Lecture Notes

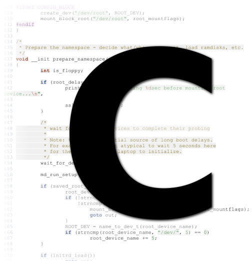
“Scientific Computing I”

winter term 2024/2025

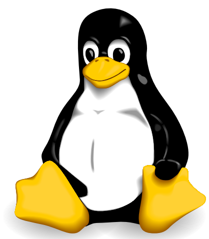
Dr. Martin Köhler
`koehlerm@mpi-magdeburg.mpg.de`

Dr. Pavan Laxmipathy Veluvali
`veluvali@mpi-magdeburg.mpg.de`

See the **Authors and Contributors** sections for a full list of contributions to these lecture notes.



for $j := i, \dots, n$ **do**
  for $k := 1, \dots, i-1$ **do**
    $A_{i,j} -= A_{i,k} A_{k,j}$



version from November 18, 2024

Contents

Preface

German Die Vorlesung "Wissenschaftliches Rechnen 1" verfolgt das Ziel, Verfahren und Algorithmen der Numerischen Mathematik praktisch umzusetzen. Sie soll Wissen und Strategien vermitteln, welche notwendig sind, um Ideen aus der Theorie in praktisch nutzbare Programme zu übersetzen und diese effizient zu implementieren. Dies soll mehrheitlich mit Hilfe der Programmiersprache C geschehen, da sie eine der am meisten eingesetzten Sprachen ist[1] und auch im Bereich von eingebetteten System unverzichtbar ist.

Die rein mathematische Betrachtung von Problemstellung reicht in vielen Fällen dem Urheber des Problems nicht mehr aus. Viel mehr sind Industrie und Technik an praktisch nutzbaren Ergebnissen für die Anwendung in Informatik, Ingenieurwesen und Alltagsproblemen interessiert.

Neben der Umsetzungen von mathematischen Verfahren soll der Umgang mit unixoiden Betriebssystemen (in diesem Fall Linux) erlernt werden. Diese bilden die hauptsächlich eingesetzte Klasse von Betriebssystemen auf den großen und sehr großen Installationen, wie Compute-Clustern in modernen Rechenzentren. Neben den Betriebssystem-Spezifika werden auch Hilfsmittel vorgestellt, die den Arbeitsablauf im Umfeld des wissenschaftlichen Rechnens erleichtern.

English This lecture aims at the practical implementation of methods and algortihms in numerical mathematics. Its main purpose is to convey the knowledge and strategies necessary to transfer and efficiently implement theoretical ideas into computer programs for practical application. We will focus on the

[1] https://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

C programming language since this is one of the most commonly[1] used languages, which is especially invaluable in the environment of embedded systems.

The purely mathematical consideration of problem settings often is no longer sufficient. Today partners from industry and technology are interested in practically usable results for applications in computer and engineering sciences.

Along with the practical implementation of mathematical methods the usage of unixoidal operating systems (in our case Linux) is to be learned. Those operating systems form the most important class of operating systems used on large compute clusters in modern high performance computing centers. Besides operating system specifics we also present a couple of tools that help simplifying work in a scientific computing environment.

Layout and Style

We have put some effort into creating a unique reading experience that visually supports the reader in identifying contributions to the content. Examples are typeset inside light gray background boxes to find them easily in the document. They follow a chapter-wise numbering scheme, that is also used for Theorem-like environments (i.e. definitions, theorems, lemmas, corollaries and remarks). These environments are all displayed as framed boxes where definitions are marked by a -symbol. Theorems, corollaries and lemmas can be identified by the -symbol and remarks show a . Equation numbers follow their own chapter-wise scheme.

Commands, program variables and alike are displayed in **`typewriter`** style throughout the document. When an appropriate portion of code is presented, we use color coding (of the background color) to identify the type of code that is displayed. We distinguish the following:

```
C sources
```

```
Fortran sources
```

```
Shell scripts (especially BASH)
```

```
Makefiles
```

Keystrokes or key combinations are shown as, e.g. q for the singel key “q” or ctrl + c for the combination of “c” and the “ctrl” key.

Authors and Contributors

The main authors o this document are Jens Saak and Martin Köhler. Beginning from WS2012/2013 they have updated this document on a biannual basis until WS2022/2023.

Jens and Martin would like to thank a couple of people that helped in preparing this manuscript. Some of them had major contributions. First of all Peter Benner provided the German basis for Chapter 5, which was slightly modified with material from the seminal book on “Stability and Accuracy of Numerical Algorithms” by Nicholas J. Higham.

Large parts of this manuscript would not have been possible without the student Ricardo Leese, who typeset most of Chapters 6–8 during the course given in winter term 2012/2013.

Furthermore many thanks go to Petar Milnarić for carefully reading through the manuscript in winter term 2016/2017. His suggestions have improved the content, as well as the layout of the material. In winter term 2018/2019, Jonas Schulze did another pass of careful review and further improved the presentation of both theory and layout.

Also, the authors thank all other students of the course for participation in the discussions during the lecture that helped increase the quality of the presentation a lot.

...the Linux philosophy is 'laugh in the face of danger'. Oops. Wrong one. 'Do it yourself'. That's it.

Linus Torwalds

CHAPTER 1

Linux and the Commandline

Contents

This first chapter is dedicated to an introduction to the Linux operating system and the command line. We focus on the command line operation of the system, since on many compute servers, especially in high performance computing centers, this is the only way to access the system. Furthermore, once we understand how to perform certain tasks on the command line, it is then a lot easier to write job scripts for submission of so called batch jobs to job scheduling systems used on distributed compute resources like clusters and grids.

We focus on Linux here although most Unix-like operating systems should at least behave very similar. Especially for the ones based on the GNU ("GNU's not

Unix") project everything should be more or less exactly the same. The GNU project was founded in 1983 long before the first Linux kernel came to life. A major contribution of the inventor Richard Stallman was the first version of the GNU Public License (GPL) that today is inseparably connected with the Linux operating system.

1.1 A short History of an Accidental Revolution

As a matter of fact, much later the first Linux system was developed for exactly the purpose we are pursuing here, namely a terminal emulator for accessing the universities Unix (in the special case Minix) based compute facilities. At some point the author realised that he had "accidentally" written an operating system kernel. The first version of Linux was announced by its inventor Linus Torvalds in the following news posting in a usenet news group[1] for the Minix OS that he was trying to access on August 26, 1991:

> *Hello everybody out there using minix -*
>
> *I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).*
>
> *I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)*
>
> *Linus (torv...@kruuna.helsinki.fi)*
>
> *PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.*

After this the (r)evolution has been fast as the following timeline (taken from Wikipedia[2]) shows:

1983 Richard Stallman creates the GNU project with the goal of creating a free operating system.

1989 Richard Stallman writes the first version of the GNU General Public License.

[1] https://groups.google.com/forum/?fromgroups=#!msg/comp.os.minix/dlNtH7RRrGA/SwRavCzVE7gJ

[2] https://en.wikipedia.org/wiki/History_of_Linux

1991 The Linux kernel is publicly announced by the 21 year old Finnish student Linus Benedict Torvalds.

1992 The Linux kernel is relicensed under the GNU GPL. The first so called "Linux distributions" are created.

1993 Over 100 developers work on the Linux kernel. With their assistance the kernel is adapted to the GNU environment, which creates a large spectrum of application types for Linux. The oldest currently existing Linux distribution, Slackware, is released for the first time. Later in the same year, the Debian project is established. Today it is the largest community distribution.

1994 In March Torvalds judges all components of the kernel to be fully matured: he releases version 1.0 of Linux. The XFree86 project contributes a graphic user interface (GUI). In this year the companies Red Hat and SUSE publish version 1.0 of their Linux distributions.

1995 Linux is ported to the DEC Alpha and to the Sun SPARC. Over the following years it is ported to an ever greater number of platforms.

1996 Version 2.0 of the Linux kernel is released. The kernel can now serve several processors at the same time, and thereby becomes a serious alternative for many companies.

1998 Many major companies such as IBM, Compaq and Oracle announce their support for Linux. In addition a group of programmers begins developing the graphic user interface KDE.

1999 A group of developers begin work on the graphic environment GNOME, which should become a free replacement for KDE, which depended on the then proprietary Qt toolkit. During the year IBM announces an extensive project for the support of Linux.

2004 The XFree86 team splits up and joins with the existing X Window standards body to form the X.Org Foundation, which results in a substantially faster development of the X Window Server for Linux.

2005 The project openSUSE begins a free distribution from Novell's community. Also the project OpenOffice.org introduces version 2.0 that now supports OASIS OpenDocument standards in October.

2006 Oracle releases its own distribution of Red Hat. Novell and Microsoft announce a cooperation for a better interoperability.

2007 Dell starts distributing laptops with Ubuntu pre-installed in them.

2011 Version 3.0 of the Linux kernel is released.

2012 The aggregate Linux server market revenue exceeds that of the rest of the Unix market.

2013 Google's Linux-based operating system, Android claims 75% of the smart-phone market share, in terms of the number of phones shipped.

2014 Ubuntu claims 22 000 000 users.

2015 Version 4.0 of the Linux kernel is released.

2019 Version 5.0 of the Linux kernel is released.

At first Linus Torvalds intended to name his operating system Freax, a portmanteau of the words "freak", "free", and "x" (for Unix). As of today the times when Linux was an operating system only for freaks are over. Several modern Linux distributions exist that are nowadays as easy to use and install as the main consumer market competitors MS Windows and MacOS.

1.2 The Linux Shell and Basic Commands for Handling Files

The shell is the Linux command interpreter. It serves as the basic interface to the operating system. In fact there is not only one shell but a couple of implementations like **`bash`**, **`csh`**, **`tcsh`**, **`ksh`**, **`zsh`**. We base our presentation on the **`bash`** shell. Most of the ideas directly transfer to the other ones although the commands and syntax can differ slightly. Before diving into the usage of the **`bash`** and basic tools for managing files and data, we call the attention to the list of special characters that play important roles and cannot easily be used in command, file, or directory names. They are reported in the following table.

<table>
<tr><td>*</td><td>serves as a placeholder for arbitrarily many characters</td></tr>
<tr><td>?</td><td>a placeholder for a single character</td></tr>
<tr><td>/</td><td>directory separator</td></tr>
<tr><td>\</td><td>escape character for quoting special characters and to mark line breaks</td></tr>
<tr><td>~</td><td>abbreviation for your home directory</td></tr>
<tr><td>|</td><td>the pipe operator: connects two simple commands to a new one by redirecting the output of the one on the left to the other one on the right. || represents a logic OR.</td></tr>
<tr><td><</td><td>fetches the input for a command (on the left) from a file or device (on the right)</td></tr>
<tr><td>></td><td>redirects the output of a command (on the left) to a file or device (on the right)</td></tr>
</table>

`2>`	same as above for the error output only, can be used to redirect the standard error messages to standard output so it is recognized by the **`>`** and **`|`** as well via **`2>&1`**
`1>`	same as above for the standard output without the errors
»	as **`>`** but appends the output instead of overwriting the file
`$`	used in command substitution and for referring to shell and environment variables
`&`	a single **`&`** after a command name sends the execution to the background. Double **`&&`** stand for the logic AND.
`` ` ``	accent grave is used for command substitution
`'`	single quotes removes the special meaning of all special characters enclosed by them.
`"`	double quotes act the same as single quotes with the exception of the **`$`**, **`` ` ``**, **`\`** (and sometimes **`!`**) characters keeping their special properties.
blank	the simple blank is used to separate words and thus needs to be escaped when, e.g., a file name contains it.
`#`	comment character; everything following this character on the same line will be dropped
`!`	initiates history expansion

Basic Directory Commands The basic arrangement of filesystems differs signifficantly from, e.g., a MS Windows machine. In contrast to MS Windows, where all physical discs get their own drive letter and start a local directory at the volume's root, in Unix-like environments the filesystem is arranged in one global directory tree and all physical drives are placed in a certain structure under a common root called **`/`**. The specific structure of this tree differs between the types of Unixes and even among Linux distributions it has been varying a lot. Over the recent years huge efforts have been undertaken to unify the structure. The Linux Standard Base (LSB) is the largest and most important initiated by the Linux Foundation. It is not only defining a common directory structure, but tries to unify large parts of the distribution to increase the cross distribution compatibility.

There are many commands used to work with or manipulate files and directories. We will only report on a selection of commonly used ones here. Before we get to the list of commands, however, we introduce some special directories. The directory shorthand **`\~`** was mentioned in the table above already. It stands for your home directory, i.e., the directory holding your personal files and the one directory in which you usually end up directly after logging in to the system. Every directory contains two special entries: "**`.`**" representing the current directory, and "**..**" abbreviating the directory one level above in the directory tree. The first one enables us to refer to commands in the current directory in case it is not in our default search path and the other enables the use of relative path constructs for referring to files.

pwd short for print working directory, and printing the name of the directory you are currently working in is exactly what it does.

cd change directory, switches the current working directory to the directory given as the argument. If no argument is given **cd** takes you home, i.e., switches to your user's home directory.

mkdir creates a new directory in the current working directory

rmdir removes the directories specified as arguments if they are empty.

touch creates an empty file, or sets the access date of the file to the current time and date, if it already exists.

rm removes files. It can also be used to remove directories with the **-r** (recursive) option. This is especially useful when **rmdir** does not work since the directory is not empty. The **-f** (force) option can be used to remove even protected files.

ls lists all files in the directory specified. If none is specified the current working directory is used. If the argument is a file or a list of files only those files are listed. Useful options are **-l** for a full listing including access rights and ownership information, **-a** for a listing including also hidden files. The **-h** option in combination with the two previous ones makes file sizes human readable, i.e., displayed as multiples of KiB, MiB, GiB, TiB, where all of these are representing powers of 1 024 (binary prefix). If a 1 000 based presentation is desired **--si** needs to be used instead (decimal prefix).

cp takes two or more arguments and copies the $n-1$ first arguments to the last. If more than 2 arguments are given the last one must be a directory. Absolute and relative paths are allowed.

mv Same as above but moves the files, i.e., the originals are removed after the copy has successfully finished.

ln links files to new names. By default a hard link is created. Then the new name serves as a new entry in the file system associated to the same data and the data is only removed if all hard links are removed. When used with the -s option a soft link (or symbolic link) is created that only points to the original. When the original data is removed the link becomes orphaned.

find is a powerful search tool that can hardly be fully described in a few words. We refer to the **man** and **info** pages for details. A feature often overlooked in the **man** page are the operators. Note that, e.g., **!** or **-not** allow to negate the following search expression.

locate Another search tool that uses a pregenerated database to accelerate the searches. The database may be restricted to parts of the filesystem only, or even not exist. Also it is frequently updated but may be outdated when the actual search is performed. However, for directories that do not change very frequently this is a good alternative since it is usually a lot faster than **find**.

File Permissions and Storage Amounts We have seen before that the **ls -l** command helps us learn about the permissions of files. Here we explain these permissions in detail and show how they can be changed. The command executed in the home directory storing the files of the standard user **scuser** on the virtual machine found on the lectures homepage give the following result

Example 1.1:

```
total 32
drwxr-xr-x 2 scuser scuser 4096 Sep 27 12:20 Desktop
drwxr-xr-x 2 scuser scuser 4096 Aug 27 15:14 Documents
drwxr-xr-x 2 scuser scuser 4096 Aug 27 15:14 Downloads
drwxr-xr-x 2 scuser scuser 4096 Aug 27 15:14 Music
drwxr-xr-x 2 scuser scuser 4096 Aug 27 15:14 Pictures
drwxr-xr-x 2 scuser scuser 4096 Aug 27 15:14 Public
drwxr-xr-x 2 scuser scuser 4096 Aug 27 15:14 Templates
drwxr-xr-x 2 scuser scuser 4096 Aug 27 15:14 Videos
```

The same command issued on the Desktop folder gives:

Example 1.2:

```
-rw------- 1 scuser scuser 12680 Aug 30 08:59 chromium-\
   ↪ browser.desktop
-rw------- 1 scuser scuser  4953 Sep 27 12:18 \
   ↪ lxterminalA6O6KW.desktop
-rw------- 1 scuser scuser  4953 Aug 27 16:32 lxterminal.\
   ↪ desktop
-rw------- 1 scuser scuser  5813 Sep 27 12:20 pcmanfm.\
   ↪ desktop
```

In both cases the output contains the same important groups information. The **drwxr-xr-x**, and **-rw------** show the file type and permissions. Here the **d** in the first set shows that the corresponding line relates to a directory. The **-** marks a normal file. Another commonly found symbol is **l** for symbolic links.

There are many more that are described in the **info** pages (see also Section 1.3). The following three groups of three characters describe the file permissions of the owner (first three), the related group (second three) and everyone else (remaining three). Here the **r** stands for the possibility to read a file, or directory and the **w** stands for write access. The **x** on a file makes that file executable, i.e., interpreted as a program. For a directory the flag stands for the ability to change into the directory. If a flag is unset, i.e., the access is not granted it is replaced by a **-** in the corresponding position. The **scuser scuser** part represents the owner (first) and the related user group (second) for the file. In the examples above the user **scuser** has read and write access on all objects and for the directories is also allowed to change into them. The group **scuser**, however, is only allowed to read and change into the directories, but can not read or manipulate the files in the Desktop directory.

To determine whether a certain user group permission set applies to your user you may use one of the two commands **id** or **groups**. The second one simply prints all group names the current user is in. The first one in addition prints the numeric ids that are used by the system to represent the user, its primary and all the other groups.

In case the group a file is related to needs to be changed, this can be done using the **chgrp** command. The command takes two or more arguments. The first argument needs to be the new group for which the association should be performed. After this a list of elements (files, directories, links) follows that should be associated to the new group. Several optional command line switches exist that influence the way, for example links are treated. Alternatively the **chown** (change ownership) command may be used. This can also be used to change the owning user. For the latter task normally superuser privileges are required. The calling sequence is mainly the same. The only difference is that instead of a group owner and group are given in the form **owner:group**. Here both owner and group are optional, but the syntax needs to be **:group** if only the group is to be changed.

The standard Unix file permissions can be changed by the **chmod** command. The standard format to perform simple changes is for example

```
chmod u+w file1
chmod g+rw file2
chmod o-wx file3
```

to grant the user write permission to **file1**, the group read and write permission on file **file2** and remove the write and execute permission from **file3** for the rest of the users (o for others). These changes are performed relative to the existing file permissions. Sometimes it is however easier to perform absolute changes. To this end read, write and execute flags have corresponding numerical values. Read permission counts 4, write permission 2 and execute permission 1. All combinations of read, write and execution permissions can then be formed as sums of those values. That means 7 represents **rwx**, 6 stands for **rw-**, 5 for **r-x** and 3 is **-wx**. This way, changing the file permissions to **rwxrw-rw-** for **file** from an arbitrary prior setting can be done via

```
chmod 755 file
```

On the Andrews filesystem (AFS) which is also used at the Magdeburg University file permissions are stored on a per directory basis. Also the above command is useless there. The corresponding command for checking and setting file permissions there is called **fs** and the command for group handling is **pts**. Their in depth explanation would exceed the space limitations here and we refer to the man pages or web based AFS quick reference[3] for getting started.

Often the disk space per user is limited by the operating system. To check the amount of space on a Unix file system that a user is currently using and is allowed to use at maximum can be found via the **quota** command. On the lectures virtual machine the disk space is the only limit for the space. The **quota** command is therefore not even installed.

The more important limit to the disk usage is obviously given by the capacity of the physical drives available in the machine or the servers our network filesystems are residing on. We can get an overview of those filesystems currently used (mounted) on our machine by typing **df**, which on the virtual machine gives

```
Filesystem     1K-blocks    Used Available Use% Mounted on
/dev/sda1        9804120 2256688   7049412  25% /
udev              246672       4    246668   1% /dev
tmpfs             101576     748    100828   1% /run
none                5120       0      5120   0% /run/lock
none              253936       0    253936   0% /run/shm
```

This tells us that we are currently using 25% of the maximum capacity of our main disk mounted at the file system root **/**. The other entries are used by the operating system and not related to physical drives.

Once we have determined we are consuming a certain percentage of our allowed or possible space it may get important to find out where all the space is going, i.e., which files are using it. The **du** command can be employed to find this out. Started in a certain directory the command recursively descends into all directories below that point in the filesystem tree and checks and reports their disk usage in bytes. At the end it provides a grand total. As for the **ls** command a **-h** flag exists to make the output a bit more user readable. The **--max-depth=** command line parameter can be used to limit the descend

[3] https://its.ucsc.edu/unix-timeshare/tutorials/afs-quickref.html

depth for which the disk usage is reported. Still the maximum depth is searched but only the selected ones are reported in detail.

Influencing the Working Environment The shell uses variables to store information about your working environment. Variables are elements referenced with a $ sign and usually written in all capital letters. One can find out which variables are currently set using the command **`env`**. If one knows the name of the variable beforehand the content can be printed out using the **`echo`** command. Some important environment variables are

`$HOME` containing the path to the users home directory,

`$USER` the user name of the user (also found in **`$LOGNAME`**, or **`$USERNAME`**),

`$PATH` a **`:`** separated list of directories that are used to search for executable programs

`$HOSTNAME` the name of the computer the shell is running on.

```
echo $HOME
```

Other important variables used by the GNU compilers and linkers will be introduced in Chapter 3. Environment variables can be set by simply assigning a value to them at the command line. For example

```
PATH=$PATH:$HOME/bin
```

appends the **`bin`** directory in the users home directory to the current executable search path. If one intends to have this setting inherited by processes started from the shell the same has to be done as

```
export PATH=$PATH:$HOME/bin
```

Also if we set variables in a script file and we want them to persist after the execution we have to use the **`export`** statement.

Two examples of such script files are the files **`.profile`** and **`.bashrc`**. Both these files are executed upon login to a new **`bash`** shell. They can thus contain settings that should always be active. For example if the above **`bin`** directory should always be contained in the search path, we would simply add the export line to one of the files. In this case this should preferably be **`.bashrc`** since the **`.profile`** will also be read by other shells which in some cases do not understand **`export`** but use a command called **`setenv`** instead.

The configuration files can also be used to define command abbreviations. For example one would often call the command **`ls`** with the **`-l`** and **`-h`** parameters and probably want to have it a little colorful to distinguish between files and directories more easily, as well as see at the first glimpse what files are executable. Adding the simple line

```
alias ll='ls␣-lh␣--color='auto'␣--group-directories-first'
```

defines a shortcut **ll** that does all this automatically.

Viewing Files The simplest file viewer is probably the **cat** command it takes the contents of the argument files, concatenates them, and displays the result at the standard output. It will not stop printing until the end of the last file is reached. Since this is not very useful for reading the content of longer files, **cat** is usually used in combination with other command or for redirecting the result to a new file (see also Section 1.4).

Two slightly more usable viewers are **head** and **tail** which by default display the ten first and last lines in the argument file. Both take the **-n** parameter that is used to change the number of lines displayed. **tail** is often used in combination with the **watch** command that periodically executes a certain command to watch the status of log files. For example

```
watch -n 60 tail -n 50 mylog.txt
```

displays the final 50 lines of the files mylog.txt every 60 seconds.

A fairly helpful file viewer is the **less** command. It uses the full height of the terminal window to display the leading part of the file. It then lets you scroll through the files content with the cursor keys, jump to the beginning or the end using the pos1 [4] and end keys, or search through the files content with / followed by the search expression. One can then navigate through the matches using the n (for next) and p (for previous) keys. The view can be exited by simply pressing the q key.

When one has two versions of the same file, e.g., subsequent iterations of the same source code, it is usually not easy to find the differences by simply comparing the content in two neighboring **less** views. To help simplify this task **diff** is the tool of choice. There are many command line switches that help to configure how the comparison is performed and how the result is displayed. By default the two files are compared and only differing lines with a little bit of context around them are displayed. There also exist several graphical user interfaces that help you compare and merge files even more easily. **xxdiff** and **kdiff3** are just two of those.

Compressing Files The common compression formats **zip** and **rar** most people know in the MS Windows world are available on Unix-like platforms as well. For example

```
zip -r folder.zip folder
```

[4] sometimes also called home

takes the directory **folder** and its entire content and creates a compressed archive **folder.zip**. After that

```
unzip folder.zip
```

can be used to unpack the directory somewhere else again.

The same task can be performed with **rar** using

```
rar a -r folder.rar folder
```

for the archiving and

```
unrar x folder.rar
```

for the extraction. If the extraction should be done flat, i.e., all files should go to the current directory ignoring the directory structure of the archive this can be achieved by

```
unrar e folder.rar
```

Alternatives, found on Unixes more classically, are **gzip**, **gunzip** for compression and decompression of single files using the Lempel-Ziv coding. If **gzip** is supplied with multiple files they will be compressed separately, however. Every compressed files gets an additional suffix **.gz** to show the compression. Similarly **bzip2** and **bunzip2** are used to compress single files using Burrows-Wheeler block sorting text compression algorithm, and Huffman coding, which usually leads to better compression rates, but takes more time to complete. The compressor adds a **.bz2** suffix. Both **gunzip** and **bunzip2** remove the additional suffixes again after decompression.

If many files are to be compressed in a single file, they can be bound together in a *tape archive* using the **tar** command. Again returning to our example above we would perform the task by

```
tar -cf folder.tar folder
```

where **-c** tells the command to create the archive and the **-f** is used to specify the resulting file name. We can combine this directly with the two compression formats above using

```
tar -czf folder.tar.gz folder
```

or

```
tar -czf folder.tgz folder
```

to create a **gzip** compressed tape archive, or

```
tar -cjf folder.tar.bz2 folder
```

to do the same using **bzip2** compression.

The corresponding decompression is then done by

```
tar -cf folder.tar
tar -czf folder.tgz
tar -czf folder.tar.gz
tar -cjf folder.tar.bz2
```

respectively.

Since the file extensions(suffixes) do not mean anything to the system in Unix environments, they can be seen as a reminder for the user. To really see what type a file has the **file** command can be used. Again we use an example for clarification. Running **file** in the above **.tgz** file by

```
file folder.tgz
```

results in something like

```
folder.tgz: gzip compressed data, from Unix, last modified\
    → : Tue Oct 9 21:38:02 2012
```

Downloading Files An easy way to download files via the command line is given by the command **wget**. The virtual appliance for the lectures virtual machine can be downloaded via

```
wget https://www.mpi-magdeburg.mpg.de/mpcsc/lehre/2016\
    → _WS_SC/vm/ubuntu-16.04.ova
```

for example. The tool is, however, much more powerful. It can also be used to mirror entire websites. For the details we refer to the man page.

Processes and System Usage Once you logout of the system, all your processes are usually terminated. Especially in the case of large computing tasks we would, however, prefer if they would continue running. One tool that helps avoiding this is the **nohup** command. It basically tells the operating system not to terminate a certain job when the user logs out. However the output of the command needs to be redirected and we cannot easily reattach to the running process. The GNU **screen** utility is a better alternative. It will be decribed in Section 1.8.

Especially for compute jobs that are running for a very long time it can be advantageous to not block the CPU of the machine they are runing on entirely. For example when one uses the local machine to start the job and wants to continue working on it, it is a good idea to manipulate the job such that it will only use such CPU cycles that are not needed by any other task. This can easily be done using the program **nice**.

```
nice -19 large-computation
```

starts the program **large-computation** with nice level 19, i.e., the lowest possible priority. Any level between 0 (the default for a user process) and 19 can be used. If the program is already running and one decides to lower the priority this can be done using **renice** as in

```
renice -n 19 12345
```

where 12345 is the process identification number (PID) of the program to be reniced.

An easy way to find out the PID for an already running task is the **pgrep** utility

```
pgrep large-computation
```

provided only one instance of the computation program is running.

A good overview of which processes are currently running is given by the **top** tool. It produces a full screen view showing the current entries of the operating systems process table. This is by default sorted by the percentage of CPU usage. The view is periodically updated and the ordering can be manipulated by the larger and smaller keys, which move the column of the display used for sorting to the left or right. The **top** view can also be used to identify jobs and find PIDs for renicing. Some tasks like renicing and terminating processes can even be performed from inside **top** using certain shortcut keys (found in the man page). As for **less**, or **man** the q is used to exit **top**.

In script files **top** can obviously not be used. There the **ps** command is the tool of choice. The tool has a huge number of switches selecting the processes to display. For a general view of the users processes

```
ps ax
```

can be used. The list is then usually rather long on the other hand. Therefore the output of **ps** is often processed further as can be seen in Section 1.4.

If we are not so much interested in the exact processes running on a machine but only want to know who is currently working on it, we can find this information via the command **who**. It simply prints a list of all active users.

Some tasks need superuser privileges to be able to execute. Systemwide installation of certain software would be one such example. A convenient way of performing such tasks is the **sudo** tool. It starts a command with the same privileges that the superuser **root** would have. To be able to do so one needs to be registered in a list of users allowed to this however. On our lectures virtual machine the **scuser** is allowed to perform mainly any tasks using **sudo**. In a general environment the permission to do so will, on the other hand, be very limited.

One thing a user is always allowed to do is the termination of tasks. If this is not done from within the task, it can be forced from the outside by the **kill** command. For the above task that we reniced already we can use

```
kill -QUIT 12345
```

to tell it to safely terminate. If for some reason it does not do so, **kill** knows a couple of other signals it can send to the process. The KILL signal is the most drastic of those and should be used only if all others fail.

1.3 Getting Help

The two most important local resources for documentation of linux commands are the **man** and **info** systems. Both simply take the command name as their argument and display documentation information in a simple text-based command line browser view. The **man** documentation page can be navigated and searched through just like the **less** view described above. In an **info** page additionally there may be cross references in the form of hyperlinks to further details and related commands.

If one does not remember the command name but knows the purpose, then **apropos** can help finding the command. Called with a keyword as the argument **apropos** searches the short descriptions at the beginnings of all man pages for the keyword and displays a list of all commands where it finds appropriate matches.

1.4 Manipulation of Simple Commands

In many situations especially in script files one is interested in passing the results of certain operations directly into the next operation. The pipe operator **|** in the linux shell can be used to do this.

```
program1 | program2
```

can be used whenever **program1** writes its output to the standard output and **program2** reads its input from the standard input. Unfortunately this is not always the case. For example if we want to remove all PDF files from the current directory and all its subdirectories, we can use **find** to generate a list of all those files. Now we would like to use **rm** to remove them. **rm**, however, takes it arguments directly from the command line and only uses the standard input if we force it to use the interactive mode asking for permission to delete every file. The task can be completed anyway using the **xargs** utility, which takes a list from standard input and splits it into a list of arguments to another command. So all in all we want to do

```
find . -name '*.pdf'| xargs rm
```

or if the number of files is very large we can force **xargs** to pass the files to **rm** one after the other

```
find . -name '*.pdf'| xargs -n 1 rm
```

The parameter **-n** here takes the number of simultaneously passed arguments. There are two more important parameters. The maximum number of parallel executions can be set with **-P** and **-d** is used to specify the delimiter used for the spliting of the list if it should not be a single space.

We have seen another example of such a contruction before since **pgrep** can be made up the same way

```
pgrep = ps ax | grep [x]xx | awk '{␣print␣$1;␣}'
```

The **grep** and **awk** utilities will be described in the following section.

In other situations it is necessary to store the output of a certain command as a text file or read the input from it. The redirection operators **>** and **<** can be used to do this. Again we use some examples to clarify this. To simply write the output of a command that would appear on the screen to a file **output.txt** we use

```
program > output.txt
```

To preserve the current content of the file we need to call

```
program >> output.txt
```

to tell the system to append the new information to the end of **output.txt**. Otherwise the file is replaced. Non existing files are created prior to writing to them.

If at a later point another program that usually reads inputs from user interaction needs this output as its input we can read it by

```
other_program < output.txt
```

We can also do both, i.e., read from a file **input** and write to another file **output**

```
program <input >output
```

There are two special variants of the output operator that allow to separate between standard outputs and error messages.

```
program 1>output 2>errors
```

will create a file **`output`** containing the standard messages of the program and another file **`errors`** where all the error messages are stored.

We can also directly reuse the output of a command to make up new strings or commands by command substitution. This is performed if a simple command is enclosed by one of the two types of command substitution characters. For example the **`date`** command can be used to return the current time and date. If we want to directly use it in the output of a script we can use the **`echo`** command to print a message containing the current time and date:

```
echo Yeah, today is `date`, the term is almost over!
echo Yeah, today is $(date), the term is almost over!
```

both will give an output similar to

```
Yeah, today is Mo Oct 16 14:45:32 CEST 2023 the term is \
    → almost over!
```

One big problem using the pipe and the redirect operators is that one can not see the output that is redirected. This might, however, be useful in some cases. The problem can be solved by the **`tee`** command, which reads data from the standard input and writes to the standard output and a file simultaneously. Consider the case where you want to list all files in the current directory and store the result in a file:

```
ls > file
```

If we also want to have the output on the screen as well we can use:

```
ls | tee file
```

`tee` can be used to to create copies of the data processed by a sequence of pipes:

```
ls | tee output_of_ls | grep "[Hh]ello.c"
```

Per default **`tee`** overwrites the given file. If it should append the output to a given file use:

```
... | tee -a outputfile
```

1.5 Script File Basics

In large computing centers the devices are usually not directly accessible, but the computation tasks have to be submitted to a job scheduling system. There, one has to provide a job script along with the executable that is used to run the computation with the desired parameters. Such job scripts are simple text files of a certain structure that we are explaining in this section. Such script files can also be helpful on the local desktop computer to automate certain actions that one has to perform on a regular basis. The following is a minimal hello world **bash** script that already contains all the important ingredients.

```
#!/bin/bash
echo "Hello World!"
```

Saving this as a file **hellow.sh** and making that file executable, we can simply run

```
hellow.sh
```

to get the response

```
Hello World!
```

The file suffix **.sh** here is only used for our convenience. That means it is only used to make it easier for the user to identify it as a shell script. The system itself identifies which interpreter (in our case the **bash** shell) needs to be executed to run the remainder of the file by the special statement **#!/bin/bash** on the first line. The **#!**, here, tells the system that the following should be read as the interpreter. It is necessary to use the full path from the root of the filesystem to make sure the interpreter is found upon execution of the script. Similarly we can specify that the interpreter should be **awk** (described in the next section) by using **#!/usr/bin/awk** or the python language **#!/usr/bin/python** on the first line and filling the remainder with something written in the corresponding programming language.

Remark 1.3: Note the blank after the "**!**" in the above example. This is mandatory since otherwise **bash** may use the "**!**" to initiate a history substitution. This happens unless it is followed by a blank, newline, carriage return or **(**. The behavior is explained in the *Event Designators* section of the man page.

Inside the script files **bash** can use several control structures like loops and conditional, or even functions. Their explanation would however exceed the scope of our presentation and we refer to the man page for details.

1.6 Simple Automatic File Manipulation

One of the key ingredients for automatic treatment of files are regular expressions. They are for example used to extract certain useful information from log files, or replace expressions in source code when name changes need to be performed in large software projects. They are also the main tool for successful usage of the **grep** and **sed** utilities described later in this section.

Regular Expressions Regular expressions are strings that can be used to establish complex search and replace operations on other strings. A regular expression consists of a combination of special and basic characters that are used to match the sought after substring in the other string. There exist a number of special characters /, (,), *, ., |, +, ?, [,], ^, $, \,{,}. The following table explains them in detail. Note that **sed** and **grep** process files line by line. Thus, line and string are used synonymously in the following.

.	matches any single character except line breaks
^	matches the beginning of the string/line
$	matches the end of the string/line
[list]	any one character from list. Here list can be a single character, a number of characters, or a character range given with -
[^list]	any one character that is NOT in list.
( )	guarantees precedence of the enclosed expression. (optional)
(re)	matches the expression re
re1 | re2	matches either the expression re1 or re2
re?	matches at most one appearance of re. Note that in **sed** you need to either write \? or use the **-r** command line switch when using this.
re+	matches one or more subsequent appearances of re
re*	matches none or arbitrarily many subsequent appearances of re
re{n,m}	matches at least n and at most m subsequent appearances of re. Both n and m can be omitted either with or without the comma. Then n means exactly n matches. n, stands for at least n matches and ,m for at most m matches.
(re1)(re2)	matches re1 followed by re2 — in search and replace operations the corresponding matches can be referred to by \1 and \2
\	escapes, i.e., removes the special meaning of the following special character.

The next table contains some enlightening examples. More examples and an insight to the magic that can be performed using those expressions can be found on the **sed** homepage[5]

a?b	matches a string of one or two characters eventually starting with a but necessarily ending on b
^From	matches a line/string beginning with From
^$	matches an empty line/string
^X*YZ	matches any line/string starting with arbitrarily many X characters followed by YZ

[5] https://sed.sourceforge.net/

linux	matches the string linux
[a-z]+	matches any string consisting of at least one but also more lower case letters
^[^aA]	any line/string that does not start with an a or A.

Some scripting languages have more powerful regular expressions than others. It is always best to check the documentation about the details. The above mentioned should be the smallest intersection of all extended regular expression sets. Note the following remark from the **grep** manual page:

> *In basic regular expressions the meta-characters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \).*

The Swiss Army Knifes for Scripting Gurus Although we refer to scripting gurus in the section title the following tools are powerful helpers in scientific computing for everyone, as well. They can be used to easily scan large log-files for the important data. For example in a large computing task we may have created a file containing all kinds of status information of our code/algorithm. For the corresponding publication we might, on the other hand, only be interested in the execution times of the single steps. The tools presented in this section can then be employed to find and print those times in the proper form required for further processing. All three of them are so extremely mighty that our presentation can only scratch the surface of their possible applications. There are many online tutorials introducing them from different points of view.

grep is basically used for printing lines in a number of input files matching a given pattern. That pattern can be a simple keyword but also an arbitrarily complicated regular expression. The easiest way to use it in the introductory example would be

```
grep Time logfile
```

If you are not sure whether Time was written with capital T you can use

```
grep -i Time logfile
```

which switches of case sensitivity, or

```
grep [tT]ime logfile
```

as an example for a simple regular expression. In the case you do not remember which file in your large software project contains the definition of a certain function you can have **grep** search a complete directory recursively

```
grep -r function-name *
```

returning all lines containing **function-name** preceded by the corresponding file name. You can also negate the output of **grep** by the switch **-v** to suppress all lines that match the pattern.

sed the **S**tream **Ed**itor is a basic text editor that, in contrast to the usual text editors (like **vi**, **emacs**, **nano**, ...), is not interactive but uses certain command strings to manipulate the text file, streamed into it, automatically, i.e., without user interaction. It is especially useful when, e.g., a variable or function (or any other identifier) in a large software project is supposed to be renamed. Consider the name of variable called **complicatedname** is to be replaced by **simplename** for better readability of the code in a large C project.

The search and replace string in **sed** takes the form **s/foo/bar/**. In this form the incoming stream is searched line by line and every first match of the regular expression **foo** is replaced by **bar**. If we expect more than one possible matches per line, we should however use **s/foo/bar/g** to replace all of them. In case we only want every third appearance in a row to be replaced, the string becomes **s/foo/bar/3**. So getting back to our example C project the call for the main file might be

```
sed -i 's/complicatedname/simplename/g' main.c
```

To complete the picture we can use **find** to search for all **.c** and **.h** files (see also Chapter 3) and execute the above line for every single one of them.

```
find . -name '*.[ch]' -exec sed -i 's/complicatedname/\
    → simplename/g' {};
```

The **-i** switch in both versions is used to perform the manipulations in place, i.e., replacing the original file by the modified result. We can advise **sed** to create backup copies with a user defined suffix by simply specifying the suffix directly after the **-i** parameter as in

```
sed -i.orig 's/foo/bar/4' filename.txt
```

which copies **filename.txt** to **filename.txt.orig** prior to the manipulation. Here the 4 advises **sed** to replace only the forth match by **bar**.

sed can behave like a couple of tools we already learned about earlier. For example to print the first 10 lines of **file** like

```
head file
```

we can use

```
sed 10q file
```

as well. Also we can make **sed** emulate **grep** by using a simple search string instead of the replace string.

```
grep foo file
```

can be written as

```
sed -n '/foo/p' file
```

in **sed** and **grep -v** is performed by replacing **p** with **!p** above.

