FYSS5120

Efficient Numerical Programming

Department of Physics

Vesa Apaja

October 16, 2023

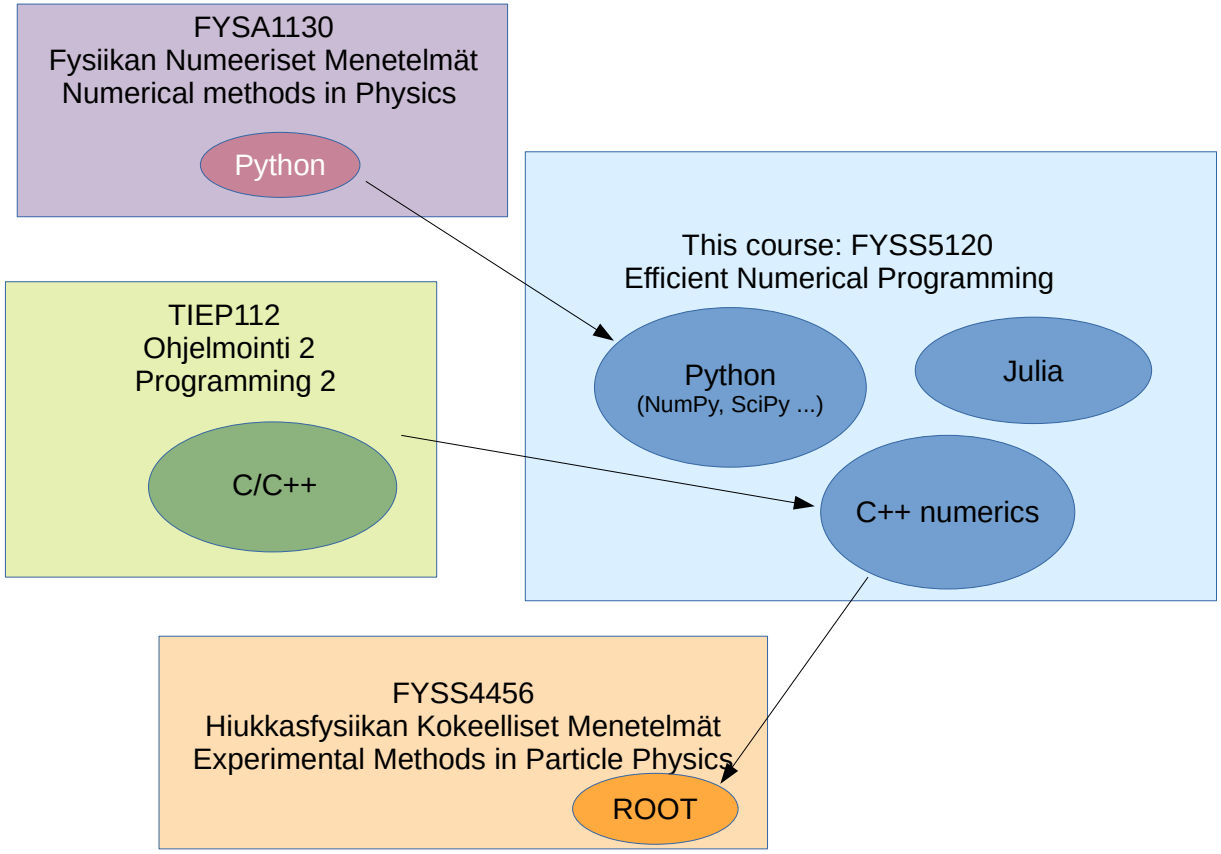
JYVÄSKYLÄN YLIOPISTO
UNIVERSITY OF JYVÄSKYLÄ

Contents

1 Course itinerary

See the course web page.

1.1 Relation to other courses in JYU



1.2 Popularity of programming languages

One answer is the Tiobe index

> *The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the best programming language or the language in which most lines of code have been written.*

Another answer is the PYPL,

> *The more a language tutorial is searched, the more popular the language is assumed to be. It is a leading indicator. The raw data comes from Google Trends.*

Count of searches of each language is presumably a good popularity indicator, as opposed to the count of existing web pages.

2 Version control using git

Many times I've made changes to a code, just to find out it stopped working. Small code improvements are the most dangerous ones. They seem benign, but can introduce a major flaw. Furthermore, they are very difficult to spot from 10000 lines of code after a few days. You need a way to get back to an older version, or at least view what you've done recently.

I'm using git (link). Let's assume the stuff you want to track is in the current directory and subdirectories `lectures/`, `pythoncodes/`, `juliacodes/`, and `c++codes/` Do this:

```
$ sudo apt-get install git # or dnf -y install git
$ git status
fatal: not a git repository (or any parent up to mount point /)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
```

This means you need to initialize git.

```
$ git init
Initialized empty Git repository in .../.git/
$ git add lectures    # add everything under subdirectory lectures
$ git add pythoncodes
$ git add juliacodes
$ git add c++codes
```

Now `git status` gives a list of new files "not staged for commit". You need to ***git commit*** them. Some files or directories are listed as "Untracked files", which won't be tracker by git, unless you `git add` them. I've added all I want, so commit:

```
$ git commit -a -m "First commit" # commit git database with a comment
```

Check status again,

```
$ git status
On branch master
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
(commit or discard the untracked or modified content in submodules)
modified:   c++codes/ParallelSTL (untracked content)
modified:   lectures/git_input.tex

Untracked files:
(use "git add <file>..." to include in what will be committed)
.f2py_f2cmap
demos/
```

This text is in file `lectures/git_input.tex`, that's why it's marked `modified`. I've also used `gitk`, a graphical git repository browser.

The difference between the committed and the current `git_input.tex` is

```
$ git diff lectures/git_input.tex
diff --git a/lectures/git_input.tex b/lectures/git_input.tex
index 188689b..772ea76 100644
--- a/lectures/git_input.tex
+++ b/lectures/git_input.tex
@@ -32,5 +32,23 @@ Now \mytt{git status} gives a list of
new files "not staged for commit". You nee
Some files or directories are listed as "Untracked files",
which won't be tracker by git, unless you \mytt{git add} them.
I've added all I want, so commit:
```

```
...
```

showing all lines I've changed since commit.

```
$ git commit -a -m "edited git_input"
$ git diff lectures/git_input.tex
(no output)
```

A simple do - undo test

Let's make a simple coding mistake and undo it, reverting to the earlier version. Store this to file `git-testing.py`

```
print('this is ok')
```

add it to git,

```
$ git add git-testing.py
```

Edit the code to

```
print('this is NOT ok')
```

Now git notices the change,

```
$ git diff  git-testing.py
diff --git a/pythoncodes/git-testing.py b/pythoncodes/git-testing.py
index 9d4fcc0..f1e7ba3 100644
--- a/pythoncodes/git-testing.py
+++ b/pythoncodes/git-testing.py
@@ -1 +1 @@
```

```
-print('this is ok')
+print('this is NOT ok')
```

Minus sign tells I removed the "ok" line, and plus that I added the "NOT ok" one. The output of `git diff` is designed to be used as a ***patch***, it's a recipe how the old version can be converted to the new one.

If I want to ***stash*** all uncommitted changes, I type `git stash`. This forgets all I've changed after previous commit. A bit dangerous, so be careful.

Git works on branches, the current one is `master`,

```
$ git  branch
* master
```

Let's create a new branch,

```
$ git checkout -b stupid
Switched to a new branch 'stupid'
```

Now I'm free to do experimenting, without messing up my old code or other stuff. Like in `pythoncodes/git-testing.py`, change content to

```
print('stupid mistake')
```

and

```
git commit -a -m ``stupid-branch-commit''
```

If I decide it was stupid, I can get back to `master`,

```
$ git checkout master
Switched to branch 'master'
$ git branch
* master
stupid
$ git branch -d stupid
error: The branch 'stupid' is not fully merged.
If you are sure you want to delete it, run 'git branch -D stupid'.
$ git branch -D stupid
Deleted branch stupid (was 5a6c607).
```

where I deleted the `stupid` branch. The error tries to warn that I haven't ***merged*** edit on branch `stupid` to branch `master`. Merging (`git merge ...`) tries to do it's best in bringing the two branches together. A quite delicate task, which often needs manual interference.

Many editors are aware of the git branch you are on. Notice, that changing branches or stashing changes really ***changes the content of your files***. If you had an editor open it may start complaining about "file xxx changed on disk, do you really want to edit the current buffer?" and upon saving it want you permission, too.

Every commit has a `commit hash`, a unique name of the commit. You'll find it by typing

```
$ git log
commit 4f81ae5c4b92c3f5cc086f506dcbf71de8a29f52 (HEAD -> master)
Author: Vesa Apaja <vesa.apaja@gmail.com>
Date:   Tue Aug 17 12:48:09 2021 +0300

stop for lunch

...
commit adc4695ad4aaa0736169d5f2c9cc5e5a6997d10a
```

```
Author: Vesa Apaja <vesa.apaja@gmail.com>
Date:   Tue Aug 17 11:46:42 2021 +0300

First commit
```

The comments (”stop for lunch” and ”First commit”) are supposed to be informative. Knowing the commit hash, you can always come back to an old commit,

```
$ git checkout <commit hash>
```

or revert to the previous commit.

I have the following alias:

```
alias gitdate='git for-each-ref --sort=committerdate refs/heads/
               --format='\''%(color: red)%(committerdate:short) %(color: cyan)%(refname:short)'\'''
```

which lists branches and their dates (with nice colors, not visible here):

```
$ gitdate
2021-08-17 master
```

This is not quite enough, but should convince you learning git is a good investment.

3 Python

3.1 About Python

Python was invented by Guido Van Rossum (1956-) in December of 1989, while working at Centrum Wiskunde & Informatica (CWI) in the Netherlands. He took some features of the ABS language, fixed some issues, and finally the language was released 1991. As you might have anticipated, Guido named the language Python after the TV show *Monty Python's Flying Circus*. He was a "benevolent dictator for life" until stepping down from the position in 2018. After retiring from Dropbox 2019, he joined Microsoft's Developer Division in 2020. Python is lead by a yearly elected steering council. Nominees are nominated by a core team member of CPython (see below) developers. The steering council appoints the Python Software Foundation leadership.

Python development relies on ***Python Enhancement Proposals (PEPs)***, which, in some form, may be invoked to a new Python version. For example, the style of Python programs is described in PEP 8 .

These notes are about Python 3, more details in The Python Standard Library . Python is an interpreted language, the interpreter is the program `python`, which runs the so-called `CPython` interpreter. Interpreting instructions takes longer than executing compiled code (as in C, Java etc.). Python is a dynamically typed language, meaning the types of objects are determined by the Python interpreter. The lack of typing prevents code optimization. Still, Python is ever so popular, and after these lectures you may join the growing Python community.

3.2 Installation

3.2.1 Linux

Here and ever after `$` stands for the (bash) shell prompt.

1. Install Python - you need root or sudo rights -
   - Fedora:[1]

[1] This also installs plenty of frequently used packages to `/usr/lib/python3.10/site-packages/`. Caveat: Packages get eventually outdated and updates have to be done as root or sudo.

```
$ sudo dnf install python3
```

- Debian, Ubuntu (untested, don't have those OS's):

```
$ sudo apt-get install python3
```

The command `python` should call `python3`,

```
$ which python
/usr/bin/python
$ ll /usr/bin/python
lrwxrwxrwx 1 root root 9 Jun 10 03:31 /usr/bin/python -> ./python3
```

so `python` is a soft link to `python3`.

3.2.2 Windows 10

Type "python" or "python3" in the command prompt, and you will be directed to Microsoft Store. [2]

Remark: It wasn't that smooth in my Win10, though:

1. Typed `python3` and got just the common error message *python is not recognized as ....*
2. Clicked the Microsoft Store icon and found python 3.9. Installed it.
3. Typed `python3` and got the error message `missing msvcp140.dll`. Fathomed I'm missing a MS Visual C package.
4. Found `vc_redist.x64.exe` from a MS web site. Download and run. Now `python3` starts, exit with `exit()` or with ctrl-Z.

---

[2] devblogs.microsoft.com: *Who put Python in the Windows 10 May 2019 Update?*

3.2.3 Python 3 vs. Python 2

There's old Python 2 code around, so I point out a few differences. More of them in
whats new in 3.0 .

Python has a compatibility module `future`, which aims to make Python code run in both Python 2 and Python 3. Python 3 differs from Python 2 most notably in that `print` is a function, not a statement:

```
print "oh dear"    # Python 2
```

```
print("oh dear")  # Python 3
```

so typically a compatible code has the line

```
from __future__ import print_function
```

Remark: Some codes also have the lines

```
from __future__ import absolute_import
from __future__ import division
from __future__ import unicode_literals
```

In Python 2 `map()` and `filter()` return lists, in Python 3 they return ***iterators*** (we'l get to them later). Also `reduce()` was moved to `functools.reduce()`.

One change not spotted by Python interpreters is the behaviour of division,

```
#Python 2: integer divided by integer is an integer
>>> 5/2
2
```

```
#Python 3: integer divided by integer is a float
>>> 5/2
2.5
>>> 6/2
3.0
>>> type(6/2)
<class 'float'>
```

Aaron Meurer shows some significant features in Python 3 . Such as this one,

```
def naivesum(N):
    # Naively sum the first N integers
    A = 0
    for i in range(N + 1):
        A += i
    return A
```

Python 3 can compute `naivesum(1000000000)` but don't try it in Python 2 because `range()` is expanded by Python 2 interpreter, taking many gigabytes of memory. Python 2 `xrange()` is can handle these large ranges.

3.3 Python package managers

Python programming relies on existing code base, which are ***Python packages*** imported to code as ***modules***. For numerical Python programs you need at least NumPy, SciPy, and Matplotlib. Python packages are stored in ***repositories***, the most commonly used is the ***Python Package Index, PyPI***. Package managers are programs that download and install packages from a repository, and, upon request, update packages to newer versions.

An important package standard is ***wheel***,[3] which makes package installation a lot faster and more reliable. Package with a wheel, suffix `.whl`, don't need a working C compiler to use a `C extension`, a Python module written in C for faster execution. If a package doesn't offer a `.whl`, installation takes the slower `setup.py` route and possibly needs a compiler.

