grows linearly with the number of digits of the error reduction. Convergence gets worse as the spectral radius approaches one, with $\rho(\mathbf{M}) \to 1$ we have $\log\rho(\mathbf{M}) \to 0$ and $k \to \infty$.

The algorithm converges **linearly** with convergence factor $\rho(\mathbf{M})$; asymptotically the error is reduced by $\rho(M)$ in each iterative step.

To actually compute the reduction factor for the error (which is unknown), equivalently the reduction in the residual vector (which is known) can be used.

### 2.4.2 CG for symmetric/hermitian positive definite problems

The *conjugate gradient method* (CG) is a method designed for symmetric positive definite (SPD) problems (in case of complex numbers, hermitian positive definite problems, HPD). Recall that parantheses $(\cdot,\cdot)$ denote the relevant inner product for real and complex vectors, i.e.

$$(\boldsymbol{a},\boldsymbol{b}) = \begin{cases} \boldsymbol{a}^T\boldsymbol{b} & \text{in } \mathbb{R}^n, \\ \overline{\boldsymbol{a}}^T\boldsymbol{b} = \boldsymbol{a}^H\boldsymbol{b} & \text{in } \mathbb{C}^n. \end{cases} \tag{2.132}$$

For SPD/HPD problems, solving $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ is equivalent to the minimization problem

$$W(\boldsymbol{x}) = \frac{1}{2}(\boldsymbol{x}, \mathbf{A}\boldsymbol{x}) - Re(\boldsymbol{x},\boldsymbol{b}) \to \min. \tag{2.133}$$

The conjugate gradient method is initialized for some vector $\boldsymbol{x}^{(0)}$, setting $\boldsymbol{r}^{(0)} = \mathbf{A}\boldsymbol{x}^{(0)} - \boldsymbol{b}$ and choosing the first search direction $\boldsymbol{p}^{(0)} = \boldsymbol{r}^{(0)}$ the direction of steepest descent.

Iterates in the $k^{th}$ step are generated as follows

- $\boldsymbol{x}^{(k+1)}$ is chosen among all $\boldsymbol{x}^{(k)} + \alpha\boldsymbol{p}^{(k)}$ such that $W(\boldsymbol{x}^{(k)} + \alpha\boldsymbol{p}^{(k)})$ is minimized. This results in a one-dimensional quadratic optimization problem for $\alpha$, leading to $\alpha = Re(\boldsymbol{p}^{(k)}, \boldsymbol{r}^{(k)})/(\boldsymbol{p}^{(k)}, \mathbf{A}\boldsymbol{p}^{(k)})$. Due to orthogonalities of search and residual directions (see below), the following simpler formula is valid,

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha\boldsymbol{p}^{(k)} \qquad \text{with } \alpha^{(k)} = \frac{(\boldsymbol{r}^{(k)}, \boldsymbol{r}^{(k)})}{(\boldsymbol{p}^{(k)}, \mathbf{A}\boldsymbol{p}^{(k)})}. \tag{2.134}$$

- the residual $\boldsymbol{r}^{(k+1)}$ can be computed in terms of $\boldsymbol{r}^{(k)}$ and $\mathbf{A}\boldsymbol{p}^{(k)}$, which have both been computed previously,

$$\boldsymbol{r}^{(k+1)} = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(k+1)} = \boldsymbol{r}^{(k)} - \alpha^{(k)}\mathbf{A}\boldsymbol{p}^{(k)}. \tag{2.135}$$

- the next search direction is defined in (2.136), we discuss this choice in more detail in the following,

$$\boldsymbol{p}^{(k+1)} = \boldsymbol{r}^{(k+1)} + \beta^{(k)}\boldsymbol{p}^{(k)} \qquad \text{with } \beta^{(k)} = \frac{(\boldsymbol{r}^{(k+1)}, \boldsymbol{r}^{(k+1)})}{(\boldsymbol{r}^{(k)}, \boldsymbol{r}^{(k)})}. \tag{2.136}$$

The CG method belongs to the larger class of *Krylov subspace methods*. For any matrix $\mathbf{A}$ and initial residual $\boldsymbol{r}^{(0)}$, the $k^{th}$ Krylov subspace $K^k$ is defied as

$$K^k(\mathbf{A}, \boldsymbol{r}^{(0)}) = \text{lin}(\boldsymbol{r}^{(0)}, \mathbf{A}\boldsymbol{r}^{(0)}, \mathbf{A}^2\boldsymbol{r}^{(0)}, \ldots, \mathbf{A}^k\boldsymbol{r}^{(0)}). \tag{2.137}$$

It can be shown that the $k^{th}$ iterate $x^{(k)}$ minimizes the potential $W(\boldsymbol{x})$ over the $(k-1)^{th}$ Krylov space,

$$\boldsymbol{x}^{(k)} = \arg \min_{x \in K^{k-1}(\mathbf{A}, \boldsymbol{r}^{(0)})} W(\boldsymbol{x}). \tag{2.138}$$

Concerning efficient implementation, we note that in each iterative step, the matrix $\mathbf{A}$ is applied only once on computation of $\mathbf{A}\boldsymbol{p}^{(k)}$. Four vectors $\boldsymbol{x}^{(k)}, \boldsymbol{r}^{(k)}, \boldsymbol{p}^{(k)}$ and $\mathbf{A}\boldsymbol{p}^{(k)}$ are sufficient, two inner products $(\boldsymbol{r}^{(k)}, \boldsymbol{r}^{(k)})$ and $(\boldsymbol{p}^{(k)}, \mathbf{A}\boldsymbol{p}^{(k)})$ are computed.

Crucial to the convergence properties of the CG methods are the following characteristics.

- Residuals are orthogonal, $(\boldsymbol{r}^{(k)}, \boldsymbol{r}^{(j)}) = 0$ for any $k \neq j$.
- Search directions are *conjugate*, $(\boldsymbol{p}^{(k)}, \mathbf{A}\boldsymbol{p}^{(j)}) = 0$ for $k \neq j$.
- The residual is orthogonal to previous search directions, $(\boldsymbol{r}^{(k)}, \boldsymbol{p}^{(j)}) = 0$ for any $k > j$.

From the orthogonality relations, one directly deduces that the CG algorithm stops at the exact solution after $n$ iterations: there are only $n$ orthogonal directions in an $n$-dimensional vector space, it is necessarily $\boldsymbol{r}^{(n)} = 0$. Also, the $n$-th Krylov subspace $K^n(\mathbf{A}, \boldsymbol{r}^{(0)})$ spans $\mathbb{R}^n$, such that $\boldsymbol{x}^{(n)}$ minimizes $W(x)$ over $K^n(\mathbf{A}, \boldsymbol{r}^{(0)}) = \mathbb{R}^n$. This is not the main motivation for using the CG method. Additionally, it can be shown that it converges essentially faster than the variants of Richardson's method.

**CG and preconditioning** The CG method can be further accelerated by preconditioning, using some preconditioner $\mathbf{P}^{-1}$. In this case, the iteration needs one additional vector $\boldsymbol{z}$ and is conducted as follows:

- the new iterate is computed as

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha \boldsymbol{p}^{(k)} \qquad \text{with } \alpha^{(k)} = \frac{(\boldsymbol{r}^{(k)}, \boldsymbol{z}^{(k)})}{(\boldsymbol{p}^{(k)}, \mathbf{A}\boldsymbol{p}^{(k)})}. \tag{2.139}$$

- the residual can be computed via

$$\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} - \alpha^{(k)} \mathbf{A}\boldsymbol{p}^{(k)}. \tag{2.140}$$

- the new iterate for $\boldsymbol{z}^{(k+1)}$ is computed from the preconditioner

$$\boldsymbol{z}^{(k+1)} = \mathbf{P}^{-1} \boldsymbol{r}^{(k+1)}, \tag{2.141}$$

- the next search direction

$$\boldsymbol{p}^{(k+1)} = \boldsymbol{z}^{(k+1)} + \beta^{(k)} \boldsymbol{p}^{(k)} \qquad \text{with } \beta^{(k)} = \frac{(\boldsymbol{r}^{(k+1)}, \boldsymbol{z}^{(k+1)})}{(\boldsymbol{r}^{(k)}, \boldsymbol{z}^{(k)})}. \tag{2.142}$$

Note that the preconditioner $\mathbf{P}^{-1}$ needs to be *symmetric positive definite*! Therefore, Jacobi's preconditioner works fine, while e.g. non-symmetric Gauss-Seidel or successive over-relaxation SOR do not work!

For an implementation of the CG method see the source code provided separately

### 2.4.3 Krylov spaces and Arnoldi iteration

Krylov subspaces, and orthogonal bases to these spaces, are important in several applications in computational mathematics. In this section, we discuss *Arnoldi iteration* to find such an orthogonal basis. This basis will be used in the generalized minimal residual method (GMRES) from the next section, as well as in eigenvalue computations.

Arnoldi [1] proposed a method to compute the eigenvalues of an $n \times n$ matrix $\mathbf{A}$. This iteration can also be interpreted as a method that generates an orthogonal basis for the sequence of Krylov spaces. Given some starting residual $\boldsymbol{r}^{(0)}$, in the $k^{th}$ step, vector $\boldsymbol{v}^{(k)}$ is added such that $\{\boldsymbol{v}^{(i)}, i \leq k\}$ form a basis for $K^k(\mathbf{A}, \boldsymbol{r}^{(0)})$.

Compute the starting residual $\boldsymbol{r}^{(0)}$. The basis of $K^0(\mathbf{A}, \boldsymbol{r}^{(0)}) = \text{lin}(\boldsymbol{r}^{(0)})$ is naturally given by setting $\boldsymbol{v}^{(0)} = \frac{1}{\|\boldsymbol{r}^{(0)}\|}\boldsymbol{r}^{(0)}$. If there is no starting residual given (e.g. the residual vector to the system to be solved), a random starting vector $\boldsymbol{r}^{(0)}$ should be used.

In the $k^{th}$ step, given $\boldsymbol{v}^{(0)} \dots \boldsymbol{v}^{(k-1)}$ orthonormal and spanning $K^{k-1}(\mathbf{A}, \boldsymbol{r}^{(0)}) = \text{lin}(\boldsymbol{r}^{(0)}, \dots, \mathbf{A}^{k-1}\boldsymbol{r}^{(0)})$, an additional orthonormal vector $\boldsymbol{v}^{(k)}$ is generated such that $K^k(\mathbf{A}, \boldsymbol{r}^{(0)}) = \text{lin}(\boldsymbol{r}^{(0)}, \dots, \mathbf{A}^{k-1}\boldsymbol{r}^{(0)}, \mathbf{A}\mathbf{A}^{k-1}\boldsymbol{r}^{(0)})$ is spanned.