We can also employ **sed** to imitate the behavior of the tool **basename** that can be used to truncate filenames by cutting of the extension. Calling

```
basename /usr/include/stdio.h .h
```

produces the output

```
stdio
```

The same can be done by

```
ls /usr/include/stdio.h |
sed -r 's/^(.*\/)*([^\/]*)\.h/\2/g'
```

which requires the **-r** flag for extended regular expressions in order to grab the second match using **\2**.

Often **sed** is employed in conjunction with the other tools presented in this section to perform pre or post processing for those. This is for example nicely seen in the **pgrep** example in Section 1.4. There instead of using the file name argument **sed** reads the input from a pipe. So the last example above could as well be written as

```
cat file | sed -n '/foo/p'
```

The sed-one-liners list[6] gives a first impression of the real power this small tool has. We refer to the various web tutorial for earning deeper knowledge. For local information confer the info pages rather than the manual pages, since they are by far more detailed and structured.

awk The AWK utility is an interpreted programming language typically used as a data extraction and reporting tool. Its name is derived from the family names of its inventors – Alfred Aho, Peter Weinberger, and Brian Kernighan. The current specifications can be found in the IEEE 1003.1-2008[7] standard. It is invoked using

[6] https://sed.sourceforge.net/sed1line.txt

[7] https://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html

```
awk 'awk-statements' filename
```

to analyze a file. It can also read its input from a pipe:

```
... | awk 'statements'
```

Instead of specifying the **awk** statements directly on the command line an **awk** script can be used. To this end the **-f scriptfile** switch is appended to the call.

awk reads the input, processes it row by row and splits it into columns. The values of the columns are accessed using ***$columnnumber*** inside an awk-statement. For example the first column is accessed by **$1**. The pseudo column **$0** represents the complete row. The separation into columns is performed based on white spaces by default. We will se later how this behavior can be changed.

An awk-statement has the following format:

```
Condition { Action }
```

Multiple statements are used writing them one after another. The condition selects a data set on which the action is applied to. A condition can be **Expression Operator Expression** where **Expression** is a column identifier, a numeric value or a string enclosed by double quotes. The **Operator** is one of **==**, **!=**, **<**, **>**, ...

Another condition type is **Expression Operator /RegEx/**. This selects a data sets with respect to a regular expression. The **Operator** can be **\~** if the regular expression should match or **!\~** if it should not match. Two special conditions exists: **BEGIN** is executed before the first row is processed and **END** is evaluated after the last row is processed. The **print** command is the only action we need. For complex ones we refer to the IEEE Standard or literature.

Consider the following file containing some measured data

```
1 0.02 0.43
2 0.03 1.03
3 0.55 0.30
```

If we want to extract only the second column we invoke **awk** as

```
cat file | awk '{␣print␣$2;␣}'
```

All rows where the third column is larger than one are returned by

```
cat file | awk '$3>1.0␣{␣print␣$0;␣}'
```

If the column separator is not a space or a tabulator it can be redefined with **FS="Separator"** inside the begin action. If we consider the same data file as above but with **|** characters to separate the values it changes to

```
cat file | awk 'BEGIN{␣FS="|";␣}␣$3>1.0␣{␣print␣$0;␣}'
```

1.7 Remote Computing on Encrypted Connections

We have used the job execution on a possibly far away compute server in a high performance computing center as a motivating example in the above, but we have never explained how this is done. We are catching up on this here. Classicaly two commands have been used to log into a remote machine. These have been **rlogin** and **rsh**. Both names suggest what they were doing. Their main purpose was to simply open a remote terminal and start a shell on the remote machine. Both lacked certain security features like encrypted communication. Therefore they have been replaced by a modern version of **rsh** called **ssh** (for **s**ecure **sh**ell). The new **ssh** tool features higher security for user logins and encrypted data transfer between the local and remote host. It is used as in

```
ssh username@remote.machine.somewhere
```

If your local machine supports it, you can use

```
ssh -X username@remote.machine.somewhere
```

to even redirect graphical user interfaces to the local machine. Note that the latter does only make sense if the two host are connected via a rather fast network connection, because it usually generates high traffic on the connection.

There is also a command for copying files to or from the remote machine that comes along with **ssh**. The **s**ecure **c**o**p**y (**scp**) features the same security mechanisms as **ssh** itself and works very similar to the basic **cp** command. Obviously you have to add user and host information to the calling sequence. This is demonstrated in the next example

```
scp localfilename user@remote.host.somewhere:\
    → remotefilename
scp user@remote.host.somewhere:remotefilename \
    → localfilename
```

The local file name is specified relative to the current working directory or absolute (i.e., relative to the file system root). The remote files by default end up in the remote users home directory. Therefore all remote file names are specified relative to the home directory or absolute. The **scp** command can also be used to copy entire directories. Then the source file name is replaced by the directory name and **scp -r** is used instead of plain **scp** to indicate the recursive operation.

1.8 screen – an Online/Offline Terminal

We have dicussed the **nohup** utility in a previous section. There, we pointed out the disadvantages of the utility. Here, we recommend an alternative approach pursued by the GNU **screen** project that the projects web page[8] describes as follows:

> *Screen is a full-screen window manager that multiplexes a physical terminal between several processes, typically interactive shells. Each virtual terminal provides the functions of the DEC VT100 terminal and, in addition, several control functions from the ANSI X3.64 (ISO 6429) and ISO 2022 standards (e.g., insert/delete line and support for multiple character sets). There is a scrollback history buffer for each virtual terminal and a copy-and-paste mechanism that allows the user to move text regions between windows. When screen is called, it creates a single window with a shell in it (or the specified command) and then gets out of your way so that you can use the program as you normally would. Then, at any time, you can create new (full-screen) windows with other programs in them (including more shells), kill the current window, view a list of the active windows, turn output logging on and off, copy text between windows, view the scrollback history, switch between windows, etc. All windows run their programs completely independent of each other. Programs continue to run when their window is currently not visible and even when the whole screen session is detached from the users terminal.*

The main strength of **screen** for our purposes is summarized in the final sentence. It gives the ability to detach the users terminal from the **screen** session, i.e., the shell in which the computation is running. At any later time and even from a completely different terminal and location the user can then reattach to the **screen** session and continue working as if he/she had never left the screen.

Basic Usage Open a (usually remote) terminal and just type

```
screen
```

A welcome message appears. Now press the space-key and you are in a standard terminal. You can now start your favourite process, e.g.,

```
top
```

[8] https://www.gnu.org/software/screen/screen.html

and detach the **screen** session by typing ctrl + a d

You should get a

```
[detached]
```

message. You can now close the terminal and come back to your session anytime later by saying

```
screen -r
```

in a terminal.

Multiple Windows **screen** allows you to use several windows in which you can run separate processes. To open a new window, just type ctrl + a c . To switch between several windows, you can either use ctrl + a n to go to the next or ctrl + a p to go to the previous window. Alternatively, you can also say, e.g., ctrl + a 2 to go to the second window.

Which screen Processes / Sessions Are Currently Running? To get an overview about **screen** sessions we have running on a certain machine we just type

```
screen -list
```

and we will get a list of the form

```
There are screens on:
30714.pts-5.<host>      (Detached)
30769.pts-5.<host>      (Attached)
2 Sockets in /var/run/uscreens/S-<user>.
```

where `<host>` is the name of your computer and `<user>` is our user name.

Terminating screen Type ctrl + d or enter

```
exit
```

and you will get back to the terminal from which you started.

screen and ssh Probably the most useful feature of **screen** is that you can use it to start processes remotely, then log out of the remote computer and log back in (even using a different computer) and continue the session. This is useful for long computations that do not need to be monitored. Consider the following example.

We log in to a remote server via **ssh**:

```
ssh user@remote.pc.somewhere
```

Next we start **screen** on the remote host:

```
screen
```

In the newly started shell in **screen**, we then start, e.g., MATLAB® without display:

```
matlab -nodisplay
```

This has to be done because you can not log out of the remote machine without killing your processes if they use the graphical display. We then start our MATLAB computation

```
start_long_matlab_computation
```

and detach the **screen** session (ctrl + a d). We can now close the **ssh**-connection and after logging back in to the remote machine, we can pick up the MATLAB session by saying

```
screen -r
```

Other Features **screen** can also be used in a multiuser-mode which, e.g., allows one user to act as a teacher for some other user who can sit at a different computer. **screen** also offers Copy&Paste and Regions. We however refer to the **screen** documentation for details here.

1.9 tmux — a screen Alternative

The terminal multiplexer **tmux** is a tool that provides all the features of **screen**, and adds tiling features to the display. On high resolution terminals, this can be beneficial, since one can keep an eye on several things simultaneously. Note that the session commands in **tmux** are using ctrl + b instead of ctrl + a , but otherwise coincide with the ones from **screen**. The tiling mentioned above is achieved through subsequent horizontal or vertical splitting of the current view or pane as **tmux** calls them. The two key sequences for splitting the pane in two are ctrl + b " for a top and bottom split, or ctrl + b % for a left and right split. One can then navigate the panes, i.e., move the focus, or active status another pane by combining ctrl + b with the arrow or cursor keys. As usual, the man page has much more to say about options and more possible commands. Also the **tmux** wiki has lots of helpful information and a nice getting started guide.

1.10 The Toolchain

The toolchain is a wrapper expression for a set of tools that are used in programming tasks. It usually consists of

- a tool for automation of the build process,
- a compiler suite containing compiler for a set of programming languages,
- tools for generation and manipulation of binaries, libraries and assembler codes,
- a debugger helping the user in evaluating wrong code and fixing it,
- a build system that simplifies the usage of external dependencies, e.g., by automatic search for libraries and header files.

In the special case of the GNU toolchain developed by the GNU project the list reads like this:

- GNU make,
- GCC (GNU Compiler Collection),
- GNU binutils and GNU assembler,
- GDB (GNU Debugger),
- GNU autotools.

We present more detailed descriptions of the single tools or proper alternatives in Chapter 3, wherever they are needed in the process of working with a C program.

Bibliography

[1] J. Bambenek and A. Klus, *grep Pocket Reference*, O'Reilly Media, 1st ed., 2009.

[2] D. J. Barrett, *Linux Pocket Guide*, O'Reilly Media, 2nd ed., March 2012.

[3] A. Robbins, *sed and awk Pocket Reference*, O'Reilly Media, 2nd ed., June 2002.

[4] A. Robbins, *bash Pocket Reference*, O'Reilly Media, 1st ed., April 2010.

[5] T. Stubblebine, *Regular Expression Pocket Reference*, O'Reilly Media, 2nd ed., July 2007.

> Walking on water and developing software from a specification are easy if both are frozen.
>
> Edward V Berard

CHAPTER 2

Revision Control

Contents

Revision Control, also known as *Version Control* or *Source Control* is a task that is becoming more and more important also in Scientific Computing. It describes the process of monitoring changes in sets of information. The sets of information are usually documents, source codes, large web repositories or alike. The set of all information (usually files) under revision control makes a *repository*, a set of changes to a single or multiple pieces of information (files) constitutes a *revision* of the repository, and in the case of software a set of revisions defines a new *version*. For general information the terms revision and version are often used synonymously. The revisions get assigned a unique name that may be an identification number or a human readable text. The main purposes of revision control can be summarized as the following items:

1. Logging of changes: at any later stage of development of the information it is clear which change has been added by whom and when this happened.
2. Recovery of earlier states of the single pieces of information: accidental or erroneous changes can be identified and rolled back.
3. Archiving: It is possible to get back to each state of the set of information, e.g. to make computational results reproducible.
4. Coordination of joint work on the information by several collaborators.
5. Parallel development of multiple branches of the information with the possibility to merge single branches back to a main development stream.

In order to achieve this functionality the systems follow either of the two strategies

Lock Modify Write The rather restrictive *pessimistic revision control* strategy is also called *Lock Modify Unlock*. It grants single authors exclusive access to the item and thus avoids conflicts.

Copy Modify Merge This is the *optimistic revision control* strategy. It allows joint access to the items for several authors. Thus it can not avoid conflicts but will provide facilities to automatically merge easy conflicts and support the authors in resolving more complicated ones. In the case of files on a computer, binary data is often difficult for this kind of approach since there the merge step is usually not possible without additional tools.

2.1 Types of Revision Control Systems

The existing tools for revision control, of computer files, can be categorized in three large groups. These groups will be introduced in the following subsections

2.1.1 Local Revision Control

As the name suggests this version is completely local. Usually only single files are under revision control and the version information is stored locally. Often one can find the version information directly inside the file in the form of comments at the beginning or end of the file. Prominent implementations of local revision control are the classic **S**ource **C**ode **C**ontrol **S**ystem (SCCS) or the more well known **R**evision **C**ontrol **S**ystem (RCS)[1]. Both systems have classically been employed on Unix-like systems for revision control of single source code files. Local revision control is also implemented in modern office applications like Microsoft Word or OpenOffice/LibreOffice Writer to track changes of ones collaborators.

2.1.2 Central Revision Control

This type of revision control is different from the previous in that it stores the version information in a central (possibly remote/online repository). Users connect in a client server way to this central resource. The actual local copy of the files the user is then manipulating is usually called *working copy*. The basic concept of central revision control goes back to the open source project **C**oncurrent **V**ersions **S**ystem (CVS)[2] and has been made even more popular by the Subversion (SVN)[3] system. The working copy usually contains information about a single version. This version is either the one the central repository was in while the local copy was created, or the one it had when the local version was last synchronized to it. This version is usually called *HEAD* revision. Local changes can usually only be determined with respect to this HEAD revision. These are the changes that are merged into the central repository when the local changes are submitted. This procedure is generally called *commit*.

2.1.3 Distributed Revision Control

The major disadvantage of central revision control systems, that use an online server for storing the central repository, is the requirement for an active network connection for determining version information and changes during revisions other than the HEAD revision. Distributed revision control systems are a way to overcome this drawback. They feature local repositories in which the entire version history is stored. Local working copies are synchronized against these local repositories. The local repositories are then synchronized to either the repositories of collaborators or central repositories in online resources.

The local repositories feature a very quick access and allow for fine grained version management and logging of changes. Therefore, usually the distributed revision control systems often have much more powerful merge facilities.

Important distributed revision control systems in the open source world are Git[4] which has among other authors been developed by Linus Torvalds, Bazaar[5] that

[1] `https://www.gnu.org/software/rcs/`
[2] `https://www.nongnu.org/cvs/`
[3] `https://subversion.apache.org/`
[4] `https://git-scm.com/`
[5] `https://github.com/bazaar-community/bazaar3`

is mainly developed by Canonical Ltd. (who are the driving force behind Ubuntu and distributions derived from it.), and Mercurial[6].

2.2 Collaborative Work on Projects

Especially the central and distributed revision control systems are very attractive for collaborative work on entire projects. While for local revision control all collaborators require access to the same file or need to exchange it, the latest version of an entire project is always accessible for all coworkers in a central repository or independent local repositories. This allows for a highly increased flexibility in editing the files.

2.2.1 Conflicts

When editing different files of the same project, or a common file in disjoint positions, usually these systems can automatically merge the changes of several authors into a single repository. In case of changes in common locations of single files, these systems offer conflict management facilities that support users in resolving the conflicts possibly generated by editing the same locations in the file.

2.2.2 Branches

A common way to avoid conflicts is the technique of *branching*. The main development line of a project is often called *trunk*, or *master*. Just like the trunk of a natural tree this version is the fundamental part of the project. A branch is then splitting off of this main version as an exact copy of the trunk. Then, it can be used to develop, e.g., a certain feature without harming the main development. In contrast to the biological tree, the branches do in general return to the trunk after a while, e.g., when the feature is ready to enter the main development stream. In the case of central revision control, these branches are usually linear sequences of revisions. For distributed systems with enhanced merging capabilities, the branches are often even branched further, such that the entire object becomes a directed acyclic graph of revisions.

2.2.3 Tags

Especially when developing software, certain revisions are more important than others, e.g. because they are used as release versions. It is then important to create so called *tags*, i.e., named revisions to have an easy means to reproduce this exact state of the repository. The way tagging is implemented, or being used is differing among the systems, but it is always possible in one way or another.

[6] https://repo.mercurial-scm.org/hg/help

2.3 Revision Control meets Social Networking

Some of the most prominent providers of Revision Control services for the open source community are GitHub[7], BitBucket[8], and GitLab[9]. Today also many scientific codes are hosted on, e.g. GitHub. In contrast to the others, GitLab also offers their software for personal use in an open source version with limited features.

All off the above have one thing in common. They do not only provide plain revision control features, but add social networking type functionality to their systems. This, for example, allows users to monitor the changes of projects, including those that they are not participating in themselves, and check the activity of other users, such as their fellow developers.

2.3.1 Issues

The expression *Issue* is used as a collective term for

bug: A bug report means a report about faulty or erroneous code. It can be as specific as a code snippet for a suggested correction, or as vague as "this unexpected behavior was observed".

feature request: No software will ever fully satisfy its users. As a consequence users will come up with lists of additional functionality the software could respectively should feature in future releases. They are collected as feature request issues.

suggestion: Similar to the above but mixing the two for things that are not exactly wrong but rather can be done, e.g., more optimal or efficient.

2.3.2 Pull Request / Merge Request

In the course of collaborative work on a joint software project, usually some developers are more priviledged than others. The so called *maintainers* or *core developers* are handling the master branch of the software, while their fellow developers create bug fixes or new features in their own branches. Whenever they are finished, the standard developers can not merge their changes back to the master branch themselves. In order to notify the maintainers about their work being ready to get merged, they post *pull requests* (also called *merge requests* in some systems). Usually the `git` systems allow the maintainers to easily review the pull request in the online system. This feature usually shows them all the changes made to the code base such that they can easily and quickly decide

[7] https://github.com
[8] https://bitbucket.org
[9] https://gitlab.com

whether or not to take the changes into the master, or maybe request additional changes to maintain project consistency or the like.

2.3.3 Forks

In case one wants to contribute to a project one is not yet part of, the easiest way to get involved is to just grab a copy of the project. This can be done using a *fork*. Acting similar to a branch, the fork is created as ones own project still maintaining the connection to the original project. Then obviously one can work freely on the own copy. Once the work is finished, again a pull request to the original projects maintainers can make the changes flow into the original project and get oneself involved.

A second scenario where forks are important is the situation, where one wants to pick up the work of another project that has been discontinued. In that case the fork enables new maintainers to continue the work on a project that has been orphaned by its original developers. This is a common situation in academia when people finish their theses and leave academia to industry, while at a later point in time other people at possibly other academic institutions want to continue their research.

2.3.4 A generic workflow

While the `git` system in general gives you maximal power with no actual prescription on how to use and distribute this power, a certain workflow is commonly observed on the above mentioned platforms. The workflow consists of the following few steps and usually maintains project and branch integrity, thus avoiding conflicts and problems.

1. someone opens an issue
2. a maintainer assigns the issue to a developer
3. the developer creates a new project branch connected to the issue
4. the developer changes the code to close the issue
5. once finished, the developer files a pull request
6. the maintainers review and merge the pull request, and close the issue.

2.4 What to put into a version control repository

Most modern version control systems for code are based on monitoring changes in files via tools like **`diff`**. Some additional features about detecting name changes for files and their addition, or removal complete the picture. This gives us a rather clear idea about what to put into a repository. Since derived binary files often change all over the place when only little is actually changed, diff will have a hard time with them. That means, PDF-files due to their encryption/compression of the content are bad. Similarly, JPEG images will have widespread changes due to the integral folding used to generate them, even when only few pixels change. The same holds true for compressed archives in ZIP, BZIP2 or similar formats. On the other hand, **`diff`** works best with plain text files.

This allows us to formulate some simple rules of thumb:

1. plain text files are good, especially when they do not change much.
2. image files and compressed files should only be added if they at most change VERY rarely; at best never, especially when they are large.
3. automatically generated files, e.g. compiled versions of source code, can always be generated again and should not be put under version control.

Some systems allow special treatment for large binary files. Then they are less harmful.

Bibliography

[1] bleeptrack and blinry, *OhMyGit – an open source game about learning git*, `https://ohmygit.org`. See also `https://github.com/git-learning-game/oh-my-git/`.

[2] S. Chacon and B. Straub, *Pro Git*, APRESS, 2nd ed., Sept. 2023, `https://github.com/progit/progit2/releases/download/2.1.411/progit.pdf`. version 2.1.411.

[3] git Community, *The official git tutorial*, `https://git-scm.com/docs/gittutorial`.

It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

How do we tell truths that might hurt?
Edsger Wybe Dijkstra

CHAPTER 3

Concise Introduction to the C Programming Language and the GNU Toolchain

Contents

One of the main goals of the lecture is to understand how mathematical algorithms are translated into a high-level programming language. This includes an overview how efficient implementations basically work. We chose C for many reasons instead of other high-level languages like C++, Java, or Fortran:

- C is easy to learn. It has only about 30 keywords.
- C has been one of the most often used programming language for a long period of time[1]. Even thirty years old programs work today.
- C is standardized by ISO in ISO/IEC 9899 (see [5, 7, 9]).
- C works on embedded systems, as well as, on the largest super computers.
- C can be combined with nearly all other popular programming languages. Even scripting languages or assembler code can be embedded.
- A large variety of libraries exists: GUI-programming, networking, mathematical algorithms.

The first version of C was developed by Ken Thompson, Dennis Ritchie and Brian W. Kernighan in the early 1970s for developing their UNIX operating system. Since then the concepts and the syntax of C have influenced many programming languages. In 1978 the K&R book [11] appeared. This book defines the first quasi standard of the C syntax. Caused by the popularity and its strong connection to UNIX many vendors have created their own subsets of C with different extensions. This became a major problem for exchanging code and lead to the standardization of C by an ANSI committee, founded in 1983. The committee released the first standard in 1989. This standard directly became an ISO standard in 1990 [5]. The standard was revised and extended in 1995, 1999 [7], 2011 [9], 2018 [10] and 2023 (upcoming). Currently, the C23 standard is only experimental and will likely see the light of day as ISO/IEC 9899:2024 next year. ISO C17 (actually ISO/IEC 9899:2018) is mainly a clarification and revision of the C11 standard, which has been implemented in the all major compilers.

C does not restrict the programmer to a fixed programming style. This allows nearly unreadable code which works correctly. Although with IOCCC[2], there is a contest focusing on the exploitation of this freedom, one of the aims of the present text is to also create awareness of the curse that hides within this freedom.

In the remainder of our presentation we assume that a Unix-like operating system (such as Linux, *BSD or MacOS X) with the GNU Compiler Collection (Version

[1] `https://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

[2] `https://www.ioccc.org/`

8.1.0 or later for full C17 support) is used.

3.1 The Programming Environment

Before we can run our first self-written program we have to understand how to process a human readable source code to an executable program. A C program consists of at least one text file with extension `.c`. This is created with a normal text editor like vim, emacs, kate, gedit, ... or an integrated development environment (IDE) like VS Code[3], kdevelop[4], eclipse[5],.... Word processors like MS Word, LibreOffice, OpenOffice, etc. are *not suitable* for this job.

Four steps are necessary to transform the human readable source code to an executable program:

1. **The Preprocessor** searches the source code for special directives beginning with **#**. These directives can include other libraries, dynamically include and exclude code, or modify the program using a complex pattern matching search and replace mechanism. The output of this phase stays human readable but the code is filled with additional statements and data from other files.
2. **The Compiler** is the main tool. It checks whether the source code is syntactically correct. Afterwards the preprocessed source is translated into assembler code. An optimization phase may speed up the code and adapt it to the features of the CPU. The assembler output is still human readable, and it expresses the same instruction as the C source on a much lower abstraction level.
3. **The Assembler** turns the assembler output into machine code. This can theoretically be executed by the CPU, but missing external libraries prevents this. The output of the step are *object files*. An archived collection of object files is used as static library. See Section 3.10.
4. **The Linker** finally merges the object files and the libraries into one executable program. It checks if all necessary functions and symbols are found in the object files and the specified libraries.

These four steps are usually performed by single compiler call. The compiler performs all steps and creates the executable directly from the source code.

The GNU Compiler collection provides one command for all steps. The **`gcc`** command invokes preprocessor, compiler, assembler and linker. Sometimes it is necessary to invoke the linker separately with **`ld`** or **`gcc`**.

[3] https://code.visualstudio.com/
[4] https://kdevelop.org/
[5] https://www.eclipse.org/

The C compiler is invoked in the shell:

```
gcc <options> -o outputfilename input1.c ... <libraries>
```

This compiles all given input files to one executable. If the output filename is omitted the compiler uses **`a.out`**. The behavior of the compiler is influenced by a variety of compiler options. Some important ones are:

Binary code optimization:

`-Os`	Optimize the code to reduce the size of the binary.
`-O1`	Turn on basic optimizations. The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
`-O2`	Optimize even more. GCC performs nearly all optimizations that do not involve a space-speed trade-off. As compared to **`-O1`**, this option increases both compilation time and the performance.
`-O3`	Aggressive optimization. It tries to unroll loops constructs and inlines small functions. It can cause unexpected effects in the program. The output is usually larger then using **`-O2`**.
`-march=native`	Automatically determines the code generation options to optimally exploit your local CPU features. **Code may not be executable on other machines.**

Debugging:

`-g`	Include the debug symbols in the output. This is necessary for tools like **`gdb`**, **`ddd`** or **`valgrind`**.
`-pg`	Include the profiling information for the GNU profiler. Execution in **`gprof`** then produces the desired information.

Floating Point Arithmetics related:

`-ffast-math`	Turns off the IEEE754 floating point arithmetics. **This option is dangerous.**
`-ffloat-store`	Floating point operations store the results to the memory instead of keeping them in high accuracy CPU registers.
`-mfpmath=sse` **`-msse2`**	Use the SSE2 registers for floating point operations instead of the classical x86/x87 floating point unit. **Only available on x86 and x86_64 platforms. `-mfpmath=sse` default on x86_64.**

-mavx **-mavx2**	as above but for the more recent AVX and AVX2 registers.

Warnings and C Standards:

-Wall	The compiler displays all warnings about malformed code.
-std=XXX	Defines the C standard to use. Normally explicit usage is not necessary. Possible values: **c89**, **c99** or **c11**.

Finding libraries and header files:

-I*path*	Set an addtional search path for the **include** directive. This can be used multiple times.
-L*path*	Set an additional search path for the linker.
-l*NAME*	Link a specified library to the program. The **lib** prefix is automatically added to the library.

Compilation of own libraries:

-c	Compile the source code to object files without linking it. The default output name is **inputname.o**.
-fPIC	Generate *position independent code*. This flag influences the assembler code production to use relative addresses. It is necessary for libraries.

Code Preprocessing and basic shared memory parallelism:

-D*NAME*=*VALUE*	Defines a preprocessor variable *NAME* and sets it to *VALUE*
-fopenmp	The OpenMP support is enabled.
-pthread	The PThread support is enabled.

If a program consists of many source files, or they need different compiler options, it is more convenient to create the single *object files* first:

```
gcc -c input1.c
gcc -c input2.c
...
```

Afterwards the *object files* are linked with libraries to the final executable:

```
gcc -o output input1.o input2.o ... <options>
```

External libraries are added using the **-l** option. The standard C library and system dependent ones are added automatically. A library named **libNAME** is linked using **-lNAME**. The linker adds the **lib** prefix automatically. The libraries must be specified in the order they depend on each other (rightmost libraries are the most independent). Cyclic dependencies are solved by adding the libraries more then once to the linker invocation.

Example 3.1: A program depends on **libone**, **libtwo** and **libthree**, where **libtwo** depends on **libone**. The resulting compiler call is:

```
gcc -o output input.c -ltwo -lone -lthree.
```

Libraries are existing in two types. The classic approach of combining single object files in a reusable library is to glue them together in a static library (usually ending on **.a**). Upon linking, all of the object contained in the library are added to the program executable. This usually results in fairly large binary commands. The more modern approach is to use so called shared object libraries (usually ending on **.so**) or also dynamic link libraries. These are kept external and library symbols and commands are included only upon execution of the program. The dynamic loader loads all external libraries when a program is executed. It searches for them in the standard paths of the operating system. If a library does not reside in these directories the search path can be extended by setting the **LD_LIBRARY_PATH** environment variable.

Example 3.2: A program uses a library in a non standard location. It is compiled and linked using

```
gcc -o output input.c -L/path/to/the/library -lthelib
```

and executed with adding the path to **LD_LIBRARY_PATH**:

```
export LD_LIBRARY_PATH=/path/to/the/library:\
    → $LD_LIBRARY_PATH
./output
```

Many tools exists to support the programmer during development and debugging. The basic ones are:

gdb The GNU Debugger is a command line tool that helps executing a program step by step, and enables to look into variable values at runtime, or view the machine code. It allows a deep analysis of what is going on in the program. Available at https://www.gnu.org/software/gdb/

ddd The Data Display Debugger is a graphical user interface for **gdb**. Available at https://www.gnu.org/software/ddd/

valgrind Is a suite of debugging tools which analyze the memory access, check for memory leaks, create call graphs,...Its graphical front end is called **valkyrie**. Available at https://www.valgrind.org

nm Lists all symbols (functions or variables) in an object file or a library.

ldd Lists all external libraries required by a program. It also checks if they are found in the current search paths and shows which ones will be used upon execution of the program.

make An automatic build utility. Details can be found in Section 3.9.

3.2 C Statements, Types and Operators

The basic structure of a C program looks like

```
#include <stdio.h>
#include <stdlib.h>
// more includes
...
// type definitions (see Section 3.4)
...
// function definitions (see Section 3.5)
...
int main (int argc, char **argv) {
  // Here comes the code.
  return 0;
}
```

The **include** statements above are called *preprocessor statements* (see Section 3.8). They include so-called *header files* containing information about external libraries or functions and variables in the current source files. **stdio.h** and **stdlib.h** are two header files from the standard C library. They provide basic input and output, access to files and other basic actions. They are necessary for essentially every program.

main() is the function that is called when a program starts. All statements are executed in the order in which they appear. The **return 0;** statements exits the **main()** function and returns a status code to the operating system. The 0 as a general convention means that a program terminated successfully. All other values are treated as errors.

Comments. Lines beginning with "**//**" are comments. The compiler ignores them but they should be used to help human readers to understand the code. Comments can also be used to prevent the compiler from including certain parts of the code. Possible comment structures are:

```
// A single line comment

/* Another single line comment */

/* This
```

```
is
a multi-line comment */

#ifdef GRAPHICS
Some code fragment
#endif /*GRAPHICS*/
```

Here the last one is a pre-processor based comment. So it is not a comment in the original sense. On the other hand, they allow to exclude large portions of code based on Macro definitions. Here, **`GRAPHICS`** is a pre-processor macro that could, e.g., be used to enable certain graphical output only when the macro is defined. This is a common way to exclude graphical interfaces from compilation for compute servers that do not supply the corresponding libraries. More details regarding this can be found in Section 3.8

Statements and Blocks. A *statement* in C can be one of the four kinds:

- variable declaration

  ```
  data-type varname;
  ```

- function call

  ```
  dosomething();
  ```

- assignment

  ```
  x = 3;
  ```

- control structure (see also Section 3.3).

All statements are case sensitive and must end with a semicolon. Line breaks are ignored by the C compiler. This allows more than one statement per line. Statements are grouped to code blocks using { and }:

```
{   // begin of the code block
  Statement1;
  Statement2;
  ...
}   // End of the code block
```

Basic Data Types and Variable Declaration. A variable needs to be declared prior to its first usage. The declaration consist of a data-type followed by a comma separated list of variable names. A valid variable name begins with a alphabetic character, only contains "_" as special character and is not used for another variable or function in the context. Variables need to be declared at the beginning of a block or a function following the C89 standard. The C99 standard allows this everywhere. Nevertheless for better readability it is recommended to follow C89. A variable only exists inside the {}-parentheses where it is declared. Variables are **not** initialized with a default value. Common built-in data-types are:

`int`	Stores one signed integer value. Normally, this is 4 byte large, that means it can store one 32-bit number.
`long`	Stores one large signed integer value. This must have at least the size of an `int` variable but it can be larger. On a 64-bit architecture this is normally 8 byte.
`unsigned int`	Stores an integer without a sign, that means only positive but larger numbers.
`unsigned long`	Stores a `long` without a sign, that means only positive but larger numbers.
`char`	Stores one character from the ASCII table. Internally, it is a one-byte integer value and holds values from -127 to 128.
`size_t`	An unsigned integer value which is large enough to store the size of the largest theoretically possible memory object. Its size depends on the hardware of the platform used.
`float`	A single precision floating point number, 4 Bytes.
`double`	A double precision floating point number, 8 Bytes.
`void`	Non specified type for function with no return value or generic pointers.

There was no boolean data-type in C until the C99 standard. Boolean values are therefore expressed as integers where zero means `false` and all other values are evaluated as `true`. The definitions of variables of basic data types can also contain initial assignments.

Example 3.3:

```
int x = 1, y;
```

The above definition declares two integers x and y and initializes x with the value 1. The character type `char` is assigned using single quotes:

```
char c = 'A';
```

The single quotes implicitly convert the given character in to the corresponding ASCII value. We introduce strings in Section 3.4.

Operators. The basic arithmetic operations $+$, $-$, $*$, and $/$ are known to C. The modulo operator $\%$ exists only for integers. If both operands are integers then the operations expression is evaluated in integer arithmetic. The division discards the fractional part in this case. The compiler pays attention to the arithmetic priority rules. Parentheses influence the evaluation order.

Example 3.4:

```
int x,y,z,r;   // Declares x,y,z, and r to be integers
x = 4;         // Sets x to 4
y = 3;         // Sets y to 3
z = x / y;     // Integer Division of x and y
r = x % r;     // Modulo, the remainder of the division
```

If the left side of an assignment is the same as the first operand of a binary operation this can be abbreviated as in:

```
x += y;  // same as x = x + y;
```

This is possible with all binary operators. The **++** and **––** operators increment or decrement a variable by one. They are used as pre- or postfix to a variable. The prefix increments the variable before its value is used. The postfix does it the other way around.

Example 3.5:

```
int x = 1, y;
x++;              // x = 2;
y = ++x;          // y = 3; x = 3;
y = x++;          // y = 3; x = 4;
```

Bitwise operators are available in C too:

`x & y`	Perform a bit-wise *and* operation.
`x | y`	Perform a bit-wise *or* operation.
`x ^ y`	Perform a bit-wise *xor* operation.
`~x`	Perform a bit-wise *not* operation.
`x << y`	Bit-Shift on **x**. Move **y** bits to the left.
`x >> y`	Bit-Shift on **x**. Move **y** bits to the right.

A *typecast* is used to convert one data-type into another one. It is performed by putting the new data-type in parentheses in front of a variable.

```
int y; double x;
x = (double) y;  //  converts y from int to double
```

Besides dealing with variables one usually needs input and output operations, e.g. for printing computation results to the screen, or reading user inputs from the keyboard. The standard C library provides **printf** and **scanf** for this purpose. The syntax of **printf** is

```
int printf("format␣string", list, of, variables, ...);
```

The first argument is the string printed to the screen. Variables are embedded to this string using placeholders. The placeholders are replaced in the order of the occurrence with the variable from the list of variables. The placeholders need to be chosen in correspondence to the data-types of the variables. Placeholder start with % followed by a type specifier (see Table 3.2). A new line is created with the "\n" escape sequence. The "\t" (tabular) is used for alignment of the output.

Example 3.6:

```
int x = 1;
double y = 1.8;
printf("x␣=␣%d␣and␣y␣=␣%g\n", x, y);
```

prints:

x = 1 and y = 1.8

The **scanf** function reads variable values from the standard input (usually the keyboard, or redirected outputs from other programs). It works analogous to **printf**. The syntax is

```
int scanf("format␣string", list, of, variables, ...);
```

where the format string is similar to **printf**. **scanf** tries to match the inputs with the placeholders and stores them to the variables in the order of their appearance. Because the variables are modified by **scanf**, they need to be prefixed with the *address-of operator* &. Details about & are given in Section 3.4 and 3.5. The return value is the number of variables read during the function call.

Example 3.7: To read one integer and one floating point number from the standard input and print them on standard output one needs to do the following:

```
int x;
double y
scanf("%d␣%lg", &x, &y);
printf("You␣typed␣%d␣and␣%g\n", x, y);
```

3.3 Control Structures

The program flow is controlled with statements of two categories. The first ones are conditionals, the second ones are loops.

Conditionals. C has two conditional statements: **`if`** and **`switch`**. The **`if`**-statement realizes an alternative. The simplest one is:

```
if ( condition ) {
  Statements evaluated if the condition is true;
}
```

The **`condition`** is an expression which is evaluated to be either **`false`**, i.e., equal to 0 as integer, or **`true`**, i.e., not equal to 0 as integer. Comparison operators exist for all numerical data-types, such as **`int`** or **`double`**:

`<`	smaller than
`<=`	smaller than or equal to
`==`	equal to
`!=`	not equal to, same as `~=` in MATLAB
`>=`	greater than or equal to
`>`	greater than

Boolean operators combine different conditions:

`&&`	boolean and
`||`	boolean or
`!`	boolean negation, prefix operator

Remark 3.8: Conditions are evaluated from left to right. The evaluation is stopped if the result is obvious. The `&&`-operator cancels the evaluation as soon as the first expression evaluates **`false`**. The `||`-operator cancels the evaluation when the first expression evaluates **`true`**.

Remark 3.9: The assignment operator **=** is true for every non zero right side.

```
if ( x = 5 ) {
  // executed independently of x
}
```

Some compilers are able to detect such errors (the authors intention in the example would most likely have been to check whether x equals 5 via **`x == 5`**) and print a corresponding warning.

The **`if`** statement can be extended to an **`if-else`** construct. This full alternative is:

```
if ( condition ) {
  Statements evaluated if the condition is true;
} else {
  Statements evaluated if the condition is false;
}
```

If more than two cases are necessary this extends to:

```
if ( condition1 ) {
  Statements evaluated if the condition1 is true;
} else if ( condition2) {
  Statements evaluated if the condition2 is true;
} else {
  Statements evaluated if the condition1 and 2 are false;
}
```

This concept works for more than two conditions analogously.

A conditional assignment

```
if ( condition ) {
  a = value1;
} else {
  a = value2;
}
```

can be reduced with the help of the **`?`**-operator to:

```
a = (condition)? value1:value2;
```

This is the only ternary operator in C.

The discrete decision statement in C is **`switch`**. The syntax is

```
switch(variable){
  case const_1:
  Statements if variable == const_1;
  break;
  case const_2:
  Statements if variable == const_2;
  break;
  default:
  Statements if none of the other cases matched.
}
```

The appropriate block is executed according to the variable compared to the constant expressions in the **`case`**-statement. The **`break`**-statement ensures that the statements in the following cases will be ignored. If there is no **`break`**-

statement the program runs trough all other following cases until a **break** statement is detected. This is used to merge different cases easily:

```
switch(variable){
  case const_1:
  case const_2:
  Statements if variable == const_1 or variable == const_2;
  break;
  default:
  ....
}
```

The **default**-statement defines a special case. It is executed if none of the other **case**-statements matched the value of the variable. **switch** only works on discrete data. Interval conditions like **x>4 && x<4.5** require an **if-else** construction.

Loops. C provides three different loop constructions: The **for**, the **while**, and the **do-while**-loop. A loop repeats a group of statements until certain conditions are met. The easiest one is the **while**-loop. It repeats a block as long a condition is true. The syntax is

```
while (condition) {
  Statements executed as long as the condition is true;
}
```

The condition is tested every time the loop is entered. If it is false at the beginning the **while**-loop is not executed. The condition works exactly as in the **if**-statements.

A slight modification of the **while**-loop is the **do-while**-loop. It repeats a block as long a condition holds true but the block is guaranteed to be executed at least one time and the condition is tested upon exiting the code block. The syntax is:

```
do {
  Statements executed as long as the condition is true.
} while (condition);
```

The semicolon at the end of the statement is untypical but mandatory.

The most general loop statement in C is the **for**-loop. It is mostly used for enumerations but it can emulate every other loop construction. The syntax is:

```
for (initialization; condition; action) {
  Statements inside the loop;
}
```

The *initialization* is executed once before the body of the loop is entered for the first time. It is used to initialize variables (most commonly the loop counter).

The loop is continued as long as the conditions stays true. The *action*-statement is executed at the end of every loop. This is mostly an increment or decrement statement. A **for**-loop is equivalent to a **while**-loop of the form:

```
initialization;
while (condition) {
  Statements inside the loop;
  action;
}
```

Each of the three parts inside the **for**-definition can be made up of multiple expressions separated by commas. They are evaluated from left to right and represent the value of the last expression.