3.3.1 `pip` package manager

`pip` is the builtin Python package manager, it installs packages from PyPI. There are two ways to use it, either directly

```
$ pip install packagename
```

or, loading `pip` as a module,[4]

```
$ python -m pip install packagename
```

For example, NumPy, SciPy, and Matplotlib can be installed typing either

```
$ python -m pip install numpy --user
$ python -m pip install scipy --user
$ python -m pip install matplotlib --user
```

or

```
$ pip install numpy --user
$ pip install scipy --user
$ pip install matplotlib --user
```

[3] Wheel in PEP427 . Wheels intend to replace eggs, the previous package standard.

[4] I prefer the `pip` module because in some machines `pip` points to the Python 2 installer `pip2` as opposed to `pip3`.

With the `--user` option modules will be installed in your home directory, under `$HOME/.local`. Recent Python installations do this automaticallyif you don't have root priviliges. You can install all three packages at once,

```
$ pip install numpy scipy matplotlib --user
```

> Avoid installing Python packages as root/admin.
> I will drop the `--user` option from now on, assuming you are not root.

Be warned, that the Python *system package* usually contains many common Python packages. Once you're done installing python3 to your OS, you may already have a system-wide installation of NumPy, Matplotlib and friends. Sooner or later these cause annoying mismatches in package versions.

Testing: start the Python interpreter and import a module. If this works, everything is fine:

```
$ python
>>> import numpy as np
>>> (press cntrl-D to quit)
```

or on the command line

```
$ python -c "import numpy as np"
```

Now and then `pip` suggests you should upgrade to the latest version,

```
$ python -m pip install upgrade pip
# or
$ pip install upgrade pip
```

3.3.2 Conda and Anaconda

You may have heard about the package manager conda :

> *Package, dependency and environment management for any language-Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, FORTRAN, and more.*
> *Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux.*

Conda installs and manages conda *binary* packages from the Anaconda repository or from the Anaconda Cloud. Anaconda is

> *Anaconda is a distribution of the Python and R programming languages for scientific computing that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS.*

Anaconda installer can be downloaded from the anaconda distribution . You need some disk space, the full 2023 installer for linux is 1015.6 MB. Run it,

```
$ sh Anaconda3-2023.07-2-Linux-x86_64.sh
```

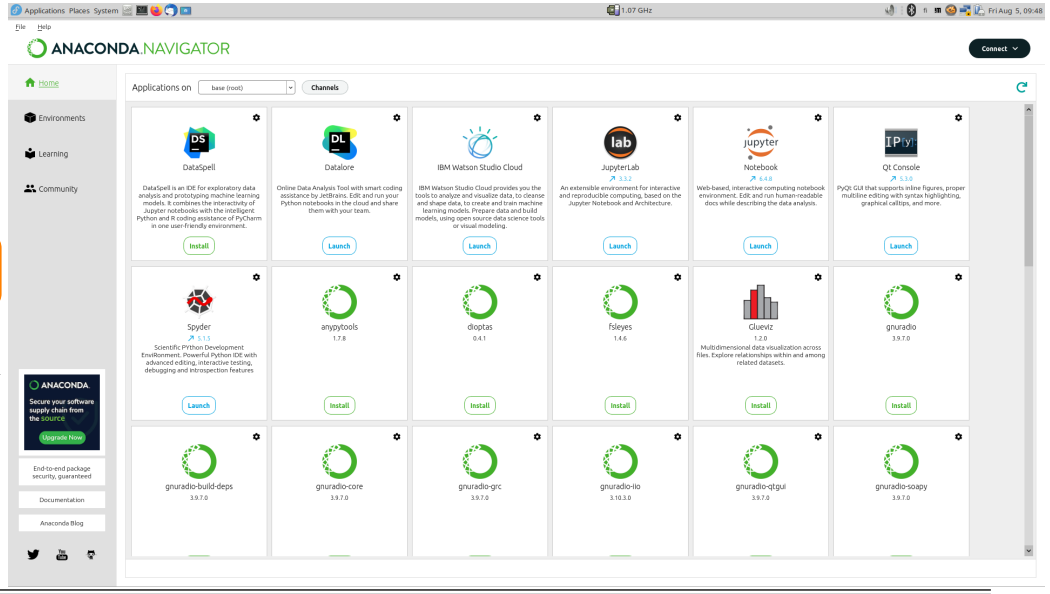
and agree with the license terms. The default installation goes to `$HOME/anaconda3`. As of August 2023, Anaconda supports Python 3.11. I tried upgrading an existing anaconda3 with the `-u` option, and it failed; in my case the fix was setting permissions:

```
chmod +w -R $HOME/anaconda
```

Anaconda comes with a GUI,

```
$ anaconda-navigator
```

which offers installers for Spyder, Jupyter, and more.



Remark: After installation of conda/anaconda you may see the prompt change to `(base)` when you open a new shell. This means the base environment is autoactivated. You can change that by typing

```
$ conda config --set auto_activate_base false
```

This writes to `~/.condarc`,

```
$ cat ~/.condarc
auto_activate_base: false
```

More about Conda at conda@docs.conda.io . We'll return to conda in the section 3.4.2, Virtual Environments.

Conda channels

Conda downloads packages from sources called ***channels***, the default channel is `https://repo.anaconda.com/pkgs/`. There

are hundreds of channels but I'm using mostly just three: `default`, `conda-forge`, and `intel`. A channel can be ***activated***,

```
$ conda config --add channels conda-forge
$ conda config --add channels intel
$ conda config --show channels
channels:
- intel
- conda-forge
- defaults
$ conda config --get channels
--add channels 'defaults'   # lowest priority
--add channels 'conda-forge'
--add channels 'intel'   # highest priority
```

or you can pick up a code from a channel without activation,

```
$ conda install gpaw -c conda-forge
```

3.4 Spyder, the scientific Python IDE

From the Spyder web page,

> *Spyder is a free and open source scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts. it features a unique combination of the advanced editing, analysis, debugging, and profiling functionality of a comprehensive development tool with the data exploration, interactive execution, deep inspection, and beautiful visualization capabilities of a scientific package.*

Spyder contains a variety of scientific Python packages, including NumPy, SciPy, Matplotlib, pandas, IPython, SymPy and Cython. To see what's installed, type `pip list` or `conda list` in the Spyder console. You can install Spyder from the anaconda-navigator, or from the download link on the web page. Most Linux releases have it, too,

```
$ dnf search spyder
Last metadata expiration check: 0:17:06 ago on Fri 05 Aug 2022 04:29:11 PM EEST.
==== Name & Summary Matched: spyder ==========
python3-pyls-spyder.noarch : Spyder extensions for the python-language-server
python3-spyder-kernels.noarch : Jupyter kernels for the Spyder console
===== Name Matched: spyder ==================
python3-spyder.noarch : Scientific Python Development Environment
```

Remark: Spyder from Anaconda complained about `libstdc++.so.6: version 'GLIBCXX_3.4.29' not found`";. Turned out Anaconda `libstdc++.so.6` library was wrong; fixed the problem by taking it away with

```
$ mv ~/anaconda3/lib/libstdc++.so.6{,.bak}
```

3.4.1 Where does Python search for modules?

The error message

```
>>> import nnn
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'nnn'
```

means you haven't installed a module `nnn`, or it's not in the Python search path.

The search path can be printed with the command[5]

[5]Or use the hard-to-remember `python -c "import sys; print('\n '.join(sys.path))"`. Copy-pasting this may get the quotation marks wrong.

```
$ python -m site
```

The output is something like

```
  sys.path = [
    '/wrk/vap/texst/opetus/FYSS5120_Efficient_Numerical_Programming/lectures',
    '/usr/lib64/python311.zip',
    '/usr/lib64/python3.11',
    '/usr/lib64/python3.11/lib-dynload',
    '/home/vap/.local/lib/python3.11/site-packages',
    '/usr/lib64/python3.11/site-packages',
    '/usr/lib/python3.11/site-packages',
]
USER_BASE: '/home/vap/.local' (exists)
USER_SITE: '/home/vap/.local/lib/python3.11/site-packages' (exists)
ENABLE_USER_SITE: True
```

The environment variable `PYTHONPATH` may also be used to tell where to look for packages, by pointing it to a `site-packages` in a non-standard location. Try not to set `PYTHONPATH`, it may cause confusion.

How to exclude packages under `.local`
One way is to define an environment variable,

```
$ export PYTHONNOUSERSITE=1
```

and *in this shell* Python won't use anything under `$HOME/.local`.

3.4.2 Python (virtual) environments

There are plenty of tutorials online, you can start from venv . A Python environment is dedicated for your project, where life is almost detached from outside world and code run on fixed module versions.

Basic Python virtual environment

The vanilla Python environment is create with the command

```
$ python -m venv path_to_new_virtual_environment
```

will create a directory for the environment. For example

```
$ python -m venv virtual
$ ls virtual/
bin  include  lib  lib64  pyvenv.cfg
$ source  virtual/bin/activate
(virtual) [vap@suikka lectures]$ python
Python 3.10.5 (main, Jun  9 2022, 00:00:00) [GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
(hit ctrl-D)
$ deactivate
```

The fact that the virtual environment can't see the packages installed outside the sandbox is intentional, it's the whole point of the virtual environment. The packages will install under the virtual environment directory, in my example to `./virtual/lib/python3.10/site-packages/`.

`pip freeze`

If you want to copy the environment,

```
(old env)$ pip freeze > requirements.txt
```

and in the other environment

```
(new env)$ pip install -r requirements.txt
```

and you'll get the same packages and versions of them.

Conda environments

The downside of virtual environments created with `python -m venv` is the dependence on system Python. Conda environments are truly standalone virtual environments, and you can create one with any Python version.

Let's create, activate, test, and deactivate a Python 3.10 environment `p3.10`.

```
$ conda create -n p3.10 python=3.10
$ conda activate p3.10
(p3.10) [vap@suikka lectures]$ python
Python 3.10.5 | packaged by conda-forge | (main, Jun 14 2022, 07:06:46) [GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
(press cntr-D)
(p3.10) [vap@suikka lectures]$ conda deactivate
```

If you happen to need Python 3.9, use something like

```
$ conda create -n p3.9 python=3.9
$ conda activate p3.9
(p3.9) $ python
Python 3.9.13 | packaged by conda-forge | (main, May 27 2022, 16:58:50)
[GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Nothing prevents from creating a conda environment while in another conda environment. The new environment is independent of the active environment, and you can hop around between environments,

```
$ conda activate p3.10
(p3.10) $ conda create -n p3.5 python=3.5
(p3.10) $ conda activate p3.5
(p3.5) $ python
Python 3.5.5 | packaged by conda-forge | (default, Jul 23 2018, 23:45:43)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Apparently conda environments are not virtual sandboxes.

Conda environments are stored by default in `$HOME/anaconda3/envs/`, So far I have

```
$ ls ~/anaconda3/envs/
p3.10 p3.5 p3.9
```

I will use conda environments later in section 7.3 to test NumPy with a few BLAS libraries. Removing an environment is done with

```
$ conda env remove -n p3.10
```

3.4.3 Spyder, iPython, Jupyter Notebook, and Jupyter Lab

Spyder is an open-source cross-platform IDE (Integrated Desktop Environment), with editing, shell, graphics output, and online help on the same platform. **iPython** is the "official", more interactive command line Python interpreter, described at length in discussion @Stackoverflow . Many use iPython with the **Jupyter notebook**, installation is

```
$ python -m pip install ipython --user
$ python -m pip install jupyter --user
```

Basic ipython has syntax highlighting and suggests line completion. Jupyter notebook starts on your web browser,[6]

```
$ jupyter notebook
```

[6]Starting Jupyter using `ipython notebook` is deprecated.

Remark: The output is quite verbose,

```
$ jupyter notebook
[I 08:56:45.052 NotebookApp] Serving notebooks from local directory: /home/vap
[I 08:56:45.052 NotebookApp] The Jupyter Notebook is running at:
[I 08:56:45.052 NotebookApp] http://localhost:8888/?token=12ad24d43bbf91289564eb38bbebe403e4a74ad8694dec55
[I 08:56:45.052 NotebookApp]  or http://127.0.0.1:8888/?token=12ad24d43bbf91289564eb38bbebe403e4a74ad8694dec55
[I 08:56:45.052 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 08:56:45.085 NotebookApp]

To access the notebook, open this file in a browser:
file:///home/vap/.local/share/jupyter/runtime/nbserver-9296-open.html
Or copy and paste one of these URLs:
http://localhost:8888/?token=12ad24d43bbf91289564eb38bbebe403e4a74ad8694dec55
or http://127.0.0.1:8888/?token=12ad24d43bbf91289564eb38bbebe403e4a74ad8694dec55
```

In the notebook, find a small button *new* somewhere in the upper right corner. There are other ways to use Jupyter, as you find typing `jupyter --help`. The graphics-capable qtconsole is also nice,

```
$ jupyter qtconsole
```