If we add $\boldsymbol{q}^{(k-1)} = \mathbf{A}\boldsymbol{v}^{(k-1)}$ to the set of basis vectors, certainly $K^k$ is spanned. But $\boldsymbol{q}^{(k-1)}$ is not orthonormal w.r.t. the rest of the basis vectors. We apply Gram-Schmidt orthogonalization, setting

$$\boldsymbol{w}^{(k)} := \boldsymbol{q}^{(k-1)} - \sum_{i=0}^{k-1} \left(\boldsymbol{q}^{(k-1)}, \boldsymbol{v}^{(i)}\right) \boldsymbol{v}^{(i)}, \qquad \boldsymbol{v}^{(k)} := \frac{1}{\|\boldsymbol{w}^{(k)}\|}\boldsymbol{w}^{(k)}. \tag{2.143}$$

Computing the update $\boldsymbol{w}^{(k)}$ directly as proposed above (2.143) leads to numerical instabilities. More stable is the modified Gram-Schmidt process, which is (theoretically) equivalent,

$$\text{for } i = k-1 \dots 0 : \boldsymbol{q}^{(k-1)} := \boldsymbol{q}^{(k-1)} - \left(\boldsymbol{q}^{(k-1)}, \boldsymbol{v}^{(i)}\right) \boldsymbol{v}^{(i)}, \qquad \boldsymbol{v}^{(k)} := \frac{1}{\|\boldsymbol{q}^{(k-1)}\|}\boldsymbol{q}^{(k-1)}. \tag{2.144}$$

We store the factors from the Gram-Schmidt procedure in matrix form, for $i = 0 \dots k$ and $j = 0 \dots k-1$

$$H_{i,j} = \left(\boldsymbol{q}^{(j)}, \boldsymbol{v}^{(i)}\right). \tag{2.145}$$

One can verify that the according matrix $\mathbf{H}$ is a *Hessenberg* matrix: We call $\mathbf{H}$ a Hessenberg matrix if it is $(k+1) \times k$, an upper right triangular matrix with an extra subdiagonal on the lower left,

$$\mathbf{H} = \begin{bmatrix} * & * & \dots & * \\ * & * & \dots & \vdots \\ 0 & * & \ddots & \vdots \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{bmatrix}, \qquad \text{or } H_{ij} = 0 \text{ for } j < i-1. \tag{2.146}$$

Why is $\mathbf{H}$ as defined above of Hessenberg form? See the definition,

$$H_{ij} = \left(\boldsymbol{q}^{(j)}, \boldsymbol{v}^{(i)}\right). \tag{2.147}$$

Then $H_{ij} = 0$ if $\boldsymbol{q}^{(j)} \perp \boldsymbol{v}^{(i)}$ are orthogonal. The $i^{th}$ basis vector $\boldsymbol{v}^{(i)}$ was constructed such that it is orthogonal to $K^{(i-1)}$, and the $j^{th}$ update vector $\boldsymbol{q}^{(j)}$ was constructed such that it lies in $K^{j+1}$. Thus, for $j + 1 \leq i - 1$, we have orthogonality, which is equivalent to $j < i - 1$.

Moreover, the sub-diagonal entry equals the norm of the next basis vector,

$$H_{j+1,j} = \left(\boldsymbol{q}^{(j)}, \boldsymbol{v}^{(j+1)}\right) = \|\boldsymbol{w}^{(j+1)}\|, \tag{2.148}$$

which can directly be verified using the orthogonality of the $\boldsymbol{v}^{(j)}$ basis vectors.

We collect the basis vectors $\boldsymbol{v}^{(j)}, j = 0 \ldots k-1$ column-wise in the matrix $\mathbf{V}^{(k)}$ (which is then an $n \times (k+1)$ matrix). If we review the Gram-Schmidt process, we verify that not only are the columns of $\mathbf{V}^{(k)}$ are a basis for $K^k$, but also

$$\mathbf{A}\mathbf{V}^{(k-1)} = \mathbf{V}^{(k)}\mathbf{H}^{(k)}. \tag{2.149}$$

Above, $H^{(k)}$ is the $(k+1)\times k$ Hessenberg matrix that was generated by the according step.

When does the Arnoldi iteration terminate? Obviously, the maximum number of iterations is given by $k = n - 1$, as then $K^{n-1} = \mathbb{R}^n$ the complete space. For most applications, it is sufficient to compute either a predefined number of basis vectors ($k \leq k_{max}$), or stop whenever the next update vector is sufficiently small ($\|\boldsymbol{w}^{(k)}\| \leq \text{tol}$). Note that the Arnoldi iteration can be performed for symmetric (hermitian) and non-symmetric (non-hermitian) as well as regular and singular matrices alike.

In case of symmetric (hermitian) matrices, instead of a Hessenberg matrix we obtain a tridiagonal matrix $\mathbf{H}$.

**Implementation** A crucial point in the implementation of the Arnoldi iteration is the ever-growing size of the generated matrices $\mathbf{V}^{(k)}$ and $\mathbf{H}^{(k)}$. Re-allocating these matrices in each iterative step is not the method of choice. Usually, matrices for some initial size $k_0$ are allocated, and re-allocation is necessary only if $k$ exceeds $k_0$.

### 2.4.4 GMRES

The *generalized minimal residual method* (GMRES) is an iterative method for solving non-symmetric (non-hermitian) systems of equations. Like the CG method, it is a Krylov space method. The idea of the GMRES method is to *minimize the Euclidean norm of the residual* $\|\boldsymbol{r}\|$ over the series of Krylov subspaces.

Given some starting vector $\boldsymbol{x}^{(0)}$, the initial residual is given as $\boldsymbol{r}^{(0)} = \boldsymbol{b} - \mathbf{A}\boldsymbol{x}^{(0)}$.

- Use the Arnoldi iteration to generate orthonormal basis vectors $\boldsymbol{v}^{(j)}$. Break the iteration if $\|\boldsymbol{w}^{(k)}\| < \varepsilon$ and some tolerance is reached. This provides $\mathbf{V}^{(k)}$ and $\mathbf{H}^{(k)}$.

- Idea: choose the iterate $\boldsymbol{x}^{(k)}$ such that it minimizes the residual over the Krylov space $K^{k-1}(\mathbf{A}, \boldsymbol{r}^{(0)})$. Recall that we have a basis for the Krylov space, $K^{k-1}(\mathbf{A}, \boldsymbol{r}^{(0)}) = \text{range}(\mathbf{V}^{(k-1)})$. The iterate $\boldsymbol{x}^{(k)}$ is of the form $\boldsymbol{x}^{(0)} + \mathbf{V}^{(k-1)}\boldsymbol{y}^{(k)}$ with an unknown $k$ dimensional vector $\boldsymbol{y}^{(k)}$. Minimizing the residual can be reformulated to

$$\begin{aligned} \|\boldsymbol{r}^{(k)}\| &= \|\boldsymbol{b} - \mathbf{A}(\boldsymbol{x}^{(0)} + \mathbf{V}^{(k-1)}\boldsymbol{y}^{(k)})\| && (2.150) \\ &= \|\boldsymbol{r}^{(0)} - \mathbf{A}\mathbf{V}^{(k-1)}\boldsymbol{y}^{(k)}\| = \|\boldsymbol{r}^{(0)} - \mathbf{V}^{(k)}\mathbf{H}^{(k)}\boldsymbol{y}^{(k)}\| && (2.151) \end{aligned}$$

  Note that due to the initialization of the Arnoldi iteration, $\boldsymbol{r}^{(0)} = \|\boldsymbol{r}^{(0)}\|\boldsymbol{v}^{(0)} = \|\boldsymbol{r}^{(0)}\|\mathbf{V}^{(k)}\boldsymbol{e}_0$. Inserting this above, the minimization becomes

$$\|\boldsymbol{r}^{(k)}\| = \left\|\mathbf{V}^{(k)}\left(\|\boldsymbol{r}^{(0)}\|\boldsymbol{e}_0 - \mathbf{H}^{(k)}\boldsymbol{y}^{(k)}\right)\right\| = \left\|\left(\|\boldsymbol{r}^{(0)}\|\boldsymbol{e}_0 - \mathbf{H}^{(k)}\boldsymbol{y}^{(k)}\right)\right\| \to \min_{\boldsymbol{y}^{(k)}}. \tag{2.152}$$

  The equality above holds because rotation by $\mathbf{V}^{(k)}$ does not change the Euclidean norm of a vector. Equation (2.152) is of the form of a least squares problem which we treated applying the QR solver in Section 2.2.4.

- We compute the minimizer $\boldsymbol{y}^{(k)}$: We perform a Householder QR decomposition on $\mathbf{H}^{(k)} = \mathbf{QR}$. Then we get the minimizer by solving

$$\mathbf{R}\boldsymbol{y}^{(k)} = \|\boldsymbol{r}^{(0)}\|\mathbf{Q}^H\boldsymbol{e}_0, \tag{2.153}$$

  where, slightly abusing notation, the last line of zeros in $\mathbf{R}$ is omitted (compare the QR algorithm for overdetermined systems, Section 2.2.4). As $\mathbf{H}^{(k)}$ is of Hessenberg form, its QR decomposition can be computed within $O(k)$ manipulations.

- Compute the solution iterate $\boldsymbol{x}^{(k)}$,

$$\boldsymbol{x}^{(k)} = \boldsymbol{x}^{(0)} + \mathbf{V}^{(k-1)}\boldsymbol{y}^{(k)}. \tag{2.154}$$

The algorithm above can be used to solve general linear systems. For the CG method, we have seen that the solution is exact on $k = n$ iterations. This is also true for GMRES, if the Arnoldi iteration is carried out up to $k = n - 1$. However, the computational cost grows with $k$: the $k^{th}$ Arnoldi step is of complexity $O(kn)$, thus $k$ steps lead to $O(k^2 n)$. If one iterates up to $k = n$, the costs sum up to $O(n^3)$, which is the same as for a direct solve.

Usually, GMRES is *restarted* at $k = k_0$, meaning that the Arnoldi iteration is computed only up to $k = k_0$, and a minimizer $\boldsymbol{y}$ is generated for $\mathbf{H}^{(k_0)}$. Then, the process is restarted, another Arnoldi iteration is performed for the new residual vector.

Unlike the CG method, there are no convergence estimates for restarted GMRES for general matrices. It can even be shown that there exist systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ such that the residual stays constant up the last but one step, and drops to zero only in the last step. In practice, we see bad convergence behavior for completely unstructured systems.

Other iterative methods for general matrices are BiCGStab or MINRES. While they are less costly than (restarted) GMRES, their convergence behavior is usually even worse.

# 3 Applied computational methods

## 3.1 Eigenvalues

Consider a square matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$. We call $\lambda$ an eigenvalue and $\boldsymbol{v}$ a (right) eigenvector if

$$\mathbf{A}\boldsymbol{v} = \lambda\boldsymbol{v}. \tag{3.1}$$

Obviously, eigenvectors are not unique, but any multiple of $\boldsymbol{v}$ will be another eigenvector to the same eigenvalue as $\boldsymbol{v}$.

**Eigenvalue problem for general matrices** If $\lambda$ is an eigenvalue, it holds that $\mathbf{A}-\lambda\mathbf{I}$ is singular (as there exists a non-trivial solution $\boldsymbol{v}$ to the homogeneous equation $(\mathbf{A}-\lambda\mathbf{I})\boldsymbol{v} = \mathbf{0}$). Thus, eigenvalues can be characterized through the characteristic equation

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0. \tag{3.2}$$

As the determinant of $\mathbf{A} - \lambda\mathbf{I}$ is a polynomial of degree $n$ in $\lambda$, an $n$ by $n$ matrix has exactly $n$ eigenvalues (due to the fundamental theorem of algebra). For a real matrix, eigenvalues are either real or come in complex-conjugate pairs. For each eigenvalue, there exists at least one eigenvector. The multiplicity of this eigenvalue as the root of the characteristic equation is termed *algebraic multiplicity* of the eigenvalue.