Example 3.10: Output all square numbers from 1 to 10:

```
int i;
for (i = 1 ; i <= 10; i++) {
  printf("␣%d␣*␣%d␣=␣%d\n", i, i, i*i);
}
```

Loops can be influenced via the **break**- and the **continue**-statement. The **break**-statement is an emergency exit inside a loop. It exits the loop immediately and stops its repetition neglecting the condition. The program continues in the first statement after the loop.

```
while ( condition ) {
  Statements;
  if ( special condition ) {
    break;  //Exits the loop regardless of the while-
        condition
  }
}
// Control jumps here on the break
```

The **continue**-statement causes the control to jump to the end of the code block defining the loop immediately skipping the remaining statements. If the condition allows it the next iteration is then started. If a **continue**-statement is called inside a **for**-loop it still evaluates the *action* statements.

```
while ( condition ) {
  Statements;
  if ( special condition ) {
    continue;
  }
  Statements;
  // Control jumps here on the continue;
}
```

Remark 3.11: Control structures can be nested inside each other as often as desired.

Remark 3.12: If a control structure only executes one statement, the surrounding brackets {} defining the code block can be omitted.

3.4 Complex Data Types and Arrays

Simple scalar values or characters are not sufficient for the applications. This section extends the basic data types by structures, arrays, strings and pointers. For enumeration, type definition and unions we refer to the literature [14, 15, 7].

Structures. Data-structures are collections of different variables within a common context. They are defined using the `struct`-statement:

```
struct NameOfTheStructure {
  data-type1 variable1;
  data-type2 variable2;
  ...
};
```

We replace the data-type of a variable by `struct NameOfTheStructure` to declare a variable to be a data-structure.

```
struct NameOfTheStructure variable;
```

The `.`-operator provides access to the components of a structure:

```
variable.member = ...;
x = variable.member;
```

Example 3.13: We define a structure representing a point in $\mathbb{R}^3$ and let $P = (0, 1, -1) \in \mathbb{R}^3$ of this type:

```
struct point3d {
  double x, y, z;
};
struct point3d P;
P.x = 0.0;
P.y = 1.0;
P.z = -1.0;
```

The normal assignment operator copies a structure to another one. However the comparison operator **==** does not work this way. If we want to compare two structures we need to compare all components separately.

Arrays. Arrays provide a multi-dimensional storage for data of the same data-type. The data is accessed using a zero-based indexing scheme in each dimension. A one-dimensional array is declared using:

```
data-type name[NumberOfElements];
```

The bracket **[]**-operator provides the access to the elements:

```
x[0] = y;        // Assignment of the first element
h = x[i-1];      // Access to the i-th element
```

The array-elements are indexed from 0 to **NumberOfElements**-1.

Remark 3.14: The access to an array is not checked for violation of the array bounds. Neither the compiler, nor the runtime environment can detect violations. Accessing elements that lie outside the declared region can crash your program, or manipulate other data of your program unintentionally. The typical error message in the first of these two cases is a **Segmentation Fault**, resulting from the attempt to access a memory segment that is not belonging to your program, which is detected by the memory management facilities of the operating system.

Example 3.15: We declare a vector $a \in \mathbb{R}^4$:

```
double a[4];
```

It consists of four values **a[0]**, **a[1]**, **a[2]** and **a[3]**.

The same scheme allows to declare n-dimensional arrays. A two-dimensional array can be declared using 2 brackets, a three-dimensional with three brackets and so on. The array data is arranged with the elements of the right most index next to each other in the memory. That means the element **x[i][j]** resides right before **x[i][j+1]**.

Remark 3.16: This is a difference to Fortran where the data is arranged the with regard to left-most index.

Every data-type can be made up to an array. Arrays of structures are possible and arrays can be used as members of structures.

Example 3.17: We declare an array of 10 Points in $\mathbb{R}^3$:

```
struct point3d {
  double x, y, z;
};
struct point3d points[10];
points[0].x = 10.0;      // Set the x value of the first
    point.
points[9].z = -1.0;      // Set the z value of the last
    point.
```

Strings. Strings are a special case of arrays. Per definition a string is only an array of characters. Since a string does not necessarily have to be as long as the surrounding array storing it, C uses a special technique to determine the end of the string. The end of a string is marked adding a 0-byte (ASCII: NIL). Every string operation stops reading when it reaches the 0-byte. As a consequence, a string of n characters requires a character array of $n + 1$ elements. In contrast to single **`char`** constants a string is assigned using double quotes. The double quote operators automatically terminate the string by the trailing 0-byte.

Example 3.18: The string "Hello!" is stored in an array of 10 characters:

```
char string[10] = "Hello!";
```

This will be stored as

Index:	0	1	2	3	4	5	6	7	8	9
Value:	'H'	'e'	'l'	'l'	'o'	'!'	0	*	*	*

in memory. The *s are undetermined values that are left over from earlier usage of the memory segment.

String manipulation functions are presented in Section 3.6.3.

Pointers. Pointers are the most powerful concept of C and at the same time the most difficult for beginners using the language. A pointer is a variable which contains a memory address instead of a normal value. It is a reference to a memory segment where the actual data is located. The following metaphor explains this in a more natural way:

> Imagine the memory as a big long street with houses on it. Each variable in a program is a house on this street. Each household can hold a number of people (which is the value of the variable). The address of the house is the memory location of the data. Now a

pointer is a variable which contains such an address.

A pointer is declared like a normal variable with an additional `*` in front of the variable name:

```
data_type *a_pointer_to_data_type;
```

A pointer needs to be assigned to a valid memory location. The operating system takes care of this. An illegal access will kill the program just like in Remark 3.14. The *address-of operator* `&`, which was already mentioned in Section 3.2 for the **scanf**-statement, returns the address of a variable. In the case of an integer this looks like:

```
int var_x;       //declares an int variable
int *ptr_x;      //declares a pointer to a int variables
var_x = 2;        //Sets the value of var_x
ptr_x = &var_x; //Assigns the pointer to the location of
    var_x
```

ptr_x contains the memory address of **var_x**. The dereferencing operator `*` is the counterpart to the `&`-operator. It allows to access the data inside the given address. Continuing the previous example

```
*ptr_x = 12;
```

will overwrite the value in the memory location stored in **ptr_x** with 12. That means **var_x** is now 12. Unused pointers should be set to **NULL** which represents 0 in the pointer context. This allows checks if a pointer is used, or not. The **void*** pointer is the generic pointer which can be type cast to any other pointer.

From the basic data type point of view pointers are not very useful. However, there is a close relation between pointers and arrays in the C language. This is best explained by following code:

```
int field[10];
int *ptr;
ptr = &field[0];
```

Then the pointer refers the first element of the array. Now we can access **field** by **ptr**:

```
int x = ptr[3];
ptr[4] = 4711;
```

In this way a pointer is simply an alternative representation of an array without a previously known size. A pointer to a single value can be considered as a pointer to an array of one element. The array-style access is, however, **not** valid for **void*** pointers.

Remark 3.19: Note that in expressions as **ptr[3]** above the brackets represent a dereferencing operation for the element chosen by the enclosed index and thus no additional * is needed

A pointer to a structure is used similarly. Dereferencing the pointer is done using the *-operator and the access to the components is done using the .-operator:

```
struct point3d p;
struct point3d *sptr;
sptr = &p;
(*sptr).x = 0.0;
```

This type of notation **(*sptr).x** looks a bit confusing and complicated. The C syntax therefore has an equivalent representation as in:

```
sptr->x = 0.0;
```

Pointers can also be cascaded. That means, constructs like **int** ptr;** are valid. Following the above example this contains a pointer to a pointer to an **int**. Dereferencing one time give the pointer to an **int** and double dereferencing gives the integer. This corresponds to a two-dimensional array. Analogously three or more * can be used to implement higher dimensonal dynamic arrays. Note that to really exploit the dynamic features of pointers one needs to employ the **malloc()** and **free()** functions (introduced below) from the standard library (both in **stdlib.h**) described in Section 3.6.1.

Pointers are also necessary if a function should be able to modify an argument passed to it. The **scanf**-example in Section 3.2 showed this already. The Section 3.5 describes this technique in more detail.

Some arithmetic operations can be applied to pointers too. We however consider this a dangerous technique for accessing elements in the memory that should only be used by experts where it is unavoidable. For details see one of the numerous tutorials on the Internet.

Type Definitions. Type definitions are one way to create abbreviations for existing data types in C. They are used to get short version of structure definitions or to create meaningful abbreviations for existing types. The syntax of a type definition is

```
typedef original type name aliasname;
```

The aliasname is used afterwards instead the original type like any other data type in a variable declaration or function declaration. The newly defined type is compatible to its original type and the compiler performs all type casts automatically. Additionally the compiler throws an error if an aliasname is used twice in a namespace.

Example 3.20: An unsigned index type for an array can be defined using

```
typedef unsigned int indextype;
```

Example 3.21: A structure definition can be abbreviated using

```
typedef struct {
  datatype component1;
  datatype component2;
} structname;
```

and then instantiated via

```
structname variable1, variable2;
```

Furthermore, type definitions can be used to hide pointers. In this case the type definition is done using:

```
typedef original type * pointer_type;
```

Then a pointer to the original type can be defined using the **pointer_type** which adds the * in the variable definition.

Remark 3.22: If the defined type already contains a pointer like

```
typedef int* pointer_to_int;
```

the variable definition

```
pointer_to_int *x;
```

makes x a double pointer.

Memory Management. Until now every pointer needed to have a predeclared variable to refer to. In many practical examples it is, however, not possible to know a priori how much space will be consumed by the data. The standard C library provides a set of functions to allocate memory dynamically.

Since the size of data-types may vary on different hardware platforms the memory allocation needs to be done relative to their sizes. The **sizeof(type)**-operator returns the size of a data-type in bytes. It can be applied to basic data types as well as structures.

Example 3.23: Print the size of the `double` and the `struct point3d` type:

```
printf("sizeof␣(double)␣=␣%lu\n",sizeof(double));
printf("sizeof␣(struct␣point3d)␣=␣%lu\n",
sizeof(struct point3d));
```

The `malloc` function allocates contiguous memory blocks of arbitrary size[6]:

```
void *malloc(size_t size);
```

This requests a memory location of `size` bytes and returns the start address. If the allocation fails it returns `NULL`. `malloc` does not care about the data-type. The returned `void*` pointer needs to be transformed to the desired data-type using a type cast.

```
double *x;
x = (double *) malloc(sizeof(double));
```

If a memory location is no longer used it should be made available again. The `free`-function deallocates the memory referred to by a pointer:

```
void free (void *ptr);
```

Example 3.24: Allocate an array with 100 `double` entries, sum them up, and free the array:

```
double *array;  // declare the pointers
// Allocate 100*sizeof(double) bytes memory
array = (double *) malloc(sizeof(double)*100);
// sum them up
double sum = 0;
for ( i = 0; i < 100; i++) {
  sum += array[i]; }
free(array); // free the memory
```

If an allocated memory location is too small or too large it can be resized using the `realloc`-function:

```
void *realloc(void *oldptr, size_t newsize);
```

It takes the old pointer and the new size of the array and returns the pointer to the resized array. The data in the part that is kept remains unchanged. If the old pointer is the special `NULL` value, `realloc` behaves exactly like `malloc`. Statically allocated arrays, such where the size is known before the program is compiled, can not be resized.

[6] Only restricted by the availability of memory.

A few other memory allocation operations exists. For example **calloc** and **mmap** are two of these.

> **Remark 3.25:** **valgrind** is an excellent tool to detect errors with wrong access to pointers or wrong usage of the memory management function.

3.5 Functions

Nearly all programming languages have a construct to separate a package of code blocks. This is necessary to get a well-arranged reusable code avoiding copy and paste orgies. The **main**-function is the starting function of every program. It is called automatically when a program is executed. Statements like **printf** and **scanf** are functions, too. Some important standard functions are introduced in Section 3.6.

Functions are called using their name followed by a list of arguments in parentheses. If the return-value is needed it is used like a variable in an expression or a function in a mathematical context.

Example 3.26: Check if **scanf** has read two integers correctly:

```
int i1, i2, r;
r = scanf("%d␣%d", &i1, &i2);
if ( r != 2 ) {
  printf("scanf␣did␣not␣read␣2␣integers␣successfully.\n");
}
```

A function consists of two parts. The header defines the input/output arguments and the return type. The second part is the body where the function is implemented. This gives the following layout:

```
return-type function-name(argument-list) {
  // Local declarations
  Statements;
  Statements;
  return return-value;
}
```

The **return-type** can be any simple data-type, including structures and pointers. If the function does not have a return value the **return-type void** is used. Obviously, the **return-value** must be of the type **return-type**. The naming conventions for variables also apply to functions. The argument list is a comma-separated list of the format **data-type variable** which defines the arguments for the function. The function header without the body is called *signature of a function*. The compiler checks if the calling sequence is compatible with its signature, i.e., the number of arguments is correct and the data-types can be type cast correctly.

Example 3.27: Define a function named "**sqr**" operating on a double precision number and returning the square of the argument:

```
double sqr(double x) {
  double a;
  a = x * x;
  return a;
}
```

The signature of this function is **double sqr(double x);**

Normally the arguments are copied to the function when it is called. The function works on a copy of the data not modifying the original. This behavior is called *Call by Value*. If a function has to change a given argument at its original location the arguments needs to be a pointer to the variable. We call this behaviour *Call by Reference* because only a reference to a variable is passed. A function can return more than one value or complex data types using this technique. The **scanf**-function again serves as an example for this. Another popular example is the **swap**-function:

Example 3.28: We define a function which takes two integer values as arguments and swaps their values. The straight forward solution would be:

```
void swap (int a, int b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
// in main()
int x = 4;
int y = 5;
swap(x, y);
```

This looks correct but the **swap**-function only exchanges a copy of **x** and **y**. The correct solution would be:

```
void (int *a, int *b) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

```
// in main()
int x = 4;
int y = 5;
swap(&x, &y);
```

In this case **a** and **b** are used as a reference to **x** and **y**. Exchanging the values in the memory locations where **a** and **b** point to will change the values of **x** and **y** immediately.

Example 3.29: The **main**-function of a C program is a special case of a function that takes two arguments, the first **int argc** argument contains the number of command line arguments passed to the program including the program name itself. The second argument **char **argv** is an array of strings. Each string contains one command line argument. The element **argv[0]** contains the name of the program.

Remark 3.30: Arrays are always passed to a function *Call by Reference* because they are equivalent to pointers. There is no way to pass an array using *Call by Value* except of creating a copy of the array before manipulating it inside the function. By default modifications are directly performed in the original array.

3.6 An Introduction to the Standard Library

The ISO C Standard [5, 7, 9] defines a standard library to provide basic functions on every platform and allow portable programming. It consists of about 20 different header files and around 200 function for input/output, basic math, string manipulation and memory management. This sections gives an overview about some important predefined functions. The functions are presented using their signature and a short description.

The POSIX C Library [6] is an important extension to the standard C library which provides more operating system dependent operations on Unix-like operating systems. It contains functions for networking, inter process communication, threading and many more. Due to space limitations it can however not be included in this presentation. Starting with the C11 standard, threading has also become part of the standard C library.

d	integers of the type **int**
ld	integers of the type **long**
u	integers of the type **unsigned int**
g	floating point numbers of the type **float** or **double**
e	floating point number in **[-]d.ddde+dd** notation
c	a single character of type **char**
s	strings (see Section 3.4)
%	the % sign.

Table 3.2: Format specifiers

3.6.1 stdio.h and stdlib.h

These two headers files provide the basic functionality of the C library. They provide input/output operations, control statements and memory management. The file-io operations are demonstrated in Section 3.7 again.

The input/output functions introduced later in this section contain *format strings* determining what is to be read or printed. These format strings contain *format specifiers* for the representation of the variable's contents. Some important specifiers are given in Table 3.2.

The full format specification has the form

```
% [flags][width][.precision][l]type
```

The **[l]type** part is what is shown in Table 3.2. The bracketed specifiers are optional. They can be used to further influence the output representation. The **width** parameter for example determines the length in the corresponding output string. For floating point numbers **precision** determines the number of digits in **width** that is used for the decimals.

Example 3.31:

```
double pi = 3.14159265;
printf("pi␣=␣%8.6g\n",pi);
```

prints:

pi = 3.141593
Note that the decimal dot is consuming one of the 8 digits.

The other placeholders and modifiers are described in the man page of the **printf** function, see:

```
man 3 printf
```

or [15, 14] in detail.

The following is a list of the most important functions contained in **stdio.h** and **stdlib.h**.

```
int printf(const char *formatstring, arguments, ...);
int fprintf(FILE *f, const char *formatstring, arguments,
    ...);
int sprintf(char *buf, const char *formatstring, arguments,
     ...);
```

The function **printf** writes a text to the standard output. The equivalent function for writing into files is **fprintf**, and **sprintf** stores the result in the output string **buf** in the memory. The format string is explained above and mentioned in Section 3.2. The return-value in all cases is the number of characters written.

```
int scanf(const char *formatstring, arguments, ...);
int fscanf(FILE *f, const char *formatstring, arguments,
    ...);
int sscanf(const char *string, const char *formatstring,
    arguments, ...);
```

The **scanf**-function reads a formatted input from the standard input. This is the keyboard in most cases. The arguments are pointers to the variables where the values read from the input are stored. The **fscanf**-function is the equivalent to read data from a file and **sscanf** reads from another string. The functions return the number of values read. **fscanf** stops reading when either the end of a line, or the end of the file is reached. **sscanf** terminates upon reaching the 0-byte.

```
FILE *fopen(char *filename, char *mode);
```

The **fopen**-function opens the file specified by the **filename** and returns a pointer to the file stream. The **mode** argument is a string determining the access to the file: **fopen** returns NULL in case of an error.

```
int fclose(FILE *stream);
```

The **fclose**-function closes a given file stream. Any buffered data is written to the file. The **stream** is no longer associated with the file.

```
int feof(FILE *stream);
```

The **feof**-function returns true if the given file stream reached the end of the file otherwise false is returned.

```
void perror(const char *s);
```

Mode	Meaning	Remarks
r	open for reading	Only possible if the file exists otherwise NULL is returned.
w	create a file for writing	If the file already exists the content is destroyed.
a	append data to a file	If the file already exists, the new data is appended to the end. If it does not exist the behavior is like w.
r+ / w+ / a+	open/create file for read and write access.	basic behavior is as above
t	text mode	Only valid in combination with the above. Produces human readable output files. This is the default if neither t nor b is given.
b	binary mode	Only valid in combination with the above. Produces machine readable output. Usually gives smaller output files.

The **perror**-function displays the most recent error from the C library. The string **s** may contain an explanatory message that is printed before the actual error message.

```
void *malloc(size_t size);
void *realloc(void *ptr, size_t new_size);
void free(void *ptr);
```

The memory management functions explained in Section 3.4.

```
void abort();
void exit(int exit_code);
```

The **abort**-function terminates a program immediately without any clean up. The **exit**-function terminates a program immediately with clean up. It is the same as **return** in the main function but can be called anywhere in the code.

```
int atoi(char *s):
double atof(char *s);
```

The **atoi**-function converts a string to an integer if possible. The **atof**-function does the same with a floating point number.

3.6.2 math.h and complex.h

These two header files provide common mathematical functions and constants. If a program uses at least one of them it needs to be linked against the math part of the standard C library. This is done using the "-lm" linker flag when the compiler/linker is invoked (see also Section 3.1). All of the following functions take **`double`** arguments and produce **`double`** return values.

`fabs(x)`	returns absolute value of **`x`**
`exp(x)`	returns e^x
`exp2(x)`	returns 2^x
`log(x)`	returns $\ln x$
`log10(x)`	returns $\log_{10} x$
`log2(x)`	returns $\log_2 x$
`sqrt(x)`	returns $\sqrt{x}$
`hypot(x,y)`	returns $\sqrt{x^2+y^2}$
`pow(x,y)`	returns x^y
`sin(x)`	returns $\sin x$
`cos(x)`	returns $\cos x$
`tan(x)`	returns $\tan x$
`asin(x)`	returns $\sin^{-1} x$
`acos(x)`	returns $\cos^{-1} x$
`atan(x)`	returns $\tan^{-1} x$

The C99 standard [7] introduces the new data types **`float complex`** and **`double complex`** for handling complex numbers. These data types are defined in the header file **`complex.h`**, along with the imaginary unit as **`I`** and the following functions for double precision complex arguments and return values:

`creal(x)`	real part of **`x`**
`cimag(x)`	imaginary part of **`x`**
`carg(x)`	computes the phase angle of a complex number
`cabs(x)`	computes the magnitude of a complex number
`conj(x)`	returns $\bar{x}$
`cexp(x)`	returns e^x
`clog(x)`	returns $\ln x$
`csqrt(x)`	returns $\sqrt{x}$
`cpow(x,y)`	returns x^y
`csin(x)`	returns $\sin x$
`ccos(x)`	returns $\cos x$
`ctan(x)`	returns $\tan x$
`casin(x)`	returns $\sin^{-1} x$
`cacos(x)`	returns $\cos^{-1} x$
`catan(x)`	returns $\tan^{-1} x$

The list of mathematical functions presented here is not complete. More can be found in the man pages or the C standard [7]. For nearly all double precision functions there exists a corresponding single precision function with an **f** as suffix. For example the single precision square root is computed by **sqrtf(x)**.

Some predefined constants are:

M_PI	$\pi = 3.14159265358979323846$
M_PI_2	$\frac{\pi}{2} = 1.57079632679489661923$
M_E	$e = 2.7182818284590452354$
M_SQRT2	$\sqrt{2} = 1.41421356237309504880$

3.6.3 string.h

The **string.h**-header file contains various functions to manipulate and work with strings. The important ones are:

```
size_t strlen(char *s);
```

The **strlen**-function returns the length of the string not including the terminating 0 character.

```
char *strcpy(char *dest, char *src);
```

The **strcpy**-function copies a string from **src** to **dest** and returns the **dest** pointer again. **dest** needs to be a preallocated string with a length of at least **strlen(src)+1** elements. The destination string is not 0-terminated if the source string does not contain the 0-byte within the length of the destination string. The behavior in case the destination is to short is unspecified and may depend on the actual implementation of the compiler.

```
char *strcat(char *dest, char *src);
```

The **strcat**-function appends the string from **src** to **dest** and returns the **dest** pointer again. **dest** needs to be a preallocated string that has space for at least **strlen(src)+strlen(dest)+1** elements.

```
int *strcmp(char *lhs, char *rhs);
```

The **strcmp**-function compares two strings lexicographically. It returns a negative value if **lhs**<**rhs**, a positive value if **lhs**>**rhs** and 0 if they are equal.

Additional Memory Manipulation Functions in `string.h` In addition to the string operations, **`string.h`** defines a variety of memory related actions like:

```
void *memcpy(void *dest, void *src, size_t n);
```

The **`memcpy`**-function copies **`n`** bytes from **`src`** to **`dest`** and returns the **`dest`** pointer again. **`dest`** needs to be a preallocated with **`n`** bytes. **`src`** and **`dest`** must not overlap each other. **`memmove`** does the same but allows overlapping. It is slower than **`memcpy`**.

```
void *memset(void *dest, int ch, size_t count);
```

The **`memset`**-function converts the value **`ch`** to an **`unsigned char`** and copies it into each of the first **`count`** characters of the location referred to by **`dest`**.

3.7 File Input and Output

The basic functions for file-io have already been mentioned in Section 3.6. In this section we present some examples for their usage. They mostly behave like their corresponding standard-io counterparts.

`fopen` opens a specified file in the desired mode. To avoid undefined behavior we have to check if **`NULL`** was returned.

Example 3.32: We create file "test.txt" for writing:

```
FILE *fp;
fp = fopen("test.txt","w");
if ( fp == NULL ) {
  perror("can not open test.txt for writing.");
  return -1;
}
```

If we want to read data from a file we have to use **`r`** instead.

The access modes `"w"` and `"a"` open files for writing. **`fprintf`** is used like **`printf`** on this file:

```
int x = 10;
double y = 145.1;
fprintf(fp, "x = %d , y = %lg\n", x, y);
```

The access mode `"r"` allows **`fscanf`** to read data from it. It works like **`scanf`** but reads a line from a file and tries to assign the values like specified in the format string. If the **`feof()`**-function evaluates to true, no more data can read from the file.

Example 3.33: We consider a human-readable file with the following layout:

```
x1 y1
x2 y2
...
```

The code-snippet to read all values and print them to the screen will be:

```
FILE *fp;
double x, y;
fp = fopen("test.txt","r");
if ( fp == NULL ) {
  perror("can␣not␣open␣test.txt␣for␣reading.");
  return -1;
}
while (!feof(fp)){
  fscanf("%lg␣%lg", &x, &y);
  printf("x=%g␣\t␣y=%g\n",x,y);
}
```

After reading or writing to a file it needs to be closed by **fclose(fp)**.

The **fprintf** and **fscanf** functions are only useful for human readable files. For individual access to binaries we refer to **fread**, **fwrite** and other functions from **stdio.h**.

Error Handling The file-io functions mostly return only that an error has happened but they do not describe this error in detail. In order to achieve this additional work is necessary. Example 3.32 already shows the usage of **perror**, which shows the last error of the standard C library on the standard error output. This error message is prefixed with the string passed to **perror**. The error number is also available in a variable **int errno**, which gets visible to the user when including the **errno.h** header file. This error number can be used inside the code to realize a proper error handling. The whole set of possible error numbers is given in the manpage **man 3 errno**. Using the **strerror** function one can retrieve a human readable representation from a given error number as a string. Since the **errno** variable is accessed by almost all function from the standard library, its value need to be saved immediately after an error is detected and before calling the next function from the standard library. Regarding this rule. Example 3.32 can be rewritten to:

```
FILE *fp;
int err;
fp = fopen("test.txt","w");
if ( fp == NULL ) {
  err = errno;     // save the last error
```

```
  fprintf(stderr,"fopen␣-␣errno␣=␣%d,␣errmsg␣=␣%s\n",
  err, strerror(err));
  return -1;
}
```

3.8 The Preprocessor and Header Files

Before a C compiler translates the source code into the machine code the input is processed by the preprocessor. It performs search-replace operations and includes other files into the current source code. All preprocessor statements begin with a **#** and end with a newline. The most frequently used one is **#include**. It includes other files into the current source code. Other common statements are **#define** and **#ifdef**.

#include As we have seen above already, **#include** is used to include other files into the current source code. These are mostly header files of libraries which contain function-headers, data-structures or constants. A C header file has the extension **.h**. The entire content of the included file is temporarily copied to the position of the **include**-statement in the source file. Two different variants of **#include** are possible:

```
#include <header.h>
```

searches the system include path[7] first and then it uses the additional ones given by the **-I** option on the command line. This is used to include standard headers and other external libraries. The second one is

```
#include "header.h"
```

which searches in the current directory first. This one is used for local, in-project, include files. It is also possible to include other **.c**-files. This can, however, cause conflicts.

#define is used in three ways. The first one is to set up symbolic replacements in the source. This is used to define constants for example.

Example 3.34: The preprocessor statements:

```
#define PI 3.14519
#define SQRT2 sqrt(2)
```

will replace any occurrence of **PI** with **3.14159** and of **SQRT2** with **sqrt(2)** in the current source file.

[7] usually **/usr/include** and **/usr/local/include**

The second way is to define parameter-depended replacements, so called *preprocessor macros*. They depend on at least one parameter and perform all replacements with respect to the given parameters. The parameters in the macro are filled up with the expressions from where the macro is used. The parameter list is appended directly to the macro-name without any white-space. The parameters should be enclosed in parentheses when they are used. The whole macro should be enclosed with parentheses again to avoid errors after the replacement.

Example 3.35: The following macro will give the absolute value of the parameter:

```
#define ABS(X)  (((X)>0)?(X):(-(X)))
```

This replaces `y = ABS(z+1);` with:

```
y = (((z+1)>0)?(z+1):(-(z+1)));
```

If `X` is not enclosed with parentheses this is evaluated to:

```
y = ((z+1>0)?z+1:-z+1));
```

This is not the desired behavior because the minus in the second part is only applied to `z` and not to the whole expression as it was intended.

The third way to use the `define`-directive is as boolean variables for the `#ifdef`-statement. It evaluates to true when the define exists. The preprocessor variables can be set using the `-D` command line option of the compiler.

Remark 3.36: The preprocessor acts stupid on replacements of all appearances of `define` statements. It does not check whether or not the result actually is valid C code. The programmer has to make sure that the `define` statements are extended to the correct intended C code.

`#ifdef` The `ifdef`-directive, a short form of `#if defined`, allows conditional compiling of the source code. It works like the `if-else` construct in a normal program but is evaluated by the preprocessor at compile time:

```
#ifdef PREPROCESSOR_DEFINE
// Code compilied if PREPROCESSOR_DEFINE exitsts
#else
// Code compiled otherwise
#endif
```

The `#else`-part can be left out. The code in the unused case is temporarily removed from the source code during the preprocessing. This technique is used to handle different environment situations in one source file.

Example 3.37: In order to debug a program easily somebody defined a **INFO**-macro which prints the given parameter to the screen. In the final version of the program this is not necessary. However removing all outputs in the code may be unwanted to be able to insert them again for debugging purposes:

```
#ifdef DEBUG
#define INFO(X) printf(X)
#else
#define INFO(X)
#endif
```

If **DEBUG** is defined the **INFO**-macro is expanded to a **printf**-statement otherwise it is replaced with nothing.

The **#ifndef** statements is the opposite of **#ifdef**. It simply negates the condition of the **#ifdef** statement.

Header-Files. If a C program is split into several source files, the header file tells the compiler which functions, data-structures and constants exist in other source files. This is necessary because the compiler can only check the function headers and the calling sequence in the current file. Header files can also be used to share data structures and variables. It is similar to a normal source file but consists only of definitions without any implementation. A cyclic inclusion should be avoided using the preprocessor commands **#define** and **#ifndef**. The following example shows how a function can be moved to an external file and how the header looks like:

Example 3.38: **exfct.c** implements the function **something**:

```
#include <math.h>  // for sqrt
#include "exfct.h" // Ensure that the function header
// fits to the one from exfct.h
double something(double x, double y, double z){
  return sqrt(x*x+y*y+z*z);
}
```

The header file **exfct.h** only contains the function header (its signature) and a preprocessor trick ensuring that it can not be included twice in one file:

```
#ifndef EXFCT_H
#define EXFCT_H
double something(double x, double y, double z);
#endif
```

The main program can now include the header and knows how the function **something** is called correctly.

Splitting a large program into different source files makes the whole project well arranged and easily maintainable. The different files should have a meaningful name.

A software project consisting of many source files can be compiled adding all **.c**-file to the compiler call. This works but is not the best way when searching for compilation errors. A better and faster way is to define a makefile which automates the build. The next Section 3.9 shows how this basically works.

3.9 Makefiles

Make is a utility that automates the build process for executable programs and libraries from source code. It is, by default, controlled by a text file called **Makefile**, which contains the build instructions. It can deal with dependencies between different source code files and compiles only files that have been modified since the last build. There exists different versions of **make** such as GNU Make, BSD Make and Microsoft's **nmake**.

A makefile is not a script as we have seen it in the case of **bash**, but rather works as a dependency tree. It compiles the files that are outdated in the order they depend on each other. The makefile consists of so called targets, which may depend on each other. A target is defined by a rule:

```
targetname: dependencies
	command1
	command2
	...
```

The indentation before the commands must be <tab> characters; not spaces! The **targetname** should be equal to or closely related to the output file generated by the commands. **dependencies** is a space separated list of other targets that need to be compiled prior to the target or names of files which need to exist. A target is only built if it is older than at least one of its dependencies. There can be more than one target in a single makefile.

Example 3.39: Consider a small software project consisting of **main.c**, **file1.c** and **file1.h**. A makefile to create the final program **prog** looks like:

```
prog: main.c file1.c file1.h
```

```
    gcc -c main.c
    gcc -c file1.c
    gcc -o prog main.o file1.o
```

In the case that the makefile is named **Makefile** or **makefile** the make process may be invoked executing

```
make targetname
```

If the makefile is called differently, use:

```
make -f makefilename targetname
```

If no **targetname** is specified, the first one found in the makefile is used.

In order to be more flexible, we can introduce variables. Mostly they contain the list of source files, object files or compiler and linker options. A variable is set by

```
VARNAME=VALUE
```

and it's contents is accessed with **$(VARNAME)**. To change the extension of all files listed in a variable the substitute command is used. The syntax is

```
NEWVAR = ${OLDVAR:.old=.new}
```

This replaces the extension of every file ending with **.old** in **OLDVAR** to **.new** and stores the list to **NEWVAR**. This is normally used to create a list of object files from the list of source files. Additionally, one can define conditional variables. In this case the value is only set if the variable does not already exist. This is helpful if the user should be able to set options when he invokes **make**. A conditional variable is set by

```
VAR?=FOO
```

If **make** is called without any argument then **VAR** will contain "FOO", if make is called like

```
make VAR=BAR
```

the variable **VAR** contains "BAR".

Because it takes too long to define a rule for every input file, suffix rules are used. They create a target for every file matching the rule. They apply to files that match the suffix and have not been processed by a separate target before.

```
.SUFFIXES: .in .out
.in.out:
    command1
    command2
```

```
    ...
```

These rules create a target for every file ending on **.in** to transform it into the same filename with the extension **.out**. This is used to compile source code from **file.c** to an object file **file.o**. Two placeholders exist referring to the input and the output filenames. The input file is referred to using **$<** and the output file using **$@**.

Finally we define a clean up target. The target **clean** removes all object files or intermediate outputs. Because this target does not produce an output file or does not depend on a file called **clean** it needs to be declared as **.PHONY** target.

Example 3.40: We consider again Example 3.39. Inserting variables, suffix rules and the extension replacement we can turn it into a more generic one:

```
SRC=main.c file1.c
OUTPUT=prog
CC=gcc
CFLAGS= -O2
OBJECTS=${SRC:.c=.o}

$(OUTPUT): $(OBJECTS)
    $(CC) -o $(OUTPUT) $(CFLAGS) $(OBJECTS)

.SUFFIXES: .c .o
.c.o:
    $(CC) -c -o $@ $(CFLAGS) $<

clean:
    rm -f $(OBJECTS)

.PHONY: clean
```

From time to time **make** does seem to do strange things and not work as intended. In most cases this is due to the fact that the actual command issued by **make** are not what we intended in the first place. Then the **-n** or **--just-print**, **--dry-run**, **--recon** flags (all aliases of each other) can help identify what is going on. In contrast to normal operation, this will make **make** print all commands that would be issued. So, we can crosscheck where the discrepancy between actual and intended operation is and update the rules in the corresponding targets.

There exist many other techniques to extend the make file such as automatic dependency creation using the GCC compiler, pattern rules as a generalization of the suffix rules, include statements, if directives and many more. See [12] for details. Other tools like CMake[8] or the GNU Autotools[9] provide high level scripting languages to create complex makefiles automatically.

3.10 Writing Own Libraries

Libraries are collections of precompiled functions and predefined data structures and constants together with the header files, containing the function headers and the data structures. In contrast to a normal C program a library does not provide a **main** function. The standard C library is an example for a library which was already used in the previous sections.

Two different types of libraries exists. The first ones are the static libraries and the other ones are the dynamic or shared ones. Both of them have advantages and disadvantages. The static ones are easy to create but need more space on the mass storage and cause problems with cyclic dependencies between libraries. On the other hand, the dynamic libraries are a bit more complicated to create but take less space on the mass storage and can be exchanged without recompiling the program. Many programs can refer to a single shared library and use it independent of the specific version or implementation.

Static Libraries Static libraries are collections of object files combined in a specially structured archive. This archive is a classical UNIX **ar**-file containing all **.o**-files of the library and a search index. The source code only needs to be compiled to object code using the -c compiler option. Afterwards, all object files are combined to a **.a**-file:

```
ar crs libNAME.a *.o
```

The **c** options creates an archive, the **r** option replaces existing files inside the archive, if it already exists and the **s** options adds an object index. This index speeds up the linking procedure. For completeness we mention that running **ar** with the **s** option is completely equivalent to using the command **ranlib** for the index generation.

A static library is linked to a program by adding the **.a**-file to the compiler call:

```
gcc -o program main.c libname.a
```

All functions referenced in **main.c** are copied from **libname.a** to the final program. If more than one static library is used the compiler resolves the symbols from left to right. That means if two or more libraries depend on each other they have to be added in their order of dependence. If there is a cyclic dependency the files need to be added multiple times.

[8] https://www.cmake.org
[9] https://en.wikipedia.org/wiki/GNU_build_system

Remark 3.41: If a static library is used in conjunction with a dynamic one or on a 64-bit architecture like x86_64 all source files must be compiled with the **-fPIC** flag.

Example 3.42: We consider the minimal external function from Example 3.38. The following steps create a static library and link it against a program.

```
gcc -c -fPIC exfct.c
ar crs libexfct.a *.o
gcc -o prgm main.c libexfct.a
```

Dynamic/Shared Libraries Dynamic or shared libraries are nearly the same as normal programs. The only difference is the missing **main** function. When they are linked to a program a cross reference is placed in the program indicating in which dynamic library the functions actually resides. The dynamic loader reads this cross references on execution and loads the necessary libraries into the same address space as the program. If the program now calls an external function it executes the code loaded from the libraries.

The dynamic linker searches for the dynamic libraries only in standard system paths. Typically, these are **/lib**, **/usr/lib/** and **/usr/local/lib/**. If a library does not exist in these standard paths, the **LD_LIBRARY_PATH** environment variable can be used to set additional search paths. An alternative way is to add additional search paths to the program during the linking phase. The addition of **-Wl,-rpath=PATH** to the compiler call allows this.

Dynamic libraries can be replaced without relinking program as long as they use a compatible binary interface. If at least one function head, i.e. the functions signature, changes or a data structure in a header file changes, the program needs to be recompiled and relinked.

Dynamic libraries are created using the compiler and the linker. The source code needs to be compiled with the **-fPIC** compiler flag. Additionally, the **-shared** option advises the compiler and the linker to create a shared library instead of a normal executable. The output file name for a shared library must follow the **libNAME.so** naming convention.

Example 3.43: We consider the minimal external function from Example 3.38 again. The following steps create a dynamical library and link it against a program.