Jupyter Lab

A more advanced environment from Jupyter Notebook is Jupyter Lab, `jupyterlab`,

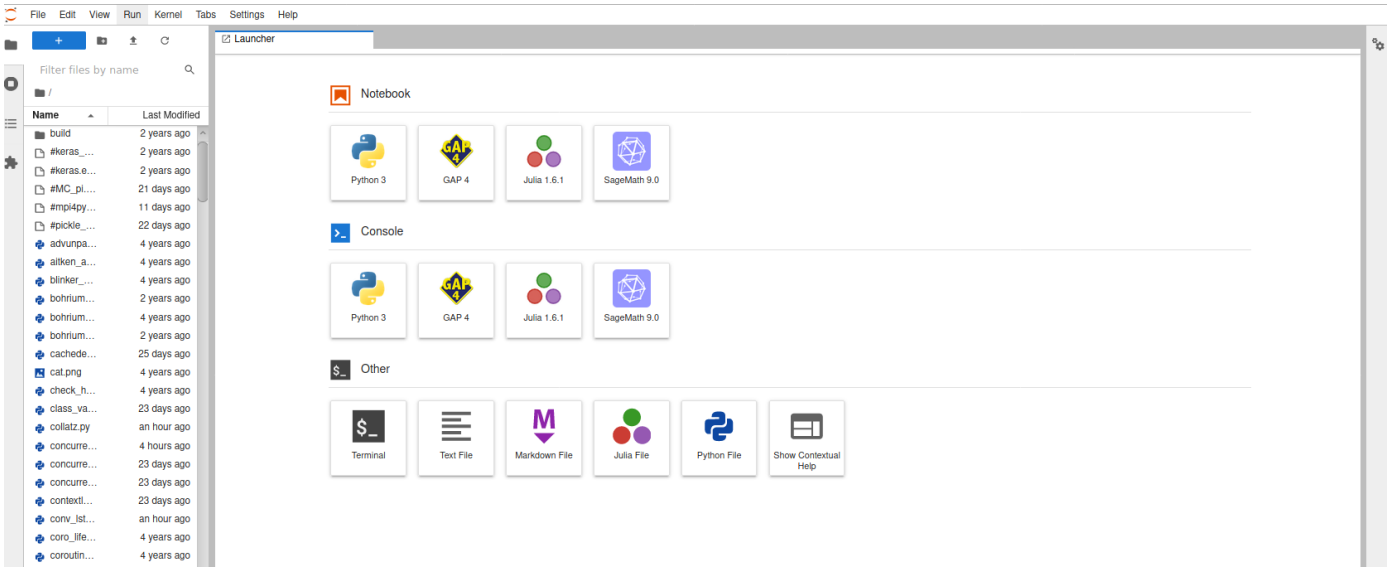
```
$ python -m pip install jupyterlab
```

If you work rather in a `conda` environment, you can

```
$ conda create -n condaenv
$ conda activate condaenv
(condaenv) $ conda install jupyterlab
```

```
(condaenv) $ jupyter lab
```

and you get a browser window,



Jupyter notebooks have a file suffix `.ipynb`. They can be converted to Python scripts using

```
$ jupyter nbconvert --to script your_notebook.ipynb
```

The output `your_notebook.py` can be run using `ipython`. If you want to show the notebook output, you can convert it to html,

```
$ jupyter nbconvert --to html your_notebook.ipynb
```

The output is `your_notebook.html`.

3.5 Updating Python packages

3.5.1 Updating Python packages using `pip` package manager

A single package update as non-root user:

```
$ python -m pip install --upgrade packagename --user
```

or

```
$ pip install --upgrade packagename --user
```

How about updating *all* packages? One possibility is

```
$ pip freeze > requirements.txt
$ pip install -r requirements.txt --upgrade
```

Remark: Other ways: This one-liner might be useful:

```
$ pip list -o --format columns|  cut -d' ' -f1|xargs -n1 pip install -U
```

Here `pip list -o` lists outdated packages, the rest parses relevant information from the list and feeds it to pip update `pip install -U`. An interactive package update can be done using

```
$ pip install pip-review
$ pip-review --local --interactive
```

3.5.2 Updating Python packages using `conda` package manager

A single package update is

```
$ conda update packagename
```

and to update all installed packages,

```
$ conda update --all
```

If these complain about

> NoBaseEnvironmentError: This conda installation has no default base environment. Use 'conda create' to create new environments and 'conda activate' to activate environments.

just do as you're told, and activate one of your conda environments. The reason you're not in the base environment is that you may have applied the configuration change

```
$ conda config --set auto_activate_base false
```

You can activate the `base` environment and do the updates,

```
$ conda activate base
(base) $ conda update --all
```

Troubleshooting

If the `You're not in the base environment` error still pops up, you probably have `CONDA_PREFIX` set, and pointing to a wrong place. In my case, I had installed `conda` from a linux repo, and had

```
$ echo $CONDA_PREFIX
/usr
$ conda info -all
...
CONDA_PREFIX = /usr
```

which is wrong because then `conda update --all` looks for the file `/usr/conda-meta/history`, which is not there, and causes `conda` to think your `base` environment is inactive. [7] If you got `conda` by running the downloaded install script of `anaconda3` as descibed earlier, you should have empty `CONDA_PREFIX`,

```
$ echo $CONDA_PREFIX
```

and now there is a file `$HOME/anaconda3/conda-meta/history`

```
$ conda update --all
```

works as expected.

[7] The reason was left open in the discussion anaconda-wont-update-no-default-base-environment-error @Stackoverflow. I pinpointed the problem by running `strace conda update --all`.

3.6 Python file extentions

- `.py` are Python source files
- `.pyc` are compiled bytecode, produced by `python`, stored in directory `__pycache__`
- `.pyo` are optimized compiled bytecode, produced by `python -O` of by `python -OO`

Importing a bytecode module is faster, hence *only modules that have been imported get a bytecode file* [8]. You can inspect the bytecode of module `short` by using the disassembler:

```
import dis  # Python disassembler
import short # this will cause short.py to be bytecoded
dis.dis(short)
```

where `short.py` could be

```
def testing():
    print('hello')
```

Bytecode can give hints how to make faster code. For example, why creating a dictionary with `{}` is a tiny bit faster than with `dict()`? From bytecode,

```
>>> import dis
>>> dis.dis("{}")
  1           0 BUILD_MAP                0
              2 RETURN_VALUE
>>> dis.dis("dict()")
  1           0 LOAD_NAME                0 (dict)
              2 CALL_FUNCTION            0
              4 RETURN_VALUE
```

[8] Manually `python /usr/lib64/python3.x/py_compile.py short.py` compiles to `__pycache__/short.cpython-3x.pyc`.

3.7 Timing and watching memory usage

3.7.1 Timing with the `timeit` module

Code snippets can be timed with `timeit` , for example

```
python -m timeit '"-".join(str(n) for n in range(100))'
```

A common usage of `timeit` in a module is

```
from timeit import Timer as T
def fun():
    ...
print(T(fun).repeat()) # default repeat() executes fun 1000000 times and repeats it 3 times
# output: list of the three cumulative times, the minimum value is most informative
```

and system time with `time.time()` or interpreter time `time.clock()`) or `time.process_time`

```
from time import process_time as T
start = T()
myfunction()
print('Executed in',T()-start,'seconds.')
```

One way to take a peak at the bottlenecks of a code was suggested by Mike Dunlavey @Stackoverflow :

If your Python code is running slow, hit `ctrl-C` and the current call stack is printed.

Repeat this a few times and you get an idea what is going on.

Detailed profiling of short-running Python codes
A native Python profiler is the module `profile`. Profiling an entire code, [9]

```
$ python -m profile -s cumtime myprogram.py
```

sorted in decreasing cumulative time. In Python console,

```
>>> import profile
>>> import myprogram
>>> profile.run(myprogram.main())  # assuming the program is run executing main()
```

Detailed profiling of long-running Python codes
The module `cProfile` - notice the capital P - is a C extension and has smaller overhead than `profile`. Usage is similar to `profile`,

```
$ python -m cProfile -s cumtime myprogram.py
```

or

```
>>> import cProfile
>>> import myprogram
>>> cProfile.run('myprogram.main()')
```

I'll return to timing Python later, and set up timing of functions as a ***decorator*** in section 3.15.

[9] For details, see profile @python.org.

3.7.2 Timing and watching memory usage in iPython

Timing can be done with the iPython magic `%timeit`, Start the iPython interpreter, and test timing of the potential energy calculation in `numerics/potential_energy.py`. You can also inspect memory usage, but for that you need a module,

```
$ python -m pip install ipython_memory_usage
```

Usage is in iPython console

```
In [1]: import ipython_memory_usage.ipython_memory_usage as ipymem
In [2]: ipymem.start_watching_memory()
# run the code
```

(Sorry, this is a picture.)

```
[vap@suikka lectures]$ ipython
Python 3.10.5 (main, Jun  9 2022, 00:00:00) [GCC 11.3.1 20220421 (Red Hat 11.3.1-2)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.26.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import numpy as np
   ...: import ipython_memory_usage.ipython_memory_usage as ipymem
   ...: from potential_energy import PotentialEnergy, PotentialEnergyLoop

In [2]: x = 100*np.random.random((1000,3))

In [3]: ipymem.start_watching_memory()
In [3] used 1.0391 MiB RAM in 15.67s, peaked 0.00 MiB above current, total RAM usage 59.44 MiB

In [4]: %timeit PotentialEnergy(x)
45.7 ms ± 465 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [4] used 0.9180 MiB RAM in 3.83s, peaked 53.25 MiB above current, total RAM usage 60.36 MiB

In [5]: %timeit PotentialEnergyLoop(x)
2.34 s ± 6.94 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [5] used 0.0391 MiB RAM in 18.87s, peaked 0.00 MiB above current, total RAM usage 60.39 MiB
```

With 10000 particles in 3D the broadcasting method peaked 5378.75 MiB. NumPy broadcasting is faster but memory intensive.

3.8 General advice to speed up Python

- Favor ***local variables*** over global ones
- Represent numerical data using ***NumPy***
- ***Vectorize***, don't do item by item. Vectorized function have deserved the name *Universal function (ufunc)*. As a downside, vectorization consumes memory.
- ***Keep data contiguous in memory***: Store data in memory in the order it's most often accessed.
- Testing membership in a ***dictionary*** is faster than in a list; dictionaries have hash tables of their members, so finding an entry in a dictionary is an O(1) operation. Finding and entry in a list is very slow!
- Testing membership in a ***set*** is as fast as testing membership in a dictionary.
- ***Cache*** results of slow, repeated operations.
- ***List comprehensions*** are usually faster than `for` loops. The interpreter does a better job with such a limited list comprehension loop compared with a general `for` loop. Speed gain may not be that big, though.
- *Anything* that avoids `for` loops is good. See ***Vectorize***.
- Try ***built-in methods*** first. Some data type have their own optimized methods for common tasks, such as sorting.

3.9 List comprehensions

List comprehensions are often compact one-liners that are frequently used in list manipulations and filtering. Besides, a `for` loop has a side effect, namely a loop variable may overwrite a global variable or create a new one,

```
x = 1
squares = []
for x in range(10):
    squares.append(x**2)
# here x=9
```

The value of (global) `x` was overwritten. This won't happen if you use a list comprehension,

```
x = 1
squares = [x**2 for x in range(10)]
# here x=1
```

Some like to use an underscore as a "dummy variable",

```
squares = [_**2 for _ in range(10)]
```

More list comprehensions:

```
# combine two lists to a list of tuples with non-equal integers:
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
# [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```
# using list comprehension as a filter to find common members
a = [1,2,3,5,7,9]
b = [2,3,5,6,7,8]
print ([x for x in a if x in b])
# [2, 3, 5, 7]
```

IMHO filtering with a list comprehension is nicer than using a lambda. [10]

```
a = [1,2,3,5,7,9]
b = [2,3,5,6,7,8]
print(list(filter(lambda x: x in a, b)))
# need list(); Python 3 has filter() as <filter object at 0x7f848d729d68>
```

Set comprehension works similarly,

```
# set comprehension
a = {x for x in 'some lecture notes' if x not in 'abcdefg'}
print(a)
# {'t', 'l', 'o', 's', 'n', ' ', 'm', 'r', 'u'}
# notice: no duplicates
```

[10] I haven't got a chapter about Python lambda function. I just casually throw them in and assume you find them plausible.

How fast is list comprehension?

Sebastian Witowski @switowski.com timed filtering certain numbers from a list of million numbers with a for loop, a list comprehension, and a `filter()` function of Python 3.8. The timings were 65.4 ms, 44.5 ms, and 104 ms, respectively. In August 2023 I tried Python 3.11, got timings 25.3 ms for loops, 21.9 ms list comprehension, and 96.5 ms for filter. For loops are obviously getting faster in new Python, but the main assett is that the list comprehension filter is a one-liner:

```
result = [number for number in MILLION_NUMBERS if not number % 2]
```

Remark: How does the filter work? Since `number % 2` is zero for even numbers, and zero is `False`, then `if not number % 2` is `True` for even numbers, thus the filter picks all even numbers. A short test using NumPy (this uses the legacy `np.random.randint()`, see section 3.23),

```
>>> import numpy as np
>>> MILLION_NUMBERS = np.random.randint(0,10000000,size=1000000)
>>> result = [number for number in MILLION_NUMBERS if not number % 2]
>>> result
[2393076, 6394306, 5062132, 2072312, 393650, 7329480, 3207092, ...
4187270, 4375422, 7783536, 8514914]
```

Nitpicking: What makes list comprehension sometimes faster than for-loops?

Both for-loops and list comprehensions have loops, and it's not really looping that makes any difference. Neither is there any more translated C-code in comprehensions. What happens inside a loop can make a small, insignificant difference.

Let's look at the bytecode for two functions ; run `appending_loop_vs_list_comprehension.py`: [11] Appending to a list translates to

```
       >>   36 FOR_ITER                23 (to 84)
            38 STORE_FAST                1 (i)

 6          40 LOAD_FAST                 0 (l)
            42 LOAD_METHOD               1 (append)          <--- THIS IS SLOW
            64 LOAD_FAST                 1 (i)
            66 PRECALL                   1
            70 CALL                      1
            80 POP_TOP
            82 JUMP_BACKWARD            24 (to 36)

 5     >>   84 LOAD_CONST                0 (None)
```

while list comprehension translates to

```
      >>    6 FOR_ITER                 4 (to 16)
            8 STORE_FAST                1 (i)
           10 LOAD_FAST                 1 (i)
           12 LIST_APPEND               2                  <--- THIS IS FASTER
           14 JUMP_BACKWARD             5 (to 6)
      >>   16 RETURN_VALUE
```

In the comprehension, the optimized `LIST_APPEND` bytecode is utilized to add elements to the list. This is more efficient than repeatedly loading and using the `append()` method. Fine, but it's all boils down to how well the present day Python interpreter optimizes the code. Adding 5 numbers to a list with `append()` method can be a bit faster, and even appending 10000 numbers to a list takes in my desktop 193 μs vs. comprehension 146 μs, and for a million numbers it takes 30.3 ms vs. 26.5 ms. It's not a game changer. However, in section 7.7 I'm talking about NumPy broadcasting, and that makes a difference.