All eigenvectors $\boldsymbol{v}_i$ to a single eigenvalue $\lambda$ span an eigenspace. The dimension of this eigenspace is called *geometric multiplicity* of the eigenvalue For a general matrix $\mathbf{A}$, the algebraic multiplicity is always greater or equal to the geometric multiplicity: this means, the eigenvectors for multiple eigenvalues are not necessarily distinct.

The eigenvalue problem (3.1) can be *shifted* by a scalar $\tau \in \mathbb{C}$,

$$\mathbf{A}\boldsymbol{v} + \tau\boldsymbol{v} = (\mathbf{A} + \tau\mathbf{I})\boldsymbol{v} = (\lambda + \tau)\boldsymbol{v}. \tag{3.3}$$

This implies that the shifted matrix $\mathbf{A} + \tau\mathbf{I}$ has the same eigenvectors $\boldsymbol{v}$ as the original matrix $\mathbf{A}$, but eigenvalues $\lambda + \tau$.

*Similarity transforms* are transformations that keep the eigenvalues, but under which eigenvectors change. Let $\mathbf{Z}$ be a regular matrix, then the eigenvalue problem (3.1) is equivalent to

$$\mathbf{Z}^{-1}\mathbf{A}\boldsymbol{v} = \lambda\mathbf{Z}^{-1}\boldsymbol{v} \qquad \text{or} \qquad \mathbf{Z}^{-1}\mathbf{A}\mathbf{Z}\,\mathbf{Z}^{-1}\boldsymbol{v} = \lambda\mathbf{Z}^{-1}\boldsymbol{v}. \tag{3.4}$$

Thus, the transformed matrix $\mathbf{Z}^{-1}\mathbf{A}\mathbf{Z}$ has the same eigenvalues as $\mathbf{A}$, but eigenvectors $\mathbf{Z}^{-1}\boldsymbol{v}$.

We collect this basic knowledge on eigenvalues:

- a square matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ has exactly $n$ eigenvalues,
- if $\mathbf{A}$ is real the eigenvalues are real or complex-conjugate pairs,
- the geometric multiplicity of an eigenvalue is always smaller or equal to its algebraic multiplicity.
- if $\mathbf{A}$ is *shifted* to $\tilde{\mathbf{A}} = \mathbf{A} + \tau\mathbf{I}$, all eigenvalues are equally shifted, while the eigenvectors are kept: the matrix $\tilde{\mathbf{A}} = \mathbf{A} + \tau\mathbf{I}$ has eigenvalues $\tilde{\lambda} = \lambda + \tau$ and eigenvectors $\tilde{\boldsymbol{v}} = \boldsymbol{v}$,
- under a *similarity transform* the eigenvalues remain unchanged: for any regular $\mathbf{Z}$, the matrix $\tilde{\mathbf{A}} = \mathbf{Z}^{-1}\mathbf{A}\mathbf{Z}$ has the same eigenvalues as $\mathbf{A}$, $\tilde{\lambda} = \lambda$, and eigenvectors $\tilde{\boldsymbol{v}} = \mathbf{Z}^{-1}\boldsymbol{v}$.

Let $\mathbf{V}$ be a matrix that is formed from all $m$ existing eigenvectors $\boldsymbol{v}_i$ column-wise, and let $\mathbf{\Lambda} = \mathrm{diag}(\lambda_1 \dots \lambda_m)$ contain the corresponding eigenvalues on the diagonal,

$$\mathbf{V} = \begin{bmatrix} \vdots & \dots & \vdots \\ \boldsymbol{v}_1 & \dots & \boldsymbol{v}_m \\ \vdots & \dots & \vdots \end{bmatrix}. \tag{3.5}$$

Then,

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{\Lambda}. \tag{3.6}$$

If each eigenvalue corresponds to a distinct eigenvector (and $n = m$ above), the matrix $\mathbf{V}$ is square and invertible, and we get

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}. \tag{3.7}$$

We call a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ *diagonizable* if there exists some regular $\mathbf{V} \in \mathbb{C}^{n\times n}$ and a diagonal matrix $\mathbf{\Lambda} = \mathrm{diag}(\lambda_1 \dots \lambda_n)$ such that

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}. \tag{3.8}$$

The eigenvalues of $\mathbf{A}$ are then given by $\lambda_1, \dots \lambda_n$ and the (right) eigenvectors are formed by the columns of $\mathbf{V}$.

Not every general $\mathbf{A}$ is diagonizable. The following statement will provide another means of finding eigenvalues of $\mathbf{A}$, even if $\mathbf{A}$ is not diagonizable.

**Schur form** For any $\mathbf{A} \in \mathbb{C}^{n\times n}$ there exists some unitary $\mathbf{U}$ such that

$$\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H \qquad \text{with } \mathbf{T} = \begin{bmatrix} \lambda_1 & * & \dots & * \\ 0 & \lambda_2 & \ddots & * \\ \vdots & \ddots & \ddots & * \\ 0 & \dots & 0 & \lambda_n \end{bmatrix}. \tag{3.9}$$

The upper right triangular matrix $\mathbf{T}$ is called *Schur form* of $\mathbf{A}$.

**Eigenvalues for symmetric/hermitian matrices** Symmetric real-valued or hermitian complex-valued matrices are an important sub-class of matrices for which we consider eigenvalue problems. For a real-symmetric or complex-hermitian matrix $\mathbf{A}$ all eigenvalues are real and have distinct mutually orthogonal eigenvectors (real eigenvectors for real $\mathbf{A}$, complex eigenvectors for complex $\mathbf{A}$).

Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be hermitian (or real symmetric), then

- all eigenvalues are real,
- for all eigenvectors, algebraic and geometric multiplicity are equal,
- the eigenvectors are mutually orthogonal, $(\boldsymbol{v}_i, \boldsymbol{v}_j) = 0$ for $i \neq j$, and scaled such that $\|\boldsymbol{v}_i\|_2 = 1$,
- $\mathbf{A}$ is diagonizable with $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^H$, where $\mathbf{V}$ contains the eigenvectors column-wise,
- Schur form and diagonalization coincide, such that $\mathbf{T} = \mathbf{\Lambda}$ and $\mathbf{V} = \mathbf{U}$.

The matrix $\mathbf{V}$ consisting of eigenvector is unitary (orthogonal), $\mathbf{V}\mathbf{V}^H = \mathbf{I}$; however, it is necessary to scale all eigenvectors to Euclidean norm $\|\boldsymbol{v}_i\|_2 = 1$!

**The generalized Eigenvalue problem** In engineering applications, often the *generalized eigenvalue problem* is of importance,

$$\mathbf{K}\boldsymbol{v} = \omega^2\mathbf{M}\boldsymbol{v}. \tag{3.10}$$

Above, $\mathbf{K}$ is e.g the stiffness matrix and $\mathbf{M}$ the mass matrix, while $\omega$ is the eigenfrequency (in rad/s). The generalized eigenvalue problem can be restored to the classical eigenvalue problem setting $\mathbf{A} = \mathbf{M}^{-1}\mathbf{K}$ and $\lambda = \omega^2$ if $\mathbf{M}$ is invertible (which is usually the case for mass matrices).

For the classical eigenvalue problem, we have stated that for hermitian/symmetric matrices $\mathbf{A}$ all eigenvalues are real and the eigenvectors are mutually orthogonal. This characteristic holds also for the generalized eigenvalue problem:

- if $\mathbf{K}$ is symmetric/hermitian and $\mathbf{M}$ is symmetric/hermitian positive definite, all generalized eigenvalues are real,
- if $\mathbf{K}$ and $\mathbf{M}$ are symmetric/hermitian positive definite, all generalized eigenvalues are real positiv,
- in both cases, all eigenvectors are $\mathbf{M}$ orthogonal, i.e.

$$(\boldsymbol{v}_i, \mathbf{M}\boldsymbol{v}_j) = 0 \quad \text{if } i \neq j \quad \text{and} \quad (\boldsymbol{v}_i, \mathbf{M}\boldsymbol{v}_i) = 1. \tag{3.11}$$

Why? The above statements cannot be deduced from the transformation to the classical eigenvalue problem $\mathbf{M}^{-1}\mathbf{K}\boldsymbol{v} = \lambda\boldsymbol{v}$. Even if $\mathbf{M}$ and $\mathbf{K}$ are symmetric/hermitian, the matrix product $\mathbf{M}^{-1}\mathbf{K}$ is not!

The following consideration provides a proof nevertheless – a purely mathematical proof which is *not* applied to practical problems. If $\mathbf{M}$ is symmetric/hermitian positive definite, it is diagonizable, $\mathbf{M} = \mathbf{U}\boldsymbol{\Lambda}_M\mathbf{U}^H$. The diagonal matrix $\boldsymbol{\Lambda}_M$ contains the classical eigenvalues $\lambda_M$ of $\mathbf{M}$, these eigenvalues are all positive. Then, $\mathbf{M}$ can be factorized in the following way,

$$\mathbf{M} = \mathbf{U}_M\mathbf{U}_M^H, \qquad \text{with } \mathbf{U}_M = \mathbf{U}\,\mathrm{diag}(\sqrt{\lambda_M}). \tag{3.12}$$

The factors $\mathbf{U}_M$ can be seen as the *matrix square root* of the positive defnite matrix $\mathbf{M}$. The matrix square root is not unique, also the Cholesky factors $\mathbf{M} = \mathbf{L}\mathbf{L}^H$ (cmp. Section 2.2.3) can be interpreted as matrix square roots.

Using the above factorization, we can rewrite the generalized eigenvalue problem (3.10) in the following way,

$$\mathbf{K}\boldsymbol{v} = \omega^2\mathbf{U}_M\mathbf{U}_M^H\boldsymbol{v}, \qquad \text{or} \qquad \underbrace{\mathbf{U}_M^{-1}\mathbf{K}\mathbf{U}_M^{-H}}_{=:\mathbf{K}_M}\underbrace{\mathbf{U}_M^H\boldsymbol{v}}_{=:\boldsymbol{w}} = \omega^2\mathbf{U}_M^H\boldsymbol{v}. \tag{3.13}$$

This results in the standard eigenvalue problem for the symmetric/hermitian matrix $\mathbf{K}_M = \mathbf{U}_M^{-1}\mathbf{K}\mathbf{U}_M^{-H}$,

$$\mathbf{K}_M\boldsymbol{w} = \omega^2\boldsymbol{w}. \tag{3.14}$$

Any eigenvalue $\omega^2$ of the above problem (3.14) is an eigenvalue of the generalized eigenvalue problem (3.10). Any eigenvector $\boldsymbol{w}$ of (3.14) corresponds to an eigenvector $\boldsymbol{v} = \mathbf{U}_M^{-H}\boldsymbol{w}$ of (3.10).

We can thus verify the statements above. As $\mathbf{K}_M$ is symmetric/hermitian, all eigenvalues are real. If $\mathbf{K}$ is positive definite, so is $\mathbf{K}_M$, and all eigenvalues are positive. Last, all eigenvectors $\boldsymbol{w}$ of (3.14) are mutually orthogonal, which proves,

$$\delta_{ij} = (\boldsymbol{w}_i, \boldsymbol{w}_j) = (\mathbf{U}_M^H\boldsymbol{v}_i, \mathbf{U}_M^H\boldsymbol{v}_j) = (\boldsymbol{v}_i, \mathbf{U}_M\mathbf{U}_M^H\boldsymbol{v}_j) = (\boldsymbol{v}_i, \mathbf{M}\boldsymbol{v}_j). \tag{3.15}$$

In practice, $\mathbf{M}$ is *not* factorized to solve the generalized eigenvalue problem! However, the proof above shows that for real matrices, using real (`double`) arithmetics is sufficient, no `complex<double>` is needed.