```
gcc -shared -fPIC -o libexfct.so exfct.c
```

```
gcc -o prgm -L. -lexfct main.c
```

If the additional search path should be integrated in the binary add **-Wl,-rpath=.** to the second compiler call. The **libexfct.so** can be modified without relinking it to the output program as long as the function signature does not change. A very nice summary of the topic can be found in [13].

3.11 Interfacing Fortran

Many mathematical libraries, especially numerical linear algebra ones, have been written in Fortran. Fortran is the oldest high-level programming language which is still in use. It is currently specified in ISO/IEC 1539-1:2010 [8]. The newer versions of Fortran provide an interface to C[10], but this is not supported by all compilers and many Fortran codes rely on old standards. Due to this, the old fashioned way of interfacing Fortran is presented by an example in this section.

Fortran code can be compiled using the **gfortran** command. This invokes the Fortran compiler of the GNU Compiler Collection. It takes nearly the same command line arguments as the C compiler. Fortran files typically use **.f**, **.f90** or **.f95** as extensions.

The DAXPY[11] operation taken from the Basic Linear Algebra Subroutine library (BLAS)[12], which we introduce in Section 6.4.1, is used as an example to explain how a Fortran subroutine is called from C. The DAXPY operation computes

$$y = y + \alpha x$$

for two vectors $x, y \in \mathbb{R}^n$ and a scalar $\alpha \in \mathbb{R}$. The Fortran function header is

```fortran
SUBROUTINE DAXPY(N,DA,DX,INCX,DY,INCY)
      DOUBLE PRECISION DA
      INTEGER INCX,INCY,N
      DOUBLE PRECISION DX(*),DY(*)
```

First of all, we have to translate the Fortran data-types to the corresponding C types. Because Fortran passes values to a function using *Call by Reference*, all arguments will be pointers no matter if they are scalar values or vectors. The data-types of the arguments translate to:

[10] https://de.wikibooks.org/wiki/Fortran:_Fortran_und_C
[11] https://www.netlib.org/blas/daxpy.f
[12] https://www.netlib.org/blas

Fortran type	C type
INTEGER	**int**
REAL	**float**
REAL*8	**double**
DOUBLE PRECISION	**double**
COMPLEX	**float complex**
COMPLEX*16	**double complex**
DOUBLE COMPLEX	**double complex**
CHARACTER or **CHARACTER(*)**	**char ***

The second step is to translate the function name. Different compilers use different conventions for this. As long as only the GNU Compiler Collection is used the rules are:

- The function name is translated to lower case.
- A trailing underscore "_" is added to the function name.
- If the function name contains an underscore, a second underscore is added.

A Fortran subroutine is like a C function with a **void** return-type. If it is a function instead of a subroutine the return-type needs to be translated according to the list above, as well. The return-type is then not a pointer.

Applying these rules to the **DAXPY** subroutine gives:

```
void daxpy_(int *N, double *DA, double *DX,
int *INCX, double *DY, int *INCY);
```

This function header is necessary in every C source code which uses the Fortran routine. It can also be moved to a header file.

The following code computes

$$y = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \qquad y = y + 2 \cdot \begin{pmatrix} 4 \\ 3 \end{pmatrix}$$

using the DAXPY subroutine:

```
#include <stdio.h>
#include <stdlib.h>
void daxpy_(int *N, double *DA, double *DX,
int *INCX, double *DY, int *INCY);
int main(int argc, char *argv) {
  double x[2] = {4 ,3};
  double y[2] = {1 ,2};
  double alpha = 2.0;
  int n = 2, incx = 1, incy = 1;
  daxpy_(&n, &alpha, x, &incx, y, &incy);
  printf("y = [ %g, %g ]\n", y[0], y[1]);
  return 0;
```

```
}
```

The program is compiled calling:

```
gfortran -c daxpy.f
gcc -c main.c
gcc -o prgm main.o daxpy.o -lm -lgfortran
```

The math (**-lm**) and the Fortran runtime library (**-lgfortran**) need to be added to the program.

If the list of arguments contains a **CHARACTER(*)** argument, we have to adjust the rules a bit since Fortran does not use the **NUL**-termination of strings. For this reason, the length of the string is passed as a hidden argument at the end of the argument list [13]. For each string argument of the subroutine one hidden argument is added. This argument is an integer and passed by-value instead of the by-reference. The length of the string is counted without the terminating **NUL** character. The argument is not directly visible in the Fortran code.

Consider the Fortran header:

```
SUBROUTINE STRFUNC(MYSTR)
   CHARACTER(*) MYSTR
```

We obtain the following prototype for the C interface:

```
#if __GNUC__ > 7
typedef size_t fortran_charlen_t;
#else
typedef int fortran_charlen_t;
#endif

void strfunc_(char *mystr, fortran_charlen_t length);
```

Remark 3.44: Although the calling convention between C and Fortran has been defining hidden argument for **CHARACTER** arguments since at least 30 years, they are rarely used. Especially, if only characters of length one are passed or the length is clear to the subroutine, these arguments are very often neglected. In most codes, especially where BLAS and LAPACK are interfaced, these hidden arguments are missing. With some version of the GCC compiler and some compiler flags this can cause trouble[a][b].

[a] https://lwn.net/Articles/791393/
[b] https://developer.r-project.org/Blog/public/2019/05/15/gfortran-issues-with-lapack/

[13] https://gcc.gnu.org/onlinedocs/gfortran/Argument-passing-conventions.html

3.12 Automatic Generation of Code Documentations Using DOXYGEN

Documenting code and writing a manual for a software project can be even more time consuming than the real programming job. **doxygen** is a documentation generator tool which allows the programmer to write the documentation directly inside the source code. It extracts the documentation from specially structured comments and outputs it to HTML files, a LaTeX document, an RTF document or man pages. A large variety of programming languages such as C, C++, Java, Fortran or Python are supported.

Modified multi line comments are mostly used for **doxygen** in a C source. Instead of **/*** they have to start with **/****. Depending on the programming language other comments must be used. These comments are interpreted by **doxygen**. When a **doxygen**-comment stands directly in front of a function, a structure definition or a similar construct, it refers to this object. The documentation is improved with special statements inside the comment. The basic ones are:

@brief	Set the brief documentation of the object.
@param	Document a parameter of a function.
@return	Document the return value of a function.
@author	Set the author of a function.
@version	Set the version of an object.
@see	Create a cross reference to an other function, struct,...

Alternatively, the commands can start with a **** instead of the **@** character. All lines not beginning with a **doxygen**-command are extracted as normal documentation text. Normal C comments are not recognized by **doxygen**.

Additionally, HTML tags or LaTeX-style formulas can be used in the documentation. A LaTeX formula is enclosed by **\f$** or **\f[** and **\f]** in order to create an in-line or a separated formula. If the outputs are HTML files the LaTeX-formulas are rendered and included as images. On the other hand, if the output is a LaTeX document the basic HTML tags are converted to the corresponding LaTeX-commands.

Example 3.45: We want to document the **sqr** function from Example 3.27. This is done adding a **doxygen** comment block right before the function header begins:

```
/**
\brief Squares a given double value.
\param x Input value.
```

```
\return the square of the input value x.

The sqr function returns the square \f$ x^2 \f$ of a
given number x. <i>The intermediate result is stored
in an internal variable.</i>
*/
double sqr(double x) {
  /* This is not for doxygen. */
  double a;
  a = x * x;
  return a;
}
```

Beside the special comments inside the source code **doxygen** is controlled by a so called **Doxyfile**. This specifies the source directory, the output format and other in- and output related options. A template of this file is generated using:

```
doxygen -g config_filename
```

The newly generated file is well documented and easily customizable using a text editor. The documentation of a software project is created by simply calling

```
doxygen config_filename
```

If **doxygen** is invoked without any configuration file it searches for a file named **Doxyfile** in the current directory.

More information about **doxygen** and its use within a software project are available in [4]. A good starting point for beginning readers can be found in [3].

Bibliography

[1] *Wikibook: C*, https://de.wikibooks.org/wiki/C-Programmierung.

[2] *Wikibook: Fortran*, https://de.wikibooks.org/wiki/Fortran.

[3] *Doxygen: Getting started*, https://www.doxygen.org/manual/starting.html.

[4] *Doxygen: Website*, https://www.doxygen.org/.

[5] ISO, *ISO/IEC 9899:1990: Programming languages — C*, International Organization for Standardization, Geneva, Switzerland, 1990, https://www.iso.org/standard/17782.html.

[6] ISO, *ISO/IEC 9945-1:1996: Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, International Organization for Standardization, Geneva, Switzerland, 1996, https://www.iso.org/standard/24426.html.

[7] ISO, *ISO/IEC 9899:1999: Programming Languages — C*, International Organization for Standardization, Geneva, Switzerland, Dec. 1999, https://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[8] ISO, *ISO/IEC 1539-1:2010 Information technology — Programming languages — Fortran — Part 1: Base language*, International Organization for Standardization, Geneva, Switzerland, June 2010, http://fcode.cn/download/N1830.pdf.

[9] ISO, *ISO/IEC 9899:2011: Programming Languages — C*, International Organization for Standardization, Geneva, Switzerland, Dec. 2011, https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[10] ISO, *ISO/IEC 9899:2018 Information technology — Programming languages — C*, International Organization for Standardization, Geneva, Switzerland, fourth ed., June 2018, https://www.iso.org/standard/68564.html.

[11] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall Software Series, Prentice Hall, 1988, https://books.google.de/books?id=161QAAAAMAAJ.

[12] R. Mecklenburg, *Managing Projects with GNU Make*, O'Reilley Media, 3rd ed., 2004.

[13] A. Rachum, *Shared libraries: Understanding dynamic loading*, Sept. 2016, https://amir.rachum.com/blog/2016/09/17/shared-libraries/.

[14] C. K. Ulrich Kaiser, *C/C++ Das umfassende Lehrbuch*, Gallileo Computing, 2005.

[15] J. Wolf, *C von A bis Z*, Gallileo Computing, 2009, http://openbook.galileocomputing.de/c_von_a_bis_z/.

[16] J. Wolf, *Linux-UNIX-Programmierung*, Gallileo Computing, 3rd ed., 2009, http://openbook.galileocomputing.de/linux_unix_programmierung/. 2nd Edition available as OpenBook.

640K is more memory than anyone will ever need on a computer.

among the top 5 myths about
Bill Gates

CHAPTER 4

Memory Architecture and Memory Management

Contents

Several different layers of memory exist in a modern computer environment. Each of the layers in this hierarchy has a certain relevance in, and special properties for, scientific computing tasks. This chapter is dedicated to a brief introduction of the single layers with their most important properties. The presentation of these properties will help understand the motivation behind the storage structures and tiled operation strategies introduced in Chapter 6.

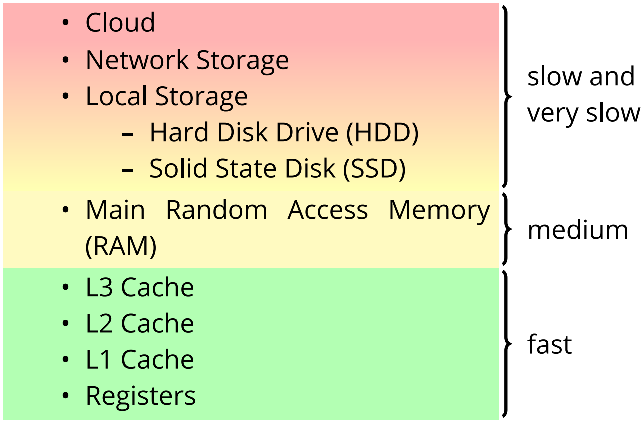


Figure 4.1: Memory Classes in Scientific Computing

Hardware sided the relevant memory comes mainly in four types

- Static Random Access Memory
  (SRAM)
- Dynamic Random Access Memory
  (DRAM)
- Flash Electrically Erasable Programmable Read-Only Memory
  (Flash-EEPROM)
- Magnetic surfaces

Here, the first two types are so called volatile memory devices, which only hold the information as long as they are supplied with electric power. The other two are designed to preserve their content during phases where the power is switched off. Naturally, the secure storage of data (with respect to power-off) comes at a cost. The cost we have to pay is the increased time for, especially, write accesses. The magnetic storage types here are the slowest. This is mainly due to the mechanic subsystems involved in the process. On a hard disk drive the magnetic read/write-head has to be positioned at the right place prior to operation. This equivalently has to be done with the tapes in a tape drive. Both types are, therefore, mainly usable for long term storage of final results. Hard disks are to some extent also useful during computations, when the main memory is running short. Special techniques often called *cache to disk* or *double buffering* are used to store data portions, that will not be used for a longer time in the computation, to the local storage and so free up main memory for intermediate computations.

Nevertheless, in basic operation the static and dynamic random access memory types are the more important ones. Both are electronic memory devices consisting of integrated circuits (ICs) as basic realizations. Their main difference is that the SRAM circuits are transistor based and the DRAMs are capacitor based. It is now easy to imagine that SRAMs can switch essentially instantaneous, whereas DRAMs have to wait for the capacitors to charge completely and require periodic refresh signals to prevent the capacitors from discharging. On the other hand, DRAMs are producible in higher density at lower costs, and have a smaller energy consumption. The main properties are compared in Table 4.1.

Feature	SRAM	DRAM
Storage Circuit Base	Transistor	Capacitor
Speed	Same as CPU	Slower than CPU
Latency	Low	High
Density	Low	High
Power Consumption	High	Low
Cost	High	Low

Table 4.1: Comparison of Volatile Memory Types

Due to the low cost the largest part of a modern computer's memory, namely the main memory, is made out of DRAM chips. The faster and more expensive SRAM chips are only used on the part of the memory that is closest to the actual processing units on the CPU. That means the Cache (see Figure 4.1) is made out of SRAMs, which is one reason why it is usually very limited.

The main concerns in this chapter will be:

- memory organization (pages, page sizes),
- swapping,
- memory related error signals,
- memory transfer and alignment,
- virtual memory concept.

4.1 Virtual Memory Concept

Definition 4.1 (Virtual memory and memory pages)**:** *Virtual memory* is an operating system abstraction layer, that allows to access the various memory layers as one large device. It usually consists of *memory pages*, the smallest accessible units of memory (normally 4 or 64 kBytes).

Virtual memory covers:

- main memory
- cache (via CPU memory management unit (MMU))
- memory mapped files
- SWAP (usually specially structured part of disks)

Data relocation relies on hardware support, mainly implemented in the memory management unit of the CPU.

Definition 4.2 (swapping and double buffering)**:** Relocation of potentially unused data to the local storage by the operating system is called *swapping*. Moving data to the local storage may cause large overhead in waiting time. Any technique that moves that data at strategically better times to avoid swapping is called *double buffering*.

4.1.1 Paging

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory, thus minimizing issues like fragmentation.

- paged virtual memory is the most common implementation,
- page size mostly 4 kBytes,
- generally data can be located anywhere in a page,
- some operations expect the data to be located at the start of a memory page,
  → *page aligned* memory
  → increases memory fragmentation
- *page locked memory* is a special type of memory that is not allowed to get swapped,
- pages can be moved between RAM and permanent storage like harddisks.

Paging on the CPU works in the following way:

1. Logical and Physical Address Space:
    - The logical address space is divided into fixed-size units called pages.
    - The physical address space is divided into blocks of the same size, called frames.
2. Page Table:

   - Each process has a page table that maps logical pages to physical frames.
   - The page table keeps track of where each page is stored in physical memory.

3. Address Translation:
   - When a process needs to access a memory location, the CPU translates the logical address into a physical address using the page table.
   - The logical address is split into a page number and an offset. The page number is used to find the corresponding frame in the page table, and the offset specifies the exact location within the frame.

Advantages of paging:

- **No Physical Fragmentation:** Since pages and frames are of fixed size, there is no fragmentation in the physical memory.
- **Efficient Memory Use:** Allows for efficient use of memory by loading only the necessary pages into memory.

Disadvantages of paging:

- **Page Table Overhead:** Each process requires its own page table, which can consume a significant amount of memory.
- **Page Faults:** If a page is not in memory, a page fault occurs, requiring the page to be loaded from disk, which can be slow.

4.1.2 Memory Related Error Signals

The two important memory related signals are:

- SIGSEGV
  - segmentation violation or segmentation fault signal
  - usually leads to immediate abortions of the process
  - caused by accessing memory segments in foreign address spaces.
- SIGBUS
  - Bus error signal
  - abortion also immediate
  - one common cause: using a processor instruction with an address that does not satisfy its alignment requirements

4.2 Volatile memory

4.2.1 Registers

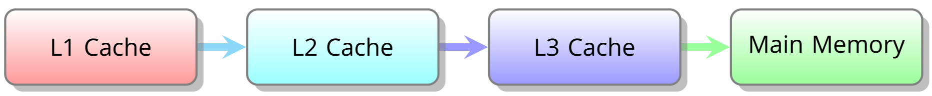
- very small number
- small (<100 Bytes)
- $\underbrace{\text{MMX, SSE, AVX}}_{\text{local vectorization}}$
- we rely on compiler capabilities

4.2.2 Cache

- **L1**: typically 32 or 64 kBytes, split into a data and an instruction part, **installed per core**, direct access to the registers, transfer-rate: 1TB/s.
- **L2**: $\approx 256 - -2048$ kBytes, **installed per core**, keeps frequently used data and instructions of the current core, transfer-rate: 1TB/s
- **L3**: $\approx$ few MBytes per core, same as L2 **for a group of cores** making a processor, connects to RAM, transfer-rate: $>$400 GB/s
- **L4**: only on few CPU architectures, cache of the memory controller, transfer-rate: 400 GB/s

Cache is small, high speed memory made out of SRAM.

Data-Lookup:



Successful lookup is called **Cache Hit**, and the data item is transferred to the registers at maximum speed.

Cache Miss:

- data not available in cache
- needs to be loaded from main memory
- results in a **miss penalty** (Cache Latency)

Hit ratio: percentage of memory accesses satisfied by the cache ($\approx 80 - 90\%$).

Miss ratio: $100\% -$ Hit ratio

Arranged in so called cache lines of $4 - 128$ Bytes.

The cache behaviour can be explored using valgrind's **`cachegrind`** component.

Cache line replacement: e.g.

- LRU — least recently used
- random

Rules of thumb:

cache transfer rate [Bytes/s] $=$ width (no. bits) $\times$ clockrate $\times$ data per clock / 8

The secret of a fast method is program locality, i.e., as many operations as possible on data already residing in the caches.

4.2.3 Main Memory

made of DRAM mainly availabe in 3 types

- asynschronous
  (FPRAM, EDORAM) (outdated)
- synchronous
  (SDRAM, DDRSDRAM, DDR2SDRAM, DDR3SDRAM, DDR4SDRAM, DDR5SDRAM)
- Rambus
  (RDRAM, XDRDRAM, XDR2DRAM)

Standard PCs today mostly use DDR4SDRAM or DDR5SDRAM.

Memory clock	2000–4000 MHz
Data rate	4000–6000 MT/s
Peak transfer rate	32–64 GB/s
CAS Latency	30–40 cycles (at best) $\approx$ 7.5–20ns

The latest DDR5SDRAM chips feature double the manufacturing density, lower operation voltage (1.1V compared to 1.2V for DDR4 and 1.5V DDR3) and higher operation frequencies (DDR4: 1600–3200MHz).

Columns Address Stroke Latency (**CAS Latency**): time for waiting between a request of data and their availability at the memory pins.

Currently available sizes: 256 MB – 2 TB

4.3 Non-Volatile Storage

4.3.1 Local Storage Media

Maximum possible transfer rates are bounded by the capabilities of the bus interface

Type	theoretic peak transfer	release / introduction
ATA 33/66/100	33/66/100 MB/s	
SATA I	150 MB/s $\widehat{=}$ 0.15 GB/s	
SATA II	300 MB/s $\widehat{=}$ 0.30 GB/s	$\approx$ 2005
SATA 3.0	600MB/s $\widehat{=}$ 0.60 GB/s	05.2009
SATA 3.2	up to 1969MB/s $\widehat{=}$ 1.97 GB/s	08.2013
SAS	300 MB/s – 22.5 GB/s	current developments

Solid State Disk vs. Hard Disk Drive Both are connected to the same host/bus interface.

Feature/Property	SSD	HDD
Noise	$+$	$-$
Reliabilty, Lifetime	$-$	$+$
Price	$-$	$+$
Capacity	$-$	$+$
Fragmentation	$+$	$-$
mechanical delay	$+$	$-$
practical transfer rates	100–600 MB/s	$\leqslant$ 140 MB/s
random access time	0.1 ms	2.9–12 ms

Developments connecting the SSD to the PCIe bus (see SATA 3.2 above) get almost 2 GB/s.

Currently available sizes: $\leqslant$8–16 TB (HDD).

RAID (Redundant Array of Independent Disks)

- can increase total storage capacity by grouping disks to larger logical volumes
- can increase the performance and data safety by multiply/redundantly storing the same data.

4.3.2 Local Network

High variance in speeds from 100–10 000 Mb/s on local network to 10–40 Gb/s on high speed Infiniband server networks. 56/80/100 Gb/s have recently entered the market. Higher speeds are in development.

4.3.3 Cloud and Remote Network Services

Usually only useful for storing results for post processing. Involves additional synchronization.

4.4 Non Uniform Memory Access

The non uniform memory access (NUMA) model is part of Flynn's taxonomy of parallel architectures, which will be treated in more detail in term 2. The basic characterization of a NUMA machine is the type of architecture that appears when several independent processing units have the memory associated locally to single units. The entire shared memory of all processing units is the sum of the local memories. Then parts of the memory can only be accessed indirectly with the help of other processing units and additional latencies are unavoidable.

Example 4.3: A system is equipped with 2 processors an 32 GB of main memory, which is separated into two blocks of 16 GB, one for each processor.

The MMUs each organize 16GB locally and need to access the other 16GB via the other MMU.

A less obvious appearance of this phenomenon is on Multicore processors, where each core has its own L1 and L2 Cache, and L3 cache can be available for certain core groups only.

4.4.1 Cache Coherence

Example 4.4: Consider a dual Core system with L1/L2 caches for each processor core. The situation that a memory block is present in both caches and one of the copies invalidates the other copy due to a write access, can appear.

The problem described in Example 4.4 is called **cache coherence problem**. The task of keeping different copies of the data coherent, i.e., consistent with respect to read access, is introducing additional management work that can increase read access times.

A system that is investing this extra work is called ccNUMA (for cache coherent NUMA) machines.

4.4.2 Memory Consistency

Cache Coherence ensures the same view to the global memory through the local cache for each processing unit.

$\Rightarrow$ At each point in time each processor performing a read access gets the latest data.

The corresponding problem for write accesses describes the **memory consistency problem**.

Bibliography

[1] *Auto-vectorization with gcc 4.7*. http://locklessinc.com/articles/vectorize/. accessed November 19, 2012.

[2] *Dynamic random-access memory*. http://en.wikipedia.org/wiki/Dynamic_random-access_memory. accessed November 19, 2012.

[3] *Flash memory*. http://en.wikipedia.org/wiki/Flash_memory. accessed November 19, 2012.

[4] *Paging*. http://en.wikipedia.org/wiki/Paging. accessed November 19, 2012.

[5] *Static random-access memory*. http://en.wikipedia.org/wiki/Static_Random_Access_Memory. accessed November 19, 2012.

[6] G. Torres, *How the memory cache works*. https://hardwaresecrets.com/how-the-cache-memory-works/, September 2007.

number crunching. Computations of a numerical nature, esp. those that make extensive use of floating point numbers. The only thing Fortrash is good for. This term is in widespread informal use outside hackerdom and even in mainstream slang, but has additional hackish connotations: namely, that the computations are mindless and involve massive use of brute force. This is not always evil, esp. if it involves ray tracing or fractals or some other use that makes pretty pictures, esp. if such pictures can be used as screen backgrounds.

The New Hacker's Dictionary

CHAPTER 5

Error Analysis and Machine Numbers

Contents

We have seen in the preface, that the numerical solution of mathematical tasks produces different kinds of errors. In order to be able to judge the correctness of our results and avoid or bound the errors resulting from finite precision representations, we investigate and analyze the machine numbers used for calculation on modern computers.

5.1 Machine Numbers

For calculations on, e.g., a computer, a cell phone, or a pocket calculator, real or complex numbers need to be stored in the finite memory of the device, i.e., with only finitely many digits of accuracy. For simple numbers like 1.0 or 0.5 it is easy to imagine that this is somehow possible, however, for π, which is known to have infinitely many digits, we obviously need to truncate somewhere and thus introduce a certain representation error.

There exist a number of known representations for storing real numbers. Most of them are based on the following theorem.

Theorem 5.1 (p–adic expansion)**:** For $x \in \mathbb{R}$, $p \in \mathbb{N}\backslash\{1\}$ there exist uniquely determined $j \in \{0,1\}$, $\ell \in \mathbb{Z}$ and $\forall k \in \mathbb{Z}$ with $k \leqslant \ell$ unique $\gamma_k \in \{0,\ldots,p-1\}$, such that

$$x = (-1)^j \sum_{k=-\infty}^{\ell} \gamma_k p^k, \tag{5.1}$$

where $\gamma_\ell \neq 0$ for $x \neq 0$, $j = \ell = 0$ for $x = 0$, and $\gamma_k < p-1$ for infinitely many $k \leqslant \ell$.

Proof. See, e.g., [3]. □

In Theorem 5.1 especially the expression "$\gamma_k < p-1$ for infinitely many k" means that, e.g., for $p = 10$ the number $3.\overline{9}$ is represented as 4.0. Moreover, note that all summands in (5.1) are positive, so for $x = 0$ all γ_k need to be zero and the condition $j = \ell = 0$ only makes the representation unique.

The p–adic representation of a number given in a different number system can be expressed using the following representation:

$$(x)_p := \pm\gamma_\ell\gamma_{\ell-1}\ldots\gamma_0.\gamma_{-1}\gamma_{-2}\ldots,$$

where the digits following the separating "." are called the mantissa.

decimal system In our everyday life we are usually using the *decimal system*, i.e., the representation for $p = 10$.

$$x = \pm \sum_{k=-\infty}^{\ell} \gamma_k \cdot 10^k = \pm\gamma_\ell\gamma_{\ell-1}\ldots\gamma_0.\gamma_{-1}\gamma_{-2}\ldots = (x)_{10}$$

with *digits* $\gamma_k \in \{0,\ldots,9\}$ and *base* $p = 10$.

The important number systems for computer arithmetic systems are:

binary system $p = 2$, $\gamma_k \in \{0, 1\}$.

As an example the decimal number $x = 1123$ is translated into the binary system as follows:

$$\begin{aligned} 1123 &= 1024 + 99 = 2^{10} + 64 + 35 \\ &= 2^{10} + 2^6 + 32 + 3 = 2^{10} + 2^6 + 2^5 + 2^1 + 2^0, \end{aligned}$$

i.e., $(1123)_2 = 10001100011$.

For the decimal number $\frac{1}{10}$, on the other hand, we have

$$\left(\frac{1}{10}\right)_2 = 0.0\overline{0011}.$$

To see this we exploit $(10)_2 = 1010$ and perform the division manually in the binary system:

```
1:1010        = 0.000110011...
.
.
.
.
-----
10000
-1010
-----
  1100
 -1010
 -----
    10000
          .
           .
            .
```

So $\frac{1}{10}$ can not be written in a finite number of digits in the mantissa. Note that this does not contradict the conditions of Theorem 5.1, since we still have $\gamma_k = 0$ for infinitely many k.

hexadecimal system $p = 16$, $\gamma_k \in \{0, 1, \ldots, 15\}$.

The usual representation uses $A = 10$, $B = 11$, …, $F = 15$, and therefore the standard digits are $\{0, 1, \ldots, 9, A, B, \ldots, F\}$.

For example for the hexadecimal number $x = A1E$ it holds

$$(A1E)_{10} = 10 \cdot 16^2 + 1 \cdot 16^1 + 14 \cdot 16^0 = 10 \cdot 256 + 16 + 14 = 2590.$$

The translation of a decimal number into the hexadecimal system is especially easy if we already know its binary representation. There the binary digits can be clustered into groups of four digits for which the hexadecimal representation is computed, as in

$$(1123)_2 = \underbrace{0100}_{4\cdot 16^2}\underbrace{0110}_{6\cdot 16^1}\underbrace{0011}_{3\cdot 16^0} \quad \Rightarrow \quad (1123)_{16} = 463.$$

Representation (5.1) is equivalent to

$$x = \underbrace{\left\{(-1)^j \sum_{k=-\infty}^{\ell} \gamma_k p^{k-\ell-1}\right\}}_{=:s} \cdot p^{\ell+1} =: \underbrace{\left\{(-1)^j \sum_{i=1}^{\infty} \frac{\alpha_i}{p^i}\right\}}_{=:a} p^b, \tag{5.2}$$

where $\alpha_i := \gamma_{\ell-i+1}$, $i = 1, \ldots$ and $b := l + 1$. In (5.1) we have $\gamma_\ell \neq 0$ and thus we immediately get $\frac{1}{p} \leqslant |s| < 1$.

Definition 5.2: The representation of any $x \in \mathbb{R}$ as in (5.2) is called *normalized floating point representation* of x with respect to p. Here

$$a \quad := \quad (-1)^j \sum_{i=1}^{\infty} \frac{\alpha_i}{p^i} \quad \text{where } \alpha_i \in \{0, 1, \ldots, p-1\} \tag{5.3}$$

is called the *significand* and

$$b \quad := \quad (-1)^s \sum_{i=1}^{m} \beta_i p^{m-i}, \quad \text{for } s \in \{0, 1\},\ \beta_i \in \{0, 1, \ldots, p-1\} \tag{5.4}$$

the *exponent*.

This floating point representation is called normalized since $\alpha_1 \neq 0$.

In contrast to the representation above, on a computer we can only store finitely many digits in the significand. In case $\alpha_i = 0$ for all $i > t \in \mathbb{N}$, x can be encoded by saving j, s (for determining the signs of significand and exponent) and the digits in the p-adic representation of significand and exponent. This motivates the schematic representation

j	α_1	$\ldots$	α_t	s	β_1	$\ldots$	β_m

Thus we require $1 + t + 1 + m$ memory positions.

Example 5.3: For $p = 10$ the normalized floating point representation of the real number $35\,657.23$ is given as

$$0.3565723 \cdot 10^5 = \left(\frac{3}{10^1} + \frac{5}{10^2} + \frac{6}{10^3} + \frac{5}{10^4} + \frac{7}{10^5} + \frac{2}{10^6} + \frac{3}{10^7}\right) \cdot 10^5,$$

encoded as

0	3	5	6	5	7	2	3	0	5
j	α_1	α_2	α_3	α_4	α_5	α_6	α_7	s	β_1

.

In this example $t = 7$ and $m = 1$.

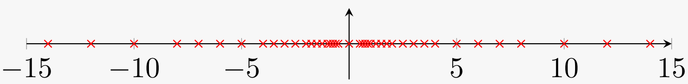
This now allows to define the representation of real numbers in sets of computer representable numbers.

Definition 5.4: For $p \in \mathbb{N}\backslash\{1\}$, $e_{\min}, e_{\max} \in \mathbb{Z}$, $t \in \mathbb{N}$ we denote the *set of normalized floating point numbers of length* t *with respect to the base* p *and range of exponents* $\{e_{\min}, e_{\min} + 1, \ldots, e_{\max}\} \subset \mathbb{Z}$ by

$$\mathbb{M}(p, t, e_{\min}, e_{\max}) := \{ \pm 0.\alpha_1\alpha_2 \ldots \alpha_t \cdot p^b \quad | \, \alpha_i \in \{0, \ldots, p-1\}, \alpha_1 \neq 0, \\ e_{\min} \leqslant b \leqslant e_{\max}\} \cup \{0\} \, .$$

$x \in \mathbb{M}(p, t, e_{\min}, e_{\max})$ is called *computer number* or *machine number*.

Example 5.5: The elements in $\mathbb{M}(2, 3, -1, 4)$ are shown in the following number ray



Note that machine numbers are not equally distributed.

5.2 Rounding Errors and Error Propagation

Real numbers need to be represented as machine numbers on a computer. They can not always be represented exactly due to the fact that the significand of a machine number has only t digits of accuracy, as we have for example seen in the translation of 0.1 to binary representation. In cases where these t digits are not sufficient, we need to either truncate the representation or round to the closest machine number. Doing this we introduce *rounding errors*.

5.2.1 Rounding Rules

The *rounding function*

$$\gamma : \mathbb{R} \to \mathbb{M}(p, t, e_{\min}, e_{\max})$$

for $x \in Z := [-x_{\max}, -x_{\min}] \cup \{0\} \cup [x_{\min}, x_{\max}]$ is determined by

$$\gamma(x) = \underset{\tilde{x} \in \mathbb{M}(p,t,e_{\min},e_{\max})}{\arg\min} |x - \tilde{x}|, \tag{5.5}$$

where

$$\begin{aligned} x_{\min} &:= \min\{|x| \mid x \in \mathbb{M}(p,t,e_{\min},e_{\max}) \backslash \{0\}\}, \\ x_{\max} &:= \max\{|x| \mid x \in \mathbb{M}(p,t,e_{\min},e_{\max})\}. \end{aligned}$$

Let $x = \pm \sum\limits_{i=1}^{\infty} \frac{\alpha_i}{p^i} \cdot p^b \in Z$ with $\alpha_1 \neq 0$. Then we have

$$\gamma(x) = \begin{cases} \pm \sum\limits_{i=1}^{t} \frac{\alpha_i}{p^i} \cdot p^b, & \alpha_{t+1} < \frac{p}{2}, \\ \pm \left(\sum\limits_{i=1}^{t} \frac{\alpha_i}{p^i} + \frac{1}{p^t} \right) \cdot p^b, & \alpha_{t+1} > \frac{p}{2}. \end{cases}$$

The special case of $\alpha_{t+1} = \frac{p}{2}$ is not uniquely determined via (5.5). There, we have, e.g., the following options:

Round up: Handle $\gamma(x)$ as if $\alpha_{t+1} > \frac{p}{2}$.

"Round-to-even": Rounds towards the closest machine number with an α_t that is even.

For example for $p = 2$, $t = 3$:

$$\begin{aligned} \gamma(0.1001) &= 0.100 && \text{(round down)} \\ \gamma(0.1011) &= 0.110 && \text{(round up)} \end{aligned}$$

The advantage as compared to rounding up is a (statistically) more equal distribution of rounding errors (they are partially negating each other). Positive effects have among others been observed in astro-physical long term computations as, e.g., in the investigation of the "Big Bang" theory.

Overflows and Underflows It still remains to specify $\gamma(x)$ for $x \notin Z$. Here we have to distinguish two cases:

$|x| < x_{\min}$: This case is called *underflow*. There are two ways to deal with this exception. On the one hand, we can round towards the closest valid machine number:

$$\gamma(x) = \begin{cases} 0 & \text{or rather} \\ \operatorname{sign}(x)\, x_{\min} & \end{cases}$$

On the other hand, one can use the so called *gradual underflow*. There we use representable but non-normalized floating point numbers, i.e., floating point numbers allowing $\alpha_1 = 0$ to circumvent the underflow. The smallest number representable in this way is $0.\underbrace{0\dots01}_{t}\cdot p^{e_{\min}}$. In this case the same rounding rules as for $x \in Z$ are used.

$|x| > x_{\max}$**:** This case is called *overflow*. Here we have the two variants

$$\gamma(x) = \begin{cases} \operatorname{sign}(x)\, x_{\max} \\ \operatorname{sign}(x) \cdot \infty. \end{cases}$$

The latter of which needs to extend our definition of $\mathbb{M}(p, t, e_{\min}, e_{\max})$ by a symbol for ∞. This is used in the IEEE 754 standard for floating point arithmetic (see also p. 107ff.).

After having defined a proper rounding function we have to ask ourselves how large the rounding errors can actually get. Here and in the following, for an exact quantity x and its machine number approximation $\tilde{x}$, we distinguish the *absolute error*

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

and the *relative error*

$$\frac{\|x - \tilde{x}\|}{\|x\|}.$$

Therein $\|\,.\,\|$ for a scalar entity in general means the absolute value, whereas otherwise it stands for a suitable norm.

For the rounding errors in $\mathbb{M}(p, t, e_{\min}, e_{\max})$ we have the following important results:

Lemma 5.6: The absolute rounding error fulfills

$$|\gamma(x) - x| \leqslant \frac{p^{-t}}{2} \cdot p^b \quad \forall x \in Z.$$

Proof. Let $x := \pm \sum_{i=1}^{\infty} \frac{\alpha_i}{p^i} p^b$ and define

$$y_1 := \operatorname{sign}(x) \sum_{i=1}^{t} \frac{\alpha_i}{p^i}\, p^b \qquad \text{(round towards zero)}$$

$$y_2 := \operatorname{sign}(x) \left(\sum_{i=1}^{t} \frac{\alpha_i}{p^i} + \frac{1}{p^t} \right) p^b \qquad \text{(round away from zero)}$$

Then apparently we have $\gamma(x) \in \{y_1, y_2\}$ and

$$x \in \begin{cases} [y_1, y_2], & x > 0, \\ [y_2, y_1], & x < 0. \end{cases}$$

Let $a_1 < a_2 \in \{y_1, y_2\}$, since $|x - a_j| \leqslant \frac{1}{2}|a_2 - a_1| = \frac{1}{2}|y_2 - y_1|$ either for $j = 1$, or for $j = 2$, or both, if $x \in [a_1, a_2]$, we find

$$|\gamma(x) - x| \leqslant \frac{1}{2}\ |y_2 - y_1| = \frac{1}{2}\frac{p^b}{p^t}.$$

□

Lemma 5.7: Let $Z = [-x_{\max}, -x_{\min}] \cup \{0\} \cup [x_{\min}, x_{\max}]$, as above. The relative rounding error for all $x \in Z \backslash \{0\}$ fulfills

$$\frac{|\gamma(x) - x|}{|x|} \ < \ \frac{1}{2}p^{1-t}.$$

Proof. The significand a of x fulfills $|a| \geqslant \frac{1}{p}$. Thus we have $|x| \geqslant \frac{1}{p} \cdot p^b$. From Lemma 5.6 we, therefore, find

$$\frac{|\gamma(x) - x|}{|x|} \leqslant \frac{1}{p^{b-1}}\frac{1}{2}\ p^{b-t} = \frac{1}{2}\ p^{1-t}.$$

From $|x| > \frac{1}{p}p^b$ we have strict inequality unless $x = \pm\frac{1}{p} \cdot p^b$. In the latter case, however, $x \in \mathbb{M}(p, t, e_{\min}, e_{\max})$ and so $\gamma(x) = x$, i.e., $\frac{|\gamma(x)-x|}{|x|} = 0$. □

Remark 5.8: Note that $x \in Z\backslash\{0\}$, excludes the case of gradual underflow.

Definition 5.9: The quantity $\mathbf{u} := \frac{1}{2}p^{1-t}$ is called *"unit round off"*.

The unit round off describes the relative error that can result from rounding operations. It should not be mistaken for the *machine epsilon* eps.

$$\mathrm{eps} := \min\{|\tilde{x} - 1| \mid \tilde{x} \in \mathbb{M}(p, t, e_{\min}, e_{\max}), \tilde{x} > 1\} = p^{1-t} = 2\mathbf{u},$$

determines the distance of 1 to the next larger machine number.

Remark 5.10: To be able to talk about the accuracy of an approximate quantity we have to estimate the relative error.

For example

$$x = 25.317, \ \tilde{x} = 25.313 \quad \text{(i.e., } \tilde{x} \text{ has 4 correct digits)}$$
$$\Longrightarrow \frac{|x-\tilde{x}|}{|x|} = \frac{0.004}{25.317} \approx 0.16 \cdot 10^{-3}.$$

It is an easy argumentation to find that the number of correct digits coincides with the negative exponent of the relative error (± 1).

The absolute error does not carry any information about the accuracy! For example for $y = 0.001, \tilde{y} = 0.002$: $|y - \tilde{y}| = 10^{-3}$ is rather small, but $\tilde{y}$ has no correct digit as we can see from the relative error

$$\frac{|y-\tilde{y}|}{|y|} = 1.$$

Remark 5.11: In C99 a set of commands and settings for influencing the computation with floating point numbers have been added to the C standard[a]. Especially the behavior of the rounding function $\gamma(.)$ can be influenced using the functions