[11] See why-is-a-list-comprehension-so-much-faster-than-appending-to-a-list @Stackoverflow.

3.10 String concatenation

stringconcat.py

```
'''
Strings are immutable, so changing them must be done by creation of a new string.
Two ways to create a random string
'''
import random
import string

N=10
# long version
s = ""
for i in range(N):
    s += random.choice(string.ascii_lowercase)
print(s)

# short version since python 3.6:
s = ''.join(random.choices(string.ascii_lowercase, k=N))
# older python:
#s = ''.join(random.choice(string.ascii_lowercase) for _ in range(N))
print(s)
```

3.11 Counting hashable objects

What are hashable objects? Any `object` you can type `hash(object)`, which is the same as `print(object.__hash())`. Obviously hashable objects have the attribute `__hash__()`. Not much wiser? All immutable objects (integers, strings, tuples, ...) are hashable because they have a permanent id and a `__hash__()` for their entire lifetime. Apart from immutable objects, you can always define a class with the method `__hash__()`,

```
class Foo:
   def __hash__(self):
      return 42

print(hash(Foo))
# output possibly 1452873573337
```

Interestingly, the hash value is not 42. Also functions are hashable,

```
def f(x):
  return 1

print(hash(f))
# output possibly 1439470782047
```

Enough talk, let's get back to the topic of counting hashable objects. The `collections` package contains very fast routines, such as `Counter`, a dict subclass container for counting in list, tuple, dictionary, *etc.*.

```
from collections import Counter
name='Vesa Apaja'
VA_counter = Counter(name)
print(VA_counter)
```

```
# Counter({'a': 3, 'V': 1, 'e': 1, 's': 1, ' ': 1, 'A': 1, 'p': 1, 'j': 1})
print(VA_counter['a'])
# 3
VA_counter.most_common()
# [('a', 3), ('V', 1), ('e', 1), ('s', 1), (' ', 1), ('A', 1), ('p', 1), ('j', 1)]
VA_counter.update('aa') # the opposite is subtract
print(VA_counter['a'])
# 5
dic = {'1':'aaa','2':100,'3':11}
dic_counter = Counter(dic)
print(dic_counter)
# Counter({'1': 'aaa', '2': 100, '3': '11'})
dic2 = {'a':11,'b':100,'c':11,'d':200}
print(Counter(dic2))
# Counter({'d': 200, 'b': 100, 'a': 11, 'c': 11})
```

The method `most_common()` returns a list of tuples (element,frequency) arranges according to frequency in decreasing order. The method `elements()` returns an iterator,

```
print(VA_counter.elements())
# <itertools.chain object at 0x147bac9dc790>
print(list(VA_counter.elements()))
# ['V', 'e', 's', 'a', 'a', 'a', 'a', 'a', ' ', 'A', 'p', 'j']
VA_counter.clear()  # empty the counter
print(VA_counter)
# Counter()
```

Comment: `dequeu` in `collections` package is nice piece of code. Also C++ has the `deque` container.

3.12 Sorting

There are two ways to sort:

1. `a = sorted(b)` gives `a` as sorted `b`. Objects `a` and `b` are separate entities.
2. `b.sort()` sorts `b` ***in place***

After `b.sort()` there is no "original" `b` around , sorting overwrites `b`:

```
b = [1,4,6,3,1,2,6,7]
b.sort()
print(b) # output is [1, 1, 2, 3, 4, 6, 6, 7]
```

3.12.1 Sorting by a key

We can use the `key` attribute in sorting,[12]

[12]Historical note: In old Python the idiom was DSU - decorate, sort, undecorate, known also as the Schwarzian transformation. For curiosity I have added the code `dsu_example.py`.

key_sort_example.py

```
import random
import string

# list of random ascii letters of random length
N = 10
lenlist = (int(random.random()*20)+1 for i in range(N))

gibberish = [''.join(random.choices(string.ascii_lowercase, k=kk)) for kk in lenlist]
print('list to sort:\n',gibberish)
# sorting:
gibberish.sort(key=len)
#
print('list, sorted in increasing length:\n',gibberish)
```

Here `gibberish` is a list of strings, and strings have a `__len__` attribute. Sorting compares `__len__`'s.

3.12.2 Sorting by a key in a given element

Frequently you want to sort by some element, for example sort by the third element in a list of tuples. There's a way and a faster way,

key_sort_example2.py

```
# list of tuples to sort
data = [(x,y,z) for x in [1,2,3] for y in [4,5,2] for z in [5,4,6,8,3,1,9]]
print('unsorted:\n',data)

# one way to sort by 3rd element in tuple (that is, by values in z)
sdata = sorted(data,key=lambda d: d[2])
print('sorted\n',sdata)

# a bit faster way to sort by 3rd element in tuple
from operator import itemgetter
ssdata = sorted(data,key=itemgetter(2))
print('sorted\n',ssdata)
```

Here speed differences are insignificant but something to keep in mind if the list is very long.

3.12.3 NumPy Sorting

NumPy method `numpy.ndarray.sort` sorts the array in place, while `numpy.sort()` returns a sorted copy of an array. These correspond to `.sort()` method and `sorted()` function in basic Python. One of the sorting arguments is `axis` which is defined for arrays with more than one dimension:

2D Array:
axis=0 runs vertically downwards across rows
axis=1 runs horizontally across columns

numpy_sort_example.py

```
import numpy as np
origdata = np.array([[5,2,1],[4,6,7],[3,2,8]])
data = np.copy(origdata)
print(data)
# [[5 2 1]
#  [4 6 7]
#  [3 2 8]]
data.sort() # sort by last axis (axis=-1, default); now same as axis=1
print('\n',data)
# [[1 2 5]
#  [4 6 7]
#  [2 3 8]]
data = np.copy(origdata)
data.sort(axis=0)
print('\n',data)
#[[3 2 1]
# [4 2 7]
# [5 6 8]]
```

If a 2D array represents a matrix, ***sorting along an axis mixes both row vectors and column vectors***. This is unwanted if the matrix columns are, for example, eigenvectors. If you need to ***sort an array by an arbitrary column***, take a look at discussions @Stackoverlow . There you find a particularly neat solution suggested by Steve Tjoa,

numpy_argsort_example.py

```
import numpy as np
origdata = np.array([[5,2,1],[4,6,7],[3,2,8]])
data = np.copy(origdata)
print('\n',data)
# [[5 2 1]
#  [4 6 7]
#  [3 2 8]]
data = data[data[:,0].argsort()] # sort by 1st column
print('\n',data)
# [[3 2 8]
#  [4 6 7]
#  [5 2 1]]
data = np.copy(origdata)
data = data[data[:,1].argsort()] # sort by 2nd column
print('\n',data)
# [[5 2 1]
#  [3 2 8]
#  [4 6 7]]
```

Let's look at this piece by piece.

1. `data[:,1]` is `array([2, 6, 2])`, second elements in `data`.
2. `data[:,1].argsort()` is `array([0, 2, 1])`, the indices of sorted second elements.
3. `data[data[:,1].argsort()]` pick rows of data in the order of the index array.

This sorting keeps the *row* vectors intact. In demos I'll figure out how to sort so that column vectors remain intact. Don't sneak in anything like `data[:,1].sort()` because that sorts `data` in-place, and your `data` array is messed up ever since. You can convince yourself that this really is the case:

```
>>> data = np.array([[5,2,1],[4,6,7],[3,2,8]])
>>> data[:,1].sort()
>>> data
array([[5, 2, 1],           <- this vector was in data
       [4, 2, 7],           <- this vector was not there
       [3, 6, 8]])          <- this vector was not there

>>> data = np.array([[5,2,1],[4,6,7],[3,2,8]])
>>> data[data[:,1].argsort()]
array([[5, 2, 1],           <- this vector was in data
       [3, 2, 8],           <- this vector was in data
       [4, 6, 7]])          <- this vector was in data
```

3.13 Arbitrary precision calculations

3.13.1 Large integers

Arbitrarily large integers are built-in,

```
>>> i = 9385429038590283589028941894901849018490184901849081904819028491283 4901
>>> i
9385429038590283589028941894901849018490184901849081904819028491283 4901
```

```
>>> i*2
18770858077180567178057883789803698036980369816380963805698256698 02
```

diophantine.py

```
# Solution to the Diophantine equation
# a**3+b**3+c**3 = k, a,b,c and k are integers, k=1...100
# Tough ones: k=33 and k=42
# Andrew Booker, University of Bristol, and Andrew Sutherland, MIT
# Solution found 6.9.2019
# using Charity Engine, about 500 000 home PC network

a=-80538738812075974
b=80435758145817515
c=12602123297335631
print(a**3+b**3+c**3)
```

3.13.2 Long floats with mpmath

An arbitrary precision floating point arithmetics is provided by mpmath .

Example: Riemann hypothesis

The trivial zeros of the Riemann zeta function are real, $\zeta(s) = 0$ for $s = -2, -4, -6, \ldots$. According to the Riemann hypothesis, all nontrivial zeros are on the critical line $s = \frac{1}{2} + i\,t$. The following code uses `findroot()` to numerically find a root near given point, while `mpmath.zetazero()` is a built-in list of zeros.

```
>>> from mpmath import *
>>> findroot(zeta, 0.5+14j) # solve for zero near 0.5+14j
mpc(real='0.5', imag='14.134725141734694')
```

```
>>> zetazero(1) # built-in list of zeros
mpc(real='0.5', imag='14.134725141734695')
>>> zetazero(1000)
mpc(real='0.5', imag='1419.4224809459956')
```

Example: Internal precision in numerical integration

mpmath_integral.py

```
# 2D integral using quadrature
from mpmath import mp  # context object
print(mp)
f = lambda x, y: 1/(1-x**2 * y**2)
res = mp.quad(f, [0, 1], [0, 1])
print(res)          # 1.23370055013617
mp.nprint(res,100) # 1.2337005501361697490381175157381221652030944824218 75
# more precision:
mp.prec = 120
mp.dps=100
print(mp)
# calculate with better precision:
res=mp.quad(f, [0, 1], [0, 1])
mp.nprint(res,100)
#1.2337005501361698273543113749845188919142124 25
#   9050988283016686720275056028024006553752783892670132 32
mp.nprint(mp.pi**2/8,100)
#1.2337005501361698273543113749845188919142124 25
#   9050988283016686720275056028024006553752216754648190 29
```

By default, `mp.prec = 53` (double precision).

3.14 Advanced unpacking

3.14.1 A word about function arguments

Functions can have two kinds of arguments,

- ***positional arguments a.k.a non-keyword arguments*** are identified based on their position in the argument list. For example

```
def f(x,y,z):
    ...
f(1,2,4) # calls f with x=1, y=2, and z=4
```

- ***keyword arguments*** are (key, value) pairs (dictionaries),

```
def f(x,y,z):
    ...
f(1,z=4,y=2) # calls f with x=1, y=2, and z=4
```

  Here `z=4` uses key `z` and value 4.

3.14.2 The beauty of the extended call syntax: `*args` and `**kwargs`

Great examples in what-does-the-star-operator-mean @Stackoverflow . Highlights:

- `*arg` unpacks `arg` to positional arguments (see code `advunpacking.py`)

```
# this function works with any number of arguments (as long as they can be printed)
def f(*args):
   for x in args:
      print(x)
```

The function call `f(1,4,6,7,8,3,2,'4','131234134')` works just fine.

Remark: I often collect (x, y)-data like this,

```
res = []
for i in range(10):
   #... some computations that give x and y
   res.append([x,y])
```

I then want separate lists for x and y values. Zipping with `zip` should do that,

```
res = [[1,5],[2,3]]
x,y = zip(res)  # not quite what I want, gives x = ([1, 5],) and y = ([2, 3],)
x,y = zip(*res) # right! *res is [1, 5] [2, 3] and so x = [1,2] and y = [5,3]
```

On the above-mentioned web page Donald Miner gives an example of an ***object factory***, a special case of the so-called ***factory methods***,

```
def make_car(*args):
   return Car(*args)
```

that creates objects from the class `Car`. Now `make_car('red', 'bmw', '335ix')` creates `Car('red', 'bmw', '335ix')`. It so simple because `*args` can hold any number of arguments. An object factory is a ***design pattern*** for creation of objects with a common interface.

- **kwargs unpacks keyword arguments (dictionaries) kwarg to positional arguments. Consider

```
def g(**kwargs):
   for key, value in kwargs.items():
      print(f'key {key} value {value}')
```

  The function calls g(x=2,y=13) or g(x=2,y=13,z=33) work fine.

Using *args to absord extra arguments

Let's take a sneak peek at the function fun1() soon to be used in a Fibonacci numbers code. Here the dictionary _cache is not supposed to be given any values as arguments,

```
def fib1(n,*args,_cache={}):
    print(n,_cache)
fib1(10)    # output: 10, {}
fib1(10,12) # output: 10, {}  <= _cache has not changed
```

Without the *args you will try to feed _cache a value and you get

```
def fib1(n,_cache={}):
    print(n,_cache)
fib1(10)     # output: 10, {}
fib1(10,12)  # output: 10, 12 <= _cache has changed
```

Here *args absorbs all positional arguments after n, so essentially (n,*args) means "n and other positional arguments". All keywork arguments, such as _cache in this example, are left untouched unless explicitly set.

With *args you are safe from accidentally setting keyword arguments:

```
def f(a,*args,test=False):
    if test:
        print('Since you insist, I kill the engine')
    else:
    print('a = ',a)

f(1,2)
# safe, output is a = 1. The extra 2 is absorbed by *args and discarded
```

Here you can also use plain `f(a,*,test=False)`. If you leave the absorbing argument out you beg for trouble:

```
def f(a,test=False):
    if test:
        print('Since you insist, I kill the engine')
    else:
        print('a = ',a)

f(1,2)
# unsafe!
# output: Since you insist, I kill the engine
```

Later in these notes we write ***wrapper functions***, which collect all possible positional and keyword arguments, packed as `args` and `kwargs`,

```
def wrapper(*args, **kwargs):
   ...something...
```

You can extract parts easily

```
a = range(10)
first, *mid, last = a  # or : first,*_,last = a
print(first,last)
#0 9
# another way
first, last = a[0], a[-1]
```

3.15 Decorators

Decorators are wrappers that can be used to add extra functionality to a function, without modifying the body of the function. You don't want to write an address on a gift, instead you wrap the gift and write the address on the wrapper. Since all functions have arguments and keywords, *any* function can be wrapped like this:

```
def universal_decorator(any_function):
    def wrapper(*args, **kwargs):
       return any_function(*args, **kwargs)
    return wrapper
```

This needs editing to actually do something, apart from calling `any_function()`. The `*args` passes arguments (if any) and the `**kwargs` passes keyword arguments (if any) to the actual function. Here I added printing of the function name:

deco.py

```
def mydecorator(any_function):
    def wrapper(*args, **kwargs):
        print('entering ',any_function.__name__)
        return any_function(*args, **kwargs)
    return wrapper

@mydecorator
def func(x):
    return x**2


if __name__=='__main__':
    print(func(10))
    # entering  func
    # 100
```

Nothing prevents re-decorating a decorated function.