**Solving the eigenvalue problem** In the following, we discuss different approaches to solving eigenvalue problems. There are two different use-cases for eigenvalue solvers: the first of finding *all* eigenvalues and vectors of (relatively) small dense matrices, the second of finding a small number (e.g. the smallest or largest eigenvalues and corresponding eigenvectors) of large (sparse) systems. Where small eigenvalue problems for dense matrices are concerned, we discuss the algorithms only for educational reasons. In real-life applications, it is seriously recommended to use LAPACK implementations of eigenvalue solvers!

We discuss two different approaches to solving eigenvalue problems: the first algorithm is designed for finding all eigenvalues and eigenvectors of a dense matrix $\mathbf{A}$. It is based on the QR factorization of $\mathbf{A}$, which is repeated iteratively. Convergence of the iterative process can be enhanced by *shifting* the eigenvalue problem.

As the QR factorization is rather costly ($O(n^3)$) for generalized matrices, the above algorithm is improved using the Arnoldi iteration. In a first (preprocessing) step, matrix $\mathbf{A}$ is transformed to Hessenberg form, and the QR factorization is then carried out for this Hessenberg matrix. The computational cost for factorizing a Hessenberg matrix is only $O(n^2)$. For symmetric/hermitian matrices, instead of a Hessenberg matrix we even get a tridiagonal matrix, factorization can be carried out in $O(n)$. As a factorization has to be carried out in each iterative step, but the Arnoldi iteration is performed only once in the beginning, this is indeed an improvement.

As a second method we study *inverse iteration*, which works well for finding the smallest eigenvalues to symmetric positive definite generalized eigenvalue problems. Within the inverse iteration algorithm, solving a small dense eigenvalue problem is required ($2m \times 2m$ if $m$ eigenvalues are to be found) – where one can use the Arnoldi solver developed previously, or rely on LAPACK routines.

### 3.1.1 The QR algorithm for eigenvalue problems

The QR algorithm for solving eigenvalue problems yields a series of matrices $\mathbf{A}^{(i)}$ that are *similar* to $\mathbf{A}$ and that converge to diagonal form (hermitian $\mathbf{A}$) or Schur form (general $\mathbf{A}$).

The algorithm is as follows:

- Initialize $\mathbf{A}^{(0)} = \mathbf{A}$,
- in step $k \geq 0$, compute
  - QR factorization of $\mathbf{A}^{(k)}$, $\mathbf{A}^{(k)} = \mathbf{Q}^{(k)}\mathbf{R}^{(k)}$,
  - set $\mathbf{A}^{(k+1)} = \mathbf{R}^{(k)}\mathbf{Q}^{(k)}$.

This way, all $\mathbf{A}^{(k)}$ are *similar*:

$$\mathbf{A}^{(k+1)} = \mathbf{R}^{(k)}\mathbf{Q}^{(k)} = \mathbf{Q}^{(k),H}\mathbf{A}^{(k)}\mathbf{Q}^{(k)} \qquad \text{since } \mathbf{Q}^{(k),H}\mathbf{A}^{(k)} = \mathbf{R}^{(k)}. \tag{3.16}$$

It can be shown that the sequence $\mathbf{A}^{(k)}$ converges to the Schur form $\mathbf{T}$ of $\mathbf{A}$, such that the eigenvalues of $\mathbf{A}$ are found on the diagonal of $\mathbf{T}$.

**Complex arithmetics** A general, non-symmetric real-valued matrix $\mathbf{A}$ can have complex-valued eigenvalues and eigenvectors. Thus, for non-symmetric matrices, *complex arithmetics* have to be used for the QR iteration above also if the matrix is real-valued!

**Computational cost** For general $\mathbf{A}$, each step in the QR algorithm means computing an QR decomposition. This is in itself computationally expensive ($O(n^3)$). If $\mathbf{A}$ is of special form, e.g. if $\mathbf{A} = \mathbf{H}$ is a Hessenberg matrix, then the QR decomposition can be achieved in $O(n^2)$ steps. If $\mathbf{A}$ is tridiagonal hermitian, the QR decomposition needs only $O(n)$ steps. Therefore, in general the matrix $\mathbf{A}$ is first transformed to Hessenberg or tridiagonal form using Arnoldi's method, and then the QR algorithm is applied to the transformed matrix, see Section 3.1.1.

In any case one should take care that the QR factorization is initialized (i.e. memory is assigned to $\mathbf{Q}^{(k)}$ and $\mathbf{R}^{(k)}$) only once in the beginning of the iteration. These matrices can be overwritten in each iterative step.

**Convergence** When is the QR algorithm "converged", i.e. at which $k$ can we assume that $\mathbf{A}^{(k)}$ is sufficiently close to upper triangular or diagonal form to stop the iteration?

A sufficient criterion is to check whether the sub-diagonal entries $A^{(k)}_{i+1,i}$ are small compared to their neighboring diagonal entries $A^{(k)}_{i,i}$ and $A^{(k)}_{i+1,i+1}$,

$$|A^{(k)}_{i+1,i}| < \varepsilon(|A^{(k)}_{i,i}| + |A^{(k)}_{i+1,i+1}|), \tag{3.17}$$

for some accuracy $\varepsilon > 0$.

The following estimate tells how fast the sub-diagonal entries converge to zero,

$$|A^{(k)}_{i+1,i}| \leq c \left| \frac{\lambda_{i+1}}{\lambda_i} \right|^k. \tag{3.18}$$

This means, the convergence is faster if eigenvalues are far from each other in absolute value. If two eigenvalues have the same absolute value, the above estimate does not guarantee convergence. With some additional effort, one can show that

- the eigenvalues are ordered by their absolute value, starting from the largest eigenvalue at position $A^{(k)}_{0,0}$ to the smallest at position $A^{(i)}_{n-1,n-1}$,
- if two eigenvalues are close in absolute value, the algorithm will in general converge poorly,
- the algorithm converges also for multiple eigenvalues $\lambda_i = \cdots = \lambda_j$, but in an arbitrarily slow manner,
- if $\lambda_i = -\lambda_j$ the algorithm does not converge,
- if the matrix $\mathbf{A}$ is real non-symmetric, complex-conjugate eigenvalues $\lambda_i, \bar{\lambda}_i$ have the same absolute value; no convergence can be achieved when using real arithmetics,
- in any of the above cases, a *shift strategy* can help to ensure (faster) convergence.

**Computing the eigenvectors** The algorithm discussed above yields all eigenvalues of the matrix $\mathbf{A}$. For a *symmetric/hermitian* matrix $\mathbf{A}$, the eigenvectors can be computed using additional book-keeping. In the symmetric/hermitian case, the QR iteration transforms $\mathbf{A}$ into a diagonal matrix $\mathbf{T}$, which contains the eigenvalues on the diagonal. Consequently, the eigenvectors are found if one multiplies all orthogonal transformations $\mathbf{Q}$ found throughout the whole process:

$$\mathbf{T} = \lim_{k\to\infty} \mathbf{A}^{(k)} = \lim_{k\to\infty} \mathbf{Q}^{(k),H}\mathbf{Q}^{(k-1),H}\cdots\mathbf{Q}^{(0),H}\mathbf{A}\mathbf{Q}^{(0)}\cdots\mathbf{Q}^{(k-1)}\mathbf{Q}^{(k)} \tag{3.19}$$

The unitary matrix $\mathbf{U}$ defining the Schur form $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H$ is therefore approximated by the sequence

$$\mathbf{U}^{(k)} = \mathbf{Q}^{(0)}\cdots\mathbf{Q}^{(k-1)}\mathbf{Q}^{(k)}. \tag{3.20}$$

For $\mathbf{A} = \mathbf{A}^H$ real symmetric/hermitian, $\mathbf{U}^{(k)}$ contains the eigenvectors column-wise as the algorithm converges. In each step, $\mathbf{U}^{(k)}$ is updated by computing the matrix product with $\mathbf{Q}^{(k)}$, which can be done in-place for the Householder reflections in the QR factorization.

If $\mathbf{A}$ is a general matrix, $\mathbf{T}$ is an upper triangular matrix. The eigenvalues are found as the diagonal entries. If they exist, the eigenvectors of $\mathbf{T}$ can be computed easily in consecutive order (staggered system) for a tridiagonal matrix

- the first (right) eigenvector of $\mathbf{T}$ is the first unit vector, $\boldsymbol{w}_0 = \boldsymbol{e}_0$,
- the second (right) eigenvector of $\mathbf{T}$ is found solving

  $$\mathbf{T}(\alpha_0\boldsymbol{e}_0 + \boldsymbol{e}_1) = \lambda_1(\alpha_0\boldsymbol{e}_0 + \boldsymbol{e}_1) \tag{3.21}$$

  then $\boldsymbol{v}_1 = \alpha_0\boldsymbol{e}_0 + \boldsymbol{e}_1$ is normalized and forms the second eigenvector.
- and so on, compute the $j^{th}$ eigenvector $\boldsymbol{w}_j$ as a linear combination of $\boldsymbol{e}_i, i < j$ and $\boldsymbol{e}_j$.
- if the geometric multiplicity of an eigenvalue is smaller than its algebraic multiplicity, the according equation above is not solvable.

Collecting the eigenvectors of the Schur matrix $\mathbf{T}$ in $\mathbf{W}$ leads to the decomposition

$$\mathbf{W}\mathbf{\Lambda} = \mathbf{T}\mathbf{W} = \lim_{k\to\infty} \mathbf{U}^{(k),H}\mathbf{A}\mathbf{U}^{(k)}\mathbf{W}. \tag{3.22}$$

The matrix $\mathbf{V} = \mathbf{U}^{(k)}\mathbf{W}$ contains the eigenvectors of $\mathbf{A}$. If all eigenvectors are linearly independent, $\mathbf{W}$ is invertible, and so is $\mathbf{V}$, and we have

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}. \tag{3.23}$$

**Improving the QR algorithm—the Arnoldi iteration revisited** Recall the Arnoldi iteration from Section 2.4.3. In the $k$th step, it produced a factorization

$$\mathbf{A}\mathbf{V}^{(k-1)} = \mathbf{V}^{(k)}\mathbf{H}^{(k)}, \tag{3.24}$$

where $\mathbf{H}^{(k)}$ is a $(k+1) \times (k)$ Hessenberg matrix, and $\mathbf{V}^{(k-1)}, \mathbf{V}^{(k)}$ collect the orthogonal vectors from the Gram-Schmidt process. If we go all the way to the $n$th step, $\mathbf{V}^{(n-1)}$ is an orthogonal square $n \times n$ matrix, and $\mathbf{V}^{(n)}$ contains one additional column consisting only of zeros. $H^{(n)}$ is an $n+1 \times n$ Hessenberg matrix with only zeros in the last line. Therefore, the last column of $V^{(n)}$ and last row of $H^{(n)}$ can be eliminated, we get square matrices only, and see

$$\mathbf{A}\mathbf{V}^{(n-1)} = \mathbf{V}^{(n-1)}\mathbf{H}^{(n)}, \qquad \text{or} \qquad \mathbf{A} = \mathbf{V}^{(n-1)}\mathbf{H}^{(n)}\mathbf{V}^{(n-1),H}. \tag{3.25}$$

As this is a similarity transform of $\mathbf{A}$, the Hessenberg matrix $\mathbf{H}^{(n)}$ (reduced to $n \times n$ as stated above) has the same eigenvalues as $\mathbf{A}$. If we find eigenvalues $\lambda$ and eigenvectors $\boldsymbol{w}$ of $\mathbf{H}^{(n)}$, we immediately have that $\lambda$ are also eigenvalues of $\mathbf{A}$. Eigenvectors of $\mathbf{A}$ are then $\boldsymbol{v} = \mathbf{V}^{(n-1)}\boldsymbol{w}$.