```
int fegetround(void);
int fesetround(int round);
```

Available rounding models, i.e. values for the **round** argument, are

- **FE_DOWNWARD**,
- **FE_UPWARD**,
- **FE_TONEAREST** (default),
- **FE_TOWARDZERO**.

[a] see, e.g., http://openbook.galileocomputing.de/c_von_a_bis_z/030_c_anhang_b_005.htm for a list

5.2.2 Computer Arithmetic

We have introduced the relative and absolute rounding errors in the previous section and proved basic results regarding their sizes in Lemma 5.6 and Lemma 5.7.

> How do these rounding errors evolve under elementary arithmetic operations $(+, -, \cdot, /)$?

This question is investigated in the following.

As a direct consequence of Lemma 5.7 it follows

$$\gamma(x) = x(1+\varepsilon), \quad |\varepsilon| \leqslant \mathbf{u} \quad \forall x \in Z.$$

This is the error resulting from simply storing the number in the computers memory. For example for $p = 2$ we have seen before that $(0.1)_2 = 0.0\overline{0011}$. In normalized representation this is $0.11\overline{0011} \cdot 2^{-3}$. Now rounding to six digits (i.e. $t = 6$) we get

$$(\gamma(0.1))_2 = 0.110011 \cdot 2^{-3},$$

which means that in decimal representation we have $\gamma(0.1) = \frac{51}{512}$ which equals the decimal fraction 0.099609375.

Computers are only equipped with a so called *pseudo arithmetic*, since we can not expect in general that the result of $x\triangle y$ for $\triangle \in \{+,-,\cdot,/\}$ and machine numbers $x,\ y \in \mathbb{M}(p,t,e_{\min},e_{\max})$ will also be a number in $\mathbb{M}(p,t,e_{\min},e_{\max})$. This becomes obvious in the following example.

Example 5.12: Both $x = 0.12$ and $y = 0.34$ are from the set of machine numbers $\mathbb{M}(10,2,e_{\min},e_{\max})$, but for their product we easily see

$$x \cdot y = 0.0408 = 0.408 \cdot 10^{-1},$$

which requires a 3 digit mantissa and thus is not in $\mathbb{M}(10,2,e_{\min},e_{\max})$.

To put the result into $\mathbb{M}(10,2,e_{\min},e_{\max})$ we thus need to round. Denoting the result of a *floating point operation*, i.e., the result of a calculation $x\triangle y$ in a system of machine numbers by $x \circledtriangle y$ one usually determines the result as in

$$x \circledtriangle y = \gamma(x\triangle y), \quad \triangle \in \{+,-,\cdot,/\}. \tag{5.6}$$

That means the operation is performed exact first and rounded to a valid machine number afterwards. Doing this we achieve the

Standard Model of the Floating Point Arithmetic: For all floating point numbers $x,\ y \in \mathbb{M}(p,t,e_{\min},e_{\max})$ and any arithmetic operation $\triangle \in \{+,-,\cdot,/\}$ it holds:

$$x \circledtriangle y = (x\triangle y)(1+\delta), \quad \text{for a } |\delta| \leqslant \mathbf{u}. \tag{5.7}$$

In the following we will always assume the validity of (5.7) and that the same also holds for $\sqrt{x}$, i.e., $\gamma(\sqrt{x}) = \sqrt{x}(1+\delta)$ for a $\delta \in \mathbb{R}$ with $|\delta| \leqslant \mathbf{u}$.

Remark 5.13: Note that the standard model is not valid on all computers or electronic devices. However, on devices fulfilling the IEEE 754 standard, which are for example most modern CPUs, it is true.

For the realization of the standard model the storage of the intermediate results (before rounding) requires three extra digits in the significand. This can be implemented in various manners in the computational units of the CPU. More details regarding this issue can be found in [4].

5.2.3 Error Propagation

The main question we are treating next is how the errors we found in the above are propagating through a more complex computation. Since the standard model for the floating point arithmetic (5.7) only holds for machine numbers, for an arbitrary calculation for an elementary operation $x \triangle y$ already up to three errors play a role. Often in a computation a single elementary operation is not enough to get the result. Thus the rounding errors *accumulate* in the course of the computation.

Let us first treat addition and subtraction. Note that we can safely ignore the case where either of the involved numbers is 0. Then the final error reduces to the representation error for the other number, i.e., it is bounded by **u**.

Addition: Let $x, y \in \mathbb{R}\backslash\{0\}$, $\operatorname{sign}(x) = \operatorname{sign}(y)$ and

$$\begin{aligned} \tilde{x} &:= \gamma(x) = x(1+\delta_x), & |\delta_x| &\leqslant \mathbf{u}, \\ \tilde{y} &:= \gamma(y) = y(1+\delta_y), & |\delta_y| &\leqslant \mathbf{u}. \end{aligned}$$

Then we have

$$\begin{aligned} \tilde{x} \oplus \tilde{y} &= (\tilde{x} + \tilde{y})(1+\delta_{x+y}) \qquad (\text{where } |\delta_{x+y}| \leqslant \mathbf{u}) \\ &= (x(1+\delta_x) + y(1+\delta_y))(1+\delta_{x+y}) \\ &= ((x+y) + (x\delta_x + y\delta_y))(1+\delta_{x+y}) \end{aligned}$$

and

$$\begin{aligned} |\tilde{x} \oplus \tilde{y} - (x+y)| =& |(x+y)\delta_{x+y} + (x\delta_x + y\delta_y)(1+\delta_{x+y})| \\ \leqslant& |x+y|\mathbf{u} + (|x|\cdot\mathbf{u} + |y|\cdot\mathbf{u})(1+\mathbf{u}) \\ =& |x+y|\mathbf{u} + |x+y|\mathbf{u}(1+\mathbf{u}) \qquad (\text{using } \operatorname{sign}(x) = \operatorname{sign}(y)) \\ =& |x+y|(2\mathbf{u} + \mathbf{u}^2). \end{aligned}$$

Thus we find

$$\frac{|(\tilde{x} \oplus \tilde{y}) - (x+y)|}{|x+y|} \leqslant 2\mathbf{u} + \mathbf{u}^2.$$

The relative error is (up to a negligible higher order term $\mathbf{u}^2$) at most twice as large as the relative representation errors of the summands x and y. Accordingly, very many additions may lead to a noticeable accumulated error.

Subtraction: Corresponds to the addition of x, y as above, but with $\operatorname{sign}(x) \neq \operatorname{sign}(y)$. Instead of adding two numbers with different signs here we treat the subtraction of two numbers with a common sign.

Let x, y, or $\tilde{x}$, $\tilde{y}$ as above respectively. Without loss of generality we assume $x \neq y$. Since we assume validity of (5.7) we have

$$\begin{aligned} \tilde{x} \ominus \tilde{y} =& (\tilde{x} - \tilde{y})(1 + \delta_{x-y}) \qquad \text{(where } |\delta_{x-y}| \leqslant \mathbf{u}\text{)} \\ =& ((x-y) + (x\delta_x - y\delta_y))(1 + \delta_{x-y}) \end{aligned}$$

It follows

$$\begin{aligned} |(\tilde{x} \ominus \tilde{y}) - (x-y)| &= |(x-y)\delta_{x-y} + (x\delta_x - y\delta_y)(1+\delta_{x-y})| \\ &= |(x-y)\delta_{x-y} + (x\delta_x - y\delta_x + y\delta_x - y\delta_y)(1+\delta_{x-y})| \\ &= |(x-y)\delta_{x-y} + (x-y)\delta_x + y(\delta_x - \delta_y) \\ &\qquad + (x-y)\delta_x\delta_{x-y} + y(\delta_x - \delta_y)\delta_{x-y}| \\ &\leqslant 2|x-y| \cdot \mathbf{u} + 2|y|\mathbf{u} + |x-y| \cdot \mathbf{u}^2 + 2|y|\mathbf{u}^2 \end{aligned}$$

and

$$\frac{|(\tilde{x} \ominus \tilde{y}) - (x-y)|}{|x-y|} \leqslant \left(\frac{2|y|}{|x-y|} + 2\right)\mathbf{u} + \left(\frac{2|y|}{|x-y|} + 1\right)\mathbf{u}^2.$$

Thus for $x \approx y$ we have to expect an especially large relative error. This effect is called *cancellation*.

To avoid cancellation it is necessary to try and rewrite the expression in a way that avoids the subtraction of two almost equal numbers.

Example 5.14: Let

$$p = 10,\ t = 10,\ x = 1.2 \cdot 10^{-5} = 0.12 \cdot 10^{-4}$$

and

$$y = f(x) = \frac{1 - \cos(x)}{x^2}.$$

The evaluation of f in x gives

$$\begin{aligned} \cos(x) &= 0.99999999992800 \cdot 10^0 =: c \approx 1 \\ \implies \tilde{c} &:= \gamma(c) = 0.9999999999 \\ \implies \tilde{y} &= (1 \ominus \tilde{c}) \oslash (x \odot x) = 10^{-10} \oslash (0.144 \cdot 10^{-9}) = 0.6944444444. \end{aligned}$$

The correct result rounded to ten digits of accuracy, however, is

$$\gamma(f(x)) = 0.4999997300.$$

The reason for the wrong result is the cancellation in the evaluation of $1 \ominus \tilde{c}$. The result here has only one correct digit. The information about all the other digits got lost (was canceled) while rounding c. Then the subtraction is performed exact, but the error $1 \ominus \tilde{c}$ is amplified by a factor of 10^{10}. The second to tenth digits in the intermediate result are not carrying any information about correct values.

$$1 \ominus \tilde{c} = 0.1\underline{000000000} \cdot 10^{-9}$$

↑ information about these values is lost

Using the alternative formulation

$$f(x) = \frac{1}{2}\left(\frac{\sin(\frac{x}{2})}{\frac{x}{2}}\right)^2,$$

which uses the identity $\cos x = 1 - 2\sin^2\left(\frac{x}{2}\right)$, one gets the much better result $\tilde{y} = 0.5$.

Multiplication: We are now investigating the multiplication, of x, y, $\tilde{x}$, $\tilde{y}$ as above, in a similar manner. Note that here the sign does not play a role, and the case of either x, or y being 0 is even easier, since then the result is 0, too, and thus exact. With a $|\delta_{x \cdot y}| \leqslant \mathbf{u}$ we have

$$\begin{aligned} \tilde{x} \odot \tilde{y} &= \tilde{x}\tilde{y}(1 + \delta_{x \cdot y}) = x(1 + \delta_x)y(1 + \delta_y)(1 + \delta_{x \cdot y}) \\ &= xy(1 + \delta_x)(1 + \delta_y)(1 + \delta_{x \cdot y}) = xy + xy(\delta_x + \delta_y + \delta_{x \cdot y}) + \mathcal{O}(\mathbf{u}^2). \end{aligned}$$

So it immediately follows

$$\frac{|\tilde{x} \odot \tilde{y} - x \cdot y|}{|x \cdot y|} \leqslant 3\mathbf{u} + \mathcal{O}(\mathbf{u}^2).$$

We thus find that the multiplication behaves similar to the addition. The case of an actual division is again following analogously. Note that it should be avoided to divide by a very small value, since this might amplify rounding errors accumulated and present in the enumerator analogous to the cancellation in Example 5.14. However, in contrast to the case of cancellation in the subtraction, here only the absolute error is affected, but not the relative.

The most important difference of computer arithmetic as compared to exact arithmetic is the following:

Computer arithmetic is neither associative nor distributive.

That means in general we have

$$
\begin{aligned}
&(x \circledtriangle y) \circledtriangle z \neq x \circledtriangle (y \circledtriangle z) \\
&x \odot (y \oplus z) \neq (x \odot y) \oplus (x \odot z), \text{ etc.}
\end{aligned}
$$

Example 5.15: Given $\mathbb{M}(10, 5, e_{\min}, e_{\max})$ and $a = 4.2832, b = 4.2821, c = 5.7632$, we want to evaluate the expression $d := (a - b) \cdot c$. In exact calculation we find:

$$
d = (0.0011) \cdot 5.7632 = 0.00633952 \quad \Longrightarrow \quad \gamma(d) = 0.63395 \cdot 10^{-2}.
$$

The relative error is

$$
\frac{|d - \gamma(d)|}{|d|} \approx 0.3 \cdot 10^{-6}.
$$

In pseudo arithmetic using $\mathbb{M}(10, 5, e_{\min}, e_{\max})$ we have two options:

(i) $(a \ominus b) \odot c = (0.11 \cdot 10^{-2}) \odot (0.57632 \cdot 10^{1}) = 0.63395 \cdot 10^{-2} = \gamma(d)$, which gives the correct rounded result.

(ii) $(a \odot c) \ominus (b \odot c) =: e \ominus f =: g$

$$
\begin{aligned}
e &= a \odot c = \gamma(0.24684932824 \cdot 10^{2}) = 0.24685 \cdot 10^{2} \\
f &= b \odot c = \gamma(0.2467859872 \cdot 10^{2}) = 0.24679 \cdot 10^{2} \\
\Longrightarrow g &= e \ominus f = \gamma(0.00006 \cdot 10^{2}) = 0.6 \cdot 10^{-2}
\end{aligned}
$$

$$
\Longrightarrow \frac{|d - g|}{|d|} \approx 0.054,
$$

which leaves us with only a single correct digit.

The problem in the second approach is the cancellation in the subtraction of the two almost equal numbers e and f. During their computation we already performed rounding, which erased the information about the truncated digits. This information would have had to take the digits 2–5 in g to get to the correct result.

Computer	p	t	$e_{\min}$	$e_{\max}$
Univac 1108	2	27	-128	127
PDP – 11	2	24	-128	127
Cray – 1	2	48	-16384	16383
HP – 45	10	10	-98	100
TI – SR5x	10	12	-98	100
IBM System/360	16	6	-64	63

Table 5.1: Different ancient floating point implementations before IEEE 754.

```
half: (16 bit)
      S EEEEE MMMMMMMMMM
      0 1   5 6       15
single: (32 bit)
   S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
   0 1      8 9                     31
double: (64 bit)
S EEEEEEEEEEE MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
0 1        11 12                                                63
```

Figure 5.1: Storage patterns for **`half`**, **`single`** and **`double`** precision variables.

In conclusion we recognize that to avoid cancellation one needs to carefully work with the associativity and distributivity.

5.2.4 The IEEE Standard 754

Manufacturers usually standardize the usage of computer arithmetic to make computation results comparable. To this end, in 1985 the IEEE[1] fixed the standard 754 that is today used by almost all computer manufacturers.

IEEE 754–1985 The standard prescribes that $\mathbb{M}$ should be closed under the operations $+$, $-$, $\cdot$, $/$, $\sqrt{\cdot}$. That means any of these operations has to lead to a result in $\mathbb{M}$. Further contributions of the standard are:

- rounding is performed as “round-to-even”.
- the standard model for floating point arithmetic holds, i.e., the result of an elementary operation is behaving as if the exact result had been rounded.
- overflows result in $\gamma(x) = \pm\infty$.
- underflows are treated using subnormal numbers as described with the gradual underflow above.

[1]The Institute of Electrical and Electronics Engineers.

- two data types have been fixed: `double` (8 byte) and `single` (4 byte), both using $p = 2$.
- Since $\alpha_1 = 1$ has to hold due to normalization, it is not stored, which gives an extra bit for the significand.
- The **`single`** data type has the following properties; for **`double`** the corresponding values in Table 5.3 have to be inserted.
  - An exponent $E = 255$ is used to encode the elements $\pm\infty$ or **`NaN`** (not-a-number) that are necessary to ensure closedness of $\mathbb{M}$.
  - The exponent b of the machine number is derived from E via $b = E - 127$, which saves another bit for the sign of the exponent.
  - $E = 0$ is used to encode subnormal numbers.

  Summarizing we get the representation

  $$x = (-1)^S \cdot (1.\gamma_2 \dots \gamma_{24}) \cdot p^{E-127}.$$

  that slightly differs from Definition 5.4. For the minimal value $E = 1$ it follows

  $$x_{\min} = 1.\underbrace{0\dots0}_{23} \cdot 2^{1-127} = 0.1 \cdot 2^{-125} \qquad \Longrightarrow \qquad e_{\min} = -125.$$

  Further, we get

  $$e_{\max} = 1 + (254 - 127) = 128.$$

  Some examples for numbers in the system of **`single`** numbers are:

`0 11111111 00000000000000000000000`	=	$+\infty$
`1 11111111 00000000000000000000000`	=	$-\infty$
`0 11111111 00000100000000000000000`	=	**`NaN`**
`1 11111111 00100010001001010101010`	=	**`NaN`**
`0 10000000 00000000000000000000000`	=	$+1.0 * 2^{128-127} = 2$
`0 10000001 10100000000000000000000`	=	$+1.101 * 2^{129-127} = 6.5$
`1 10000001 10100000000000000000000`	=	$-1.101 * 2^{129-127} = -6.5$
`0 00000001 00000000000000000000000`	=	$+1.0 * 2^{1-127} = 2^{-126} = x_{\min}$
`0 00000000 10000000000000000000000`	=	$+0.1 * 2^{-126} = 2^{-127}$
`0 00000000 00000000000000000000001`	=	$+0.0\dots01 * 2^{-126} = 2^{-149}$
	=	smallest representable number
`0 00000000 00000000000000000000000`	=	$+0$
`1 00000000 00000000000000000000000`	=	-0
`0 01111111 00000000000000000000000`	=	$1.0 * 2^{127-127} = 1.0$
`1 01111111 00000000000000000000000`	=	$-1.0 * 2^{127-127} = -1.0$

Flag	Example	Result
invalid	$0/0,\ \ 0\cdot\infty,\ \ \sqrt{-1},$ $\infty/\infty,\ \ +\infty+(-\infty)$	**`NaN`** ("not a number")
overflow	$x_{\max} * x_{\max}$	$\pm\infty$ usually denoted: $\pm$**`Inf`**
division by zero	$x/0$ for $x \neq 0$	$\pm\infty$
underflow	$x_{\min}/p^s$, $1 < s < t$	subnormal number
inexact	$\mathrm{rd}(x \circ y) \neq x \circ y$	correctly rounded result

Table 5.2: IEEE Standard 754, Exception Handling.