3.15.1 Python preprocessing and adding code for debugging

Personally, I would like to have the possibility to turn decorators on and off for debugging purposes or just for extra verbosity. Something along the lines

```
# THIS DOES NOT WORK AS INTENDED
#ifdef DEBUG
@trackcalls
```

```
#endif
def function_under_scrutiny():
```

and execute `python -DDEBUG testdebug.py` if I want `trackcalls` to be effective, else not. However, `python` won't preprocess `#ifdef` and `#endif`. There are Python preprocessors around, but I would hesitate on adding another interpreter to the mixture.

Typically debugging is done using the structure

debug_or_not.py

```
if __debug__:
    print("debugging")
else:
    print("not debugging")
```

and

```
$ python debug_or_not.py
# debugging
$ python -O debug_or_not.py
# not debugging
```

The option `-O` turns on basic optimization, sets `__debug__` to false and also ignores all `assert` statements. However, it's a bit lengthy to use an `if-else` structure.

Another way to proceed is given by Kundor @Stackoverflow : attach to the wrapper both the decorated function and the undecorated one,

deco_with_unwrapped.py

```
def trackcalls(fun):
    def wrapper(*args, **kwargs):
        print('executing',fun.__name__)
        return fun(*args, **kwargs)
    wrapper.nodebug = fun
    return wrapper

@trackcalls
def job():
    return 'job done'

print(job())            # this tracks
print(job.nodebug())    # this doesn't
```

Not quite an on/off decorator, the decorator here is always on but with added functionality. Why does it work? Functions in Python are ***objects***, and objects can return objects. Be aware that a decorator takes at least some time to interpret, and that a decorated function can't be undecorated.

3.15.2 Turning a decorator on/off using `python -O`

I haven't seen this approach suggested elsewhere but it's so simple it can't be my brainchild. Remember how `__debug__` code was thrown away by `python -O`. Define two decorators, the actual decorator and a decorator that just returns the original function,

debug_deco.py

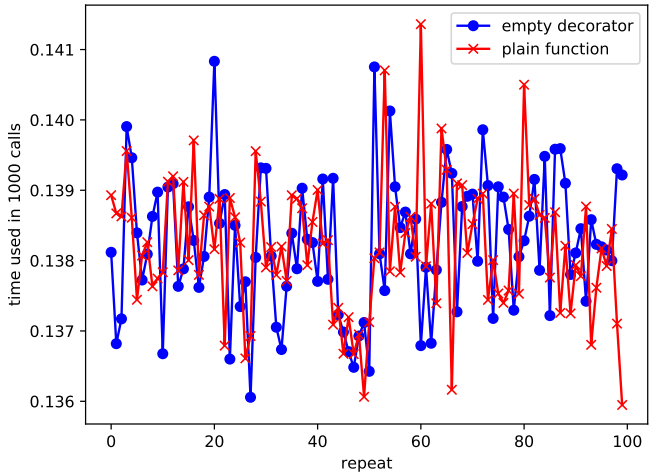
```
if __debug__:
    def trackcalls(fun):
        def wrapper(*args, **kwargs):
            print('executing',fun.__name__)
            return fun(*args, **kwargs)
        return wrapper
else:
    def trackcalls(fun):
        return fun

@trackcalls
def job():
    return 'job done'

print(job())
```

Result:

```
$ python debug_deco.py
executing job
job done
$ python -O debug_deco.py
job done
```

Add a counter of function calls and timing measurement and you have a code that profiles some predefined parts of itself unless run as `python -O`. The option `-OO` omits also docstrings and saves some space.

There seems to be no speed penalty for using the extra empty decorator vs. the direct function call.



Remark: Typing `python` takes so long, so I have aliases (defined in file .bashrc),

```
$ alias  p{,d}
alias p='python3 -O'
alias pd='python3'
```

A college used to have aliases

```
alias meak=make
alias maek=make
alias meka=make
alias mkea=make
alias mkae=make
```

because he would hastily type any of the variants wanting to run `make`. I'm no better, I have

```
alias mm=make
alias mmm='make clean;make'
```

Decorator classes

So far I've been talking about decorator functions, but ***decorator classes*** are even more versatile.

decorator_class.py

```
class MyDecorator:
    def __init__(self, f):
        self.f = f
        print('entering ',f.__name__)

    def __call__(self,*args,**kwargs):
        print('decorated with MyDecorator')
        return self.f(*args,**kwargs)

@MyDecorator
def func(x):
    return x**2

print(func(10))
# entering  func
# decorating with MyDecorator
# 100
```

The magic method `__call__` makes the class name callable, as `MyDecorator()`. Here `func(10)` decorated with `MyDecorator()` means the same as

```
MyDecorator(func)(10) # for *undecorated* func(); Notice: not the same as MyDecorator(func(10))
```

Decorators can be stacked, this will be executed in the order `deco1` calling `deco2` calling `func`.

```
@deco1
@deco2
def func(x):
```

3.15.3 `dataclass` decorator

Take a look at the code

pointclass.py

```
from dataclasses import dataclass

@dataclass
class PointClass:
    x : float
    y : float
    outlier : bool = False

point1 = PointClass(1.1,2.2,True)
point2 = PointClass(1,4)
print(point1)
print(point2)
#PointClass(x=1.1, y=2.2, outlier=True)
#PointClass(x=1, y=4, outlier=False)
```

The module `dataclasses` has the decorator `dataclass` that automatically adds an appropriate `__init__()`, saving you the trouble of typing it yourself,

```
# not needed, this was done by dataclass
def __init__(self, x : float, y : float, outlier : bool = False):
    self.x = x
    self.y = y
    self.outlier = outlier
```

The decorator `dataclass` also added `__repr__()`. All this was done by `dataclass` based on the data in the visible `PointClass`. Sometimes `dataclass` can be persuaded to add a `__hash__()` magic method.

3.15.4 Cache decorator

If a time-consuming function `f(x)` is frequently called with the same argument `xi`, it may be wise to store the value `f(xi)`. The storage is a ***cache***, a ***dictionary*** of `{key:value}` pairs. You can write your own dictionary or use a ***cache decorator***. The latter is a single line instruction to provide a function with a cache of wanted size. Recursively computed Fibonacci numbers are the canonical example.

cachedecorator.py

```
'''
Fibonacci number recursive computation
Cache decorator from functools; performance profiling
'''
from functools import cache
import profile

class Fun:
    """
    >>> Fun.fib0(10)
    55
    >>> Fun.fib1(10)
    55
    >>> Fun.fib2(10)
    55
    """
    # no cache
    def fib0(n):
        if n < 2:
            return n
        return Fun.fib0(n-1) + Fun.fib0(n-2)

    # self-made cache
    def fib1(n,*,_cache={}):  # local _cache dictionary used here
        if n in _cache:
            return _cache[n]
        if n < 2:
            return n
        result = Fun.fib1(n-1) + Fun.fib1(n-2)
        _cache[n] = result           # Store result in _cache
        return result

    # cache decorator; Python 3.9: @cache gives @lru_cache(maxsize=None)
    @cache
    def fib2(n):
        if n < 2:
            return n
        return Fun.fib2(n-1) + Fun.fib2(n-2)

if __name__ == '__main__':
    # test functionality; to see output, run with python cachedecorator.py -v
    if __debug__:
        import doctest
        doctest.testmod()

    funs = [Fun.fib0, Fun.fib1, Fun.fib2]

    n = 25  # not too large, please :^)
    for fun in funs:
        print('call function',fun.__name__)
        profile.run('fun(n)')
    print('cache information for fib2 with cache decorator:',Fun.fib2.cache_info())
```

I got `242791 function calls (7 primitive calls) in 0.760 seconds`

```
55 function calls (7 primitive calls) in 0.000 seconds
32 function calls (7 primitive calls) in 0.000 seconds

cache information for fib2 with cache decorator:  CacheInfo(hits=23, misses=26, maxsize=None, currsize=26)
```

You can tune the cache size by looking at the hit/miss info. The cache decorator is simple to invoke,

```
from functools import cache

@cache
def f(x):
... # time consuming calculation, same input x used frequently
```

As indicated in a comment, since Python 3.9 `@cache` is a thin wrapper to `@lru_cache(maxsize=None)`. `@lru_cache` has a few options that may improve performance in some cases.

Remark: Just for fun:
David Beazley writes a code using Python `lambda` (lambda calculus, see video DB @Youtube). He has `lambda` written as λ, which looks much nicer.

DB_Y_combinator.py

```
#
# Idea stolen from David Beazley
#
# Y-combinator (Haskell B. Curry) to do recursion
#
Y = lambda f:(lambda x: f(lambda z:x(x)(z)))(lambda x: f(lambda z: x(x)(z)))
#
# Factorial
#
R = lambda f: lambda n: 1 if n==0 else n*f(n-1)
fact = Y(R)
for i in range(11):
    print(f'fact({i}) =', fact(i))
#
# Fibonacci sequence
#
R = lambda f: lambda n: 1 if n<=2 else f(n-1)+f(n-2)
fib = Y(R)
for i in range(11):
    print(f'fib({i}) =', fib(i))
```

3.15.5 Decorators with arguments

Decorators with arguments won't increase speed but can improve readability and save *your* time. These ***decorator patterns*** can change the functioning of a decorator dynamically. A good intro is in decorators @Codementor . There you find a nice example of a decorator that checks for permission to execute a function:

```
@requires_permission('administrator')
def delete_user(iUserId):
    # code ...

@requires_permission('premium_member')
def premium_checkpoint():
    # code ...
```

This is achieved with ***a function that returns a decorator***,

```
def requires_permission(sPermission):
  def decorator(fn):
      def decorated(*args,**kwargs):
          lPermissions = get_permissions(current_user_id())
          if sPermission in lPermissions:
              return fn(*args,**kwargs)
          raise Exception("permission denied")
      return decorated
  return decorator
```

No need to set up several decorators, such as `@requires_admin`, `@requires_premium_member` ... .
A working example is in file `decorator_with_arguments.py`.

3.16 Iterables, generators and `yield`

If you can write

```
for x in xs:
```

then `xs` is ***iterable***, *i.e.* can be iterated element by element. For example, the list `[1,2,3]` is iterable. Iterables may be very large, so if you only need a few elements, then storing them all in memory is inefficient. Instead, you define a special type of iterator, a ***generator***, which *generates* the elements on demand. For example, the list comprehension

```
a = [x**2 for x in range(10)]
print(a)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
for elem in a: print(elem)
# 0
# 1
# 2
# ...
```

creates the list `a`, an iterable fully stored in memory, and `print(a)` prints the whole list. This is simple to convert to a ***generator expression***, just replace `[]`'s with `()`'s:

```
g = (x**2 for x in range(10)) # generator expression
print(g)
# <generator object <genexpr> at 0x7ff5565420f8>
for elem in g: print(elem)
# 0
# 1
# 2
```

```
# ...
```

creates a ***generator expression*** g, therefore `print(g)` won't print the elements but says that g is a generator object expression stored somewhere in memory. The `for` loop asks the generator to generate elements, in this case all of them.

A generator yields next output only on demand.

The `return` returns whatever you ask it to return, possibly something fully expanded in memory. Replace `return` with `yield` and the function returns a generator.

yield_example.py

```
# Functions returning a generator or a generator expression
def square1(n):
    for x in range(n):
        yield from [x**2] # yield makes this return a generator

def square2(n):
    return (x**2 for x in range(n))  # (x**2...) is a generator expression, return it

print(square1(6))
print(list(square1(6)))
#<generator object square1 at 0x7f6a2e11c9e8>
#[0, 1, 4, 9, 16, 25]
print(square2(6))
print(list(square2(6)))
#<generator object square2.<locals>.<genexpr> at 0x7f6a2e11c9e8>
#[0, 1, 4, 9, 16, 25]
```

Generators use less memory because you compute only what you need.
A generator function's body won't execute until you ask it to.

For example, in `list(g(2))` it's the call to function `list()` that tells `g` to generate element 2.
One more quirk, notice how there was no explicit `yield` in

```
def square2(n):
    return (x**2 for x in range(n))  # generator expression
```

generator expressions are objects, and so they can be passed around and also `return`'ed.