Once the Hessenberg matrix $\mathbf{H}^{(n)}$ has been computed, its eigenvalues can be computed using the QR algorithm from Section 3.1.1. To avoid to many indices, we use $\mathbf{H} = \mathbf{H}^{(n)}$, such that we are now concerned with finding eigenvalues of $\mathbf{H}$. There, one makes use of the fact that the QR decomposition

$$\mathbf{H} = \mathbf{Q}^{(0)}\mathbf{R}^{(0)} \tag{3.26}$$

can be computed within $O(n^2)$ computational steps. Why? Recall that in the QR factorization (Section 2.2.4), in each iterative step a Householder vector $\boldsymbol{w}$ was computed, such that multiplication with $\mathbf{P} = \mathbf{I} - 2\boldsymbol{w}\boldsymbol{w}^H$ zeroes all sub-diagonal entries of the matrix to be factorized. For a Hessenberg matrix $\mathbf{H}$, one finds that each Householder vector $\boldsymbol{w}$ has only two non-zero entries, namely in step $k$, $w_k$ and $w_{k+1}$, as only one sub-diagonal entry of $\mathbf{H}$ needs to be zeroed.

But will the next iterate

$$\mathbf{H}^{(1)} = \mathbf{R}^{(0)}\mathbf{Q}^{(0)} = \mathbf{Q}^{(0),H}\mathbf{H}^{(0)}\mathbf{Q}^{(0)} \tag{3.27}$$

still be of Hessenberg form? This is indeed the case, which one can again derive from the special form of the householder vectors $\boldsymbol{w}$.

The computational complexity is even lower if $\mathbf{H}$ is tridiagonal, stemming from a symmetric/hermitian matrix $\mathbf{A}$.

**QR algorithm with single shift** The QR algorithm is modified in the following way (where we use $\mathbf{A}$ in notation, but of course application to the Hessenberg form $\mathbf{H}$ from Arnoldi's iteraition is preferred, replacing $\mathbf{A}$ by $\mathbf{H}$):

- Initialize $\mathbf{A}^{(0)} = \mathbf{A}$,
- in step $k \geq 0$, choose $\mu_k$ and compute
  - QR factorization of $\mathbf{A}^{(k)} - \mu_k\mathbf{I}$, $\mathbf{A}^{(k)} - \mu_k\mathbf{I} = \mathbf{Q}^{(k)}\mathbf{R}^{(k)}$,
  - set $\mathbf{A}^{(k+1)} = \mathbf{R}^{(k)}\mathbf{Q}^{(k)} + \mu_k\mathbf{I}$.

Then, for $\mu_k = \mu$ convergence is controlled by

$$|A^{(k)}_{i+1,i}| \leq c \left| \frac{\lambda_{i+1} - \mu}{\lambda_i - \mu} \right|^k . \tag{3.28}$$

For two eigenvalues of same absolute value (but different value) convergence can be improved. A general choice is to use the last diagonal entry $\mu_k = A^{(k)}_{n-1,n-1}$ to get fast convergence of the last sub-diagonal entry (then the enumerator $\lambda_{n-1} - \mu_k$ is small). On convergence, the size of the matrix can be restricted to the upper left $(n-1) \times (n-1)$ sub-matrix and one uses $\mu_k = A^{(k)}_{n-2,n-2}$ as a shift parameter, and so forth.

For real-valued non-symmetric matrices, complex eigenvalues always appear in complex-conjugate pairs $\lambda$ and $\bar{\lambda}$ with same absolute value. In this case, a shift strategy is essential. However, choosing $\mu$ real will still not work, as still the absolute values of shifted $\lambda - \mu$ and $\bar{\lambda} - \mu$ are equal. One has to use complex arithmethics throughout the algorithm, and complex shifts $\mu \in \mathbb{C}$. Then, (fast) convergence is obtained at the cost of complex instead of real numbers computations. *Double shift strategies* using only real arithmetics have been designed for this case.

### 3.1.2 Inverse iteration for real-symmetric matrices

The inverse iteration has been initially designed to compute resonance frequencies in structural mechanics. We present the inverse iteration for computing the smallest eigenvalue $\lambda = \omega^2$ for the generalized eigenvalue problem,

$$\mathbf{K}\boldsymbol{v} = \omega^2 \mathbf{M}\boldsymbol{v}, \tag{3.29}$$

where $\mathbf{K}$ is symmetric and invertible and $\mathbf{M}$ is symmetric positive definite. Singular $\mathbf{K}$ with zero eigenvalues can be treated by a shift strategy, where $\mathbf{K} - \mu\mathbf{M}$ replaces $\mathbf{K}$, and all eigenvalues are shifted by $\mu$:

$$(\mathbf{K} - \mu\mathbf{M})\boldsymbol{v} = \lambda^* \mathbf{M}\boldsymbol{v}, \tag{3.30}$$

and $\lambda^* = \lambda - \mu$.

For convenient notation, we assume the eigenvalues are ordered with respect to their absolute value, such that $\lambda_1$ is of *smallest* absolute value,

$$|\lambda_1| \leq |\lambda_2| \leq \ldots, \quad \text{or } |\lambda_i| \leq |\lambda_j| \text{ for all } i < j. \tag{3.31}$$

When computing resonance frequencies, $\lambda_1 = \omega_1^2$ will be positive, but this is not necessarily so for the shifted or any general real-symmetric eigenvalue problem.

Assume some initial guess $\boldsymbol{v}^{(0)}$ for the eigenvector $\boldsymbol{v}_1$ corresponding to the absolute smallest eigenvalue $\lambda_1$ is given (if no such starting vector is known, some random vector can be used, a zero-vector $\boldsymbol{v}^{(0)} = \mathbf{0}$ does not work). Assume that $\boldsymbol{v}^{(0)}$ is not orthogonal to $\boldsymbol{v}_1$, i.e. $(\boldsymbol{v}^{(0)}, \boldsymbol{v}_1) \neq 0$. This is most probably the case for a random starting vector, but not for the zero vector $\boldsymbol{v}^{(0)} = \mathbf{0}$, and possibly

also not for $\boldsymbol{v}^{(0)}$ some constant vector. Based on the starting vector, a sequence of iterates is generated, which can be shown to converge to the eigenvector $\boldsymbol{v}_1$ for the eigenvalue of smallest absolute value,

$$\boldsymbol{z}^{(k+1)} = \mathbf{K}^{-1}\mathbf{M}\boldsymbol{v}^{(k)}, \qquad \boldsymbol{v}^{(k+1)} = \frac{\boldsymbol{z}^{(k+1)}}{\|\boldsymbol{z}^{(k+1)}\|} \to \boldsymbol{v}_1. \tag{3.32}$$

How do we get an estimate for the eigenvalue $\lambda$? The Rayleigh quotient can be used to compute the $k^{th}$ iterate,

$$\lambda^{(k)} := \frac{(\boldsymbol{v}^{(k)}, \mathbf{K}\boldsymbol{v}^{(k)})}{(\boldsymbol{v}^{(k)}, \mathbf{M}\boldsymbol{v}^{(k)})}. \tag{3.33}$$

The *residual* of the eigenvalue problem can be used to estimate the error,

$$\boldsymbol{r}^{(k)} := \mathbf{K}\boldsymbol{v}^{(k)} - \lambda^{(k)}\mathbf{M}\boldsymbol{v}^{(k)}. \tag{3.34}$$

Convergence $\boldsymbol{v}^{(k)} \to \boldsymbol{v}_1$ can be shown by the following consideration: For real-symmetric $\mathbf{K}$ and $\mathbf{M}$, the eigenvectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ are linear independent, and form a basis of $\mathbb{R}^n$. Thus, $\boldsymbol{v}^{(0)}$ can be represented in this basis,

$$\boldsymbol{v}^{(0)} = \sum_{i=1}^{n} x_i \boldsymbol{v}_i. \tag{3.35}$$

Moreover, we have that the eigenvectors satisfy the generalized eigenvalue problem,

$$\mathbf{K}\boldsymbol{v}_i = \lambda_i \mathbf{M}\boldsymbol{v}_i, \quad \text{and so} \quad \frac{1}{\lambda_i}\boldsymbol{v}_i = \mathbf{K}^{-1}\mathbf{M}\boldsymbol{v}_i. \tag{3.36}$$

In iteration $k$, recall $\boldsymbol{z}^{(k+1)}$ is the un-scaled iterate, from which we get $\boldsymbol{v}^{(k+1)}$ by normalization,

$$\boldsymbol{z}^{(k+1)} = (\mathbf{K}^{-1}\mathbf{M})\boldsymbol{v}^{(k)} = (\mathbf{K}^{-1}\mathbf{M})^{k+1}\boldsymbol{v}^{(0)} \qquad \text{and} \qquad \boldsymbol{v}^{(k+1)} = \frac{1}{\|\boldsymbol{z}^{(k+1)}\|}\boldsymbol{z}^{(k+1)}. \tag{3.37}$$

If we use the basis representation (3.35) in the iteration, we get

$$\boldsymbol{z}^{(k+1)} = \sum_{i=1}^{n} x_i (\mathbf{K}^{-1}\mathbf{M})^{k+1}\boldsymbol{v}_i = \sum_{i=1}^{n} x_i \left(\frac{1}{\lambda_i}\right)^{k+1} \boldsymbol{v}_i \tag{3.38}$$

$$= \frac{x_1}{\lambda_1^{k+1}} \underbrace{\left(\boldsymbol{v}_1 + \sum_{i=2}^{n} \frac{x_i}{x_1}\left(\frac{\lambda_1}{\lambda_i}\right)^{k+1} \boldsymbol{v}_i\right)}_{\text{next iterate to be scaled}}. \tag{3.39}$$

After normalization, the constant factor $x_1/\lambda_1^{k+1}$ will be dropped, therefore it is essential to see the limit value of the remaining vector $\boldsymbol{z}^{(k+1)}$, and whether it converges – in terms of direction – towards $\boldsymbol{v}_1$. We see that, for each of the finitely many $i = 2 \ldots n$,

$$\frac{x_i}{x_1}\left(\frac{\lambda_1}{\lambda_i}\right)^{k+1} \overset{k\to\infty}{\longrightarrow} 0 \quad \text{since} \quad \left|\frac{\lambda_1}{\lambda_i}\right| < 1, \tag{3.40}$$

as we assumed $\lambda_1$ to be the eigenvalue of smallest absolute value. Thus,

$$\boldsymbol{z}^{(k)} \to \boldsymbol{v}_1 \text{ at rate } \left(\frac{\lambda_1}{\lambda_2}\right)^k. \tag{3.41}$$

The convergence is better the farther the first two eigenvalues are apart. If $\lambda_1$ is a multiple eigenvalue, the algorithm computes *a linear combination of the according eigenvectors, which is again an eigenvector*, depending on the starting value $\boldsymbol{v}^{(0)}$. Convergence is controlled by the distance to the next different eigenvalue.