```
bfloat16
   S EEEEEEEE MMMMMMM
   0 1      8 9    15
```

Figure 5.2: Storage pattern for **`bfloat16`** half precision variables.

- The value of a variable can be tested for **`NaN`** since this is the only "number" for which $x \neq x$ is true.
- Whenever an incorrect result or a number that is not covered by Definition 5.4 is encountered this is causing an *exception*. Then a *flag* is raised, which can be checked by the toolchain to create the appropriate warnings according to Table 5.2.

IEEE 754–2008 The revised edition of the standard serves a multitude of purposes:

- It merges IEEE 754–1985 with IEEE 845 (a standard defining decimal floating point numbers important in finance).
- It reduces the possible implementation alternatives, as well as ambiguous formulations.
- It adds two additional $p = 2$-based precision levels for **`half`** (also known as **`fp16`**) (2 byte) and **`quad`**ruple (16 byte) precision.
- It extends **`min`** and **`max`** for the special cases ± 0 and $\pm\infty$.
- The formerly *denormalized* numbers for *gradual underflow* treatment are now consistently called *subnormal* numbers.
- Also, a combined multiplication and addition operation called fused multiply add and performing $a \leftarrow a \pm (b \times c)$ was added to the set of basic operations fulfilling the standard model for floating point arithmetic.

precision	p	t	$e_{\min}$	$e_{\max}$	$\mathbf{u}$	$x_{\min}$	$x_{\max}$
`half`	2	$10+1$	-13	16	$\approx 4.88 \cdot 10^{-4}$	$\approx 6 \cdot 10^{-5}$	$\approx 1 \cdot 10^{5}$
`single`	2	$23+1$	-125	128	$\approx 5.96 \cdot 10^{-8}$	$\approx 1 \cdot 10^{-38}$	$\approx 3 \cdot 10^{38}$
`double`	2	$52+1$	$-1\,021$	$1\,024$	$\approx 1.11 \cdot 10^{-16}$	$\approx 10^{-308}$	$\approx 10^{308}$
`quad`	2	$112+1$	$-16\,381$	$16\,384$	$\approx 9.63 \cdot 10^{-35}$	$\approx 10^{-4\,932}$	$\approx 10^{4\,932}$

Table 5.3: IEEE standard 754-2008, data types.

5.2.5 An alternate 16-bit floating point format

The IEEE 754 **`fp16`** half precision format has drawbacks for scientific computations. The main one being its very limited number range due to the rather small representable exponents (see Table 5.3). In an attempt to overcome this issue, the Google Brain project, an artificial intelligence research group at Google, created their own 16bit floating point format *brain floating point*, or in short **`bfloat16`**.

The key properties of this format are closely related to the **`fp32`** or **`single`** format, as it preserves the first 16 bit of its layout. Consequently, it uses the same exponent range and sacrifices 16 bit from the significand, ending up with only 7 bit left (+1 bit again due to the leading implicit 1 from normalization). In turn, **`bfloat16`** inherits the good range of real numbers that single precision can cover with a a lot less numbers in that range in total due to the shorter significand. The handling of exceptions is following that of the IEEE numbers. A second attractive consequence of the close relation to **`fp32`** is that format conversion from **`fp32`** single precision to **`bfloat16`** half precision is especially easy as it only requires a copy of the leading 16 bit of the 32-bit number with manipulations only necessary for proper rounding.

While other lower precision number formats have been introduced by hardware manufacturers (see [1] for a comparison), the **`bfloat16`** format appears to be the most widely accepted quasi standard. For an early discussion of the format we also refer to [5].

5.3 Error Analysis

This section is dedicated to the derivation of a general framework for the appraisal of the quality of numerically generated results of computations. The computed result can differ from the real result due to a number of errors from different categories:

data errors The data used in the computations are not known exactly, e.g., due to measurement inaccuracies.

rounding errors Errors resulting from the necessity to work with numbers from

$\mathbb{M}(p, t, e_{\min}, e_{\max})$ instead of $\mathbb{R}$ and the evaluation of expressions with a finite significand. The propagation and accumulation of these kinds of errors was already discussed in the above.

methodological errors Methodological errors depend on different factors. On the one hand, the accuracy of the model underlying the computation plays a role. On the other hand, also the solution method applied to solve or evaluate the model has a crucial contribution to this type of error.

The methodological error in any case strictly depends on the task at hand and the way it is solved. In the following we will therefore restrict to the impact of data and rounding errors on the computed result.

To this end, we will mainly employ the two concepts of *conditioning* (or *condition numbers*) and *stability*.

5.3.1 Conditioning/Condition Number

The concept of conditioning or condition numbers is a property of the mathematical problem only. It is independent of the actual algorithm or method used for solving the problem. Thus it provides the ability to derive statements about the maximum possible quality of the numerical results. Consider the following example. We want to compute the root of a linear affine function, i.e., the intersection with the x-axis. The steeper the function is the better, i.e., the more accurate, we can derive the x value of the root. This is due to the fact that small perturbations in the function value for a steep function lead to even smaller perturbation of the corresponding x value. The problem is said to be *well conditioned* in this case. On the other hand, if the function is very flat already small perturbations in the y values lead to large perturbations in the position of the computed root. This corresponds to a very bad conditioning of the problem. We thus see that the conditioning may depend on both the problem and the data.

To put this in more mathematical terms, we consider the problem of evaluating $y = f(x)$, where the function $f : D \to V$ maps the data $x \in D$ to the result $y \in V$ and $y + \Delta y = f(x + \Delta x)$ is the result for the perturbed data $x + \Delta x$. Then the relative error for an optimal result to be expected is bounded as in:

$$\frac{\|\Delta y\|}{\|y\|} \leqslant c(f, x) \cdot \frac{\|\Delta x\|}{\|x\|},$$

where $c(f, x)$ is called the *condition number* for the problem of evaluating $f(x)$.

5.3.2 Stability

The corresponding property for the algorithm is called *stability*. Its main purpose is to guarantee that the algorithm at least gives

$$\frac{\|\Delta y\|}{\|y\|} \lessapprox c(f,x) \cdot \frac{\|\Delta x\|}{\|x\|}.$$

That means we get as close to the optimal result as possible. Such an algorithm is then called *numerically stable* (We will give a precise definition in Definition 5.18). A bad algorithm would give a larger error. It is then called *numerically unstable*.

In the following we will use the notation from above:

- $x \in D$ are the data for the problem,
- $f : D \to V$ is the mathematical problem mapping data to values,
- and $y = f(x) \in V$ is the exact result, whereas
- $\hat{y}$ is the numerically computed result.

5.3.3 Forward Error Analysis

The first and obvious question that arises is how far apart y and $\hat{y}$ are, i.e.,

$$\|y - \hat{y}\| = ?, \qquad \frac{\|y - \hat{y}\|}{\|y\|} = ?$$

This question is answered by a *forward error analysis*. Here one proceeds through the computation step by step analyzing the propagation and accumulation of rounding errors by means of the methods discussed in Section 5.2. The basic procedure is best explained using a small example.

Example 5.16: Let the mathematical problem be that of solving the simple quadratic equation $y^2 - 2ay + b = 0$, for given $a, b \in \mathbb{M}(p, t, e_{\min}, e_{\max})$. The two solutions are known to be

$$y_1 = a - \sqrt{a^2 - b}, \quad \text{and} \quad y_2 = a + \sqrt{a^2 - b}.$$

We concentrate on the computation of y_1. Exactly following the solution formula above is giving the below algorithm in exact and finite arithmetic (following the standard model for floating point arithmetic):

$$\begin{array}{lllll} & \text{exact computation} & & \text{numerical realization} \\ 1. & c := a \cdot a & \Longrightarrow & \hat{c} = a^2(1+\delta_1) \\ 2. & d := c - b & \Longrightarrow & \hat{d} = (\hat{c} - b)(1+\delta_2) \\ 3. & e := \sqrt{d} & \Longrightarrow & \hat{e} = \sqrt{\hat{d}}(1+\delta_3) \\ 4. & y_1 := a - e & \Longrightarrow & \hat{y}_1 = (a - \hat{e})(1+\delta_4) \end{array}$$

Here we have $|\delta_i| \leqslant \mathbf{u}$, $i = 1, \ldots, 4$ due to the standard model assumption.

Now inserting all computed quantities we find

$$\begin{aligned}
\hat{y}_1 &= \left\{a - \sqrt{(a^2(1+\delta_1) - b)(1+\delta_2)(1+\delta_3)}\right\}(1+\delta_4) \\
&= a(1+\delta_4) \\
&\quad - \Bigg\{a^2 \underbrace{(1+\delta_1)(1+\delta_2)(1+\delta_3)^2(1+\delta_4)^2}_{\substack{= 1+\delta_1+\delta_2+2\delta_3+2\delta_4+\mathcal{O}(\mathbf{u}^2) \\ =: 1+\varepsilon_1, \quad |\varepsilon_1| \leqslant 6\mathbf{u} + \mathcal{O}(\mathbf{u}^2)}} - b \underbrace{(1+\delta_2)(1+\delta_3)^2(1+\delta_4)^2}_{\substack{1+\delta_2+2\delta_3+2\delta_4+\mathcal{O}(\mathbf{u}^2) \\ =: 1+\varepsilon_2, \quad |\varepsilon_2| \leqslant 5\mathbf{u} + \mathcal{O}(\mathbf{u}^2)}}\Bigg\}^{\frac{1}{2}} \\
&= a + a\delta_4 - \sqrt{(a^2-b) + (a^2\varepsilon_1 - b\varepsilon_2)} \\
&= a + a\delta_4 - \sqrt{a^2-b} - \frac{1}{2\sqrt{a^2-b}}(a^2\varepsilon_1 - b\varepsilon_2) + \mathcal{O}(\mathbf{u}^2)
\end{aligned}$$

The last step exploits that using a Taylor expansion of $g(x) := \sqrt{x}$ at

$$x + \Delta x = \underbrace{a^2 - b}_{=:x} + \underbrace{a^2\varepsilon_1 - b\varepsilon_2}_{=:\Delta x},$$

we get

$$g(x+\Delta x) = \sqrt{x+\Delta x} = \sqrt{x} + \frac{1}{2\sqrt{x}}\Delta x + \mathcal{O}((\Delta x)^2),$$

where $|\Delta x| \leqslant 6(|a^2| + |b|)\mathbf{u} = \mathcal{O}(\mathbf{u})$.

Using this knowledge for the numerical result it follows

$$\hat{y}_1 = y_1 - \frac{1}{2\sqrt{a^2-b}}(a^2\varepsilon_1 - b\varepsilon_2) + a\delta_4 + \mathcal{O}(\mathbf{u}^2)$$

and thus for the relative error we get

$$\begin{aligned}
\frac{|\hat{y}_1 - y_1|}{|y_1|} &= \frac{1}{|a - \sqrt{a^2-b}|} \cdot \frac{1}{2\sqrt{a^2-b}} \underbrace{\left|a^2\varepsilon_1 - b\varepsilon_2 + 2a\delta_4\sqrt{a^2-b}\right|}_{\leqslant a^2\cdot 6\mathbf{u} + \underbrace{|b|\cdot 5\mathbf{u}}_{<|b|\cdot 6\mathbf{u}} + |a|\sqrt{a^2-b}\cdot 2\mathbf{u}} + \mathcal{O}(\mathbf{u}^2) \\
&\leqslant 3\frac{a^2 + |b| + |a|\sqrt{a^2-b}}{\sqrt{a^2-b}\cdot|a - \sqrt{a^2-b}|}\mathbf{u} + \mathcal{O}(\mathbf{u}^2)
\end{aligned}$$

The forward error may be large if the denominator is small. This can happen in two cases that can both be traced back to cancellation happening in the computation of y_1.

(i)	$a^2 \approx b$	$\Longrightarrow$	cancellation in	$2.\ d := a^2 - b,$
(ii)	$\lvert b\rvert \ll a^2 \wedge a > 0$	$\Longrightarrow$	cancellation in	$4.\ y_1 = a - e.$

This example shows again why cancellation can lead to large errors in the overall computation. To avoid this effect we have to use adapted formulas, i.e. improve the numerical method. (See exercises)

5.3.4 Backward Error Analysis

The second and less obvious question that we want to investigate is the following. Given the result of the computation $\hat{y}$ — can we express $\hat{y}$ as the exact solution of a mathematical problem for slightly perturbed data? That means:

Does there exist a Δx, such that $\hat{y} = f(x + \Delta x)$?

Asking this question makes sense, since for inaccurate data x we only know the correct value up to, e.g., measurement errors. If the analysis for $\hat{y} = f(x + \Delta x)$ now provides a Δx that is of the magnitude of the data errors (i.e., measurement inaccuracies), then the computation result is as good as we can expect. An answer to the above question is derived by a so called *backward error analysis*.

Definition 5.17: $\eta := \inf\{\|\Delta x\|; \hat{y} = f(x+\Delta x)\}$ is the *(absolute) backward error* of $\hat{y}$, $\eta_{\text{rel}} := \eta/\|x\|$ is called the *relative backward error*, where $\| \,.\, \|$ is a suitable norm in the set of data D.

The relation of forward and backward errors is best described by the diagram in Figure 5.3.

The concepts of forward and backward error now enable us to give a precise definition of the corresponding notions of numerical stability as introduced in the beginning of this section.

Definition 5.18: If for any $x \in D$ a method for computing $y = f(x)$ produces a $\hat{y} = f(x + \Delta x)$ for a small relative backward error $\frac{\Delta x}{x}$, then the method is said to be *(numerically) backward stable*. The concrete definition of small depends on the problem, but might, e.g., mean Δx is of the size of the unavoidable data errors.

On the other hand, a method is called *(numerically) forward stable* if it produces a relative forward error $\frac{\Delta y}{y}$ of the same magnitude that a backward stable method would.

Remark 5.19: Note that a forward stable method does not necessarily have to be backward stable to fulfill the definition. Also the definition is

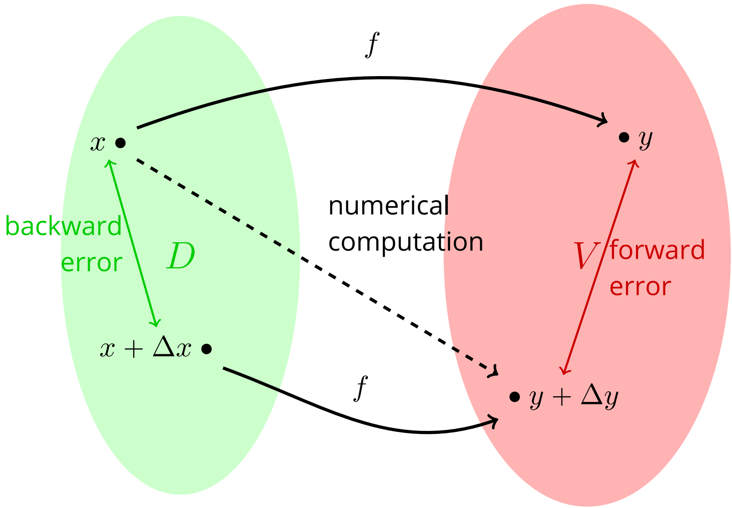


Figure 5.3: Forward/Backward Error Relations in Numerical Computations

mainly expressing the rule of thumbs that a forward stable algorithm produces an error that is approximately proportional to the data error via the condition number. Even if the backward error of the computed solution is small, this error can be amplified by a factor as large as the condition number when passing to the forward error, for a foward stable method.

We always have:

backward stable $\Rightarrow$ **forward stable**

The opposite implication does, however, in general not hold.

The verification of backward stability is performed by a *backward error analysis*. The backward error analysis treats the computed result $\hat{y}$ as that of the exact computation for perturbed data. Afterward the perturbed data and the original data are compared. The approach is introduced by revisiting the Example 5.16 and performing the analog procedure for the backward analysis.

Example 5.20 (Example 5.16 continued)**:** Consider $y_1 = a - \sqrt{a^2 - b}$ and $\hat{y}_1$ the corresponding solution of the quadratic equation for perturbed data a and b

$$y^2 - 2(a + \Delta a)y + (b + \Delta b) = 0$$

To this end, we require an expression of the form

$$\hat{y}_1 = (a + \Delta a) - \sqrt{(a + \Delta a)^2 - (b + \Delta b)}.$$

As for the forward error analysis in Example 5.16 we get

$$
\begin{aligned}
\hat{y}_1 &= a(1+\delta_4) \\
&\quad - \Big\{a^2 \underbrace{(1+\delta_1)(1+\delta_2)(1+\delta_3)^2}_{\substack{=1+\delta_1+\delta_2+2\delta_3+\mathcal{O}(\mathbf{u}^2)\\ =:1+\varepsilon_1,\quad |\varepsilon_1|\leqslant 4\mathbf{u}+\mathcal{O}(\mathbf{u}^2)}} (1+\delta_4)^2 - b\underbrace{(1+\delta_1)(1+\delta_3)^2(1+\delta_4)^2}_{=:1+\varepsilon_2,\quad |\varepsilon_2|\leqslant 5\mathbf{u}+\mathcal{O}(\mathbf{u}^2)}\Big\}^{\frac{1}{2}} \\
&= a + a\delta_4 - \Big\{(a+a\delta_4)^2 - b\big(\underbrace{1+\varepsilon_2 - \frac{a^2}{b}\varepsilon_1(1+\delta_4)^2}_{\substack{=1+\varepsilon_2-\frac{a^2}{b}\varepsilon_1+\mathcal{O}(\mathbf{u}^2)\\ =:1+\delta_b,\ |\delta_b|\leqslant 5\mathbf{u}+\frac{4a^2}{|b|}\mathbf{u}+\mathcal{O}(\mathbf{u}^2)}}\big)\Big\}^{\frac{1}{2}} \\
&= (a+a\delta_4) - \sqrt{(a+a\delta_4)^2 - (b+b\delta_b)}
\end{aligned}
$$

Now defining $\Delta a := a\delta_4$, $\Delta b := b\delta_b$ we can estimate the relative backward error as

$$
\begin{aligned}
\frac{|\eta_a|}{|a|} &\leqslant \frac{|\Delta a|}{|a|} \leqslant |\delta_4| \leqslant \mathbf{u}, \\
\frac{|\eta_b|}{|b|} &\leqslant |\delta_b| \leqslant \underbrace{\left(5 + \frac{4a^2}{|b|}\right)}_{\text{amplification factor}} \mathbf{u} + \mathcal{O}(\mathbf{u}^2).
\end{aligned}
$$

Note that the relative error is the infimum over all possible errors $\Delta x = \Delta\,[a,\ b]$. A small backward error, as we would expect it from a numerically backward stable algorithm, is derived if $a^2 \approx |b|$. The error may get large in case $a^2 \gg b$.

Remark 5.21: The separate consideration of the backward errors in a and b is called *component-wise error analysis*. For a *norm-wise* consideration one tries to estimate $\frac{1}{\left\|\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]\right\|_2}\eta$.

5.3.5 Perturbation Analysis

Knowing the limitations on the range of small expected errors, we need to find out next, whether the problematic error amplification is problem immanent or caused by the specific algorithmic approach we chose for solving the problem. The question thus is, if we can reformulate the algorithm to avoid the problem.

This question is answered employing a perturbation analysis that is used to find the condition number of the problem. We will introduce the procedure following the steps for an abstract (scalar) model example again.

To this end, let

$$f : D \to V, \quad f \in \mathcal{C}^2(D), \qquad y = f(x), \quad \hat{y} = f(x + \Delta x).$$

The question, that we are going to answer now, is in what sense the perturbation of the data is transported to the result. Geometrically, it is easy to see that the value of $\hat{y}$ is deviating from y the more, the larger the slope of the tangent of f in x, i.e., $|f'(x)|$ is. In the general case we use the total differential type of definition of the derivative of f in x to estimate the deviation. That means,

$$f(x + \Delta x) = f(x) + f'(x)\Delta x + o(\Delta x).$$

Here, $g \in o(\Delta x)$ means that $\lim_{\Delta x \to 0} \frac{g(\Delta x)}{\Delta x} = 0$. Then

$$\begin{aligned} \hat{y} - y &= f(x + \Delta x) - f(x) \\ &= f(x) + f'(x)\Delta x + o(\Delta x) - f(x) \\ &= f'(x) \cdot \Delta x + o(\Delta x) \approx f'(x) \cdot \Delta x. \end{aligned}$$

This approximation means that (neglecting an asymptotically vanishing remainder term) the factor $|f'(x)|$ amplifies the data errors in the result $\hat{y}$.

This treatment is called *asymptotic* or *local perturbation analysis* since it asymptotically gets better when successively narrowing in on x and obviously the approximation is only good in a local neighborhood of x.

Let $y \neq 0$ then we have

$$\begin{aligned} \frac{\hat{y} - y}{y} &= \frac{f'(x)\Delta x}{y} + o(\Delta x) \\ &= \frac{f'(x) \cdot x}{f(x)} \cdot \frac{\Delta x}{x} + o(\Delta x) \end{aligned}$$

and thus

$$\frac{|\hat{y} - y|}{|y|} = \underbrace{\frac{\left|f'(x) \cdot x\right|}{|f(x)|}}_{=:c(f,x)} \cdot \frac{|\Delta x|}{|x|} + o(|\Delta x|). \tag{5.8}$$

Note that in (5.8) we are not applying the triangular inequality, but equality may hold since $o(|\Delta x|)$ is allowed to be negative.

Definition 5.22: Let $f \in \mathcal{C}(D)$, $x, x + \Delta x \in D$ and $f(x + \Delta x) = \hat{y}$. The infimum of all numbers $c_{\text{abs}}(f, x)$ for which

$$\|y - \hat{y}\| \leqslant c_{\text{abs}}(f, x)\, \|\Delta x\| + o(\|\Delta x\|)$$

holds, is called *(absolute) condition number of f in x.*

Analogously, the infimum of all numbers $c(f, x) = c_{\text{rel}}(f, x)$, such that

$$\frac{\|y - \hat{y}\|}{\|y\|} \leqslant c_{\text{rel}}(f, x) \frac{\|\Delta x\|}{\|x\|} + o\left(\frac{\|\Delta x\|}{\|x\|}\right)$$

is true, is denoted as *(relative) condition number of f in x.*

If f is differentiable then in analogy to (5.8)

$$c_{\text{abs}}(f, x) = \left\|f'(x)\right\|, \quad c(f, x) = c_{\text{rel}}(f, x) \leqslant \frac{\|x\|}{\|f(x)\|} \left\|f'(x)\right\|,$$

where f' is the Jacobi matrix of $f : D \to V$ in x and the norms have to be compatible. That means, ideally, for the Jacobian the operator norm (see Definition 6.10) induced by the vector norms on V and D should be used.

Note that in (5.8) equality holds. For an inequality we would only have an upper bound to the condition number. This would only then become the condition number when it can be shown to be a *sharp* bound, i.e., when we can find at least one $x \in D$ such that equality holds (minimum case), or for every $\delta > 0$ there exists an $\hat{y} = f(x + \hat{x}) \in V$, such that for $c(f, x) \cdot \hat{x} - \delta$ violates the bound (infimum case).

Example 5.23 (Examples 5.16, 5.20 continued)**:** Let us get back to the example quadratic equation. Here we have $x = \left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] \in \mathbb{R}^2$ and

$$f(a, b) = a - \sqrt{a^2 - b}, \quad y = f(a, b), \quad \hat{y} = f(a + \Delta a, b + \Delta b).$$

Further, let us assume

$$\max\left\{\frac{|\Delta a|}{|a|}, \frac{|\Delta b|}{|b|}\right\} \leqslant \varepsilon \ll 1.$$

For the evaluation of the Taylor expansion we require the partial derivatives of

f with respect to the data a, b:

$$\begin{aligned}
\frac{\partial f}{\partial a}(a,b) &= 1 - \frac{1}{2}(a^2-b)^{-\frac{1}{2}} \cdot 2a = 1 - \frac{a}{\sqrt{a^2-b}} = \frac{\sqrt{a^2-b}-a}{\sqrt{a^2-b}} \\
&= -\frac{f(a,b)}{\sqrt{a^2-b}}, \\
\frac{\partial f}{\partial b}(a,b) &= \frac{1}{2} \cdot \frac{1}{\sqrt{a^2-b}}.
\end{aligned}$$

Further assuming that $a^2 > b > 0$ or $b < 0$, such that $\sqrt{a^2-b} \in \mathbb{R}$, we find

$$\begin{aligned}
\hat{y} - y &= f(a,b) + \frac{\partial f}{\partial a}(a,b) \cdot \Delta a + \frac{\partial f}{\partial b}(a,b) \cdot \Delta b + o(\varepsilon) - f(a,b) \\
&= -\frac{f(a,b)a}{\sqrt{a^2-b}} \cdot \frac{\Delta a}{a} + \frac{1}{2} \cdot \frac{b}{\sqrt{a^2-b}} \frac{\Delta b}{b} + o(\varepsilon)
\end{aligned}$$

and thus

$$\frac{|\hat{y}-y|}{|y|} \leqslant \underbrace{\frac{|a|}{\sqrt{a^2-b}}}_{=:c_a(f,a,b)} \cdot \frac{|\Delta a|}{|a|} + \underbrace{\frac{|b|}{2\sqrt{a^2-b} \cdot |a-\sqrt{a^2-b}|}}_{=:c_b(f,a,b)} \cdot \frac{|\Delta b|}{|b|} + o(\varepsilon) \tag{5.9}$$

$$\leqslant \frac{1}{\sqrt{a^2-b}} \left(|a| + \frac{|b|}{2|a-\sqrt{a^2-b}|} \right) \cdot \varepsilon + o(\varepsilon). \tag{5.10}$$

The inequality (5.9) here represents the component-wise perturbation analysis and (5.10) the norm-wise one. A norm-wise consideration also follows from the Cauchy-Schwarz-Inequality applied to

$$\hat{y} - y = (\nabla f(a,b))^{\mathsf{T}} \begin{bmatrix} \Delta a \\ \Delta b \end{bmatrix} + o(\varepsilon),$$

such that

$$|\hat{y} - y| \leqslant \|\nabla f(a,b)\| \cdot \left\| \begin{bmatrix} \Delta a \\ \Delta b \end{bmatrix} \right\| + o(\varepsilon).$$

Here, we are only interested in the (usually more precise) component wise consideration. The two cases of major interest are the ones that we have investigated to lead to large errors in the forward analysis (Example 5.16) and backward analysis (Example 5.20).

case 1: $a^2 \approx b$ For $a^2 \to b$ it follows $c_a(f,a,b) \to \infty$ and also $c_b(f,a,b) \to \infty$.

The problem thus is ill-conditioned, i.e., we can not expect "good" results. A large forward error is "unavoidable". The large forward errors in this case are therefore caused by the bad conditioning of the problem. This corresponds to the observation in Example 5.20 that the backward error is still small in this case.

case 2: $a^2 \gg b$ In this case $c_a(f, a, b) \approx 1$. The same can easily be seen for $c_b(f, a, b)$ when considering $\frac{b}{a^2} \to 0 \Leftrightarrow b \to 0$ and applying L'Hôpitals rule. That means, we find that the problem is well conditioned in this case. Having large forward and backward errors here, therefore, means that our computation method is unstable.

Since our method for computing y_1 in the above examples was performing well in most cases and only misbehaved in the case where $a^2 \gg b$, we also call the method *conditionally stable*.

We conclude this section with a couple of facts that we should be aware of when trying to evaluate the quality of numerical computations.

1. $c(f, x)$ in general not only depends on the problem but also on the data supplied to it. A mathematical problem thus is not generally good or bad, but it depends on where in D we evaluate it.

2. Condition numbers can be categorized as follows:
   $c(f, x) \approx 1$ $\Rightarrow$ well conditioned.
   $c(f, x) \gg 1$ $\Rightarrow$ ill-conditioned.
   $c(f, x) \ll 1$ may be bad as well since we can easily "lose information" due to the large possible backward errors.

3. An unstable algorithm can result from the decomposition of a (possibly well conditioned) mathematical problem into a concatenation of sub-tasks, i.e.,
   $$f(x) = (g_k \circ g_{k-1} \circ \ldots \circ g_1)(x),$$
   where one or more of the g_j are ill-conditioned. For example, if the g_j are elementary operations and one of them is suffering from cancellation, then the loss of information resulting from the cancellation may prevail the remaining computation.

4. The main property of the connection between forward error, backward error and condition number is sketched by the rough rule:
   $$\text{forward error} \approx \text{condition number} \times \text{backward error}.$$
   This again illustrates the implication
   $$\text{backward stability} \Rightarrow \text{forward stable}$$

The following *rule of thumb* gives a good assessment of the numerically computed results:

good conditioning & stable algorithm	$\Longrightarrow$	reliable result.
bad conditioning *or* unstable algorithm	$\Longrightarrow$	unsure result.

Bibliography

[1] *bfloat16 floating-point format*, https://en.wikipedia.org/wiki/Bfloat16_floating-point_format. accessed 2023-12-04.

[2] P. Deuflhard and A. Hohmann, *Numerical analysis in modern scientific computing. An introduction.*, no. 43 in Texts in Applied Mathematics., Springer-Verlag, New York, 2003, https://doi.org/10.1007/978-0-387-21584-6.

[3] O. Forster, *Analysis 1. Differential and integral calculus of one variable. (Analysis 1. Differential- und Integralrechnung einer Veränderlichen.)* , Wiesbaden: Vieweg+Teubner, 10th revised and expanded ed. ed., 2011.

[4] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, PA, second ed., 2002.

[5] N. J. Higham, *Half precision arithmetic: fp16 versus bfloat16*, Dec. 2018, https://nhigham.com/2018/12/03/half-precision-arithmetic-fp16-versus-bfloat16/. last accessed 2023-12-04.

[6] M. L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, Apr. 2001.

Mathematics is the queen of the sciences.

Carl Friedrich Gauß

CHAPTER 6

Basic Operations, Formats and Matrix-Norms

Contents

6.1 Vector Norms and Inner Products

Definition 6.1: Let X be a linear space over the field $\mathbb{F}$. A mapping

$$\|.\| : X \to \mathbb{R},$$

with

i) $\|x\| \geqslant 0 \quad \forall x \in X$, (positivity)

ii) $\|x\| = 0 \iff x = 0$, (definiteness)

iii) $\|\alpha x\| = |\alpha| \, \|x\| \quad \forall \alpha \in \mathbb{F}, \forall x \in X$, (homogeneity)

iv) $\|x + y\| \leqslant \|x\| + \|y\| \quad \forall x, \, y \in X$, (triangle inequality)

is called *norm* on X. A linear space together with a norm $(X, \|.\|_X)$ is called *normed linear space*.

Example 6.2: Let $X = \mathbb{R}^n$, $p \in \mathbb{N}$. The functions

$$\|x\|_p := \sqrt[p]{\sum_{i=1}^{n} |x_i|^p} \qquad p \in \mathbb{N}$$

$$\|x\|_\infty := \max_i |x_i|$$

define norms on X.

Definition 6.3: Let X be a linear space over the field $\mathbb{F} \in \{\mathbb{R}, \mathbb{C}\}$. An *inner product* on X is defined by a sesquilinear form

$$(.,.) \, : \, X \times X \to \mathbb{F}$$

with properties

i) $(x, x) \in \mathbb{R}_{\geqslant 0} \quad \forall x \in X$, (positivity)

ii) $(x, x) = 0 \iff x = 0$, (definiteness)

iii) $(x, y) = \overline{(y, x)} \quad \forall x, \, y \in X$, (symmetry)

iv) $(\alpha x + \beta y, z) = \alpha(x, z) + \beta(y, z) \quad \forall x, \, y, \, z \in X, \forall \alpha, \, \beta \in \mathbb{F}$(linearity)

A linear space with an inner product $(X, (.,.))$ is called a *pre-Hilbert space*.

Theorem 6.4: Let $(X, (.,.))$ be a pre-Hilbert space. Then

$$\|x\| := \sqrt{(x, x)} \quad \forall x \in X$$

defines a norm in X.

Proof. Homework □

Definition 6.5: Two norms $\|x\|_a$, $\|x\|_b$ on a linear space X are called *equivalent*, if and only if any sequence converging with respect to $\|x\|_a$ also converges with respect to $\|x\|_b$ and vice versa.

Theorem 6.6: $\|.\|_a$, $\|.\|_b$ on the linear space X are equivalent

$$\Leftrightarrow \exists \alpha, \beta > 0 : \alpha \|x\|_a \leqslant \|x\|_b \leqslant \beta \|x\|_a \quad \forall x \in X \tag{6.1}$$

Idea of the proof.

"$\Leftarrow$": direct consequence of (6.1) applied to $x = y_n - y_\infty$ for a sequence $(y_n)_{n\in\mathbb{N}} \to y_\infty$ in either $\|.\|_a$, or $\|.\|_b$.

"$\Rightarrow$": Assume we can not find a γ such that $\|x\|_a < \gamma$ for all $x \in X$ with $\|x\|_b = 1$. Then there exists a sequence $(x_n)_{n\in\mathbb{N}}$ with $\|x_n\|_a \to \infty$ for $n \to \infty$ and $\|x_n\|_b = 1$ for all n. Now we define $y_n := \frac{x_n}{\|x_n\|_a}$, which in $\|.\|_b$ obviously converges to 0, but $\|x_n\|_a = 1$ and thus it does not converge in $\|.\|_a$, which contradicts our assumption.

Thus, we can find such $\gamma \in \mathbb{R}_{>0}$ and $\forall y \in X \backslash \{0\}$ we have

$$\|y\|_a = \left\| \|y\|_b \frac{y}{\|y\|_b} \right\|_a = \|y\|_b \left\| \frac{y}{\|y\|_b} \right\|_a \leqslant \|y\|_b \, \gamma$$

This proves the left inequality with $\alpha = \frac{1}{\gamma}$. The other half can be shown analogously.

□

As another direct consequence of equation (6.1) we get

Corollary 6.7: The limits of a sequence with respect to equivalent norms coincide.

Theorem 6.8: Let X be a finite dimensional linear space over $\mathbb{R}$, or $\mathbb{C}$. All norms on X are equivalent.

Proof. Literature □

6.2 Linear Operators, Operator and Matrix Norms

Definition 6.9: Let $(X, \|.\|_X), (Y, \|.\|_Y)$ normed linear spaces. An operator $A : X \to Y$ is called

i) *continuous in* $x \in X$, if for all sequences $(x_n)_{n\in\mathbb{N}}$ in X with $x_n \to x$ for $n \to \infty$ we have

$$Ax_n \to Ax \text{ for } n \to \infty$$

ii) *continuous*, if A is continuous in all $x \in X$.

iii) *linear* if it fulfills

$$A(\alpha x + \beta y) = \alpha Ax + \beta Ay$$

iv) bounded if A is linear and $\exists C \geqslant 0$, such that

$$\|Ax\|_Y \leqslant C \|x\|_X \quad \forall x \in X$$

Any C with this property are called *upper bound of* A.

The norms $\|.\|_X$, and $\|.\|_Y$ allow to measure distances in X and Y. We need similar norms to measure distances of matrices or linear operators mapping between them. The most important among those norms are the induced operator or matrix norms introduced in the following definition.

Definition 6.10: Let $A : X \to Y$ be a linear operator $(X, \|.\|_X), (Y, \|.\|_Y)$

normed linear spaces. The *operator norm* of A is defined as

$$\|A\| := \sup_{\|x\|_X=1} \|Ax\|_Y = \sup_{x\in X\setminus\{0\}} \frac{\|Ax\|_Y}{\|x\|_X}$$

$\|A\|$ is also called *induced operator norm.* In case A is a matrix, one also speaks of an *induced matrix norm*.

In Chapter 5 we saw that those norms compatible with a vector norm are of special importance. We can now define this precisely.

Definition 6.11: Let $(X, \|.\|_X), (Y, \|.\|_Y)$ normed linear spaces and denote the space of linear operators from X to Y by $\mathcal{L}(X, Y)$. A norm $\|.\|$ on $\mathcal{L}(X, Y)$ is called *consistent* with $\|.\|_X$ and $\|.\|_Y$, if for any $x \in X$ and $A \in \mathcal{L}(X, Y)$ we have $\|Ax\|_Y \leqslant \|A\| \, \|x\|_X$.

In case $Y = X$, i.e. $\|Ax\|_X \leqslant \|A\| \, \|x\|_X$, the norm $\|.\|$ is called *compatible* with $\|.\|_X$.

Remark 6.12:

- The norms in Definition 6.10 fulfill the consistency, and compatibility condition by definition.
- They are not the only norm that do so.

We have talked about upper bounds to the operator A in the sense of norms of images and preimages. The operator norm takes a distinguished position among those bounds.

Theorem 6.13: $\|A\|$ is the smallest upper bound of A and A is bounded if and only if $\|A\| < \infty$.

Proof. **"$\Rightarrow$":** Let A be bounded $\rightarrow \exists \infty > C \geqslant 0$ with

$$\|Ax\|_Y \leqslant C \quad \forall x \in X, \|x\|_X = 1$$

and

$$\|A\| = \sup_{\|x\|_X=1} \|Ax\|_Y \leqslant C < \infty.$$

Especially $\|A\| \leqslant C$ for all upper bounds C.

"⇐": Let A be linear with $\|A\| < \infty$. Then, for arbitrary $x \in X \backslash \{0\}$, we have

$$\begin{aligned}\|Ax\|_Y &= \left\| \|x\|_X \, A\left(\frac{x}{\|x\|_X}\right) \right\|_Y = \|x\|_X \left\| A\left(\frac{x}{\|x\|_X}\right) \right\|_Y \\ &\leqslant \|x\|_X \sup_{\|z\|_X = 1} \|Az\|_Y = \|x\|_X \, \|A\| \,.\end{aligned}$$

That means, A is bounded with upper bound $\|A\|$.

□

Matrices are a special type of linear operator. The linear operators, as part of the operators from one linear space to another, have some very distinct properties that we will collect next.

Theorem 6.14: Let $(X, \|.\|_X)$ and $(Y, \|.\|_Y)$ be normed linear spaces, and $A : X \to Y$ a linear operator.

The following are equivalent:

i) A is continuous in $x = 0$

ii) A is continuous

iii) A is bounded

Proof. **i)⇒ii):** Let $x \in X$, $(x_n)_{n\in\mathbb{N}} \subseteq X$ with $x_n \to x$, $n \to \infty$

$$\Rightarrow Ax_n \overset{A \text{ linear}}{=} A\underbrace{(x_n - x)}_{\overset{\|\cdot\|_X}{\to} 0,\, n\to\infty} + Ax \overset{\|.\|_Y}{\to} Ax \quad \text{for } n \to \infty$$

ii)⇒iii): We prove this part using a contradiction argument. Assume A continuous, but unbounded. Then there exists $(x_n)_{n\in\mathbb{N}} \subseteq X$ with $\|x_n\|_X = 1$ and $\|Ax_n\| \geqslant n$. Define:

$$y_n := \frac{x_n}{\|Ax_n\|_Y}.$$

Then we immediately get

$$\|y_n\|_X = \left\| \frac{x_n}{\|Ax_n\|_Y} \right\|_X = \frac{\|x_n\|_X}{\|Ax_n\|_Y} = \frac{1}{\|Ax_n\|_Y} \leqslant \frac{1}{n}$$

and thus

$$y_n \overset{\|.\|_X}{\longrightarrow} 0 \quad n \to \infty.$$

On the other hand,

$$\|Ay_n\|_Y = \left\| A \frac{x_n}{\|Ax_n\|_Y} \right\|_Y = \frac{\|Ax_n\|_Y}{\|Ax_n\|_Y} = 1$$

for all $n \in \mathbb{N}$, which contradicts continuity of A in $x = 0$.

iii)⇒i): Let A be bounded and $(x_n)_{n\in\mathbb{N}} \subseteq X$ with $x_n \overset{\|.\|_X}{\to} 0$ for $n \to \infty$. Then

$$\|Ax_n\|_Y \leqslant \|A\| \, \|x_n\|_X \to 0 \quad \text{as} \quad n \to \infty$$

and thus A continuous in $x = 0$.

□

An especially appealing feature of linear operators is that their properties are inherited to product operators, since these are established through simple concatenation of the application of the involved linear operators, as we can see from the following lemma.

Lemma 6.15 (Submultiplicativity)**:** Let $(X, \|.\|_X)$, $(Y, \|.\|_Y)$, $(Z, \|.\|_Z)$ be normed linear spaces.

$$\begin{aligned} A :& X \to Y \\ B :& Y \to Z \end{aligned}$$

bounded linear operators, then the operator concatenation

$$BA : X \to Z$$

is bounded with

$$\|BA\| \leqslant \|B\| \, \|A\| \, . \tag{6.2}$$

Proof. First we note that for any $x \in X$ due to boundedness of A and B we have

$$\|BAx\| \leqslant \|B\| \, \|Ax\|_Y \leqslant \|B\| \, \|A\| \, \|x\|_X$$

The lemma, thus, is a direct consequence of

$$\begin{aligned} \|BA\| &= \sup_{\|x\|_X=1} \|BAx\| \leqslant \sup_{\|x\|_X=1} \|B\| \, \|Ax\|_Y \\ &\leqslant \sup_{\|x\|_X=1} \|B\| \, \|A\| \, \|x\|_X = \|B\| \, \|A\| \end{aligned}$$

□

A bounded linear operator from one finite dimensional linear space into another one can always be expressed as a matrix. This is due to the fact that an evaluation of the operator on a basis immediately provides the matrix representation. We collect some notation to classify matrices.

Definition 6.16: Given

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \in \mathbb{R}^{n \times m},$$

i) the *transposed* matrix A^{T} is defined as

$$A^{\mathsf{T}} = \begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \in \mathbb{R}^{m \times n},$$

ii) If $A^{\mathsf{T}} = A$, then A is called *symmetric* $(n = m)$

iii) If $A^{\mathsf{T}} A = I$, then A is called *orthogonal* $(n \geqslant m)$

iv) If $A^{\mathsf{T}} A = A A^{\mathsf{T}}$, then A is called *normal* $(n = m)$

Definition 6.17:

i) Given

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \in \mathbb{C}^{n \times m},$$

the *conjugate transposed* matrix A^{H} is defined as

$$A^{\mathsf{H}} = \begin{bmatrix} \overline{a_{11}} & \cdots & \overline{a_{n1}} \\ \vdots & \ddots & \vdots \\ \overline{a_{1m}} & \cdots & \overline{a_{nm}} \end{bmatrix} \in \mathbb{C}^{m \times n},$$

ii) If $A^{\mathsf{H}} = A$, then A is called *hermitian* $(n = m)$

iii) If $A^{\mathsf{H}} A = I$, then A is called *unitary* $(n \geqslant m)$

iv) If $A^{\mathsf{H}} A = A A^{\mathsf{H}}$, then A is called *normal* $(n = m)$

Definition 6.18: Let $X = \mathbb{R}^n$, or $X = \mathbb{C}^n$. A matrix $A : X \to X$ is called

i) *upper triangular*, if $a_{ij} = 0 \; \forall i > j$,

ii) *lower triangular*, if $a_{ij} = 0 \; \forall i < j$,

iii) *diagonal*, if $a_{ij} = 0 \; \forall i \neq j$,

iv) *positive semidefinite* if $(Ax, x)_2 \geqslant 0 \; \forall x \in X$,

v) *positive definite* if $(Ax, x)_2 > 0 \; \forall x \in X \backslash \{0\}$,

vi) *negative (semi)definite* if $-A$ is positive (semi)definite.

Two linear systems of equations are called equivalent if and only if their sets of solutions coincide.

Lemma 6.19: Let $P \in \mathbb{C}^{n \times n}$ be invertible and $A \in \mathbb{C}^{n \times n}$, then the linear systems of equations $Ax = y$ and $PAx = Py$ for $x, y \in \mathbb{C}^n$ are equivalent.

Proof.

$$\begin{aligned} P \text{ is invertible} &\Rightarrow \text{“}Px = 0 \iff x = 0\text{”} \\ &\Rightarrow \text{“}P(Ax - y) = 0 \iff Ax - y = 0\text{”} \end{aligned}$$

□

Lemma 6.20: The linear system $Ax = b$ permits a solution if and only if $\operatorname{rank}(A) = \operatorname{rank}([A \; b])$

Proof. Homework □

Some structural properties of matrices are preserved in products of matrices. This is often exploited to generate structure preserving algorithms or limit error amplification. The following two Lemmas collect such properties and will be proved in the exercises.

Lemma 6.21: Products of lower (upper) triangular matrices are lower (upper) triangular.

Lemma 6.22: Products of orthogonal matrices are orthogonal matrices.

Some matrix norm examples:

i) $\|A\| := \max_{i,j} |a_{ij}|$ (induced by the pair $(\|.\|_1, \|.\|_\infty)$ of norms, not sub-multiplicative)

ii) $\|A\|_F := \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} \left|a_{ij}^2\right|}$ (not induced, compatible with the vector $\|.\|_2$-norm)

iii) $\|A\|_1 := \max_{j=1,\dots,n} \sum_{i=1}^{n} |a_{ij}|$ (induced, column sum norm)

iv) $\|A\|_\infty := \max_{i=1,\dots,n} \sum_{j=1}^{n} |a_{ij}|$ (induced, row sum norm)

v) $\|A\|_2 := \sup_{\|x\|_2=1} \|Ax\|_2$ (induced, spectral norm)

Theorem 6.23: Any matrix $A \in \mathbb{C}^{n\times n}$ is bounded in every matrix norm.

Proof. Homework □

6.2.1 Spectral Norm and Spectral Radius

A complex number $\lambda \in \mathbb{C}$ is called *eigenvalue* of a matrix A if $\exists x \neq 0$

$$Ax = \lambda x$$

Then x is called *(right) eigenvector* of A. The set of all eigenvalues is $\Lambda(A) := \{\lambda \in \mathbb{C} : Ax = \lambda x\}$, it is called *spectrum* of A. The value $\rho(A) = \max\{|\lambda| : \lambda \in \Lambda(A)\}$ is called the *spectral radius* of A.

Remark 6.24: In the following A^* denotes either A^{H} when A is complex or A^{T} when it is real.

Theorem 6.25 (Schur decomposition)**:** Let $A \in \mathbb{C}^{n\times n}$ $(\mathbb{R}^{n\times n})$. There exists a unitary (orthogonal) matrix $U \in \mathbb{C}^{n\times n}$ $(\mathbb{R}^{n\times n})$ such that

$$T = U^*AU$$

is a (quasi) upper triangular matrix.

Proof. Homework. □

Remark 6.26:

- $\Lambda(A) = \{t_{ii} : i = 1, \dots, n\}$ $(A \in \mathbb{C}^{n\times n})$, where t_{ii} are the diagonal entries in T from Theorem 6.25.
- The Schur decomposition can be computed in a QR-algorithm in $\mathcal{O}(n^3)$ floating point operations.

Corollary 6.27: Let $A \in \mathbb{C}^{n\times n}$ $(\mathbb{R}^{n\times n})$ hermitian (symmetric). There exists a unitary (orthogonal) matrix $U \in \mathbb{C}^{n\times n}$ $(\mathbb{R}^{n\times n})$ such that

$$\begin{bmatrix} \diagdown \end{bmatrix} = \mathrm{diag}(\lambda_1, \dots, \lambda_n) = U^* A U$$

Here λ_i $(i = 1, \dots, n)$ is the i-th eigenvalue of A with the i-th column of U the corresponding eigenvector.

Theorem 6.28: The $\|.\|_2$ operator norm of A is called spectral norm since we have:

i) $\|A\|_2 = \sqrt{\rho(A^*A)}$

ii) $\rho(A) \leqslant \|A\|$ for an arbitrary induced norm $\|.\|$

iii) $A = A^* \Rightarrow \rho(A) = \|A\|_2$

Proof. i) $(A^*A) = (A^*A)^*$ thus Corollary 6.27 tells us that there exists an orthogonal matrix U with

$$U^*A^*AU = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}$$

Further, for all $x \in \mathbb{C}^n$ we find coefficients α_i, $(i = 1, \dots, n)$, such that

$$x = \sum_{i=1}^{n} \alpha_i u_i$$

Thus,

$$A^*Ax = \sum_{i=1}^{n} \lambda_i \alpha_i u_i,$$

and therefore

$$\begin{aligned}
\|Ax\|_2^2 &= (Ax, Ax)_2 = (x, A^*Ax)_2 \\
&= \left(\sum \alpha_i u_i, \sum \lambda_i \alpha_i u_i\right)_2 \\
&= \sum (\alpha_i u_i, \lambda_i \alpha_i u_i)_2 \\
&= \sum \lambda_i |\alpha_i|^2 (u_i, u_i)_2 \\
&= \sum \lambda_i |\alpha_i|^2 \, \|u\|_2^2 \\
&= \sum \lambda_i |\alpha_i|^2 \\
&\leqslant \rho(A^*A) \sum |\alpha_i|^2 \\
&= \rho(A^*A) \, \|x\|_2^2 \, ,
\end{aligned}$$

such that

$$\frac{\|Ax\|_2}{\|x\|_2} \leqslant \rho(A^*A)$$

and $\lambda_i \geqslant 0 \forall i$. Now let $\lambda_{i_0} = \rho(A^*A)$, and u_{i_0} the corresponding eigenvector, then

$$\frac{\|Au_{i_0}\|_2^2}{\|u_{i_0}\|_2^2} = \frac{\lambda_{i_0} \, \|u_{i_0}\|_2^2}{\|u_{i_0}\|_2^2} = \lambda_{i_0} = \rho(A^*A).$$

So we have proved the first statement.

ii) By definition of the induced norm we have for each pair of eigenvalue λ and corresponding eigenvector u that

$$\|A\| = \sup_{\|x\|=1} \|Ax\| \geqslant \|Au\| = \|\lambda u\| = |\lambda| \, \|u\| = |\lambda|,$$

and therefore $\rho(A) \leqslant \|A\|$.

iii) $A^* = A$:

$$\|A\|_2 = \sqrt{\rho(A^*A)} = \sqrt{\rho(A^2)} = \sqrt{\rho(A)^2} = \rho(A)$$

□

In fact the last statement is true also for normal matrices. The proof is slightly more technical, though, since it requires the full eigendecomposition of A and the knowledge that for normal matrices the left and right eigenbases coincide.

6.2.2 Condition Number and Singular Values

Recall:

$$c_{\text{rel}}(f,x) \leqslant \frac{\|x\|}{\|f(x)\|} \cdot \left\|f'(x)\right\|$$

Now let $f \equiv A$ and A invertible $\Rightarrow$

$$y = Ax \Leftrightarrow x = A^{-1}y$$

$$\Rightarrow \frac{\|x\|}{\|f(x)\|} = \frac{\|x\|}{\|Ax\|} = \frac{\left\|A^{-1}y\right\|}{\|y\|} \leqslant \sup_{y \neq 0} \frac{\left\|A^{-1}y\right\|}{\|y\|} = \left\|A^{-1}\right\|.$$

Since the Jacobian of a linear operator is the linear operator, we have

$$f'(x) = A\big|_x.$$

Such that we find

$$c_{\text{rel}}(A,x) \leqslant \|A\| \left\|A^{-1}\right\|.$$

In case $A = I$ we further have

$$c_{\text{rel}} = \frac{\|x\|}{\|x\|} \|I\| = 1 = \|I\| \left\|I^{-1}\right\|,$$

which proves that the bound is indeed sharp. This motivates the following definition.

Definition 6.29: Let $A \in \mathbb{C}^{n \times n}$ and $\|.\|_a$ an induced operator norm

$$\kappa_a(A) := \|A\|_a \left\|A^{-1}\right\|_a$$

denotes the *a-condition number* of A.

Lemma 6.30: For any induced operator norm $\|.\|_a$ it holds

$$\kappa_a(A) \geqslant \kappa_a(I) = 1$$

Proof.

$$\begin{aligned}\kappa_a(I) &= \|I\|_a \left\|I^{-1}\right\|_a = 1 = \|I\|_a = \left\|AA^{-1}\right\|_a \\ &\overset{\text{Lemma 6.15}}{\leqslant} \|A\|_a \left\|A^{-1}\right\|_a = \kappa_a(A)\end{aligned}$$

□

In the following we will, for ease of notation, leave out the index a if a property holds for all possible values of a.

Theorem 6.31: Let $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$. Let x be the exact solution of $Ax = b$ and $x + \Delta x$ the exact solution of the perturbed $A(x + \Delta x) = b + \Delta b$. Then

$$\frac{\|\Delta x\|}{\|x\|} \leqslant \kappa(A)\frac{\|\Delta b\|}{\|b\|}.$$

Theorem 6.32: Let $Ax = b$, as in Theorem 6.31. Moreover define the error $e_k := A^{-1}b - x_k$, and the residual $r_k := b - Ax_k$ in step k of an iterative solver for $Ax = b$. It holds:

$$\frac{1}{\kappa(A)}\frac{\|r_k\|}{\|r_0\|} \leqslant \frac{\|e_k\|}{\|e_0\|} \leqslant \kappa(A)\frac{\|r_k\|}{\|r_0\|} \leqslant \kappa(A)^2\frac{\|e_k\|}{\|e_0\|}. \tag{6.3}$$

Proof. Note

$$\|r_k\| = \|b - Ax_k\| = \left\|A(A^{-1}b - x_k)\right\| = \|Ae_k\| \leqslant \|A\|\,\|e_k\|$$

and analogously

$$\|e_k\| = \left\|A^{-1}b - x_k\right\| \leqslant \left\|A^{-1}\right\|\,\|r_k\|$$

Thus

$$\frac{1}{\kappa(A)}\frac{\|r_k\|}{\|r_0\|} = \frac{1}{\|A\|\,\|A^{-1}\|}\frac{\|r_k\|}{\|r_0\|} \leqslant \frac{1}{\|A\|}\frac{\|r_k\|}{\|A^{-1}r_0\|} = \frac{1}{\|A\|}\frac{\|Ae_k\|}{\|e_0\|} \leqslant \frac{\|e_k\|}{\|e_0\|}.$$

This proves the leftmost inequality in (6.3). The others can be shown similarly. □

6.2.3 Some Remarks on $\kappa_2(A)$

Theorem 6.33: Let $A \in \mathbb{R}^{n\times n}$. There exist orthogonal matrices $U, V \in \mathbb{R}^{n\times n}$ such that

$$U^\mathsf{T} A V = \begin{pmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_n \end{pmatrix} \tag{6.4}$$

where $0 \leqslant \sigma_n \leqslant \cdots \leqslant \sigma_1$. For $i = 1, \ldots, n$ we further have

$$\det(A^\mathsf{T} A - \sigma_i^2 I) = 0 \tag{6.5}$$

i.e. $\sigma_i^2 = \lambda_i$ with $\lambda_i \in \Lambda(A^\mathsf{T} A)$.

Proof. $A^\mathsf{T} A$ is symmetric and positive semidefinite, so there exists $V \in \mathbb{R}^{n\times n}$, such that

$$V^\mathsf{T} A^\mathsf{T} AV = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$$

where $\lambda_1 \geqslant \cdots \geqslant \lambda_n \geqslant 0$. Thus $\sigma_i = \sqrt{\lambda_i}$ is well defined in Theorem 6.33 and (6.5) follows from Corollary 6.27. For (6.4) we define $U = AVD^{-1}$, where $D = \operatorname{diag}(\sigma_1, \dots, \sigma_n)$. Since we have

$$U^\mathsf{T} U = D^{-T} V^\mathsf{T} A^\mathsf{T} AVD^{-1} = D^{-1} \operatorname{diag}(\lambda_1, \dots, \lambda_n)\, D^{-1} = I$$

U is ortogonal and

$$U^\mathsf{T} AV = D^{-T} V^\mathsf{T} A^\mathsf{T} AV = D^{-1} \operatorname{diag}(\lambda_i) = D$$

$\square$

In addition for invertible A we have $\sigma_n > 0$ and $\lambda_n > 0$.

Definition 6.34: The σ_i in Theorem 6.33 are called *singular values* of A. The corresponding columns in U, V are called the i-th left/right *singular vectors*.

Now from

$$\begin{aligned}
\sup_{x\neq 0} \frac{\|Ax\|_2^2}{\|x\|_2^2} &= \sup_{x\neq 0} \frac{(Ax, Ax)_2}{(x,x)_2} = \sup_{x\neq 0} \frac{x^\mathsf{T} A^\mathsf{T} Ax}{x^\mathsf{T} x} \\
&\overset{V \text{ reg.}}{=} \sup_{Vx\neq 0} \frac{x^\mathsf{T} V^\mathsf{T} A^\mathsf{T} AVx}{x^\mathsf{T} V^\mathsf{T} Vx} \\
&\overset{U,V \text{ orth.}}{=} \sup_{x\neq 0} \frac{x^\mathsf{T} V^\mathsf{T} A^\mathsf{T} UU^\mathsf{T} AVx}{x^\mathsf{T} x} = \sup_{x\neq 0} \frac{x^\mathsf{T} D^\mathsf{T} Dx}{x^\mathsf{T} x} = \sigma_1^2,
\end{aligned}$$

we analogously find for the infimum

$$\inf_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sigma_n.$$

Further we have

$$U^\mathsf{T} AV = \operatorname{diag}(\sigma_1, \dots, \sigma_n)\,,$$

and

$$V^\mathsf{T} A^{-1} U = \operatorname{diag}\left(\frac{1}{\sigma_1}, \dots, \frac{1}{\sigma_n}\right)$$

and thus $\|A\|_2 = \sigma_1$ and $\left\|A^{-1}\right\|_2 = \frac{1}{\sigma_n}$, which proves the following Corollary.

Corollary 6.35: Let $A \in \mathbb{R}^{n\times n}$ invertible, σ_1, σ_n its largest and smallest singular values, then we have

$$\kappa_2(A) = \frac{\sigma_1}{\sigma_n}$$

If A is in addition normal and λ_1 and λ_n are its largest and smallest magnitude eigenvalues, then we also have

$$\kappa_2(A) = \frac{|\lambda_1|}{|\lambda_n|}$$

Here the second part uses the fact that $A \in \mathbb{C}^{n\times n}$ normal guarantees that $\exists U \in \mathbb{C}^{n\times n}$ unitary, such that U^*AU is diagonal (compare, e.g., [3, Corollary 7.1.4]).

Definition 6.36: (compare Theorem 6.6)
$\|.\|_a, \|.\|_b$ vector norms on $\mathbb{R}^n$. The condition numbers κ_a, κ_b are called *equivalent* if one can find $\alpha, \beta > 0$ such that

$$\alpha\kappa_a(A) \leqslant \kappa_b(A) \leqslant \beta\kappa_a(A) \qquad \forall A \in \mathbb{R}^{n\times n} \text{ invertible}$$

The equivalence constants α and β coincide with the constants α, β in Theorem 6.6.

6.3 Matrix Storage Formats

In this section we will introduce different ways of storing matrices in C data structures. Depending on the type of matrix, judged by the number of non-zero entries, we apply different techniques. The varying suggested storage schemes will be demonstrated using the example matrix

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 5 & 0 & 6 \\ 0 & 0 & 7 & 0 \end{bmatrix}.$$

6.3.1 Dense Matrices

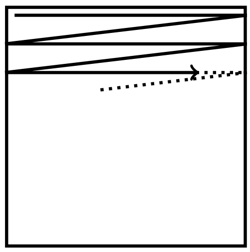
Definition 6.37: A matrix is called *dense*, or *densely populated* if essentially all its entries are non-zero.

Dense matrices should be stored as some storage type that resembles a 2d array.

2d Arrays in C We have seen this in Chapter 3. In principle for the C programming language two definitions of 2d arrays are available:

- **`double A[5][10]`** (static array),
- **`double **A + malloc()`** (dynamic array).

Both versions result in A being a 2d array. In both cases it is stored "row major", i.e., the order of elements follows the model:

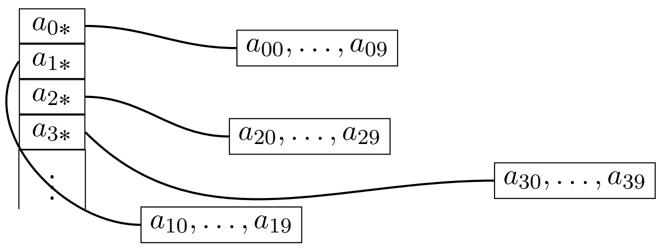


Differences of Static and Dynamic 2d Array in C

i) A static array in C is essentially one big row vector:
   `double A[5][10]`

$a_{00}, \ldots, a_{09}$	$a_{10}, \ldots, a_{19}$	$a_{20}, \ldots, a_{29}$	$a_{30}, \ldots, a_{39}$	$a_{40}, \ldots$

ii) For a dynamic array the rows may be stored somewhere (possibly) not consecutively arranged
   `double **A;`

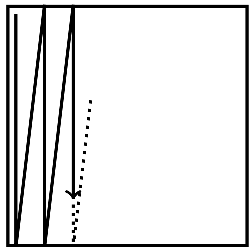


Option i) is only usable when size is known a priori.

Option ii) is more flexible, but destroys data locality. An advantage of this format, however, is easy swapping of rows, since no data needs to be copied, but only pointers are rearranged.

2d Arrays in Fortran Section 6.4 introduces basic mathematical / linear algebra operations based on Fortran 77/90 implementations.

Static Fortran arrays (all arrays in Fortran 77) are stored “column major”, i.e.,



$a_{00},\dots,a_{n0}$	$a_{01},\dots,a_{n1}$	$a_{02},\dots,a_{n2}$	$a_{03},\dots,a_{n3}$	$a_{04},\dots$

This behavior can be implemented as a 1d array with index transformation in C, as well. To this end we introduce an important expression that will play an even more important role in Section 6.4, again.

Definition 6.38: The distance, counted in the number of elements, between the beginnings of 2 subsequent columns in a 2d array is called the *leading dimension* (LD) of the array.

$$\Rightarrow a_{kl} \hat{=} A[l \cdot LD + k]$$

In Fortran 77 this behavior is already part of the language definition. The expression $A(LD,:)$ does this mapping automatically.

Advantages:

- Data locality is enforced also for dynamic arrays since the single row/column pointers can no longer be scattered around the main memory.
- More importantly, the array is now stored in Fortran 77 compliant column major format and can thus be passed directly to (optimized) Fortran libraries.

Basic Object Oriented Design Although C does not directly support object oriented programming, structures and functions on structures can be used to mimic the object oriented behavior and increase code efficiency.