A generator function remembers the state of it's local variables. Once you call a generator function it won't simply exit, it's put on hold. www.programiz.com gives the following example:

yield_next.py

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed last')
    yield n

a = my_gen()
print(next(a))
print(next(a))
print(next(a))
print(next(a))
'''
This is printed first
1
This is printed second
2
This is printed last
3
Traceback (most recent call last):
  File "yield_next.py", line 19, in <module>
    print(next(a))
          ^^^^^^^
StopIteration
'''
```

This instantiates a generator `my_gen()` called `a`, and iterates a few times using `next()`. The last `next(a)` gives the error `StopIteration` because there are no more `yield`s in the generator – we have ***exhausted the generator*** [13]. Each `yield` has only one number to yield, so successive `next()`'s move down the `yield`s until there are none left.

[13] Also `for`-loops use `next()` to iterate.

3.16.1 Generator for watching a file

A generator is an excellent choise if you want to follow and process the latest line appearing on a file. The file could be accumulating log data, stock markets, simulation results, or electricity spot prices. I show two possibilities:

- If you are working in a bash shell, the command

  ```
  $ tail -f -n 1 filename
  ```

  will put the last line on console (option `-n 1` picks one line at a time), so you basically want to run this command from Python and analyze the output line. This is done in the code `shell_watching_data.py`. The code uses `subprocess` to start `tail -f -n 1 filename` on the background and sends the output line for further analysis.

- The sample code `generator_watching_data.py` uses only Python file reading.

The function `watch_file()` is quite generic. The reason why the sample codes look a bit odd is that, for testing, the same script writes data and watches for file updates.

Remark: If watching a file is all you need, then in bash shell

```
$ watch filename
```

will show the contents, updated once in a second.

3.16.2 Generator pipelines

Generator pipelines are high-performance, memory efficient, and modular tools

They are Unix-like, consider, for example, the following command:

```
$ ls -la|grep *py|more
```

Example: Read data from file and process it

The data file has columns index, value, value. Let's find all even index y-values, add them together and print the sum. Write each sub-task as a generator function and ***pipeline*** them:[14]

pipeline.py

```
# A pipeline of generator functions

# write a data file "data.tmp" for testing
filename = 'data.tmp'
try:
    # does the data file exist?
    with open(filename,'r') as file:
        pass
except:
    # doesn't exist, create it
    import numpy as np
    with open(filename,'w') as file:
        for i in range(20):
            file.write(f"{i:<5} {np.random.random():<8.3f} {np.random.random():<8.3f}\n")

with open(filename,'r') as file:
    lines = (line for line in file)
    evenlines = (line for line in lines if int(line.split()[0])%2 == 0)
    ycol  = (float(line.split()[2]) for line in evenlines)
    print("sum(y;even index)=",sum(ycol))
```

- Testing if a file exists can be done using the `os` module, the `pathlib` module and many more. I prefer to `try` opening and closing the file, if it fails the file doesn't exist, so I create it. The power of `try-except` is strong.

[14] Warning: this uses the legacy `np.random.random()`, see section 3.23.

- The generators `lines`, `evenlines`, and `ycol` are just waiting for activation, so walk past them and look at the last line of code.
- The function call `sum(ycol)` asks the generator `ycol` to generate a value. `ycol` asks the generator `evenlines` to generate a line with even index. Finally, `evenlines` asks the generator `lines` to read a line from file.
- String `line` is split to columns using the method `line.split()`. This isn't picky about the data file format, now the assumed field separator is space. Avoid fixed indices, such as `line[13:16]`, it's too prone to errors.
- You can debug generators by printing intermediate values, but be aware that ***adding intermediate `yield`s breaks up generator pipelines*** - a leaking pipeline. Printing a generator is safe but not very informative,

```
print(evenlines)
# <generator object <genexpr> at 0x7f05f88697d8>
```

but for testing you have to ask the generator to actually *yield* values,

```
print(list(evenlines))
# ['    0   0.23  0.972\n', '    2  0.771 0.00342\n', '  ... ]
```

This, however, exhausts the generator, so all subsequent attempts to ask `evenlines` to generate more values fails. The pipeline is broken and silently fails, with

```
# output: sum(y;even index)= 0
```

After `print(list(evenlines))`, there are no more `evenlines` to generate, no values to sum, hence the sum is zero. Most generators can be exhausted, a feature to keep in mind. Generators do ***lazy evaluation***: they won't move a finger until demanded – and they are *so* lazy that they evaluate once and only once.

3.17 The versatile underscore

Underscore can mean a few things depending on the context,

- Underscore hints that this is supposed to be an internal variable,

```
def fib1(n, *args, _cache={})
    ...something...
```

- Underscore marks ignored value or values,

```
first,_,last = [1,2,3] # ignore 2
a = range(10)
first,*_,last = a # advanced unpacking, clearly discards mid values
```

  The last example can, of course, be written also as `first,last = a[0],a[-1]` .

- In the Python interpreter underscore can also refer to the previous value,

```
>>> a = 2352635251234
>>> a
2352635251234
>>> _
2352635251234
>>> _**3
13021583502190843901122159705126080904
>>>
```

Underscores have their own life in Python names, and a quite a vivid life it is.

3.18 Underscores in Python names

PEP 8 gives recommendations about Python names. Particularly intriquing things may happen if a class attribute has two undercores, namely

underscores.py

```
class Foo:
    def _bar1():
        print("this is class Foo's _bar1() \n")
    def __bar2():
        print("this is class Foo's __bar2()\n")
    def _test():
        Foo.__bar2()

# these work:
Foo._bar1()
Foo._test()
# this doesn't work; raises AttributeError
# Foo.__bar2()
# but this does:
Foo._Foo__bar2() # Python name mangling
```

Python name mangling hides names with leading double underscores a bit deeper, in order to avoid unintended use outside the class. Inside the class `Foo.__bar2()` is correct but not outside. The method is still accessible as `Foo._Foo__bar2()`, so it's hidden but not private.

3.18.1 Magic methods

Magic methods have leading and trailing double underscores (hence called also "dunder methods"). The magic method `__init__()` is executed when a class is instantiated,

init_example.py

```
class MyClass:
    def __init__(self,value):
        print('Initializing to x =',value)
        self.x = value # This x is an instance variable
    def do_things(self,y):
        print('doing things with x =',self.x,'and y =',y)

# instantiate an instance of class MyClass
my = MyClass(10)      # this calls __init__
my.do_things(20)
# Initializing to x = 10
# doing things with x = 10 and y = 20
myone = MyClass(1)     # calls __init__
myone.__init__(2) # calls __init__ directly
myone.do_things(20)
# Initializing to x = 1
# Initializing to x = 2
# doing things with x = 2 and y = 20
# doing things with x = 10 and y = 20
my.do_things(20)
# doing things with x = 1 and y = 20
```

Notice how there was no name mangling for `__init__`. In this example `x` is an ***instance variable***, meaning it's bound to a specific instance of the class. Every instance of the class `MyClass` has it's own `x`.

Magic methods enhance readability

No matter how ugly and complicated the classes may be, the bread-and-butter operations on class instances should be simple and plausible. Now suppose it makes sense to add and compare two instances of `WaveClass` like this:

wave_add1.py

```
class WaveClass:
    def __init__(self,frequency=0,amplitude=0):
        self.frequency = frequency
        self.amplitude = amplitude
    def add(self,other):
        frequency = (self.frequency + other.frequency)/2
        amplitude = self.amplitude + other.amplitude
        return WaveClass(frequency,amplitude)
    def gt(self,other):
        return self.amplitude>other.amplitude

w1 = WaveClass(0.1,2.0)
w2 = WaveClass(0.3,1.0)
w3 = WaveClass(0.23,3.0)
w4 = WaveClass(0.12,4.0)

w12 = w1.add(w2)
print('combined wave 12:',w12.frequency,w12.amplitude)
print('wave 2 higher than wave 3:',w2.gt(w3))
w1234 = w1.add(w2).add(w3).add(w4)
print('combined wave 1234:',w1234)
print('combined wave 1234:',w1234.frequency,w1234.amplitude)
"""
combined wave 12: 0.2 3.0
wave 2 higher than wave 3: False
combined wave 1234: <__main__.Wave object at 0x14db9afff190>
combined wave 1234: 0.1675 10.0
"""
```

I'm not complaining about `WaveClass` *per se*, since I won't often look at it. But lines such as

```
w1234 = w1.add(w2).add(w3).add(w4)
```

could infest my daily code and they appear unnecessarily ugly. I would prefer to see sums in the form

```
w1234 = w1+w2+w3+w4
```

and this is where magic methods come into play.

wave_add2.py

```
class WaveClass:
    def __init__(self,frequency=0,amplitude=0):
        self.frequency = frequency
        self.amplitude = amplitude
    def __add__(self,other):
        frequency = (self.frequency + other.frequency)/2
        amplitude = self.amplitude + other.amplitude
        return WaveClass(frequency,amplitude)
    def __gt__(self,other):
        return self.amplitude > other.amplitude
    def __eq__(self, other):
        return self.frequency == other.frequency and self.amplitude==other.amplitude
    def __str__(self):
        return f'{self.__class__.__name__}({self.frequency:<.3f}, {self.amplitude:.3f})' # note: 3 decimals
    def __repr__(self):
        return  f'{self.__class__.__name__}({self.frequency},{self.amplitude})'


w1 = WaveClass(0.1,2.0)
w2 = WaveClass(0.3,1.0)
w3 = WaveClass(0.23,3.0)
w4 = WaveClass(0.12,4.0)

print('w1+w2 = ', w1+w2)
print('w1+w2+w3+w4:',w1+w2+w3+w4)
print('wave 2 higher than wave 3:',w2>w3)
print('wave 2 lower than wave 3:',w2<w3)
print('repr(w1) = ',repr(w1))
print('eval(repr(w1))==w1 is ',eval(repr(w1))==w1)
"""
w1+w2 =  Wave(0.200, 3.000)
w1+w2+w3+w4: Wave(0.168, 10.000)
wave 2 higher than wave 3: False
wave 2 lower than wave 3: True
repr(w1) =  Wave(0.1,2.0)
eval(repr(w1))==w1 is  True
"""
```

The magic method `__add__` tells how the + operator acts on `Waves`. As the wave class example shows, `__gt__` definition automatically defines also `__lt__`. More magic methods are listed in Python reference. The way `__add__` is applied with the

+-operator is ***operator overloading***, and we'll see how hte same idea is implemented in C++ (section 25).

Remark: The magic method `__str__` is applied with calls to `print()` and `format()`. A close relative to `__str__` is `__repr__`. The goal of `__repr__` is to be unambiguous and `__str__` is to be readable.[15] `__repr__` is typically used for debugging and it represents all relevant information about the object that could be needed to reconstruct it, while `__str__` shows some human-readable information. Python has no idea how to print an instance without neither `__str__` nor `__repr__`. One popular sanity check is to compare if an object matches it's representation,

```
eval(repr(object)) == object   # should be True
```

This requires that `__eq__` is defined; Most builtins and my `WaveClass` pass the test.[a] The canonical example is

```
import datetime
now = datetime.datetime.now()
print(str(now))
# 2022-08-08 14:31:47.266444
print(now)
# 2022-08-08 14:31:47.266444
print (repr(now))
# datetime.datetime(2022, 8, 8, 14, 31, 47, 266444)
eval(repr(now)) == now
# True
```

[a] Without `__eq__`, the interpreter would go down comparing the *references* of the two distinct objects `eval(repr(object))` and `object`, which obviously aren't equal. You need to explicitly tell what "equal" means.

3.18.2 Context managers

A context manager typically makes sure that if you start something, you properly finish it. If you `open` a file you may want to close it when you're done, without explicitly calling `close`. Consider opening and reading a file using the ***built-in context manager***,

[15] See Discussion @Stackoverflow.

```
with open('./short.py', 'r') as f:
  f.read()
```

This will open, read, *and* close the file. A context manager interface has two magic methods

1. `__enter__` is called when interpreter meets the keyword `with`
2. `__exit__` is called when the context finishes

If you wish, provide `__enter__` and `__exit__`, and you have coded your own context manager. In the next example you want to make sure there *is* a "close" when the job is done,

```
from contextlib import closing
import urllib.request

with closing(urllib.request.urlopen('http://google.com')) as page:
    htmlpage=page.readlines()
#print(htmlpage)
```

This reads the whole page to `htmlpage` and calls `page.close()` even in case of error. Additional timeout can be added to make sure the operation ends if the connection hangs.

Generators in context managers

Generators in context managers need special attention because of lazy evaluation. Consider this example [16]

[16] See Context managers @Wiki for details.

```
with open('./short.py') as f:
    lines = (line for line in f) # generator
list(lines)  # fails, ValueError: I/O operation on closed file.
```

The last line `list(lines)` asks the generator `lines` to generate lines, but `lines` tries to read from file `f` that was already closed when the context manager `__exit__`:ed. The generator object `lines` is there (you can call `print(lines)`) but it can't generate anything any more. Generators in context managers are possible, but you need to release resources explicitly.

generator_context.py

```
def linereader(infile):
    with open(infile) as f:
        for line in f:
            yield line

from contextlib import closing
with closing(linereader('./short.py')) as lines:
    print(list(lines))
```

Example of a context manager

This context manager makes sure your thesis is sent to your thesis supervisor for inspection:

context_manager.py

```
class PhD:
    def __init__(self, thesis):
        self.thesis = 'Harry Potter and '+thesis
        print(f'__init__: Start writing thesis called "{self.thesis}"')

    def __enter__(self):
        print(f'__enter__: Read articles for my thesis.')
        return self

    def __exit__(self, exception_type, exception_value, traceback):
        if exception_type:
            print(f'exception type: {exception_type}')
            print(f'exception value: {exception_value}')
            print(f'traceback: {traceback}')
            print(f'__exit__: I wont send an empty thesis to my supervisor')
        else:
            print(f'__exit__: Send thesis "{self.thesis}" to my supervisor')

    def serious_work(self):
        print("I'm taking this seriously")

    def fooling_around(self):
        print("I'm rather playing video games")
        raise Exception('something wrong with my time management')

with PhD(thesis = 'Neutron Scattering from Aluminium Alloys') as th:
    th.serious_work()
    # th.fooling_around() # try this to see how an exception is handled
```

Output:

```
__init__: Start writing thesis called "Harry Potter and Neutron Scattering from Aluminium Alloys"
__enter__: Read articles for my thesis.
I'm taking this seriously
__exit__: Send thesis "Harry Potter and Neutron Scattering from Aluminium Alloys" to my supervisor
```

Notice how `__init__` and `__enter__` are called immediately when the context starts.

As you see, `__exit__` can handle exceptions, such as running `th.fooling_around()`: [17]

Output:

```
__init__: Start writing thesis called "Harry Potter and Neutron Scattering from Aluminium Alloys"
__enter__: Read articles for my thesis.
I'm rather playing video games
exception type: <class 'Exception'>
exception value: something wrong with my time management
traceback: <traceback object at 0x148978006700>
__exit__: I wont send an empty thesis to my supervisor
Traceback (most recent call last):
  File "context_manager.py", line 28, in <module>
    th.fooling_around() # try this to see how an exception is handled
  File "context_manager.py", line 24, in fooling_around
    raise Exception('something wrong with my time management')
Exception: something wrong with my time management
```

The exception arguments in `__exit__(self, exception_type, exception_value, traceback)` must be there whether you use them or not. Leaving them out as in `__exit__(self)` gives