Note that in each step, a linear system of equations has to be solved:

$$\mathbf{K}\boldsymbol{z}^{(k+1)} = \mathbf{M}\boldsymbol{v}^{(k)}. \tag{3.42}$$

However, the system matrix is constant, thus any factorization (LU or QR) has to be computed only once, and is applied afterwards.

**Power iteration** Originally, the inverse to the above method has been introduced before the inverse method above. It is known as *power iteration*, and yields new iterates by

$$\boldsymbol{z}^{(k+1)} = \mathbf{M}^{-1}\mathbf{K}\boldsymbol{v}^{(k)}, \qquad \boldsymbol{v}^{(k+1)} = \frac{1}{\|\boldsymbol{z}^{(k+1)}\|}\boldsymbol{z}^{(k+1)}. \tag{3.43}$$

This sequence converges to the eigenvector corresponding to the eigenvalue with *largest* absolute value. It is especially convenient if the classical eigenvalue problem with $\mathbf{M} = \mathbf{I}$ and $\mathbf{K} = \mathbf{A}$ is considered, in this case only *applications* of $\mathbf{A}$ are necessary, and not solving linear systems.

In engineering applications though, usually the smallest eigenvalues are of interest.

**Acceleration using the Rayleigh quotient** At convergence, the iterates satisfy the generalized eigenvalue problem,

$$\mathbf{K}\boldsymbol{v}^{(k+1)} = \lambda_1\mathbf{M}\boldsymbol{v}^{(k)} \qquad \text{which implies} \qquad \mathbf{K}(\boldsymbol{v}^{(k+1)} - \boldsymbol{v}^{(k)}) = (\lambda_1\mathbf{M} - \mathbf{K})\boldsymbol{v}^{(k)}. \tag{3.44}$$

This motivates the following, slightly different iteration

$$\boldsymbol{z}^{(k+1)} = \boldsymbol{v}^{(k)} + \alpha \underbrace{\mathbf{K}^{-1}(\lambda^{(k)}\mathbf{M} - \mathbf{K})\boldsymbol{v}^{(k)}}_{=:\boldsymbol{w}^{(k)}} \qquad \text{with } \lambda^{(k)} = \frac{(\mathbf{K}\boldsymbol{v}^{(k)}, \boldsymbol{v}^{(k)})}{(\mathbf{M}\boldsymbol{v}^{(k)}, \boldsymbol{v}^{(k)})}, \tag{3.45}$$

and subsequently setting $\boldsymbol{v}^{(k+1)} = 1/\|\boldsymbol{z}^{(k+1)}\|\boldsymbol{z}^{(k+1)}$. Above,

$$\boldsymbol{w}^{(k)} = \lambda^{(k)}\mathbf{K}^{-1}\mathbf{M}\boldsymbol{v}^{(k)} - \boldsymbol{v}^{(k)}, \tag{3.46}$$

can be seen as a search direction for the update, and $\alpha$ the step size parameter. The parameter $\lambda^{(k)}$ will approximate $\lambda_1$, as it is chosen as the Rayleigh coefficient. The step size $\alpha$ is chosen such that the next Rayleigh quotient $\lambda^{(k+1)}$ is minimized,

$$\alpha = \arg\min \frac{\big(\mathbf{K}(\boldsymbol{v}^{(k)} + \alpha\boldsymbol{w}^{(k)}), \boldsymbol{v}^{(k)} + \alpha\boldsymbol{w}^{(k)}\big)}{\big(\mathbf{M}(\boldsymbol{v}^{(k)} + \alpha\boldsymbol{w}^{(k)}), \boldsymbol{v}^{(k)} + \alpha\boldsymbol{w}^{(k)}\big)}. \tag{3.47}$$

To be exact, *minimizing* the Rayleigh quotient will lead to finding the *smallest* generalized eigenvalue, which is possibly negative. If one wants to find generalized eigenvalues of *minimal absolute value*, the absolute value of the Rayleigh quotient is minimized above.

Equation (3.47) translates into solving the eigenvalue problem on the two-dimensional sub-space $\mathrm{lin}(\boldsymbol{v}^{(k)}, \boldsymbol{w}^{(k)})$, and choosing the smaller of the two eigenvalues: We observe

$$\big(\mathbf{K}(y_1\boldsymbol{v}^{(k)} + y_2\boldsymbol{w}^{(k)}), y_1\boldsymbol{v}^{(k)} + y_2\boldsymbol{w}^{(k)}\big) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}^T \underbrace{\begin{bmatrix} (\mathbf{K}\boldsymbol{v}^{(k)}, \boldsymbol{v}^{(k)}) & (\mathbf{K}\boldsymbol{v}^{(k)}, \boldsymbol{w}^{(k)}) \\ (\mathbf{K}\boldsymbol{w}^{(k)}, \boldsymbol{v}^{(k)}) & (\mathbf{K}\boldsymbol{w}^{(k)}, \boldsymbol{w}^{(k)}) \end{bmatrix}}_{=:\mathbf{K}_2} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{3.48}$$

$$\big(\mathbf{M}(y_1\boldsymbol{v}^{(k)} + y_2\boldsymbol{w}^{(k)}), y_1\boldsymbol{v}^{(k)} + y_2\boldsymbol{w}^{(k)}\big) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}^T \underbrace{\begin{bmatrix} (\mathbf{M}\boldsymbol{v}^{(k)}, \boldsymbol{v}^{(k)}) & (\mathbf{M}\boldsymbol{v}^{(k)}, \boldsymbol{w}^{(k)}) \\ (\mathbf{M}\boldsymbol{w}^{(k)}, \boldsymbol{v}^{(k)}) & (\mathbf{M}\boldsymbol{w}^{(k)}, \boldsymbol{w}^{(k)}) \end{bmatrix}}_{=:\mathbf{M}_2} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}. \tag{3.49}$$

Let now $\rho$ be the smallest eigenvalue and $\boldsymbol{y} = [y_1, y_2]^T$ be the corresponding eigenvector of the $2 \times 2$ problem

$$\mathbf{K}_2\boldsymbol{y} = \rho\mathbf{M}_2\boldsymbol{y}. \tag{3.50}$$

Then, the new iterate minimizes the Rayleigh quotient,

$$\boldsymbol{z}^{(k+1)} = y_1\boldsymbol{v}^{(k)} + y_2\boldsymbol{w}^{(k)} \qquad \text{and} \qquad \frac{\big(\mathbf{K}\boldsymbol{z}^{(k+1)}, \boldsymbol{z}^{(k+1)}\big)}{\big(\mathbf{M}\boldsymbol{z}^{(k+1)}, \boldsymbol{z}^{(k+1)}\big)} \to \min. \tag{3.51}$$

If one wants to find the generalized eigenvalue of smallest absolute value, choose $\rho$ as the eigenvalue of smaller absolute value.

We observe that in each step, it is necessary to solve a linear system with constant system matrix $\mathbf{K}$, and to solve a $2 \times 2$ eigenvalue problem exactly. The latter solve can be implemented explicitely for symmetric $2 \times 2$ matrices.

A note on preconditioning: in case a preconditioner $\mathbf{P}^{-1}$ for $\mathbf{K}$ is available, instead of using the update direction $\boldsymbol{w}^{(k)} = \mathbf{K}^{-1}(\lambda^{(k)}\mathbf{M} - \mathbf{K})\boldsymbol{v}^{(k)}$, one can use $\boldsymbol{w}^{(k)} = \mathbf{P}^{-1}(\lambda^{(k)}\mathbf{M} - \mathbf{K})\boldsymbol{v}^{(k)}$. If the preconditioner approximates $\mathbf{K}^{-1}$ well, the number of necessary iterations will not grow much, however it is much cheaper in application for large systems.

The Rayleigh method can be adjusted to find $m$ smallest eigenvalues at once. Then, one uses $m$ different eigenvector iterates $\boldsymbol{v}_i^{(k)}, i = 1 \dots m$ and $m$ update directions $\boldsymbol{w}_i^{(k)}$. The generalized eigenvalue problem is then of size $2m \times 2m$, containing all inner products of the form $(\mathbf{K}\boldsymbol{v}_i^{(k)}, \boldsymbol{v}_i^{(k)})$ etc. New iterates $\boldsymbol{v}_i^{(k+1)}$ are generated using the $m$ (absolutely) smallest eigenvalues and corresponding eigenvectors of the $2m \times 2m$ problem. Attention – the number of computed eigenvectors should be significantly smaller than the system size. If $m > n/2$, the system size of the *small* problem is larger than the size of the original problem! Then, the small (indeed, larger) mass matrix is singular, and the eigenvalue solver might fail.

Shifting the problem to $\mathbf{K} - \mu\mathbf{M}$ can be used to find eigenvectors and eigenvalues close to $\mu$. Then it is important to use the eigenvalues/eigenvectors of smallest *absolute* value for the Rayleigh quotient.

### 3.1.3 Example problem

We consider the wave equation on the unit square $[0, 1]^2$, with $c$ the propagation speed,

$$-c^2\Delta u + \ddot{u} = 0, \qquad u = 0 \text{ on the boundary.} \tag{3.52}$$

A transformation to frequency domain, using the ansatz $u(x)e^{i\omega t}$ leads to the Helmholtz equation

$$-c^2\Delta u = \omega^2 u, \qquad u = 0 \text{ on the boundary.} \tag{3.53}$$

Exact solutions to this model problem are known, eigenfunctions (also called eigenmodes, *Eigenmoden, Schwingungsformen*) and eigenvalues/eigenfrequencies are given by

$$u_{kl} = \sin(k\pi x)\sin(l\pi y), \qquad \lambda_{kl} = \omega_{kl}^2 = c^2(k^2 + l^2)\pi^2. \tag{3.54}$$

A finite difference scheme on an equidistant grid similar to the scheme presented for the diffusion equation in Section 2.3 leads to an eigenvalue problem of the form

$$\mathbf{K}\boldsymbol{v} = \omega^2\boldsymbol{v}. \tag{3.55}$$

The stiffness matrix $\mathbf{K}$ is equivalent to the matrix derived in Section 2.3, exchanging $\kappa$ by $c^2$.

Note: in the more general case, when e.g. the density (and thereby the propagation speed $c$) is not constant on the whole domain, or when finite elements are used instead of finite differences, the mass matrix $\mathbf{M}$ enters, and we find a generalized eigenvalue problem.

In a first model script, we compute *all* eigenvalues on a very coarse grid, and compare the obtained values to the analytical values (3.54). We can guess that the lowest eigenvalues are found at the expected frequencies, but higher eigenvalues are probably unphysical.

In a second attempt, we compute eigenvectors (i.e., eigenfunctions) for selected eigenvalues using inverse iteration. As expected, eigenfunctions for small eigenvalues (e.g. $\lambda_{1,1} = 19.7392, \lambda_{1,2} = 49.348$ are plausible, while eigenfunctions for larger eigenvalues (e.g. $\lambda \simeq 400$) are not. Inverse iteration converges fast for single eigenvalues (such as $\lambda_{1,1} = 19.7392$), and also for double eigenvalues (such as $\lambda_{1,2} = \lambda_{2,1} = 49.348$). In each case one eigenfunction is computed. When computing several eigenvalues at once, two ($\mathbf{M}$-)orthogonal eigenvectors are found for double eigenvalues. Indeed, all eigenvectors are orthogonal, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$.