```
struct my_matrix_st{
   INT cols;
   INT rows;
   INT LD;

   double *values;

   char structure;
```

```
};
```

Thereby, `INT` can either be `int` or `long` depending on the application and, if Fortran libraries are used, the default integer size in Fortran. Typically, this is realized by a preprocessor `define`. To use 32-bit integers, being the default in Fortran, we use

```
#define INT int;
```

If 64-bit integers are required we set **`INT`** using

```
#define INT long;
```

Remark 6.39: If a double precision matrix needs more than 16GB memory and Fortran libraries, like BLAS and LAPACK, should be used to operate on this matrix, it is mandatory to use 64 bit integers in both C and Fortran.

Note that due to the non-existence of Fortran unsigned integer types we also use the signed types **`int`** and **`long`** in C to avoid conflicts when passing data to Fortran routines.

The matrix A would thus lead to **`A.cols`**$=4$, **`A.rows`**$=4$, **`A.LD`**$=4$ and

`A.values`=	1	0	0	0	2	3	5	0	0	4	0	7	0	0	6	0

The **`structure`** entry in this case could be **`NULL`** to indicate, that the matrix is not specially structured. In order to better understand the value of the leading dimension concept, consider we want to manipulate the 2×2 sub-matrix starting in the (2,2)-position in A, i.e., the matrix

$$B = \begin{bmatrix} 3 & 4 \\ 5 & 0 \end{bmatrix}.$$

The corresponding values would then be **`B.cols`**$=2$, **`B.rows`**$=2$, **`B.LD`**$=4$, again **`B.structure`**=**`NULL`** and the **`B.values`** pointer would be set to the **`A.values[5]`**. This way we know that in **`B.values`** the entry with value 4 is 4 (**`B.LD`**) positions ahead of the one where the 3 is stored.

Tiled Matrix Storage In Chapter 7, we will see that it is essential to work on small tiles covering the matrix to get optimal performance for several operations. Therefore, if the matrix is large it becomes mandatory to store it in this style already. This fact will become even more important when multi– or many-core considerations come into play. In order to have an easy access to the tiles and ensure that the tiles are small matices that can be handled using existing software, we adapt the above storage structure a bit:

```
struct tiled_matrix_st{
    INT cols;
    INT rows;

    INT Tcols;
    INT Trows;
    INT Tsize;

    double *tiles;

    char structure;
};
```

Let us revisit the example matrix A. As before we have **`A.cols`**$= 4$, **`A.rows`**$= 4$. Consider the case of a tile size **`A.Tsize`**$= 2$, that means, also, **`A.Tcols`**$= 2$, **`A.Trows`**$= 2$, and

$$\texttt{A.tiles}= \boxed{\begin{smallmatrix}1&2\\0&3\end{smallmatrix}}\boxed{\begin{smallmatrix}0&5\\0&0\end{smallmatrix}}\boxed{\begin{smallmatrix}0&0\\4&0\end{smallmatrix}}\boxed{\begin{smallmatrix}0&6\\7&0\end{smallmatrix}}$$

Note that this is a very simplified representation of **`A.tiles`**, that is supposed to illustrate the priciple. The actual list is a static or dynamic array of adresses of small tile matrices. For the single tiles the same considerations as for the 2d array above hold. Again a Fortran style storage should be preferred if high performance libraries in Fortran are aimed.

6.3.2 Sparse Matrices

Definition 6.40: We call a matrix $A \in \mathbb{R}^{n\times n}$ or $A \in \mathbb{C}^{n\times n}$ *sparse* if only a few entries of A per row or column are non-zero, in average.

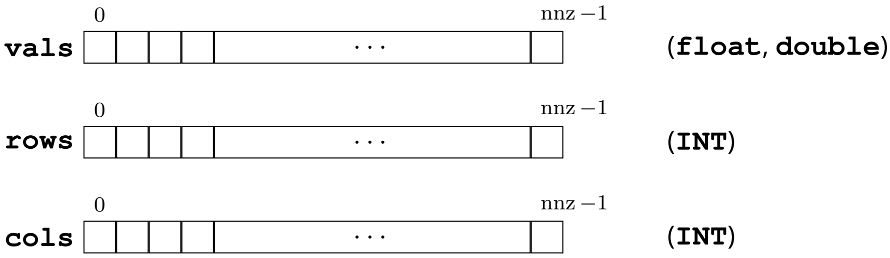
Precisely, we want A to be such that storing A uses $\mathcal{O}(n)$ storage and multiplication with A is performed in $\mathcal{O}(n)$ effort.

Both conditions boil down to the number of non-zero entries in A ($\mathrm{nnz}(A)$) being $\mathcal{O}(n)$. Several formats for storing sparse matrices exist. Some important ones are introduced below. They all follow the same fundamental principle.

Basic idea: In order to save memory we store "only" the non-zero entries and neglect the zeros.

Coordinate Storage (COO)

Stores A in 3 vectors of length $\mathrm{nnz}(A)$ for entry values, row indices, and column indices:



Advantages:

- easy to implement
- easy addition of new entries
- easy elementwise access

Drawbacks:

- non local memory access
- (atomic access to output vector in threaded implementation)

Note that the format does not prescribe any ordering of the entries, i.e., the storage for the matrix A might look like

vals	1	7	2	3	4	5	6
rows	0	3	0	1	1	2	2
cols	0	2	1	1	2	1	3

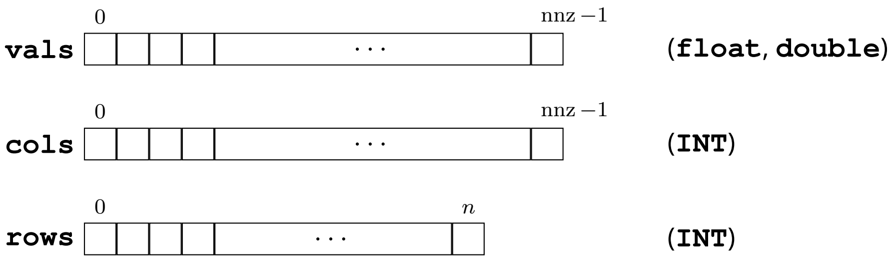
which is using C indexing starting at 0 to avoid index shifts in, e.g., matrix vector product implementations, where the indices in the vector are C, i.e., zero based.

Remark 6.41: The coordinate storage format is, for example, the basis of the sparse matrix version of the Matrix Market[a] file exchange format.

[a] https://math.nist.gov/MatrixMarket/

Compressed Sparse Row Storage (CSR/CRS)

As above the format uses three vectors to store the data. Two vectors **vals** and **cols** store the entry values and column indices. The third vector holding the row indices (**rows**) stores, where the corresponding row starts in the vectors **vals** and **cols**. Additionally, the last entry stores the number of non-zero entries $\mathrm{nnz}(A)$. Not that, since the start of the first row is evident, the first entry is actually not needed, but it simplifies implementations as discussed below.



Advantages:

- optimal storage requirements
- can exploit BLAS (Section 6.4) in per row operations
- allows multithreading

Drawbacks:

- non local memory access due to indirect indexing
- (load balancing problem in threaded implementations due to different row lengths)

Remark 6.42: An equivalent format swapping the roles of row and column pointers in the above, is used, e.g., in MATLAB. It is called *compressed sparse column storage* (CSC/CCS).

As noted above, the first entry in the **rows** pointer actually contains redundant information, since it is clear that the corresponding row starts at the first position in both other arrays. However, most implementations still use the version including the redundant value since then loops running over all entries in a row can simply be written as something like

```
for ( j = rowptr [i] ; j < rowptr[i+1]; j++) {...}
```

and the first and last rows do not need any special treatment.

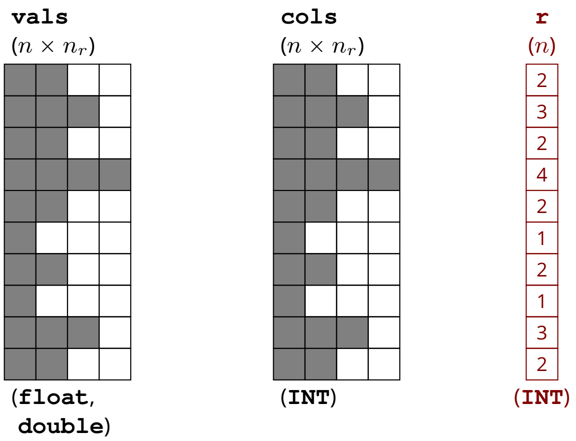
The matrix A in CSR format looks as follows:

vals	1	2	3	4	5	6	7
cols	0	1	1	2	1	3	2
rows	0	2	4	6	7		

Here again we have used zero based indexing of columns as usual in C to avoid index shifts.

Ellpack and Ellpack-R (ELLR)

This describes a format that was introduced as storage format specially tailored for vector computers. The main idea was to automatically balance the workload and exploit data parallelism[1]. Let n_r be the maximum row length. Ellpack stores two 2d-arrays **vals** and **cols** with size $n \times n_r$. The Ellpack-R (ELLR) format adds an additional vector storing the actual lengths of the single rows in order to avoid processing of zero elements.



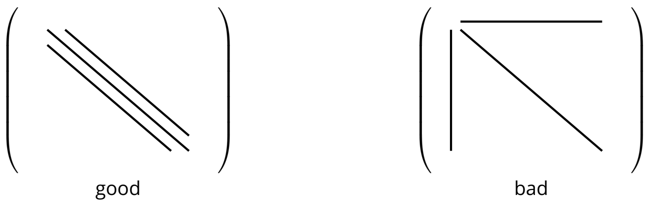
Advantages:

- constant per row length $\rightsquigarrow$ good load balancing properties
- (coalesced memory access (threads k, k+1 access consecutive memory cells))
- (no synchronization required)

Drawbacks:

- The storage requirement is dominated by the longest row. $\Rightarrow$ Possibly, many zeros are stored.

[1] Details will be introduced in Term 2.



- The zeros are actually processed without leading to new information.

Advantage of the ELLR:

- The unnecessary processing of zeros is avoided.

Drawback of the ELLR:

- Additional n integers for storing of the row lengths are required.
- Load balancing features of the Ellpack format are no longer valid.

Also here we present the matrix A in the form of the stored data for this format. As in the examples above we again use the C/zero based indexing in the **`cols`** array to avoid index shifts in loops using this matrix together with vectors implemented as 1d arrays.

`vals` (4×2)

1	2
3	4
5	6
7	0

`cols` (4×2)

0	1
1	2
1	3
2	0

`r` (4)

2
2
2
1

Remark 6.43: In the NVIDIA® CUDA® toolkit for acceleration of codes using NVIDIA® graphics adapters, or more precisely in the corresponding cusparse library used for working with sparse matrices, a hybrid matrix storage format is used. This format is using Ellpack for the short rows, i.e., rows with only few non-zero entries. The exceptionally long rows that are causing the storage problems in both Ellpack and ELLR, are stored and treated separately.

6.3.3 Complex Matrices

In the above sections we have focused on the storage schemes for real matrices. In the dense case, the structure for a possibly complex matrix could simply be extended by a second **`double`** pointer **`ivals`** for storing the imaginary parts and a second **`char`** that indicates whether the matrix is real or complex, i.e., whether `ivals` contains useful values or is simply `NULL`. Since the information in the additional `char` is in principle redundant, this could also be hidden in the `structure` field by using clever preprocessor defines indicating the different structures in addition with the information whether they are real or complex. However, this version breaks the property of the `values` field to be directly passable to the Fortran call. Therefore, in the dense complex case, the `vals` array should be twice as large and real and imaginary parts of each entry should be stored next to each other. This can, e.g., be achieved by using the `double complex`, or `float complex` types from `complex.h` (in the C99 standard). These are compatible with the Fortran types `COMPLEX*16` and `COMPLEX*8`=`COMPLEX`.

Similarly, for sparse matrices the `vals` field gets a twin `ivals`. Also, similar to the above, special structures together with the indication of real or complex data storage can be handled by an additional information member like `structure`.

6.4 Linear Algebra Software

One of the most basic tasks in most applications in scientific computing is the necessity to provide a basic set of routines dealing with the linear algebra subtasks. Due to the foresight of a couple of developers in the mid 1970s this is a rather easy task, as long as the involved linear operators can be represented as dense matrices. Then, the related functions and solutions are usually well approximated by simple vectors in $\mathbb{R}^n$, or $\mathbb{C}^n$. The basic operations that are required in this case have been grouped in three classes, the so-called levels, in the basic linear algebra subroutines (BLAS) library introduced in Section 6.4.1. Those levels are

- basic vector operations,
- matrix-vector operations,
- and matrix-matrix operations.

Each of the levels is described in a separate paragraph below. The BLAS library only contains the most basic operations like products and weighted sums. The application of those operations in more complex tasks, like linear system solves, eigenvalue computation, matrix factorizations and similar calculations, is implemented in a set of routines gathered in the linear algebra package (LAPACK). We will briefly sketch its content in Section 6.4.2. There exist several implementations of these two libraries. The main reference implementation is hosted at `https://www.netlib.org`. It provides source codes for both libraries that can be compiled on basically any machine. Hardware manufacturers have started early to provide their own implementations. The most well known one today is probably the Intel® Math Kernel Library[2] (MKL) that contains optimized versions of both libraries. AMD provides support in their AMD Optimizing CPU Libraries[3] (AOCL).

6.4.1 Basic Linear Algebra Subroutines (BLAS)

The basic linear algebra subroutines BLAS are sub-divided into three classes, called levels, that are mainly standing for the involved memory and computation complexities, but also for their historic development.

- Level 1 described in [4]: $\mathcal{O}(n)$ operation on $\mathcal{O}(n)$ data
- Level 2 described in [2]: $\mathcal{O}(n^2)$ operations on $\mathcal{O}(n^2)$ data
- Level 3 described in [1]: $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ data

While, as mentioned above, the reference implementation is available in the online library for numerical software at `https://www.netlib.org/blas`, vendor versions are available from major hardware manufacturers:

- Intel® Math Kernel Library (MKL)
- AMD BLIS (as part of AOCL)
- Apple Accelerate framework
- IBM Engineering and Scientific Subroutines Library (ESSL)
- ...

BLAS has a Fortran induced naming scheme: (Level 1)[4]

$$\underbrace{\texttt{cblas\_}}_{\text{prefix}} \quad \underbrace{\text{X}}_{\text{datatype}} \quad \underbrace{\text{XXXX}}_{\text{operation}}$$

Data types (allowed specifiers)

- **s** — single precision real
- **c** — single precision complex
- **d** — double precision real
- **z** — double precision complex

[2] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`
[3] `https://developer.amd.com/amd-aocl/`
[4] We base our presentation on the prefix used, e.g., in the Apple Accelerate framework.

Operations (examples)

- **`axpy`** $y \leftarrow \alpha x + y$
- **`dot`** $r \leftarrow x^\mathsf{T} y$
- **`nrm2`** $r \leftarrow \|x\|_2 = \sqrt{x^\mathsf{T} x}$
- **`asum`** $r \leftarrow \|x\|_1 = \sum\limits_i |x_i|$

Example 6.44: `cblas_daxpy` double precision real version of $y \leftarrow ax + y$ in the C wrapped format.

The prefix is usually only needed in C versions. It is empty for calling the F77 versions (compare also Section 3.11).

Levels 2 and 3 additionally respect/exploit matrix structures and indicate them in the correspndign function names:

$$\underbrace{\texttt{cblas\_}}_{\text{prefix}} \quad \underbrace{\text{X}}_{\text{datatype}} \quad \underbrace{\text{XX}}_{\text{structure}} \quad \underbrace{\text{XXX}}_{\text{operations}}$$

Possible values for the structure placeholder are:

GE	general	**GB**	general banded		
SY	symmetric	**SB**	symmetric banded	**SP**	symmetric packed
HE	hermitian	**HB**	hermitian banded	**HP**	hermitian packed
TR	triangular	**TB**	triangular banded	**TP**	triangular packed

Typical arguments For triangular matrix operations the type of triangular structure is controlled by the argument **`UPLO`**. It is taking character values '**`L`**', '**`U`**' for lower or upper triangular, respectively.

The operand order (e.g., decision about left or right multiplication) is steered by the **`SIDE`** arguments '**`L`**' or '**`R`**'.

For triangular matrices the **`DIAG`** argument specifies whether they have a unit diagonal '**`U`**' or not '**`N`**'.

Transposition is decided via **`TRANS`** argument taking either of the following values:

'**`N`**'	non transposed	X
'**`T`**'	transposed	X^T
'**`C`**'	conjugate transposed	X^H

Remark 6.45: Note that '`H`' is not defined by the standard and not understood by the general implementations. Although some implementations may support it, it should therefore never be used.

As two examples, we report on the double precision and double precision complex matrix-matrix-product routines that perform the operation

$$C \leftarrow \alpha \mathrm{op}(A) \cdot \mathrm{op}(B) + \beta C,$$

where $\mathrm{op}(.)$ refers to the transposition types above. The Fortran interfaces and data types are

```
SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
!.. Scalar Arguments ..
REAL*8 ALPHA,BETA
INTEGER K,LDA,LDB,LDC,M,N
CHARACTER TRANSA,TRANSB

!.. Array Arguments ..
REAL*8 A(LDA,*),B(LDB,*),C(LDC,*)
```

for the real case and

```
SUBROUTINE ZGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
!.. Scalar Arguments ..
COMPLEX*16 ALPHA,BETA
INTEGER K,LDA,LDB,LDC,M,N
CHARACTER TRANSA,TRANSB

!.. Array Arguments ..
COMPLEX*16 A(LDA,*),B(LDB,*),C(LDC,*)
```

for the complex one. Thus, the corresponding C prototypes look like

```
void dgemm_(char *transa, char *transb, int *m, int *n, int
    *k,
     double *alpha, double *A, int *lda,
     double *B, int *ldb,
     double *beta, double *C, int *ldc);
```

for the real and

```
void zgemm_(char *transa, char *transb, int *m, int *n, int
    *k,
     double complex *alpha, double complex *A, int *lda,
     double complex *B, int *ldb,
     double complex *beta, double complex *C, int *ldc);
```

for the complex case.

Vector Operations (BLAS **Level 1)**

- scaling and addition: αx, $\alpha x + y$,
- inner products: $x^* y$,
- norm expressions: $\|x\|_2$, $\|x\|_1$,$\|x\|_\infty$.

Matrix-Vector Operations (BLAS **Level 2)**

Let $\mathbb{F} \in \{\mathbb{C}, \mathbb{R}\}$, $\alpha, \beta \in \mathbb{F}$, $A \in \mathbb{F}^{m \times n}$, $x, y \in \mathbb{F}^n$:

- scaling and addition: $\alpha Ax + \beta y$, $\alpha A^* x + \beta y$,
- rank-1/2 updates: $A + \alpha xy^*, A + \alpha xx^*$, $A + \alpha xy^* + \beta yx^*$,
- triangular solves: $\alpha T^{-1} x$, $\alpha T^{-*} x$, T triangular.

Matrix-Matrix Operations (BLAS **Level 3)**

- $\alpha AB + \beta C$, $\alpha AB^* + \beta C$, $\alpha A^* B^* + \beta C$,
- rank k updates: $\alpha AA^* + \beta C$, $\alpha A^* A + \beta C$
- rank $2k$ updates: $\alpha A^* B + \alpha B^* A + \beta C$
- triangular multi-solves: $\alpha T^{-1} C$, $\alpha T^{-*} C$, T triangular.

Idea Behind the Level 3 Performance Gain The performance of Level 3 operations increases by block sub-structuring the operations. The special case $C \leftarrow C + AB^\mathsf{T}$ of the above **GEMM** operation, evaluated in a simple 2×2 block structured form becomes

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} + \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} \begin{bmatrix} B_{11}^\mathsf{T} & B_{21}^\mathsf{T} \end{bmatrix},$$

which allows to compute the single blocks in the result as:

$$\begin{aligned} C_{11} &\leftarrow C_{11} + A_{11} B_{11}^\mathsf{T}, & C_{12} &\leftarrow C_{12} + A_{11} B_{21}^\mathsf{T}, \\ C_{21} &\leftarrow C_{21} + A_{21} B_{11}^\mathsf{T}, & C_{22} &\leftarrow C_{22} + A_{21} B_{21}^\mathsf{T}. \end{aligned}$$

Analogous formulas result from further refinement of the block-subdivision. Optimal block sizes depend on the processors cache hierarchy (see Chapter 4). They are intended to keep data in the caches as long as they are required. This way the implementation aims at minimizing the transfers of single data elements between cache and main memory. This is paying off since each data element is involved $\mathcal{O}(n)$-times in the operation. Also the order of operations during calculations can influence the amount of data copied per time unit.

Tuning is done by exploiting knowledge about the hardware specifications in vendor implementations (MKL, ESSL, but also OpenBLAS), or by optimizing the block sizes at compilation time as in ATLAS[5] (automatically tuned linear algebra subroutines).

6.4.2 Linear Algebra PACKage (LAPACK)

LAPACK is a Fortran 90 based library that provides routines for

- solution of linear systems of equations,
- least squares solutions of linear systems of equations,
- solutions of eigenvalue problems,
- and singular value problems.

The associated matrix factorizations that are underlying these algorithms are also provided, as are related operations (e.g., reordering of Schur factorizations to achieve other orderings of the eigenvalues.)

LAPACK was first released Feb 1992. The latest version is 3.9.0 and was published November 24, 2023. The library is in conception an add-on to BLAS, especially BLAS Level 3. It uses the appropriate BLAS routines wherever possible. That especially means that LAPACK supports the same data types as BLAS and uses, respectively, exploits the same matrix structures as described for the BLAS above.

Just like for the BLAS, the reference implementation is available at `https://netlib.org/lapack`.

Optimized vendor versions are for example included in:

- Intel® MKL
- Apple Accelerate framework (ATLAS based)
- IBM Engineering and Scientific Subroutines Library (ESSL)
- AMD AOCL-LAPACK

The automatically tuned linear algebra subroutines (ATLAS) also cover the operations defined in LAPACK.

LAPACK routines are divided in 3 Categories

i) auxiliary routines

ii) computational routines

[5] `https://math-atlas.sourceforge.net/`

iii) driver routines

The general naming scheme follows the BLAS Level-2/3 approach.

- auxiliary routines: these routines in LAPACK provide common helper functionality: scaling, reordering, machine specifications. Examples are:
  - **disnan**, **sisnan** — check the argument for NaN
  - **dlamch**, **slamch** — retrieve machine parameters, i.e., get $\mathbb{M}$, **eps**, base, length of mantissa, $e_{\min}$, $e_{\max}$
  - **cerbla** — error handling in case of invalid inputs
- computational routines: perform simple specific tasks
  - factorizations: LU, LL^*, LDL^*, QR, $LQ, \ldots$
  - eigenvalue and singular value computations
  - recovery of eigenvectors, Schur vector
- driver routines: these routines call a set of computational routines to solve linear algebra problems
  - linear equations: $Ax = b$
  - linear least squares: $\min_x \|b - Ax\|_2$
  - generalized linear least squares
  - eigenvalue decompositions
  - generalized eigenvalue/singular value decompositions

Related software:

- CLAPACK (C wrapper to LAPACK)
  https://www.netlib.org/clapack/
- ScaLAPACK (distributed parallel version)
  https://www.netlib.org/scalapack/
- PLASMA (Parallel Linear Algebra for Scalable Multicore Architectures)
  https://icl.cs.utk.edu/plasma/software/
- MAGMA (Matrix Algebra on GPU and Multicore Architectures)
  https://icl.cs.utk.edu/magma/
- LAPACK95 (Fortran 95)
  https://www.netlib.org/lapack95/
- JLAPACK (rather outdated Fortran-Java LAPACK)

- lapack++ (native C++ implementation last update in 2000)
  https://math.nist.gov/lapack++/

6.4.3 SuiteSparse

SuiteSparse is a collection of software packages/tools related to sparse factorizations (LU, Cholesky and QR) and direct solution of sparse linear systems. The UMFPACK tool from the collection is working behind the application of backslash to sparse linear systems in MATLAB. The main authors are T. A. Davis and his team at the Texas A&M University[6].

6.4.4 ITPACK

This package is intended for solving large sparse linear systems by iterative methods. It is hosted at https://www.netlib.org/itpack.

The main library consists of three sub-packages for

- single precision,
- double precision,
- vector machines.

It uses CG, PCG, Chebyschev acceleration and generalized CG for systems with non-symmetric matrices.

The development of this Fortran based package takes place at Center for Numerical Analysis at University of Texas at Austin.

6.4.5 Trilinos

"Trilinos is a collection of open source software libraries intended to be used as building blocks for the development of scientific applications".[7]

Trilinos is developed at the Sandia National Labs. The current version is 15.0.0 from Nov. 2023. The package is licensed under the terms of the LGPL[8] and covers:

- construction and usage of sparse and dense matrices, graphs and vectors.
- iterative and direct solution of linear systems
- parallel multilevel and algebraic preconditioning
- and many more ...

[6] https://faculty.cse.tamu.edu/davis/suitesparse.html
[7] https://en.wikipedia.org/wiki/Trilinos
[8] see, e.g., https://opensource.org/licenses/lgpl-license

The basic library is written in C++ with Fortran kernels. Moreover Python bindings are provided via SWIG. Trilinos can be found online at:
https://trilinos.org

6.4.6 Native Packages for other Programming Environments and Languages

- C++
  - boost — supports threading as well
    https://www.boost.org/
  - MTL — The Matrix Template Library
    https://www.simunova.com/en/node/24
    * The library uses boost and BLAS in kernels.
    * A single computer version available as OpenSource.
    * MTL4 has distributed computing capabilities, but those are connected to a payed license release.
- Python
  - NumPy — provides proper n-d array for Python
    https://www.numpy.org/
  - SciPy — amongst many others provides LAPACK functionality (calling F90 LAPACK)
    https://www.scipy.org/
- Java
  - JaMa — Java Matrix Package provides basic linear algebra in Java
    https://math.nist.gov/javanumerics/jama/
  - JaMPack — same as JaMa
  - maintenance questionable: latest release Nov 2012, previous version July 2005.

Bibliography

[1] J. J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.

[2] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Software, 14 (1988), pp. 1–17.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, third ed., 1996.

[4] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for FORTRAN usage*, ACM Trans. Math. Software, 5 (1979), pp. 303–323.

An algorithm must be seen to be believed.

Donald Ervin Knuth

CHAPTER 7

The Solution of Moderate Size Dense Linear Systems

Contents

7.1 Important Preliminaries

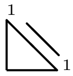
In this section we collect some facts that should be known from Numerical Analysis I

Theorem 7.1 (LU decomposition)**:** Let $A \in \mathbb{R}^{n\times n}$ and for $k = 1, \ldots, n-1$, $A_k = A(1 : k, 1 : k) \in \mathbb{R}^{k\times k}$ the leading $k \times k$ sub-matrix.

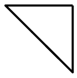
i) If $\forall k = 1, \ldots, n-1$ it holds $\det(A_k) \neq 0$, then $\exists L, U \in \mathbb{R}^{n\times n}$ such that

$$A = LU$$

with

$L =$  (unit lower triangular)

and

$U =$  (upper triangular).

ii) If $A = LU$ exists and A is regular then the LU factorization is unique.

iii) If $A = LU$ as in (ii) then

$$\det(A) = u_{11} \cdots u_{nn}$$

Proof. homework. □

Note that the simple regular 2×2 matrix $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ does not allow for an LU decomposition, but applying a single row permutation we get:

$$\tilde{A} := PA = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \text{where } P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$\tilde{A}$ has an LU decomposition by Theorem 7.1. This observation motivates the following theorem.

Theorem 7.2: Let $A \in \mathbb{R}^{n \times n}$ regular. There exists a permutation matrix $P \in \mathbb{R}^{n \times n}$ such that

$$PA = LU$$

for L, U as in Theorem 7.1.

idea of the proof. Exploit properties of Gaussian elimination procedure, that defines the L and U matrices, and permutation matrices. The full proof can be found, e.g. in [1, 3] □

Gaussian elimination is used to compute the L and U matrices. It consists of a triple loop procedure. The straight forward row-by-row elimination version reads:

Algorithm 7.1: Gaussian Elimination "kij"-formulation

Input: $A \in \mathbb{R}^{n \times n}$
Output: A overwritten by L, U

```
for k = 1 : n − 1 do
    A(k + 1 : n, k) = A(k + 1 : n, k)/A(k, k);
    for i = k + 1 : n do
        for j = k + 1 : n do
            A(i, j) = A(i, j) − A(i, k)A(k, j);
```

There are 5 other versions kji, ikj, ijk, jik, jki. The jki version is sometimes called *left looking LU*. It will become important for sparse matrices in Chapter 8.

Clever data arrangement (vector formulation) in kij-version leads to the so called **outer product Gaussian Elimination**:

Algorithm 7.2: Outer product Gaussian Elimination

Input: $A \in \mathbb{R}^{n \times n}$ fulfilling Theorem 7.1
Output: $L, U \in \mathbb{R}^{n \times n}$ such that $A = LU$ as in Theorem 7.1 A is overwritten by the factors.

```
for k = 1 : n − 1 do
    rows = k + 1 : n;
    A(rows, k) = A(rows, k)/A(k, k);
    A(rows, rows) = A(rows, rows) − A(rows, k)A(k, rows);
```

Algorithm 7.2 is a rank-1 update, i.e., BLAS Level 2 operation formulation of the Gaussian elimination process. It involves $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ flops. Solving $Ax = b$ for $x \in \mathbb{R}^n$ given $A \in \mathbb{R}^{n \times n},\ b \in \mathbb{R}^n$ now is performed as in

Algorithm 7.3: Linear System solver using Gaussian Elimination and forward/backward substitution

Input: $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$

1. Compute L, U as in Theorem 7.1, such that $A = LU$ (e.g. via Algorithm 7.2);
2. Solve $Ly = b$ by forward substitution (e.g., using Algorithm 7.5);
3. Solve $Ux = y$ by backward substitution;

Algorithm 7.4: Forward Substitution (Row Version)

Input: $L \in \mathbb{R}^{n\times n}$ (unit) lower triangular, $b \in \mathbb{R}^n$
Output: $y = L^{-1}b$ (stored in b)

$b(1) = \frac{b(1)}{L(1,1)}$;
for $i = 2 : n$ **do**
  $b(i) = \frac{b(i) - L(i,1:i-1)b(1:i-1)}{L(i,i)}$

7.2 Cache/BLAS Exploitation

7.2.1 Triangular System

Consider

$$\begin{aligned} a_{11}x_1 \qquad\qquad &= b_1, \\ a_{21}x_1 + a_{22}x_2 &= b_2. \end{aligned}$$

In case $a_{11} \neq 0$ and $a_{22} \neq 0$ this leads to

$$\begin{aligned} x_1 &= \frac{b_1}{a_{11}}, \\ x_2 &= \frac{b_2 - a_{21}x_1}{a_{22}} = \frac{b_2 - \frac{a_{21}}{a_{11}}b_1}{a_{22}} \end{aligned}$$

In the i-th equation in a system $Lx = b$ in Algorithm 7.3 we find:

$$x_i = \frac{b_i - \sum\limits_{j=1}^{i-1} l_{ij}x_j}{l_{ii}}$$

For the computation of all x_i we find a complexity of $\mathcal{O}(n^2)$ flops.

An accuracy discussion can be found in [2]. It states that the rounding error in each element of the solution vector is smaller than $n \cdot u$.

Note that row-wise access to L is “bad” in column major storage, since it destroys memory locality. Algorithm 7.5 presents a column major storage oriented version of the procedure.

Note further that the backward substitution can be derived completely analogously.

Algorithm 7.5: Forward Substitution (Column Version)

Input: $L \in \mathbb{R}^{n\times n}$ (unit) lower triangular, $b \in \mathbb{R}^n$
Output: $y = L^{-1}b$ (stored in b)
for $j = 1 : n-1$ **do**
  $b(j) = \frac{b(j)}{L(j,j)}$;
  $b(j+1:n) = b(j+1:n) - b(j)L(j+1:n,j)$;
$b(n) = \frac{b(n)}{L(n,n)}$;

Algorithm 7.6: Block Forward Substitution

Input: L, B as in (7.1)
Output: X solving $LX = B$
for $j = 1 : N$ **do**
  Solve $L_{jj}X_j = B_j$ for X_j;
  for $i = j+1 : N$ **do**
    $B_i = B_i - L_{ij}X_j$

7.2.2 Triangular Systems with Multiple Right Hand Sides and BLAS Level 3 formulation

Let $B \in \mathbb{R}^{n\times q}$ leading to a family of linear systems $LX = B$ with $X \in \mathbb{R}^{n\times q}$. L is (unit) lower triangular and we consider the block substructure as in

$$\begin{bmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{N1} & L_{N2} & \cdots & L_{NN} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix}. \tag{7.1}$$

We apply Algorithm 7.5 with the $L(1,1)$ element replaced by the L_{11} block to get

$$\begin{bmatrix} L_{22} & 0 & \cdots & 0 \\ L_{32} & L_{33} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{N2} & L_{N3} & \cdots & L_{NN} \end{bmatrix} \begin{bmatrix} X_2 \\ X_3 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} B_2 - L_{21}X_1 \\ B_3 - L_{31}X_1 \\ \vdots \\ B_N - L_{N1}X_1 \end{bmatrix}$$

after computing X_1 from $L_{11}X_1 = B_1$ by Algorithm 7.5. Now, successively, continuing with $L_{22}X_2 = \tilde{B}_2$ and so forth, we derive the block forward elimination scheme given in Algorithm 7.6

We can optimize the block sizes in (7.1) such that we get optimal performance out of the BLAS Level 3 block operations.

Again the backward substitution case allows for the analogous approach. This allows to accelerate the last two steps in Algorithm 7.3 by fast BLAS Level 3 operations.

7.2.3 BLAS **Level 3 based Gaussian Elimination**

The above raises the obvious question:

Can we do something similar for the Gaussian elimination process?

In fact we can. The following derivation will provide the block outer product formulation of the outer product Gaussian elimination in Algorithm 7.2. To this end, let $A \in \mathbb{R}^{n \times n}$ with partitioning

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{7.2}$$

Here $A_{11} \in \mathbb{R}^{r \times r}$, $A_{12} \in \mathbb{R}^{r \times (n-r)}$, $A_{21} \in \mathbb{R}^{(n-r) \times r}$, $A_{22} \in \mathbb{R}^{(n-r) \times (n-r)}$, for a blocking parameter $1 \leqslant r \leqslant n$. Now we can compute $A_{11} = L_{11}U_{11}$, e.g., using Algorithm 7.2 and solve the triangular systems

$$\begin{aligned} L_{11}U_{12} &= A_{12} \text{ for } U_{12}, \\ L_{21}U_{11} &= A_{21} \text{ for } L_{21}. \end{aligned}$$

Then it follows:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & \tilde{A}_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & I_{n-r} \end{bmatrix},$$

where

$$\tilde{A}_{22} = A_{22} - L_{21}U_{12}. \tag{7.3}$$

Now if $\tilde{A}_{22} = L_{22}U_{22}$ were the LU of the updated $(2,2)$ block, then

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

Since we did not post special assumptions on the matrix A in Equation (7.2) other than the existence of the LU-decomposition, we can proceed with $\tilde{A}_{22}$ as above. This leads to the procedure summarized in Algorithm 7.7.

Algorithm 7.7 requires $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ flops, just like Algorithm 7.2 but the rank-r update is a BLAS Level 3 operation, so optimizing the size of r according to our CPUs cache hierarchy we can expect superior performance. However, for $N = \frac{n}{r}$, [1] shows that the fraction of BLAS Level 3 operations in Algorithm 7.7 is $1 - \frac{1}{N^2}$; and $1 - \frac{1}{N}$ for the block-triangular solves. Note that this contradicts choosing r as large as possible and requires an additional level of optimization.

Algorithm 7.7: Panel Outer Product LU

Input: $A \in \mathbb{R}^{n \times n}$ as in Theorem 7.1, r as above
Output: $A = LU$ with L, U stored in A

```
k = 1;
while k ⩽ n do
    l = min(n, k + r − 1);
    Compute A(k : l, k : l) = L̃Ũ via Algorithm 7.2;
    Solve L̃Z = A(k : l, l + 1 : n) and store Z;
    Solve WŨ = A(l + 1 : n, k : l) and store W;
    Perform the rank-r update:
      A(l + 1 : n, l + 1 : n) = A(l + 1 : n, l + 1 : n) − WZ;
    k = l + 1;
```

Algorithm 7.8: iterative refinement

Input: $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, \hat{x}$ an approximate solution
Output: $\hat{x}$ a solution (approximation)