```
TypeError: __exit__() takes 1 positional argument but 4 were given
```

which tries to tell that Python always calls `__exit__` with 4 arguments and I had only one.

[17] The exception was triggered with `raise Exception()`; an easy way to cause a `ZeroDivisionError` exception is to call `print(1/0)`.

3.19 Coroutines

Things to remember (inspired by David Beazley's talk about coroutines):

- Generators ***produce*** data for iteration
- Coroutines can both ***produce*** and ***consume*** data → coroutines can be ***pipelined***

Coroutines have these properties:

1. You start, ***prime***, a coroutine and it waits for further orders.
2. A coroutine starts really fast, and costs practically nothing. As if you called a function.
3. A coroutine takes about 1 kB of memory. For instance, starting a thread takes several megabytes of memory
4. At any time, you can ***send*** data to the object and it processes it.

You may think of coroutines as ***event-driven objects*** or ***generator functions***. Coroutines won't run truly parallel in the CPython interpreter (see GIL problems later in 13.1) but they swap tasks really fast.

Here's and example of how to prime a coroutine and to send data to it.

coroutine_and_source.py

```
import numpy as np

def data_consumer(id):
    # A coroutine that eats data sent to it
    print('primed the data_consumer',id)
    while True:
        data = yield
        print(id,'eats', data)

def data_source(target):
    # An ordinary function that produces data and send is to target
    for data in ['beas','potatoes','lettuce','cucumber']:
        target.send(data)

eater = data_consumer('thehungryone')
eater.send(None) # prime the coroutine, now it's ready to receive data
print('manually feeding:')
eater.send('cookie')
eater.send('elephant')
eater.close() # stop it

print('feeding from source:')
veg = data_consumer('veggie')
next(veg) # another way to prime a coroutine, same as veg.send(None)
data_source(veg)
veg.close() # stop it
```

`yield` is where data exits from a generator and `yield` is where data enters a coroutine.

As David Beazley points out, a decorator can prime a coroutine automatically,

coroutine_decorator.py

```
# Wrapper from http://www.dabeaz.com/coroutines/coroutine.py
# edited: no method .next(), but a built-in function next()
# automate coroutine priming
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr) # prime the coroutine
        return cr
    return start

@coroutine
def data_consumer(id):
    # A coroutine that eats data sent to it
    print('primed the data_consumer',id)
    while True:
        data = yield
        print(id,'drinks', data)

eater = data_consumer('joe')
eater.send('coffee')
```

Coroutines can form a pipeline, and a pipeline that cannot branch is a rotten pipeline. Of course you must prime *all* coroutines in the pipeline or it fails without a warning. This kind of failure is easily avoided with a coroutine priming decorator.[18]

[18] Warning: this uses the legacy `np.random.random()`, see section 3.23.

coroutine_pipeline.py

```
"""
 A pipeline of coroutines

 Wrapper from http://www.dabeaz.com/coroutines/coroutine.py
 edited: no method .next(), but a built-in function next()
 Automatic coroutine priming
"""
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr) # prime the coroutine
        return cr
    return start

@coroutine
def sequencefinder(sequence,next_filter=None):
    while True:
        number = yield
        if str(sequence) in str(number):
            print(f'{number:>40} contains number sequence {sequence}')
            try:
                next_filter.send(number)
            except:
                # the last coroutine acts as a sink
                pass


def source(target):
    for i in np.arange(10000000):
        target.send(int(1e30*np.random.random()))


import numpy as np
# np.random.seed(123131313)

# Feed random numbers to coroutines that find given sequences in it
# pipeline:
# source -> f_first -> f_second

f_second = sequencefinder(31415926)        # find longer sequence 31415926 (and no further)
f_first  = sequencefinder(314159,f_second) # find short sequence 314159 and feed to second finder

# start the pipeline process
source(f_first)
```

3.20 Delegating work to subgenerators with `yield from`

Covering all exceptions and edge cases used to be very tricky, but since Python 3.3 we have the all-mighty ***`yield from`***. The idea is that a generator may receive the value it `yield`s from another generator. To fully appreciate the subtleties, I recommend main uses of `yield from` @Stackoverlow. The essence of

```
def my_generator:
    yield from my_subgenerator
```

is, AFAIK, that ***whatever comes from `my_subgenerator` is transparently copied to yield***. Moreover, it's a two-way door, `my_generator` may also send data to `my_subgenerator`, so the word "from" is a bit odd. If you consider that a generators may give out anything (results, exceptions, StopIteration, *etc.*), you realize that there are awfully many possibilities to cover: miraculously, `yield from` does it all! Obviously `yield from` saves lot's of coding, and makes splitting generators to subgenerators an easy task, similar to splitting functions to subfunctions.

3.21 Changing behaviour of a library class method

Suppose you have a Python library class, call it `LibraryClass`, that came with a downloaded package. You find that `LibraryClass` is excellent, but has a method whose implementation doesn't meet your needs.[19] The point is, that

> You definitely don't want to edit the library, because your edits may be overwritten already in the next update.

Let's say `LibraryClass` contains the method `write` that's not good enough. Create a your own class, which ***inherits*** all from `LibraryClass`, but has a it's own, improved `write` method:

[19] Inspired by a real-world problem I helped to solve some time a go.

```
class MyClass(LibraryClass):
   def write(self,*args,**kwargs):
       # whatever write is supposed to do

myinstance = MyClass()
myinstance.write(...)
```

The newly implemented `write` method overrules - ***shadows*** - the old `LibraryClass` method.

Curiosity: Poking a method to a class

Consider an empty class,

```
class EmptyClass():
    pass
```

It does nothing so far. Can we add a method to it afterwards, so that it becomes

```
class EmptyClass():
   def fun(self,*args,**kwargs):
      # some code
      print(*args)
```

Obviously the function would need to have the `self` argument, so how about writing a global function `fun()` and bind it to the class:

poke_method_to_class.py

```
# An empty class
class EmptyClass:
    pass

# A global function
def fun(self,*arg):
    print(*arg)
    print(self)

# create an instance before poking
old_instance = EmptyClass()

# poke fun to EmptyClass
EmptyClass.fun = fun

# create an instance after poking
new_instance = EmptyClass()

# the new_instance has method fun:
new_instance.fun('calling new_instance.fun')
# also the old_instance has method fun:
old_instance.fun('calling old_instance.fun')
"""
calling new_instance.fun
<__main__.EmptyClass object at 0x145b021f3be0>
calling old_instance.fun
<__main__.EmptyClass object at 0x145b021f3c40>
"""
```

The class instance created *before poking the method to the class* knows about that method after poking. One downside is that the global function has now the argument `self` but you can still call it with a dummy first argument, *e.g.* with an empty string `fun('',1.9)`.

Curiosity: Poking an attribute to an object

You can also add new attributes to objects.

poke_attribute.py

```
def hello():
    print("Hi there!")

if __name__ == '__main__':
    hello()
    print('dir(hello):\n',dir(hello))
    print(hello.__name__)
    print('inquiry about attribute:',hasattr(hello,'numerical_value'))
    # since False, add the attribute:
    print(80*'-')
    print("setting a new attribute 'numerical_value' to value 101")
    setattr(hello,'numerical_value',101)
    print('dir(hello):\n',dir(hello))
    print('inquiry about attribute:',hasattr(hello,'numerical_value'))
    print(hello.numerical_value)
```

3.22 Curiosity: Making sure only one class instance can be created: Singleton

Although you may never need one, and some argue you nver should, the `Singleton` class [20] gives me a chance to comment about a few Python aspects that are interesting *per se*.

Sometimes you want to prevent creating but one instance of a certain class.

Such an allow-only-one-instance ***design patterns*** are called singletons @wikipedia, implementation in Python is very simple, the essence is to make use of the magic method `__new__`, which is called when an instance is *about* to be created. From Wikipedia (almost),

[20] The class name is, of course, irrelevant. `Once` would do.

```
class Singleton:
     __instance = None
     def __new__(cls, *args, **kwargs):
          if not cls.__instance:
               cls.__instance = object.__new__(cls)
          return cls.__instance
```

Notice that `cls` refers to the class (now `Singleton`), while `self` would refer to the specific instance of the class. The dunder method `__new__` is called before `__init__`, initialization of an instance. The double underscore makes `__instance` a bit more private.[21] The first instance of the `Singleton` class has `__instance` set as `None`, so it returns a new instance of the `Singleton` class. Second and subsequent attempts to create new instances have `__instance` set so that `__new__` keeps returning the same first instance again and again.

The class variable, in this example `__instance` is not really a "save" or a "static" variable as in some other programming languages, where static variables inside a function remember their values between function calls. Here `__instance` is a ***class variable***, it belongs to the `Singleton` class and it's the same whenever you deal with a `Singleton`. Mostly class variables are ment to share data between all class instances, just that now there can be only one single instance.

Remark: Also `super().__new__(cls)` will be translated to `object.__new__(cls)`, which can be seen if you make a deliberate error,

```
File ".../singleton_test.py", line 19, in __new__
cls._instance = super().__new__(cls,*args)
TypeError: object.__new__() takes exactly one argument (the type to instantiate)
```

---

[21] Users *can* access `__instance` if they insist. The Python name mangling rules say it's `Singleton._Singleton__instance`; see 3.18. It's not private, just well hidden.

Remark: Maybe a bit less cryptic solution would be to use a `@classmethod`, a method bound to a class and which access and modify the state of the class,

```
class Singleton:
   __instance = None
   @classmethod
   def get(cls, *args, **kwargs):
      if not cls.__instance:
          cls.__instance = cls(*args, **kwargs)
      return cls.__instance
```

and create objects using `sing = Singleton.get()`.

Remark: Sometimes you see code explicitly inheriting `object`, `class Singleton(object)`. However, that's redundant in Python 3, where base-classes automatically inherit from object, and you may as well write just `class Singleton`.

Testing `Singleton`:

```
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)
print(id(s1),id(s2))
"""
True
231437651557792 231437651557792
"""
```

The objects `s1` and s2 are exactly the same, I just used two names for the same object to be able to superficially tell them apart. The above code actually means

```
s1 = Singleton()
s2 = s1
```

3.23 NumPy random numbers and seeds

Some topics mentioned here will be re-iterated in the C++ section. If you don't remember anything else about this section, remember at least this advice:

> Don't recycle random numbers.

Devices producing real random numbers have turned out to be too slow, therefore we rely on ***Pseudo Random Number Generators (PRNGs)***.[22]

Remark: Motivation, if you need any: Let's say you've instantiated two PRNGs `gen1` and `gen2`, and the test `gen1 is gen2` gives `False`. If the algorithm is good, `gen1` will produce numbers that are sufficiently random, meaning the generated set of numbers satisfy strict randomness criteria. Then you have `gen2`, which is an instance of the same class as `gen1`, so it, too, will produce good random numbers. The catch is that if you generate, say, 10^8 numbers with `gen1` and 10^8 numbers with `gen2`, will the 2×10^8 numbers satisfy strict randomness criteria? In the worst case, if the generators use the same ***seed***, you could actually have twice exactly the same 10^8 numbers. Even with different seeds randomness is compromised. How badly, depends on how large the internal state space of the PRNG is and how large and how different the seeds are. I have a real-world example that caused an ethernet error `''Broken cable?''` about once a month.

There's no perfect PRNG. The two popular and good ones are:

- Mersenne Twister MT19937 or MT19937-64 @Wikipedia is a well-tested, widely used generator. First version 1997. Long period ($2^{19937} - 1$), but large state buffer (2.5 KiB), and a bit slow. I'll apply MT19937 in C++.
- Permuted congruential generator PCG @Wikipedia A newcomer (2014) with long period ($2^{128} \approx 3 \times 10^{38}$).

Core Python relies on MT19937,

[22] I would have liked to call them "generators", but Python already has them in another meaning.

```
>>> import random
>>> random.random()
0.3162973958752622
>>> random.random.__doc__
'random() -> x in the interval [0, 1).'
>>> print(random.__doc__)
... long description ...
General notes on the underlying Mersenne Twister core generator:

* The period is 2**19937-1.
* It is one of the most extensively tested generators in existence.
* The random() method is implemented in C, executes in a single Python step,
and is, therefore, threadsafe.
```

while ***NumPy default is PCG64***, see pcg64 @numpy. [23]

```
>>> import numpy as np
>>> print(np.random.__doc__)
========================
Random Number Generation
========================

Use ``default_rng()`` to create a `Generator` and call its methods.

=============== ==========================================================
Generator
--------------- ----------------------------------------------------------
```

[23] The implementation follows the research of Melissa O'Neill (homepage @cs.hmc.edu).