## 3.2 Nonlinear problems

In the present section, we are concerned with systems of nonlinear equations. We assume the nonlinear system to be of general form

$$\boldsymbol{F}(\boldsymbol{x}) = 0, \tag{3.56}$$

with solution $\boldsymbol{x} \in \mathbb{R}^n$ and nonlinear residual $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$. Thus, in general we assume that there are $n$ equations for $n$ unknowns.

In general, it is neither clear whether a solution to (3.56) exists, nor that it is unique. We will not try to prove either existence or uniqueness results, but rather design iterative methods that converge towards a solution in case it exists, if the starting point is close enough.

### 3.2.1 Fixed point iteration

Fixed point iteration is a popular method for finding solutions for problems where not much is known about the structure of the equations, i.e. about $\boldsymbol{F}$ as a function. To this end, we assume that the nonlinear system of equations is given in fixed point form,

$$\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{x}. \tag{3.57}$$

This can always be achieved e.g. by setting

$$\boldsymbol{\phi}(\boldsymbol{x}) := \boldsymbol{x} - \mathbf{P}^{-1}\boldsymbol{F}(\boldsymbol{x}), \tag{3.58}$$

with an appropriate scaling tensor $\mathbf{P}^{-1}$. We will see that $\mathbf{P}^{-1}$ is closely linked to the preconditioner treated in Section 2.4.

For a fixed point equation of form (3.57), the fixed point iteration can be defined as follows,

- choose some starting value $\boldsymbol{x}^{(0)}$,
- in step $k$, set $\boldsymbol{x}^{(k+1)} = \boldsymbol{\phi}(\boldsymbol{x}^{(k)})$.

Banach's fixed point theorem can be used to show that the iteration converges, given the fixed point operator $\boldsymbol{\phi}$ is contractive on some subdomain $D \subset \mathbb{R}^n$. In this case, the solution is even unique in $D$ – however, the solution is not necessarily unique in all of $\mathbb{R}^n$.

**Theorem 1** (Banach's fixed point theorem)**.** *Let $D \subset \mathbb{R}^n$ such that $\boldsymbol{\phi} : D \to D$ maps $D$ to itself. Let moreover $\boldsymbol{\phi}$ be a contraction, such that for all $x, y \in D$,*

$$\|\boldsymbol{\phi}(\boldsymbol{x}) - \boldsymbol{\phi}(\boldsymbol{y})\| \leq \theta\|\boldsymbol{x} - \boldsymbol{y}\| \quad \textit{for some } \theta < 1. \tag{3.59}$$

*Then there exists exactly one fixed point $\boldsymbol{x}^* \in D$ with $\boldsymbol{\phi}(\boldsymbol{x}^*) = \boldsymbol{x}^*$, and the fixed point iteration converges to $\boldsymbol{x}^*$ for any starting value $\boldsymbol{x}^{(0)} \in D$.*

This implies that, if $\mathbf{P}^{-1}$ is chosen such that

$$\|\boldsymbol{x} - \boldsymbol{y} - \mathbf{P}^{-1}(\boldsymbol{F}(\boldsymbol{x}) - \boldsymbol{F}(\boldsymbol{y}))\| \leq \theta \|\boldsymbol{x} - \boldsymbol{y}\|, \tag{3.60}$$

and $\theta < 1$, the fixed point iteration converges. We used this theorem for designing iterative methods for linear problems of the form $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. In this case, the nonlinear operator $\boldsymbol{F}$ and the fixed point operator $\boldsymbol{\phi}$ are given as

$$\boldsymbol{F}(\boldsymbol{x}) := \mathbf{A}\boldsymbol{x} - \boldsymbol{b} = 0 \quad \text{and} \quad \boldsymbol{\phi}(\boldsymbol{x}) := \boldsymbol{x} - \mathbf{P}^{-1}(\mathbf{A}\boldsymbol{x} - \boldsymbol{b}). \tag{3.61}$$

For this $\boldsymbol{F}$ or $\boldsymbol{\phi}$, the fixed point iteration reads

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \mathbf{P}^{-1}(\mathbf{A}\boldsymbol{x}^{(k)} - \boldsymbol{b}), \tag{3.62}$$

and was treated in Section 2.4. Recall that we could show convergence if, when using a compatible matrix norm,

$$\|\boldsymbol{\phi}(\boldsymbol{x}) - \boldsymbol{\phi}(\boldsymbol{y})\| = \|(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A})(\boldsymbol{x} - \boldsymbol{y})\| \leq \underbrace{\|(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A})\|}_{=\theta<1} \|(\boldsymbol{x} - \boldsymbol{y})\|. \tag{3.63}$$

This reduced to $\sigma_{max}(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}) < 1$. If a general nonlinear problem is differentiable such that $\boldsymbol{F}'$ exists in the whole subdomain $D$, the condition is replaced by

$$\sigma_{max}(\mathbf{I} - \mathbf{P}^{-1}\boldsymbol{F}'(\boldsymbol{x})) \leq \theta < 1 \qquad \text{for all } \boldsymbol{x} \in D. \tag{3.64}$$

Characteristics of the fixed point iteration are

- the fixed point iteration converges linearly, in each step the error drops at least by a constant factor

  $$\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^*\| \leq \theta \|\boldsymbol{x}^{(k)} - \boldsymbol{x}^*\|. \tag{3.65}$$

  The number of converged digits grows linearly with the number of steps.

- no knowledge about differentiability, or no implementation of $\boldsymbol{F}'$ is necessary.

### 3.2.2 Newton's method

For defining Newton's method, let us assume that $\boldsymbol{F}$ is continuously differentiable. Then, in each step, the nonlinear problem is linearized around $\boldsymbol{x}^{(k)}$ and solved to obtain the next iterate,

$$\boldsymbol{F}(\boldsymbol{x}^{(k)}) + \boldsymbol{F}'(\boldsymbol{x}^{(k)})(\boldsymbol{x} - \boldsymbol{x}^{(k)}) = 0. \tag{3.66}$$

This results in the following algorithm,

- choose some starting value $\boldsymbol{x}^{(0)}$,
- in step $k$, set

  $$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - (\boldsymbol{F}'(\boldsymbol{x}^{(k)}))^{-1}\boldsymbol{F}(\boldsymbol{x}^{(k)}). \tag{3.67}$$

The $k$th update is defined as $\delta \boldsymbol{x}^{(k)} = -(\boldsymbol{F}'(\boldsymbol{x}^{(k)}))^{-1}\boldsymbol{F}(\boldsymbol{x}^{(k)})$.

We only state that there are conditions on the starting value $\boldsymbol{x}^{(0)}$ and on $\boldsymbol{F}$ itself that ensure

- the Jacobian $\boldsymbol{F}'(\boldsymbol{x}^{(k)})$ is invertible throughout the iteration,
- the iteration converges towards some solution $\boldsymbol{x}^*$,
- if close enough, it converges quadratically,

$$\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^*\| \leq \theta \|\boldsymbol{x}^{(k)} - \boldsymbol{x}^*\|^2. \tag{3.68}$$

In practice, it is probably out of question to check these conditions, but rather see whether the iteration converges or not.

### Characteristics of Newton's method

- Given some conditions, Newton's method converges locally quadratically,

$$\|\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^*\| \leq \theta \|\boldsymbol{x}^{(k)} - \boldsymbol{x}^*\|^2. \tag{3.69}$$

  This means that close to the solution, the number of converged digits doubles within a fixed number of iterations.

- Newton's method is *affine invariant*, it does not change if equations are rescaled or recombined. Let $\mathbf{A}$ be an (invertible) transformation matrix for the equations, and define $\boldsymbol{G}(\boldsymbol{x}) := \mathbf{A}\boldsymbol{F}(\boldsymbol{x})$. Then, Newton's iteration for $\boldsymbol{G}$ is

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &= \boldsymbol{x}^{(k)} - (\boldsymbol{G}')^{-1}\boldsymbol{G}(\boldsymbol{x}^{(k)}) = \boldsymbol{x}^{(k)} - (\mathbf{A}\boldsymbol{F}')^{-1}\mathbf{A}\boldsymbol{F}(\boldsymbol{x}^{(k)}) && (3.70)\\ &= \boldsymbol{x}^{(k)} - (\boldsymbol{F}')^{-1}\mathbf{A}^{-1}\mathbf{A}\boldsymbol{F}(\boldsymbol{x}^{(k)}) = \boldsymbol{x}^{(k)} - (\boldsymbol{F}')^{-1}\boldsymbol{F}(\boldsymbol{x}^{(k)}). && (3.71) \end{aligned}$$

**Making Newton's method converge** In case the initial guess $\boldsymbol{x}^{(0)}$ is not good, Newton's method might not converge. We discuss two standard approaches to make it converge for a larger array of starting values.

The first approach is doing load steps. This is a common approach in engineering problems, where some load (mechanic/electric/...) is applied to a nonlinear system. In an abstract setting, we choose a sequence of problems $\boldsymbol{F}_\lambda(\boldsymbol{x}_\lambda) = 0$ for load parameters $\lambda \in [0, 1]$. For $\lambda = 0$, no load is applied, and $\boldsymbol{x}_0 = 0$ is known as solution. For $\lambda = 1$, the load is applied in total, such that $\boldsymbol{F}_1 = \boldsymbol{F}$. Problems are solved for a sequence of growing load factors $\lambda \to 1$, where for each problem the solution to the last problem is chosen as an initial value. For small load step size, the iterations (hopefully) converge fast. This approach is applicable to problems where some kind of "natural" load can be defined. It will not help convergence in case of buckling or snap-through behavior in structural mechanics, though.

The second approach is to choose a damping parameter $\alpha_k \in (0, 1]$ in each step, such that

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \alpha_k(\boldsymbol{F}'(\boldsymbol{x}^{(k)}))^{-1}\boldsymbol{F}(\boldsymbol{x}^{(k)}) = \boldsymbol{x}^{(k)} + \alpha_k \, \Delta\boldsymbol{x}^{(k)}. \tag{3.72}$$

For $\alpha_k = 1$, the undamped method is retained. Choosing $\alpha_k$ small will help convergence also for less accurate starting values. However, $\alpha_k < 1$ means that quadratic convergence is replaced by linear convergence. Thus, different strategies to choose $\alpha_k$ somewhat optimal have been developed,

- start at $\alpha_k = 1$, while

$$\|\boldsymbol{F}(\boldsymbol{x}^{(k)} + \alpha_k \Delta \boldsymbol{x}^{(k)}\| > \|\boldsymbol{F}(\boldsymbol{x}^{(k)})\| \tag{3.73}$$

  divide $\alpha_k$ by two. This choice is easy to test (evaluation of $\boldsymbol{F}$ only), however it is not affine invariant, thus the algorithm changes when equations are rescaled/recombined.