```
repeat
    r = b − Ax̂;
    solve Ad = r;
    update x̂ = x̂ + d
until x̂ accurate enough;
```

7.3 Iterative Refinement

Iterative refinement is a fixed point type approach that seeks to improve the computed result of a linear system solve. In the notation of Chapter 5 let $\hat{x}$ be the computed solution of $Ax = b$. The iterative refinement process is summarized in the Algorithm 7.8. A common application is the iterative refinement of single precision results on a double precision architecture. This is, e.g., used in connection with accelerator devices such as graphics processing units, that are usually working a lot faster in single precision, than in double precision.

Motivation: Let $r = b - A\hat{x}$ and $d = A^{-1}r$, $\tilde{x} = \hat{x} + d$. Then in exact arithmetic we have

$$A\tilde{x} = A(\hat{x} + d) = A\hat{x} + Ad = (b - r) + AA^{-1}r = b - r + r = b$$

Thus in exact arithmetic the updated $\hat{x}$ in Algorithm 7.8 would be the exact solution after 1 step.

The literature distinguishes mainly 2 approaches:

i) fixed precision refinement

ii) mixed precision refinement

In fixed precision refinement all steps in Algorithm 7.8 are computed in the same precision ($\mathbf{u}$).

For mixed precision refinement the residual r is computed in a higher precision ($\hat{\mathbf{u}}$). Classically $\hat{\mathbf{u}} = \mathbf{u}^2$, i.e., $\mathbf{u}$ corresponds to single precision, and $\hat{\mathbf{u}}$ then stands for double precision.

Notation: Let $A \in \mathbb{R}^{n\times n}$ be a square matrix. The absolute value of A is defined component-wise:

$$|A| = (|a_{ij}|)_{i,j=1,\dots,n}.$$

Under the assumption

$$(A + \Delta A)\hat{x} = b \quad |\Delta A| \leqslant \mathbf{u}w \tag{7.4}$$

for W non-negative depending on A, n, and u (but not on b), [2] proves the following two theorems based on forward analysis:

Theorem 7.3 (Mixed Precision Refinement)**:** Let $Ax = b$ be a non-singular linear system solved with a method satisfying (7.4) and residuals in double the working precision. Moreover

$$\eta = u \left\| |A^{-1}| \left(|A| + w\right) \right\|_\infty$$

If $\eta < 1 - \delta$ for δ large enough, then iterative refinement reduces the forward error by approximately a factor of η at each stage until

$$\frac{\|x - \hat{x}\|_\infty}{\|x\|_\infty} \approx \mathbf{u}$$

Theorem 7.4 (Fixed Precision Refinement)**:** Setting as in Theorem 7.3 but with residual computation in working precision. The same reduction holds, but with limit

$$\frac{\|x - \hat{x}\|_\infty}{\|x\|_\infty} \leqslant 2nu \underbrace{\frac{\left\| |A^{-1}||A||x| \right\|_\infty}{\|x\|_\infty}}_{\mathrm{cond}(A,x)} \tag{7.5}$$

Remark 7.5:

- (7.5) is essentially the best we can expect in fixed precision.

- Note that the solver need not be of LU type and $\hat{\mathbf{u}}$ is not limited to $\mathbf{u}^2$.
- When working in $\hat{\mathbf{u}} = \mathbf{u}^2$, i.e., system solves in single precision and residual in double precision, one can reuse the LU decomposition from the outer solve. That means the iterative refinement is of $\mathcal{O}(n^2)$ complexity, i.e., one order of magnitude cheaper than the actual solve and the amount of data copied is reduced due to single precision storage.
- Fixed precision iterative refinement may be used to stabilize unstable solvers for $Ax = b$, e.g., $LU = PA$ computed with poor pivoting (see [2, Section 12.2]).
- **rule of thumb:**
  machine precision: $10^{-d} = \mathbf{u}$, $\kappa_\infty(A) \approx 10^q \rightsquigarrow k$ steps of mixed precision refinement lead to approximately $\min(d, k(d-q))$ correct digits in x.

Convergence of iterative refinement from the splitting method point of view: Splitting Methods: $A = B + (A - B)$

$$\begin{aligned}
\Rightarrow Ax = b &\Leftrightarrow B^{-1}(B + (A-B))x = B^{-1}b \Leftrightarrow (I + (B^{-1}A - I))x = B^{-1}b \\
&\Leftrightarrow x + (B^{-1}A - I)x = B^{-1}b
\end{aligned}$$

$$\begin{aligned}
\rightsquigarrow x_{i+1} &= B^{-1}b - (B^{-1}A - I)x_i \qquad (*) \\
&= x_i + \underbrace{B^{-1}\underbrace{(b - Ax_i)}_{r_i}}_{d_i}
\end{aligned}$$

If $B^{-1} = (\hat{L}\hat{U})^{-1}$ this reflects a refinement of the LU. From (*) we immediately find $x_{i+1} = B^{-1}b + B^{-1}(B - A)x_i$. As for the splitting methods in general, by the Banach fixed point theorem we then have that the iteration converges if $M := B^{-1}(B - A)$ is a contraction, i.e. $\rho(M) < 1$.

Bibliography

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, fourth ed., 2013.

[2] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, PA, second ed., 2002.

[3] A. Meister, *Numerik linearer Gleichungssysteme. Eine Einführung in moderne Verfahren.*, Vieweg+Teubner, Wiesbaden, 4th revised ed. ed., 2011, https://doi.org/10.1007/978-3-8348-8100-7.

I would rather have today's algorithms on yesterday's computers than vice versa.

Phillipe Toint

CHAPTER 8

Solving Linear Systems With Sparse Matrices

Contents

Recall:

- sparse matrix: $A \in \mathbb{R}^{n\times n}$, such that $y = Ax$ can be computed in $\mathcal{O}(n)$ complexity.

- storage:
  - only non-zero entries are stored,
  - indirect indexing is mandatory for minimal storage requirements,
  - e.g., CSR (compressed sparse row storage, with C/zero based indexing)

$$\begin{bmatrix}1&2&0&0\\0&3&4&0\\0&5&0&6\\0&0&7&0\end{bmatrix} \to \begin{array}{l} \begin{bmatrix}1&2&3&4&5&6&7\\0&1&1&2&1&3&2\\0&2&4&6&7&&\end{bmatrix} \end{array} \begin{array}{l}\text{values (double)}\\ \text{column indices (long)}\\ \text{row-pointers (long)}\end{array}$$

number of non-zeroes

Issues

"Cache" Indirect indexing requires the value, index and row-pointer vectors to reside in the cache simultaneously for optimal performance.
Consider:

- 64 bit architecture
- in average 10 entries per row
- 4MB cache
- $A \in \mathbb{R}^{24\,000 \times 24\,000}$

Required storage:[1]

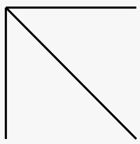
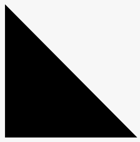
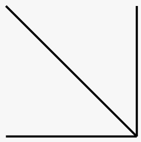
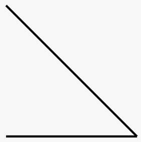
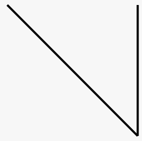
$$\begin{aligned}(24\,000 + 240\,000 + 240\,000) \times 8 \text{ Bytes} &= 504\,000 \times 8 \text{ Bytes}\\ &= 4032 \text{ kBytes}\end{aligned}$$

That means we have $(4\,096 - 4\,032)$ kBytes$=$ 64 kByte of cache left for instructions in $y = Ax$. In applications one easily wants to work with $n = 10^6 \dots 10^8$, which on modern computers usually easily fits into RAM. The execution speed of operations with A are thus strictly limited by data transfer rate from the main memory to the caches.

"Fill in" Another important issue with sparse matrices arises with direct solvers. These require matrix factorizations. However, it can not be guaranteed that the factors stay sparse if the matrix A is sparse. Usually the factors get a certain amount of new entries. The new entries are referred to as *fill* or *fill-in*. We will see more details on this phenomenon in Section 8.4.

[1] neglecting the **nnz** entry for the sake of simpler numbers

Example 8.1 (Fill-In)**:** The diagrams below show the non-zero entry distribution in A, L and U for A sparse and $A = LU$.

	A	L	U
worst case			
best case			

Definition 8.2 (pattern)**:** Let $A \in \mathbb{R}^{n\times n}$ be a matrix.
We call the set

$$\mathcal{P}(A) = \{(i,j) : a_{ij} \neq 0\}$$

the *pattern* of A.
Furthermore, we define

$$\mathcal{P}_R(A,i) = \{j : a_{ij} \neq 0\}$$

as the pattern of the i-th row of A.

Definition 8.3 (structural rank)**:** Let $\mathcal{P}(A) \subset \mathbb{N}^2$ be a pattern of a matrix $A \in \mathbb{R}^{n\times n}$. The number

$$\mathrm{rk}_S(A) = \max\{\mathrm{rank}(B) : B \in \mathbb{R}^{n\times n} \text{ with } \mathcal{P}(B) = \mathcal{P}(A)\}$$

is called the *structural rank* of A.
If $\mathrm{rk}_S(A) < n$, then A is called *structural rank deficient*

Example 8.4:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \qquad C = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix},$$

$$\mathrm{rk}_S(A) = 2 \neq 1 = \mathrm{rank}(A)\,, \qquad \mathrm{rk}_S(C) = 1 = \mathrm{rank}(C)\,.$$

Remark 8.5: The structural rank of A is:

- a property related to the pattern $\mathcal{P}(A)$,
- much cheaper to compute than the (numerical) rank,
- available via `sprank()` in MATLAB,
- an upper bound to the rank of A.

8.1 Preconditioning

In everything presented here, we will only use the so-called left preconditioning. Other versions like right, or two-sided preconditioning also exist. The ideas are very similar there, therefore we restrict the presentation to the most simple case.

Recall Lemma 6.19:

$$P \in \mathbb{C}^{n\times n} \text{ non-singular}, \; A \in \mathbb{C}^{n\times n}, x, b \in \mathbb{C}^n$$

$$Ax = b \Leftrightarrow PAx = Pb$$

The matrix P can be used to lower the condition number for finding x. The perfect candidate for such a matrix P is obviously A^{-1}, since then $PA = I$ and $\kappa(PA) = 1$.

However, A^{-1} is not accessible and especially has even worse "fill in" restrictions than the factorizations. Good approximations to A^{-1} are thus required that are:

- cheap to generate,
- easily and efficiently applicable,
- able to get stored with similar memory requirement as A.

P does not need to be a matrix, e.g., sometimes other (iterative) solvers are used.

8.1.1 Diagonal Preconditioning

$$P^{-1} = \operatorname{diag}(A)$$

- also called Jacobi preconditioning
- very simple and cheap
- might improve certain problems, e.g., diagonal dominant systems

- generally not sufficient
- more sophisticated variants use diagonal $k \times k$ ($k > 1$) blocks or multiple diagonals (e.g., tridiagonal preconditioning)

8.1.2 Splitting Methods

Recall Section 7.3. Set $A = B + (A - B)$, then

$$Ax = b \Leftrightarrow Bx = b + (B - A)x.$$

This motivates to define:

$$x_{i+1} = B^{-1}b + \underbrace{B^{-1}(B - A)}_{M} x_i.$$

If we can ensure $\rho(M) < 1$ then by a fixed point argument we can guarantee convergence.

Example 8.6: Two common examples of splitting methods are:

- $B =$ diagonal of A $\rightsquigarrow$ Jacobi method
- $B =$ lower triangle of A $\rightsquigarrow$ Gauß Seidel method

Splitting methods are often considered to be smoothers rather than preconditioners. They mainly damp out high frequency parts of the error. Therefore, often they are used in combination with multigrid techniques in order to smooth interpolation errors.

8.1.3 Multigrid approaches

If A was generated by a hierarchical approach (e.g., the finite element method (FEM) with successive mesh refinement), the multiple layers (FEM grids) can be used to successively restrict the current iterate of the outer iteration to the coarsest grid/mesh. Then one gets a good solution there and performs interpolation to get back to the finest level.

Splitting methods are used to smooth out the high frequency interpolation errors. If the hierarchy is unknown or unusable, algebraic approaches can be used to generate the hierarchy from the connectivity graph of the matrix, i.e., the graph with nodes $1, \dots, n$ and edges from i to j if $(i, j) \in \mathcal{P}(A)$. Clusters and subclusters of nodes then produce the required hierarchy.

8.1.4 Incomplete Factorizations

Computation of $LU = A$ is often infeasible due to fill-in.
Basic idea: Only allow entries in L, U corresponding to $\mathcal{P}(A)$. This leads to the ILU(0) often written simply as ILU.

- usually only provides poor approximation
- variants allow:
  - "levels (k) of fill" (ILU(k))
  - fill-in that exceeds a drop tolerance ε (ILU(ε))
  - adding dropped fill to the diagonal (MIC)

8.1.5 Sparse Approximate Inverses (SPAI)

The basic idea of the sparse approximate inverse (SPAI) is to find the matrix $M \in \mathbb{R}^{n\times n}$ that best approximates A^{-1} among all matrices with $\mathcal{P}(M) = \mathcal{P}(A)$, in the sense

$$\min_M \|AM - I\|_F^2 = \min_M \underbrace{\sum_{j=1}^{n} \|Am_j - e_j\|_F^2}_{n \text{ independent least squares problems}} .$$

The SPAI preconditioner is especially attractive in parallel computing due to the independent column-wise computation.

In order to improve the approximation quality, similar pattern-extension considerations as for the incomplete factorizations can be used.

In any case, only matrix vector products are required for the application of the preconditioner, since PAx would be evaluated as $P(Ax)$, i.e., two subsequent matrix vector products.

8.2 Krylov Subspaces and Projection Methods

Definition 8.7: $A \in \mathbb{C}^{n\times n}$ regular, $b \in \mathbb{C}^n$. A *projection method* for $Ax = b$ is a procedure for approximation of x by $x_m \in x_0 + \mathcal{K}_m$, which satisfies

$$(b - Ax_m) \perp \mathcal{L}_m. \tag{8.1}$$

Here, $x_0 \in \mathbb{C}^n$ is an arbitrary initial vector and $\mathcal{K}_m$, $\mathcal{L}_m$ are m-dimensional subspaces of $\mathbb{C}^n$.

Condition (8.1) represents orthogonality in the Euclidean sense.

In case $\mathcal{K}_m = \mathcal{L}_m$, (8.1) is called *Galerkin-condition* and one has an *orthogonal projection method*. In case $\mathcal{K}_m \neq \mathcal{L}_m$, (8.1) is called *Petrov-Galerkin-condition* and one has an *oblique projection method*.

Definition 8.8: $A \in \mathbb{C}^{n\times n}$ regular, $y \in \mathbb{C}^n$.

i) $\mathcal{K}_m(A, y) = \operatorname{span}\{y, Ay, A^2y, \dots, A^{m-1}y\}$ is called the m-th Krylov subspace of A for a seed vector y.

ii) A projection method with $\mathcal{K}_m = \mathcal{K}_m(A, y)$ is called *Krylov subspace (projection) method*.

Definition 8.9 (minimal polynomial of A)**:** Let $p_\nu(\lambda) = \sum\limits_{j=0}^{\nu} a_j\lambda^j$. The polinomial p_ν is called *minimal polynomial of A* if $\nu \in \mathbb{N}$ is the smallest degree such that $p_\nu(A) = 0$.

In exact arithmetic we get the exact solution with $m = \nu$, since

$$\sum_{j=0}^{\nu} a_jA^j = 0 \Leftrightarrow A\sum_{j=1}^{\nu} a_jA^{j-1} = -a_0I.$$

Thus

$$A^{-1} = -\frac{1}{a_0}\sum_{j=1}^{\nu} a_jA^{j-1},$$

which, in turn, means

$$x = A^{-1}b = -\frac{1}{a_0}\sum_{j=1}^{\nu} a_jA^{j-1}b \in \mathcal{K}_\nu(A, b).$$

Now we let $x_0 \in \mathbb{C}^n$ be the initial vector and $r_0 := b - Ax_0$ the corresponding initial residual. Further, let $\mathcal{K}_m = \mathcal{K}_m(A, r_0)$, $\mathcal{L}_m$ be subspaces, and the columns of $V_m, W_m \in \mathbb{C}^{n\times m}$ bases of $\mathcal{K}_m$ and $\mathcal{L}_m$, respectively.

Then, for $x_m \in x_0 + \mathcal{K}_m$ there exists a $\sigma_m \in \mathbb{C}^m$ with $x_m = x_0 + V_m\sigma_m$ and (8.1) holds if and only if

$$\begin{aligned}
&\Leftrightarrow 0 = W_m^{\mathsf{H}}(b - A(x_0 + V_m\sigma_m))\\
&\Leftrightarrow 0 = W_m^{\mathsf{H}}(b - Ax_0) - W_m^{\mathsf{H}}AV_m\sigma_m\\
&\Leftrightarrow W_m^{\mathsf{H}}AV_m\sigma_m = W_m^{\mathsf{H}}r_0\\
&\Leftrightarrow \sigma_m = (W_m^{\mathsf{H}}AV_m)^{-1}W_m^{\mathsf{H}}r_0.
\end{aligned}$$

Thus $x_m = x_0 + V_m(W_m^{\mathsf{H}} A V_m)^{-1} W_m^{\mathsf{H}} r_0$

$$\begin{aligned} r_m &= b - Ax_m \\ &= b - A(x_0 + V_m(W_m^{\mathsf{H}} A V_m)^{-1} W_m^{\mathsf{H}} r_0) \\ &= r_0 - AV_m(W_m^{\mathsf{H}} A V_m)^{-1} W_m^{\mathsf{H}} r_0 \end{aligned}$$

The projection P_m to the m-th subspace is then given as $P_m = I - Q_m$, where $Q_m = AV_m(W_m^{\mathsf{H}} A V_m)^{-1} W_m^{\mathsf{H}}$. The above derivation proves the following simple lemma.

Lemma 8.10: If $W_m^{\mathsf{H}} A V_m$ is invertible, then (8.1) has a unique solution given as

$$x_m = x_0 + V_m(W_m^{\mathsf{H}} A V_m)^{-1} W_m^{\mathsf{H}} r_0$$

with corresponding residual

$$r_m = r_0 - AV_m(W_m^{\mathsf{H}} A V_m)^{-1} W_m^{\mathsf{H}} r_0$$

The invertibility assumption is sometimes easily guaranteed. For example if A is symmetric positive definite (s.p.d.) with $\mathcal{K}_m = \mathcal{K}_m(A, r_0) = \mathcal{L}_m$

$$\begin{aligned} &\Rightarrow W_m = V_m \text{ and } \dim \mathcal{K}_m = m \\ &\Rightarrow W_m^{\mathsf{H}} A V_m = V_m^{\mathsf{H}} A V_m \text{ s.p.d.} \end{aligned}$$

Analogously, for A invertible and $\mathcal{L}_m = A\mathcal{K}_m \Rightarrow W_m = AV_m$ with $\dim \mathcal{K}_m = m = \dim \mathcal{L}_m$, we immediately see that $W_m^{\mathsf{H}} A V_m = V_m^{\mathsf{H}} A^{\mathsf{H}} A V_m$ is s.p.d..

8.3 Conjugate Gradients

Different choices of $\mathcal{K}_m$ and $\mathcal{L}_m$ lead to different methods. Let $A \in \mathbb{R}^{n\times n}$ be symmetric and positive definite. If we choose $\mathcal{K}_m = \mathcal{L}_m = \mathcal{K}_m(A, r_0)$ then $V_m = W_m$ and, as we have investigated $x_m \in x_0 + \mathcal{K}_m(A, r_0)$ and $r_m \perp \mathcal{K}_m(A, r_0)$ and $W_m^{\mathsf{H}} A V_m = V_m^{\mathsf{H}} A W_m$ is s.p.d. for all m. The resulting method is called *conjugate gradients* (CG) method and is summarized in Algorithm 8.1. We have discussed the necessity of preconditioning in Section 8.1 above. The algorithm that results from the application of left preconditioning in Algorithm 8.1 is the *preconditioned CG*, presented in Algorithm 8.2. Note that the algorithm can be formulated such that we only need one additional matrix vector product at the cost of one additional vector in memory, namely the preconditioned residual.

Algorithm 8.1: Conjugate Gradient Method

Input: $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n$

Output: $x = A^{-1}b$

$p_0 = r_0 = b - Ax_0, \alpha_0 = \|r_0\|_2^2$;

for $m = 0, \dots, n-1$ **do**

  if $\alpha_m \neq 0$ **then**

    $v_m = Ap_m$;

    $\lambda_m = \frac{\alpha_m}{(v_m, p_m)}$;

    $x_{m+1} = x_m + \lambda_m p_m$;

    $r_{m+1} = r_m - \lambda_m v_m$;

    $\alpha_{m+1} = \|r_{m+1}\|_2^2$;

    $p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} p_m$;

  else

    `STOP`;

Remark 8.11: The CG method is often derived from minimization of the functional

$$\begin{aligned} \mathcal{F}\colon \mathbb{R}^n &\to \mathbb{R}, \\ x &\mapsto \frac{1}{2}(Ax, x)_2 - (b, x)_2 \end{aligned}$$

In fact CG minimizes the error $e_m := x_m - A^{-1}b$ with respect to the norm

$$\|x\|_A := \sqrt{(Ax, x)_2}$$

induced by the matrix A due to symmetry and positive definiteness.

Theorem 8.12: Let

$$e_m = x_m - A^{-1}b$$

denote the error in the m-th step of the CG algorithm. Then it holds

$$\|e_m\|_A \leqslant 2\left(\frac{\kappa_2(A) - 1}{\kappa_2(A) + 1}\right)^m \|e_0\|_A .$$

Proof. any textbook on iterative methods. □

Algorithm 8.2: Preconditioned Conjugate Gradient Method

Input: $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n, A^{-1} \approx P \in \mathbb{R}^{n\times n}$
Output: $x = A^{-1}b$

$r_0 = b - Ax_0,\ p_0 = z_0 = Pr_0,\ \alpha_0 = (r_0, p_0)$;
for $m = 0 : n-1$ **do**
    if $\alpha_m \neq 0$ **then**
        $v_m = Ap_m$;
        $\lambda_m = \frac{\alpha_m}{(v_m, p_m)_2}$;
        $x_{m+1} = x_m + \lambda_m p_m$;
        $r_{m+1} = r_m - \lambda_m v_m$;
        $z_{m+1} = Pr_{m+1}$;
        $\alpha_{m+1} = (r_{m+1}, z_{m+1})_2$;
        $p_{m+1} = z_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} p_m$;
    else
        `STOP`;

8.4 Direct Solvers for Sparse Symmetric Systems

In the following, to ease the presentations, we will follow the general assumptions that

- $A \in \mathbb{R}^{n\times n}$ is sparse and symmetric,
- and no pivoting is used.

For non-symmetric matrices the presented concepts have to be generalized from undirected to directed graphs. We leave these details out to get a better view on the basic ideas and avoid the additional technical difficulties that would distract readers.

Definition 8.13: Two graphs are easily related to the matrix $A \in \mathbb{R}^{n\times n}$.

i) $\mathcal{V} = \{1, \ldots, n\}$ is called the set of *vertices*, i.e., variable indices.

ii) The set of *edges* $\mathcal{E} \subseteq V^2$ is the set of pairs $(i,j) \in \mathcal{E} \Leftrightarrow a_{ij} \neq 0$.

iii) The *directed connectivity graph of* A $\mathcal{G}_d(A) = (\mathcal{V}, \mathcal{E})$ associates a direction to an edge by the order of indices in the pair.

iv) The *undirected connectivity graph of* A $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$ identifies the pairs (i,j) and (j,i), i.e. considers $(i,j) = (j,i)$, and thus neglects the direction.

Remark 8.14: We collect some properties of the symmetric case treated in this chapter.

- A symmetric $\Rightarrow a_{ij} = a_{ji} \Rightarrow$ "$(i,j) \in \mathcal{E} \Leftrightarrow (j,i) \in \mathcal{E}$" $\Rightarrow$ its is sufficient to the treat the undirected graph
- If A s.p.d. then $\forall i\ a_{ii} > 0 \Rightarrow (i,i) \in \mathcal{E}$, i.e., the graph contains the trivial edges (usually not included in graphical representations of the graph)
- The number of nonzero elements in column i equals the number of neighbors of the vertex i in the graph $\mathcal{G}(A)$.
- Symmetric permutations, i.e., permutations of the matrix where both columns and rows are swapped simultaneously, are equivalent to renumbering the graph, i.e., application of a permutation to the elements of $\mathcal{V}$.
- $\mathcal{E} \mathrel{\hat{=}} \mathcal{P}(A)$

8.4.1 The Elimination Graph Model for Symmetric Matrices

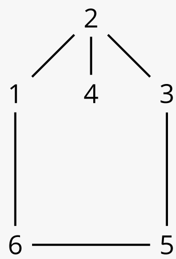
Idea: Compute LL^T from a sequence of $\mathrm{rank}\text{-}1$ reductions, following the lines of the derivation of Algorithm 7.2

$$
\begin{aligned}
A = A_0 = H_0 &= \begin{bmatrix} d_1 & v_1^\mathsf{T} \\ v_1 & \tilde{H}_1 \end{bmatrix}, \quad \tilde{H}_1 \in \mathbb{R}^{n-1 \times n-1} \\
&= \underbrace{\begin{bmatrix} \sqrt{d_1} & 0 \\ \frac{1}{\sqrt{d_1}} v_1 & I_{n-1} \end{bmatrix}}_{L_1} \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & H_1 \end{bmatrix}}_{A_1} \underbrace{\begin{bmatrix} \sqrt{d_1} & \frac{1}{\sqrt{d_1}} v_1^\mathsf{T} \\ 0 & I_{n-1} \end{bmatrix}}_{L_1^\mathsf{T}} \\
A &= (L_1 L_2 L_3 \dots L_{n-1}) I_n (L_{n-1}^\mathsf{T} \dots L_3^\mathsf{T} L_2^\mathsf{T} L_1^\mathsf{T}) \\
&= (L_1 L_2 L_3 \dots L_{n-1}) I_n (L_1 L_2 L_3 \dots L_{n-1})^\mathsf{T} \\
&= LL^\mathsf{T}
\end{aligned}
$$

$v_j v_j^\mathsf{T}$ influence the structure, i.e., pattern of H_j. It is a usually dense (but probably scattered) sub-block of H_j. If $\mathcal{P}(v_j v_j^\mathsf{T}) \backslash (\mathcal{P}(v_j v_j^\mathsf{T}) \cap \mathcal{P}(H_{j-1})) \neq \varnothing$ then step j leads to fill-in in H_j.

What does this procedure mean in terms of the graphs? The answer is best understood following a simple example.

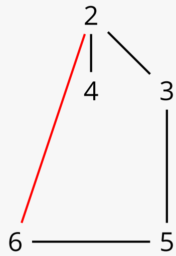
Example 8.15: This example demonstrates the graph elimination procedure and resulting fill-in for the Cholesky decomposition of a simple 6×6 example. Actual values are unimportant and thus replaced by $*$'s. The indices are indicated on the diagonal.



(a) initial graph $\mathcal{G}_0$

$$H_0 = \begin{bmatrix} 1 & * & & & & * \\ * & 2 & * & * & & \\ & * & 3 & & * & \\ & * & & 4 & & \\ & & * & & 5 & * \\ * & & & & * & 6 \end{bmatrix}$$

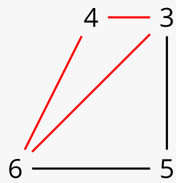
(b) corresponding submatrix 0



(c) elimination graph $\mathcal{G}_1$

$$H_1 = \begin{bmatrix} 2 & * & * & & \color{red}{*} \\ * & 3 & & * & \\ * & & 4 & & \\ & * & & 5 & * \\ \color{red}{*} & & & * & 6 \end{bmatrix}$$

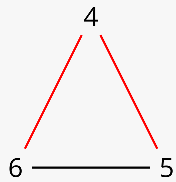
(d) corresponding submatrix 1



(e) elimination graph $\mathcal{G}_2$

$$H_2 = \begin{bmatrix} 3 & \color{red}{*} & * & \color{red}{*} \\ \color{red}{*} & 4 & & \color{red}{*} \\ * & & 5 & * \\ \color{red}{*} & \color{red}{*} & * & 6 \end{bmatrix}$$

(f) corresponding submatrix 2



(g) elimination graph $\mathcal{G}_3$

$$H_3 = \begin{bmatrix} 4 & \color{red}{*} & \color{red}{*} \\ \color{red}{*} & 5 & * \\ \color{red}{*} & * & 6 \end{bmatrix}$$

(h) corresponding submatrix 3

Figure 8.1: Basic graph elimination procedure for a symmetric matrix and the Cholesky decomposition

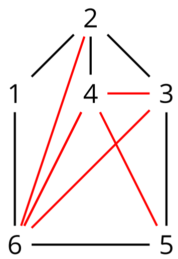
Algorithm 8.3: graph eliminations process

Input: $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$ undirected graph of A
Output: $\mathcal{G}_1, \ldots, \mathcal{G}_{n-1}$ sequence of *eliminations graphs*

for *k=1:n-1* **do**
  $\mathcal{V} = \mathcal{V} \backslash \{k\}$ (remove vertex k);
  $\mathcal{E} = (\mathcal{E} \backslash \{(k, l) : l \text{ neighbor of } k\}) \cup \{(x, y) : x, y \text{ neighbors of } k\}$;

8.4.2 The filled graph $\mathcal{G}^+(A)$

The procedure above introduces new elements. Let $F = L + L^\mathsf{T}$, then $\mathcal{P}(F)$ is the filled pattern of A and $\mathcal{G}(F)$ ist called the *filled graph* of A denoted by $\mathcal{G}^+(A)$. For the example above we have:



(a) The filled graph $\mathcal{G}^+(A) = \mathcal{G}(F)$

$$F = \begin{bmatrix} 1 & * & & & & * \\ * & 2 & * & * & & * \\ & * & 3 & * & * & * \\ & * & * & 4 & * & * \\ & & * & * & 5 & * \\ * & * & * & * & * & 6 \end{bmatrix}$$

(b) The final matrix $F = L + L^\mathsf{T}$ with fill.

Figure 8.2: The filled graph and matrix of a Cholesky decomposition example.

Obviously, the filled graph $\mathcal{G}^+(A)$ is the union of the elimination graphs $\mathcal{G}_0, \mathcal{G}_1, \ldots$ In fact one can prove:

Lemma 8.16 ([3]): $(i, j) \in \mathcal{G}^+(A) \Leftrightarrow (i, j) \in \mathcal{G}(A)$, or there exists $k < \min(i, j)$, such that $(i, k) \in \mathcal{G}^+(A)$ and $(k, j) \in \mathcal{G}^+(A)$.

8.4.3 Characterization of Fill-in

Let $L = (l_{ij})_{i,j=1,\ldots,n}$ be a Cholesky factor of A, i.e., $A = LL^\mathsf{T}$.

Theorem 8.17 (Fill-path-theorem [4]): $l_{ij} \neq 0 \Leftrightarrow \exists$ path in $\mathcal{G}(A)$ between i and j such that all nodes (vertices) in the path have indices smaller than both i and j.

We have seen in the introduction of Chapter 8, that reordering of variables can have strong impact on the amount of fill-in and consequently on the subsequent operations.

Definition 8.18: The *minimum fill-in problem* describes the problem of finding the optimal permutation of vertex labels that produces the smallest possible number of new edges in $\mathcal{G}^+(A)$ compared to $\mathcal{G}(A)$.

The article [7] shows that the minimum fill-in problem is NP-complete and thus NP-hard in general. Several heuristic approaches exist that come up with sub-optimal solutions.

8.4.4 Heuristic Fill Reduction

Mainly 3 classes of methods exist.

i) Global approaches
   - Structured permutation
   - Fill-in only in the resulting structure
   - Examples: (reverse) Cuthill-McKee, nested dissection

ii) Local heuristics
   - Incorporated into pivoting strategies
   - Symmetric case: minimum degree, minimum fill
   - General case: Markowitz criterion

iii) Hybrid variants
   (a) Permutation to block structure
   (b) Local heuristic applied on the single blocks

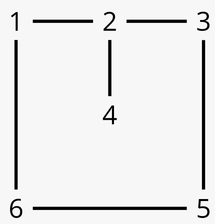
(Reverse) Cuthill-McKee Reordering (RCM)

A global strategy that approaches the minimum fill problem by bandwidth (n_b) minimization is the (Reverse) Cuthill-McKee reordering. Its general aim is to find a symmetric permutation such that

$$n_b = \max_i \max_{a_{ij} \neq 0} |i - j|$$

is minimized. Recall that a symmetric permutation is just the same as a vertex relabeling.

Example 8.19: Influence of the ordering of the degrees of freedom on the resulting fill-in in the Cholesky decomposition is demonstrated in the following two figures.

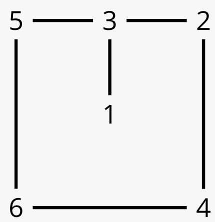


(a) Graph before reordering.

$$\begin{bmatrix} 1 & * & & & & * \\ * & 2 & * & * & & \\ & * & 3 & & * & \\ & * & & 4 & & \\ & & * & & 5 & * \\ * & & & & * & 6 \end{bmatrix}$$

(b) Bandwidth 5 pattern.

Figure 8.3: Graph and sparsity pattern before reordering.



(a) Graph after RCM reordering.

$$\begin{bmatrix} 1 & & * & & & \\ & 2 & * & * & & \\ * & * & 3 & & * & \\ & * & & 4 & & * \\ & & * & & 5 & * \\ & & & * & * & 6 \end{bmatrix}$$

(b) Resulting bandwidth 2 pattern.

Figure 8.4: Graph and sparsity pattern after RCM reordering.

Basically, RCM reordering selects a root node, forms the tree that consists of all shortest paths to all other vertices in $\mathcal{G}(A)$ and then performs an *ordered* breadth first search on that tree to fill the permutation vector.

In contrast to a standard breadth first search, here the vertices are ordered with respect to their increasing degree.

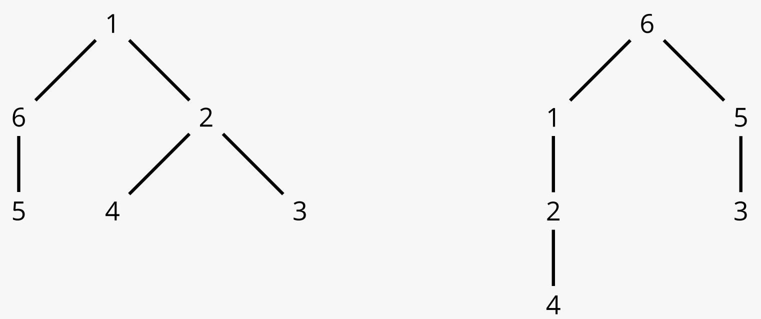
Step 10 in Algorithm 8.4 is mandatory for the reverse reordering, when avoided the algorithm implements Cuthill-McKee reordering. Both versions lead to the same bandwith, but the reverse version is observed to have better properties wth respect to fill-in. Selection of a good root node in Step 3 is crucial for a good reordering. This is demonstrated for some good starting nodes in the next example. As a rule of thumb, the root node should be chosen such that it has preferably long paths to all other nodes in the graph. We leave it as an excercise to derive the reorderings for root node 2, which is a particularly bad choice.

Algorithm 8.4: Reverse Cuthill-McKee (RCM) reordering

Input: $A \in \mathbb{R}^{n \times n}$ with $\mathcal{P}(A)$ symmetric
Output: $p \in \mathbb{R}^n$ such that $\tilde{A} = A(p,p)$ has reduced bandwidth

1 $Q = [], R = []$;
2 **repeat**
3   Select root node $P \notin R$;
4     $R = [R, P]$;
5     $Q = [Q$, nodes adjacent to P ordered by increasing degree];
6   **while** $Q \neq \emptyset$ **do**
7         $R = [R, Q(1)]$,
8         $Q = [Q(2 : \mathrm{end})$, nodes adjacent of $Q(1)$ not contained in R by increasing degree];
9 **until** *all nodes are contained in* R;
10 $p = R(n : -1 : 1)$;

Example 8.20: This example shows the importance of the selection of the root node in Step 3 of Algorithm 8.4.



$R = [1]$	$Q = [6, 2]$	$R = [6]$	$Q = [1, 5]$
$R = [1, 6]$	$Q = [2, 5]$	$R = [6, 1]$	$Q = [5, 2]$
$R = [1, 6, 2]$	$Q = [5, 4, 3]$	$R = [6, 1, 5]$	$Q = [2, 3]$
$R = [1, 6, 2, 5]$	$Q = [4, 3]$	$R = [6, 1, 5, 2]$	$Q = [3, 4]$
$R = [1, 6, 2, 5, 4, 3]$	$Q = []$	$R = [6, 1, 5, 2, 3, 4]$	$Q = []$

$$p_1 = [3, 4, 5, 2, 6, 1] \qquad\qquad p_6 = [4, 3, 2, 5, 1, 6]$$

Here the right column shows exactly the procedure that leads to the bandwidth 2 representation in Example 8.19, while the permutation on the left gives a bandwidth 3 pattern. This is the one produced by the current implementations in MATLAB and GNU Octave. Note that the matrix A is transformed into the reduced-bandwidth matrix $\tilde{A}$ as $\tilde{A} = A(p,p)$, while the graph uses the inverse permutation. That means the permutation $\tilde{p}$ such that $\tilde{p}(p) = [1, \ldots, 6]$, i.e.

Algorithm 8.5: Generic local strategy

Input: $A \in \mathbb{R}^{n \times n}$ sparse, m a metric on the nodes in $\mathcal{G}(A)$, $p = []$
Output: $p \in \mathbb{R}^n$ such that $\tilde{A} = A(p, p)$ is the reordered matrix

1 **repeat**
2   Select a node P (the pivot element) with minimal metric value $m(P)$: $p = [p, P]$;
3   Update elimination graph erasing P;
4   Update metric for all non-selected nodes;
5 **until** *all nodes selected*;

$\tilde{p}_1 = [6, 4, 1, 2, 3, 5]$, $\tilde{p}_6 = [5, 3, 2, 1, 4, 6]$.

Local heuristics

Let $A \in \mathbb{R}^{n \times n}$ sparse symmetric, $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$ the corresponding undirected graph of A and $m\colon V \to \mathbb{R}$ a metric, such that $m(i) < m(j)$ implies that vertex i is "better" than vertex j. Algorithm 8.5 presents a generic local strategy.

Remark 8.21:

- Step 4 in Algorithm 8.5 should be restricted to those nodes where m changed due to the graph update.
- The local pivot search allows combination with classic pivot strategies to improve the numerical results.

Minimum degree idea: The basic strategy behind minimum degree reordering is to choose the degree of a vertex as the metric. That means $m(i) < m(j)$ if node i has less neighbors than node j. Especially, the degrees only change for adjacent nodes of P during the elimination of P, i.e., we have a very local metric updated.

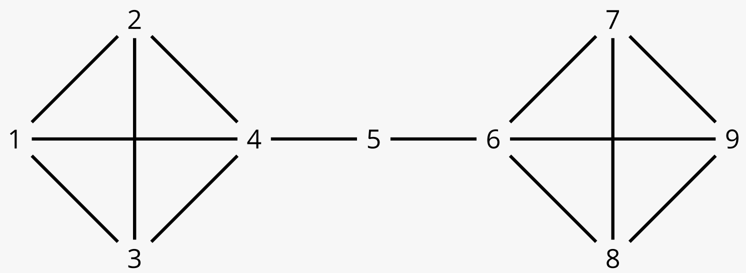
Step 3 of Algorithm 8.5 is performed as in Section 8.4.1.

Minimum degree reordering is not always optimal as we see from the following example.

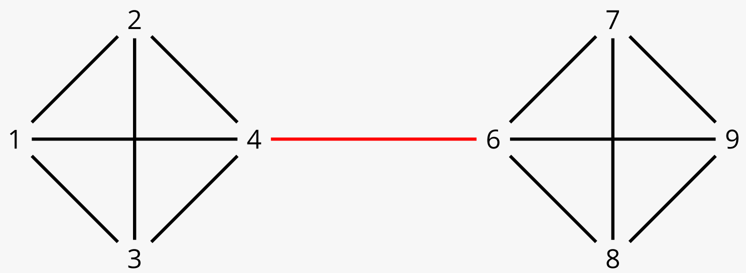
Example 8.22: We consider the following matrix $A \in \mathbb{R}^{9\times 9}$

$$A = \begin{bmatrix} 1 & * & * & * & & & & & \\ * & 2 & * & * & & & & & \\ * & * & 3 & * & & & & & \\ * & * & * & 4 & * & & & & \\ & & & * & 5 & * & & & \\ & & & & * & 6 & * & * & * \\ & & & & & * & 7 & * & * \\ & & & & & * & * & 8 & * \\ & & & & & * & * & * & 9 \end{bmatrix}$$

, for which, by construction, factorization is possible without fill-in. $\mathcal{G}(A)$ looks like this



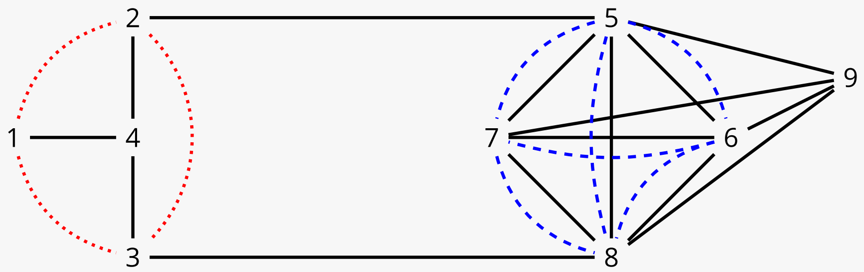
Now the minimum degree metric suggests to choose node 5 (of degree 2) for elimination, which results in:



This obviously introduces a new edge from node 4 to node 6, i.e., results in fill-in. This is still better than choosing node 4 or 6 (both degree 3), which would lead to two new edges each, i.e. more fill-in. On the other hand, all other nodes (also degree 3) could obviously be removed without causing additional edges.

All heuristic approaches to the minimum fill problem in general only produce suboptimal solutions. This is however clear, since the optimal solution is usually not accessible since it is the solution to an NP-hard problem.

Example 8.23 (minimum degree metric versus minimum fill metric)**:** The following simple graph (— edges) shows the discrepancies between minimum degree and minimum fill as metrics.



The potential fill is indicated by the colored edges. The ····· edges indicate the fill resulting from the removal of node 4. The - - - edges show that all edges that are required to preserve paths after removal of node 9 do already exist. That means, the degree and the fill measures of node 4 are both 3, while the degree of node 9 is 4, but the fill measure is 0. Below we collect a comparison of the two metrics on the entire graph.

node	degree metric value	fill metric value
1	1	0
2	2	1
3	2	1
4	3	3
5	5	4
6	4	0
7	4	0
8	5	4
9	4	0

Hybrid method and graph components

Definition 8.24 (connected)**:** In an undirected graph $\mathcal{G}$ two vertices u and v are called *connected* if $\mathcal{G}$ contains a path from u to v. Otherwise, they are called *disconnected*.

A *Graph* $\mathcal{G}$ is said to be *connected* if each pair of vertices is connected. A *connected component* is a maximal connected subgraph of $\mathcal{G}$.

That means, if u, v are vertices in $\mathcal{G}$ from different connected components, then u, v are disconnected. Thus, the corresponding degrees of freedom in the linear system are independent of each other.

Especially, reordering A corresponding to the connected components leads to a block diagonal matrix. The resulting diagonal blocks can then be treated by local strategies or dense solvers.

For general non-symmetric matrices *strongly connected components* have to be used. That means, both directed paths between two vertices need to exist. Therefore, not all diagonal blocks decouple completely, since only one direction may exist for a pair of vertices in two components. Nonetheless strongly connected components may form so-called *supernodes* that can be used to localize the memory access. This idea leads to the SuperLU algorithm and software package[2].

Sparse Matrix Vector Products and Reordering

Consider the matrix vector product of a matrix A stored in CSR format and a dense vector x.

Naively looking at the problem one might think: Even if the elements in A are scattered all over the row, in the CSR format they are stored one after the other, anyway. This would lead us to the expectation that we get no advantage due to reordering.

However, this is only half the truth. Consider an RCM reordered matrix with small bandwidth. The relevant indices corresponding to the entries are local, as well. Thus, a local portion of x is used. Additionally, the next row has a very similar set of indices containing entries. That means, in the next row product almost the entire portion of x can be reused, which leads to only little cache misses on x.

In contrast to this, scattered row entries will lead to a rather irregular and especially non-sequential access to x possibly causing lots of cache misses.

8.4.5 Related Software

- SuiteSparse (Section 6.4.3)
  - CSparse — Introductory basic direct solver library used for "The sparse backslash book" [2]
  - UMFPACK — The library behind the sparse "\" in MATLAB and the sparse direct solver in SciPy[3]
  - **A**pproximate **M**inimum **D**egree related reordering
- ITPack — see Section 6.4.4

[2] `https://portal.nersc.gov/project/sparse/superlu/`
[3] `https://www.scipy.org`

- Trilinos — see Section 6.4.5
- METIS[4] / SCOTCH[5] — 2 libraries for graph partitioning, clustering and computation of fill reducing reorderings.

Bibliography

[1] T. A. Davis, *Direct methods for sparse linear systems (lectures)*. http://www.youtube.com/playlist?list=PL5EvFKC69QIyRLFuxWRnH6hIw6e1-bBXB.

[2] T. A. Davis, *Direct Methods for Sparse Linear Systems*, no. 2 in Fundamentals of Algorithms, SIAM, Philadelphia, PA, USA, 2006, https://doi.org/10.1137/1.9780898718881.

[3] S. Parter, *The use of linear graphs in Gauss elimination*, SIAM Review, 3 (1961), pp. 119–130, https://doi.org/10.1137/1003021.

[4] D. Rose, R. Tarjan, and G. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM Journal on Computing, 5 (1976), pp. 266–283, https://doi.org/10.1137/0205021.

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2003, https://doi.org/10.1137/1.9780898718003.

[6] H. A. Van der Vorst, *Iterative Krylov Methods for Large Linear Systems*, vol. 13 of Cambridge Monographs on Applied and Computational Mathematics, Cambridge University Press, Cambridge, 2003, https://doi.org/10.1017/CBO9780511615115.

[7] M. Yannakakis, *Computing the minimum fill-in is np-complete*, SIAM Journal on Algebraic Discrete Methods, 2 (1981), pp. 77–79, https://doi.org/10.1137/0602010.

[4] https://github.com/KarypisLab/METIS
[5] https://www.labri.fr/perso/pelegrin/scotch/