- Deuflhard [2] proposes an alternative that is affine invariant: start at $\alpha_k = 1$, while

$$\left\|(\boldsymbol{F}'(\boldsymbol{x}^{(k)}))^{-1}\boldsymbol{F}(\boldsymbol{x}^{(k)} + \alpha_k \Delta \boldsymbol{x}^{(k)})\right\| > \left(1 - \frac{\alpha_k}{2}\right)\|\Delta \boldsymbol{x}^{(k)}\| \tag{3.74}$$

  divide $\alpha_k$ by two. This requires additional solves with $(\boldsymbol{F}'(\boldsymbol{x}^{(k)}))^{-1}$. In case a factorization (LU, QR) has been performed, this strategy may be considered.

- Often, the nonlinear system stems from an optimization problem, e.g.

$$\Psi(\boldsymbol{x}) \to \min \qquad\qquad \boldsymbol{F}(\boldsymbol{x}) = \Psi'(\boldsymbol{x}) = \boldsymbol{0}. \tag{3.75}$$

  In that case, $\alpha_k$ is reduced while the potential $\Psi$ is not decreased, i.e. while

$$\Psi(\boldsymbol{x}^{(k)} + \alpha_k \Delta \boldsymbol{x}^{(k)}) > \Psi(\boldsymbol{x}^{(k)}) \tag{3.76}$$

  divide $\alpha_k$ by two.

**Computation of the Jacobian** It is not always possible to compute the Jacobian $\boldsymbol{F}'(\boldsymbol{x}^{(k)})$ analytically. A viable approach is to use *numeric differentiation*. Therefore, choose $\varepsilon$ small, and set the $j^{\text{th}}$ column of $\boldsymbol{F}'$ to the difference quotient

$$(\boldsymbol{F}')_j = \frac{1}{\varepsilon}\left(\boldsymbol{F}(\boldsymbol{x}^{(k)} + \varepsilon \boldsymbol{e}_j) - \boldsymbol{F}(\boldsymbol{x}^{(k)})\right). \tag{3.77}$$

Above, the $j^{\text{th}}$ parameter $x_j^{(k)}$ is varied by $\varepsilon$. An optimal choice of the numeric differentiation parameter $\varepsilon$ is far from trivial: too large $\varepsilon$ results in an incorrect Newton Jacobian, however to avoid numeric instabilites, it is important to choose $\varepsilon$ not too small. If different components of $\boldsymbol{x}$ scale differently, this has to be reflected in the respective choice of $\varepsilon$.

Another approach is the use of automatic differentiation.

### 3.2.3 Nonlinear regression – the Gauss-Newton method

In this section, we consider the problem of finding optimal parameters to approximate measurement data by a given type of function. The following theory is an extension to the linear problem of polynomial regression in Section 2.2.5.

Let $(t_i, y_i), i \in 1 \to n$ be given (measurement) data. Let moreover $f(\boldsymbol{x}, t)$ be a family of functions, where $\boldsymbol{x} \in \mathbb{R}^m$ denotes the *parameter vector*. Our aim is to find optimal parameters $\boldsymbol{x}^*$ such that $f(\boldsymbol{x}^*, t)$ approximates the measurement data. To be precise, the optimal parameter vector is chosen such that the quadratic distance to the measurements is minimized,

$$\frac{1}{2}\sum_{i=1}^{n}(f(\boldsymbol{x}, t_i) - y_i)^2 \to \min_{\boldsymbol{x}}. \tag{3.78}$$

In the following, we assume that there are less parameters than measurment points, i.e. $m < n$.

We denote the error vector

$$\boldsymbol{F}(\boldsymbol{x}) := \begin{bmatrix} \vdots \\ f(\boldsymbol{x}, t_i) - y_i \\ \vdots \end{bmatrix} \in \mathbb{R}^n. \tag{3.79}$$

Then, the minimization problem (3.78) is equivalent to the minimization problem

$$g(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{F}(\boldsymbol{x})^T\boldsymbol{F}(\boldsymbol{x}) \to \min. \tag{3.80}$$

A necessary condition for $\boldsymbol{x}$ to be a minimizer is that the derivative of $g(\boldsymbol{x})$ vanishes, i.e.

$$\boldsymbol{G}(\boldsymbol{x}) := g'(\boldsymbol{x}) = \boldsymbol{F}'(\boldsymbol{x})^T\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}, \tag{3.81}$$

$$\text{with } \boldsymbol{F}'(\boldsymbol{x}) \in \mathbb{R}^{n\times m},\ (\boldsymbol{F}'(\boldsymbol{x}))_{ij} = \frac{\partial F_i}{\partial x_j}(\boldsymbol{x}) = \frac{\partial f}{\partial x_j}(\boldsymbol{x}, t_i). \tag{3.82}$$

Note that (3.81) is only a necessary condition, after computing a solution $\boldsymbol{x}^*$ one is obliged to check whether $\boldsymbol{x}$ is really a minimizer. Fortunately, this can usually be done by visually comparing the corresponding function $f(\boldsymbol{x}^*, t)$ with the given measurement data.

**Newton iteration** The classical Newton iteration for problem (3.81) needs the derivative of $\boldsymbol{G}(\boldsymbol{x})$. It computes formally as

$$\boldsymbol{G}'(\boldsymbol{x}) = \big(\boldsymbol{F}'(\boldsymbol{x})\big)^T\big(\boldsymbol{F}'(\boldsymbol{x})\big) + \big(\boldsymbol{F}''(\boldsymbol{x})\big)^T\boldsymbol{F}(\boldsymbol{x}). \tag{3.83}$$

Above, $\boldsymbol{F}''(\boldsymbol{x})$ is a tensor of order 3, with

$$\big(\boldsymbol{F}''(\boldsymbol{x})\big)_{ijk} = \frac{\partial^2 f}{\partial x_j \partial x_k}(\boldsymbol{x}, t_i). \tag{3.84}$$

Thus, component $jk$ of gradient $\boldsymbol{G}'(\boldsymbol{x})$ is found as

$$\big(\boldsymbol{G}'(\boldsymbol{x})\big)_{jk} = \sum_{i=1}^{n}\left(\frac{\partial f}{\partial x_j}(\boldsymbol{x}, t_i)\frac{\partial f}{\partial x_k}(\boldsymbol{x}, t_i) + \frac{\partial^2 f}{\partial x_j \partial x_k}(\boldsymbol{x}, t_i)\,(f(\boldsymbol{x}, t_i) - y_i)\right). \tag{3.85}$$

Finally, the Newton update in step $k$ reads

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \left(\boldsymbol{G}'(\boldsymbol{x}^{(k)})\right)^{-1} \boldsymbol{G}(\boldsymbol{x}^{(k)}). \tag{3.86}$$

**Gauss-Newton iteration** The Gauss-Newton iteration is an inexact variant of the above Newton iteration. In the Gauss-Newton method, the exact Newton matrix $\boldsymbol{G}'(\boldsymbol{x}^{(k)})$ is replaced by the first summand $\left(\boldsymbol{F}'(\boldsymbol{x})\right)^T \left(\boldsymbol{F}'(\boldsymbol{x})\right)$ in (3.83),

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \left(\left(\boldsymbol{F}'(\boldsymbol{x})\right)^T \left(\boldsymbol{F}'(\boldsymbol{x})\right)\right)^{-1} \boldsymbol{G}(\boldsymbol{x}^{(k)}). \tag{3.87}$$

This way, the update $\Delta\boldsymbol{x}^{(k)}$ computes as

$$\Delta\boldsymbol{x}^{(k)} = -\left(\left(\boldsymbol{F}'(\boldsymbol{x})\right)^T \left(\boldsymbol{F}'(\boldsymbol{x})\right)\right)^{-1} \boldsymbol{F}'(\boldsymbol{x}^{(k)})^T \boldsymbol{F}(\boldsymbol{x}^{(k)}), \tag{3.88}$$

which means it satisfies the normal equation

$$\left(\boldsymbol{F}'(\boldsymbol{x})\right)^T \left(\boldsymbol{F}'(\boldsymbol{x})\right) \Delta\boldsymbol{x}^{(k)} = -\boldsymbol{F}'(\boldsymbol{x}^{(k)})^T \boldsymbol{F}(\boldsymbol{x}^{(k)}). \tag{3.89}$$

This can be solved as discussed in Section 2.2.5 on overdetermined systems of equations. We perform a QR decomposition of $\boldsymbol{F}'(\boldsymbol{x})$,

$$\boldsymbol{F}'(\boldsymbol{x}) = \mathbf{QR}, \tag{3.90}$$

and solve the first $m$ staggered equations of

$$\mathbf{R}\Delta\boldsymbol{x}^{(k)} = -\mathbf{Q}^T \boldsymbol{F}(\boldsymbol{x}^{(k)}). \tag{3.91}$$

Equation (3.89) can be interpreted as the normal equation to the *linearized regression problem*

$$\|\boldsymbol{F}(\boldsymbol{x}^{(k)}) + \boldsymbol{F}'(\boldsymbol{x}^{(k)})\Delta\boldsymbol{x}^{(k)}\|^2 \to \min. \tag{3.92}$$

**Convergence** The Gauss-Newton iteration is an inexact Newton iteration. As such, we cannot expect quadratic convergence. Indeed, the Gauss-Newton method *converges linearly.*

In case the data are *compatible*, i.e. if there exists a set of parameters $\boldsymbol{x}^*$ such that the error is exactly zero,

$$\|\boldsymbol{F}(\boldsymbol{x}^*)\| = 0, \tag{3.93}$$

the iteration matrix is exact at convergence,

$$\boldsymbol{G}'(\boldsymbol{x}^*) = \left(\boldsymbol{F}'(\boldsymbol{x}^*)\right)^T \left(\boldsymbol{F}'(\boldsymbol{x}^*)\right) + \left(\boldsymbol{F}''(\boldsymbol{x}^*)\right)^T \underbrace{\boldsymbol{F}(\boldsymbol{x}^*)}_{=0} = \left(\boldsymbol{F}'(\boldsymbol{x}^*)\right)^T \left(\boldsymbol{F}'(\boldsymbol{x}^*)\right). \tag{3.94}$$

In this case, the Gauss-Newton method converges locally quadratic. In general, the better the data can be fitted, the better Gauss-Newton converges.

Implementing Newton's method in full (using $\boldsymbol{G}$ and $\boldsymbol{G}'$) is seriously discourage. Often, the problem of finding nonlinear parameters is ill-conditioned, i.e. $\boldsymbol{F}'$ has a high condition number. This ill-conditioning is further amplified if $\boldsymbol{G}' \simeq (\boldsymbol{F}')^T \boldsymbol{F}'$ is inverted.

**Scaling of equations** In case the data vector $\boldsymbol{y}$ varies over several orders of magnitude, minimizing the quadratic distance (3.78) will not yield usable results, as the different error components scale differently. Instead, it is proposed to minimize the scaled relative error

$$\frac{1}{2} \sum_{i=1}^{n} \left( \frac{f(\boldsymbol{x}, t_i)}{y_i} - 1 \right)^2 \to \min_{\boldsymbol{x}} . \tag{3.95}$$

In this case, the algorithm above is implemented for $\tilde{f}(\boldsymbol{x}, t_i) = f(\boldsymbol{x}, t_i)/y_i$ and $\tilde{y}_i = 1$.

# Bibliography

[1] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951.

[2] Peter Deuflhard and Andreas Hohmann. *Numerische Mathematik 1 – Eine algorithmisch orientierte Einführung*. Walter de Gruyter, 2008.

[3] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[4] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes - The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.