```
Generator       Class implementing all of the random number distributions
default_rng     Default constructor for ``Generator``
=============== ============================================================

============================================ ===
BitGenerator Streams that work with Generator
-------------------------------------------- ---
MT19937
PCG64
PCG64DXSM
Philox
SFC64
============================================ ===
...
>>> print(np.random.default_rng())
Generator(PCG64)
```

Sequential code

In a sequential code - applies also to codes that `import threading` thanks to GIL - you can create one PRNG instance and use that. Or, if you don't trust yourself, make it a singleton class. Most of us use the NumPy module `random` to create pseudo random numbers. The usage depends on the current implementation in NumPy, so better look at the random number manual @numpy.org. The recommended procedure is, at the moment of writing,

```
>>> from numpy.random import default_rng
>>> rng = default_rng()
>>> vals = rng.standard_normal(10) # normally distributed
>>> more_vals = rng.standard_normal(10)
>>> rng.random(5) # random floats in the half-open interval [0.0, 1.0)
```

```
Out[10]: array([0.26708778, 0.21264404, 0.77064753, 0.84541853, 0.12868363])
```

Remark: People, me included, became used to code now considered legacy, namely convinience functions such as `numpy.random.seed()` and `numpy.random.random()`. I still have them in some samples, sorry.

Parallel code

If you need only relatively few random numbers in each parallel worker, you can generate the numbers in one worker and distribute the splitted list to the workers. I wonder if there's anyone doing this.

If you need to generate lots of random numbers in each parallel worker you need to be really careful.

How can you make sure all the random numbers generated in parallel workers are sufficiently random?

Remark: Maybe I need to convince you that random numbers in parallel execution are to be taken seriously. In 2021 Tanel Pärnamaa found a nasty surprise concerning machine learning with PyTorch + NumPy (see post @github.io). He presents a minimal code for testing:

tanel_numpy_pytorch.py

```
import numpy as np
from torch.utils.data import Dataset, DataLoader

class RandomDataset(Dataset):
    def __getitem__(self, index):
        return np.random.randint(0, 1000, 3)

    def __len__(self):
        return 16

dataset = RandomDataset()
dataloader = DataLoader(dataset, batch_size=2, num_workers=4)
for batch in dataloader:
    print(batch)
```

The code overwrites the `__getitem__()` method of the inherited `Dataset` class, see Chapter 3.21. The batches are not supposed to be repeating, but in that version of PyTorch they came out exactly the same, meaning you'd be repeating exactly the same operation multiple times, and wondering how come my neural network learns so slowly. The reason for this failure, in Tanel's own words, was

> PyTorch uses multiprocessing to load data in parallel. The worker processes are created using the `fork` start method. This means each worker process inherits all resources of the parent, including the state of NumPy's random number generator.

There was a long discussion whether this is a bug or a feature. In any case, in recent PyTorch this has been fixed or changed. The take-home message: ***With some hesitation, you can copy random number generators, but never copy their states.***

Since you've decided which of the available PRNGs you want to use, to only remaining question is how to seed PRNGs in workers so that each worker receives a sufficiently different seed, whatever that means.[24]

A PRNG is just a ***deterministic*** algorithm, there's no magic source of randomness in it. The purpose of the seed is to select the internal state vector of the PRNG, and the algorithm evolves the state vector in a clever way to produce seemingly random numbers. In PCG64:

[24] Massively parallel code can dig up seeding problems that won't ever surface in smaller scales. I'll leave the massively parallel discussion to experts.

The `pcg64.RandomState` state vector consists of 2 unsigned 128-bit values, which are represented externally as python longs (2.x) or ints (Python 3+). `pcg64.RandomState` is seeded using a single 128-bit unsigned integer (Python `long`/`int`). In addition, a second 128-bit unsigned integer is used to set the stream.

One general advice: Don't use seeds such as 1, 2, 3.., and especially

Don't use seeds with lots of zeros.

An all-zero initial state has very low entropy[25], and the PRNG algorithm gets stuck there. Even if the state is only partly zeroes, you end up with less random output.[26] What may happen if workers have PRNGs with distinct yet too similar seeds? Remember PRNGs evolve their internal state and produce an output. Then, with bad luck, a generator state happens to coincide (collide) with a state some other PRNG has visited. From then on, that poor PRNG will evolve the same route the other generator did, and you are recycling random numbers. Avoiding collisions is impractical if not impossible, all we want is make them very improbable.

Back to the question how to get distinctive seeds to workers. One way is to choose an initial high entropy seed, give each worker a distinct key, and combine the initial seed with the key to get the worker seeds, (see also parallel @numpy.org)

```
>>> from numpy.random import SeedSequence, default_rng
# A big integer as initial seed
>>> entropy = 52906836728959148027957190488174817840
>>> ss = SeedSequence(entropy)
# Spawn off 10 child SeedSequences to pass to child processes.
>>> child_seeds = ss.spawn(10)
>>> streams = [default_rng(s) for s in child_seeds]
# For testing, generate 5 uniformly distributed random numbers from each stream
>>> for s in streams: print(s.random(5))
```

Spawning children copies the initial seed and adds a different key to each child, for example

[25] If you get only numbers 0, it's one bit of information, so entropy is $\log_2(1) = 0$. Very low entropy, like in absolute zero temperature in thermodynamics. Entropy is not a property of a single number, but the whole number space.

[26] Computer scientist use the poetic name *zeroland*.

```
>>> child_seeds[3]
SeedSequence(
entropy=529068367289591480279571904881748178 40,
spawn_key=(3,),
)
```

Some testing,

```
>>> default_rng(child_seeds[2]).__getstate__()
{'bit_generator': 'PCG64',
  'state': {'state': 60282425288889213924973460985733618434,
    'inc': 296823201770864150067649502116665995049},
  'has_uint32': 0,
  'uinteger': 0}
```

Don't be surprised if you get exactly the same random output every time you run this code. It's all deterministic, so *how* could it change?

Repeatable output is good for checking, but often you need to produce fresh data the *next time* you run the code, so you need seeds that change with *time*. [27] There you have it: seed that's drawn from the system clock, or something that relates to it, such as keystroke timings, file system changes etc. The system clock is deterministic by itself - a clock giving random numbers is no clock at all. Now you face the same question, how to repeat the calculation so that you're producing new independent data?

One way would be to store the state vectors of the PRNGs at the end of the run, and reload them in the next run. That would be close to continuing the calculation longer. That's not a very popular or recommended solution, so consider using the system clock, something along the lines

[27] I have a story about an academic who repeated simulations with the same seed.

parallel_seed_test.py

```
import numpy as np
from numpy.random import SeedSequence, default_rng
import datetime

# hit the number pad until you get bored
entropy = 529068367289591480279571904881748178 40
thistime = np.datetime64(datetime.datetime.now()).astype(np.uint64)
entropy = entropy*thistime # just one possibility
ss = SeedSequence(entropy)
child_seeds = ss.spawn(10)
child_rngs = [default_rng(s) for s in child_seeds]
for rng in child_rngs: print(rng.random(5))
```

The function `datetime.datetime.now()` has microsecond resolution,

```
>>> datetime.datetime.resolution
datetime.timedelta(microseconds=1)
```

so that's the maximum pace you should start new runs.

Remark: Another way is to initialize `SeedSequence` with a tuple `(entropy,key)`,

```
>>> from numpy.random import SeedSequence, default_rng
# High entropy initial seed (here 32 bits)
>>> entropy = 0x87351080e25cb0fad77a44a3be03b491
>>> keys = [key for key in range(10)] # keys for 10 workers
>>> child_seeds = [SeedSequence((entropy,key)) for key in keys]
>>> streams = [default_rng(s) for s in child_seeds]
# For testing, generate 5 random numbers U[0,1) from each stream
>>> for s in streams: print(s.random(5))
```

The `spawn` method and the tuple initialization are not quite the same thing.

Remark: The module `randomstate` provides a way to seed multiple PCG64 generators (failed to install in my computer, so haven't tested this). In principle,

```
>>> from randomstate.entropy import random_entropy
>>> import randomstate.prng.pcg64 as pcg64
>>> entropy = random_entropy(4)
# 128-bit number as a seed
seed = reduce(lambda x, y: x + y, [long(entropy[i]) * 2 ** (32 * i) for i in range(4)])
streams = [pcg64.RandomState(seed, stream) for stream in range(10)]
```

3.24 Debugging Python segmentation fault

Segmentation faults are memory handling errors, which kick you out of Python shell. Usually this happens without giving any hint of what happened and where. A bit more information can be obtained using the `faulthandler` module,

```
$ python -q -X faulthandler  code.py
```

Here's an example of a code that segfaulted,

```
$ python -q -X faulthandler Au_chain.py

Fatal Python error: Fatal Python error: Segmentation fault

Segmentation faultThread 0x

00007efe7296d740 (most recent call first):
File "...long path .../lib/python/hotbit/solver.py", line 108 in diagSegmentation fault
```

and that line was trying to solve a generalized eigenvalue proglem

```
e, wf = geig(H,S)
```

so the problem was either bad input or the underlying LAPACK library routine failed. Keep digging.

3.25 Pattern matching with `match-case` in Python version 3.10 - and a warning

Python 3.10 introduced ***pattern matching*** because it's such a great tool in functional programming languages (Scala, Rust, Haskell). The idea is to check if an object has certain characteristics, without multitude of `isinstance()` tests and complicated `if-elif-else` structures.

The proposals for `match-case` are in PEP634 and PEP635. In one example, the task is to check if `x` is a tuple with two or three elements, and to set `host`, `port`, and `mode` accordingly. The old implementation could be

```
if isinstance(x, tuple) and len(x) == 2:
    host, port = x
    mode = "http"
elif isinstance(x, tuple) and len(x) == 3:
    host, port, mode = x
```

Since Python 3.10 you can do it more elegantly,[28]

```
match x:
    case host, port:
        mode = "http"
    case host, port, mode:
        pass
```

`match-case` is often a lot faster than multiple `if-else` tests, see, e.g., examples @ tonybaloney.github.i0.

[28] Code highlighting for `match-case` is missing.

Everything was fine, until someone noticed a problem: `match-case` rebinds variables! Take a look at the following "Schrödinger's cat" problem.

schrodinger_match.py

```
# Pattern matching caveat

CAT_DEAD = False
CAT_ALIVE = True
print('before match:')
print(f'CAT_DEAD is {CAT_DEAD}')
print(f'CAT_ALIVE is {CAT_ALIVE}')
print('doing match:')
match CAT_ALIVE:
  case CAT_DEAD:
    print("CAT_ALIVE matches CAT_DEAD")

print('after match:')
print(f'CAT_DEAD is {CAT_DEAD}')
print(f'CAT_ALIVE is {CAT_ALIVE}')
```

The output in Python 3.11 is

```
before match:
CAT_DEAD is False
CAT_ALIVE is True
doing match:
CAT_ALIVE matches CAT_DEAD
after match:
CAT_DEAD is True
CAT_ALIVE is True
```

If you use `match-case` to look at the cat, the cat turns to a zombie, both dead and alive. In the code pattern matching in `match-case` set a value to a variable that was not suppose to change! This kind of ***side effect*** is against all good programming language practices, so no wonder opposition rose.

3.26 The `Property` decorator

A Python keyword you find in many codes is `property`. It's a built-in function, mostly used as a decorator `@property`. It's used for these purposes:

- Turning class attributes to properties
  `Attributes` are simple data in an object, accessed with the dot notation `object.attribute`. The problem is that you have no control how the data is accessed and you can't define special handling of the data.
  `Properties` are special kind of attributes that take care of their access and handling. In short, you can validate data.

- Defining read-only attributes, set only at `__init__`

For example ASE (Atomic Simulation Environment) (ASE @dtu.dk) has a base class for atoms and molecules,

```
class Atoms:
    """ description """
    def __init__(self, symbols=None,
                 positions=None, numbers=None,
                 tags=None, momenta=None, masses=None,
                 magmoms=None, charges=None,
                 scaled_positions=None,
                 cell=None, pbc=None, celldisp=None,
                 constraint=None,
                 calculator=None,
                 info=None,
                 velocities=None):

      self._cellobj = Cell.new()
      self._pbc = np.zeros(3, bool)
      ...
```

The underscore in the periodic boundary conditions attribute `_pbc` signals that it's supposed to be non-public, that is, not supposed to be set as `object._pbc = ...`. In `__init__`, it's initialized to the default value `array([False, False, False])`. To change those you'd use a `setter()` method. Similarly, to read the value you'd use a `getter()` method.

ASE has this code:

```
@property
def pbc(self):
    """Reference to pbc-flags for in-place manipulations."""
    return self._pbc

@pbc.setter
def pbc(self, pbc):
    self._pbc[:] = pbc

def set_pbc(self, pbc):
    """Set periodic boundary condition flags."""
    self.pbc = pbc

def get_pbc(self):
    """Get periodic boundary condition flags."""
    return self.pbc.copy()
```

Here one exposes a public property `pbc` to replace direct manipulation of `_pbc`. Calling the method `set_pbc()` falls back to `pbc`. (if for some reason z-direction can't be made periodic) the tests will be executed.

Finally, why does the `get_pbc()` return a copy of the NumPy array `pbc` and not the array itself? Suppose you had a data validation in the code, making sure the z-direction can't be made periodic:

```
@pbc.setter
def pbc(self, pbc):
    print('called pbc setter')
    if pbc[2] == True:
        raise ValueError("Z-direction can't be made periodic")
    self._pbc[:] = pbc
```

This prevents mistakes:

```
atom = Atoms()
atom.pbc=np.array([False,True,True])}       # raises ValueError
atom.set_pbc(np.array([False,False,True]))  # raises ValueError
```

However, if you had `get_pbc()` returning the original array, you can by-pass validation:

```
  def get_pbc(self):
      """Get periodic boundary condition flags."""
      return self.pbc   # <= BAD IDEA: not a copy but the array itself

atom = Atoms()
periodic[:] = atom.get_pbc()           # notice the [:]
periodic[2] = True
print(atom.pbc)
# output : [False,True,True]               # NO VALIDATION, changed pbc[2] to illegal value!
```

A method that returns the original array makes all safety measures useless. Return a copy, and the user can't get hold of the original array. Another reason to return a copy is that you may want to return a modified data array without any changes made to the original one.

You can make a read-only property by leaving out all setters.

```
class Circle:
    def __init__(self,radius):
        self._radius = radius
    @property
    def radius(self):
        return self._radius

c1 = Circle(10.0)
print(c1.radius)
c1.radius = 5.0  # AttributeError: property 'radius' of 'Circle' object has no setter
```

There're no private Python variables. You can set `c1._radius = 5.0`. Just don't, it's not *meant* to be set.

4 Simulation and Measurements in Python

Suppose I have a simulation code, and want to do measurents at a certain frequency. Some measurements are computionally so expensive that I want them done only now and then, or the simulation chokes on measuring. Here I show how to set up a simulation and a way to add measurements to it. This is me coding, so it's subject to improvements.

The base class `MeasurementBase` defines how measurements are added and executed:

```
class MeasurementBase:
  def __init__(self):
      # start with an empty list of measurements and set number of collected data to zero
      self.measurements = []
      self.steps = 0

  def add_measurement(self, measurement, frequency = 1, *args, **kwargs):
      # adds a callback function "measurement._measure"
      # default: frequency = 1 means measure at every step

      # test the function has a __call__()
      if not callable(measurement._measure):
          eprint(f'{measurement.name} has no callable method _measure')
          raise ValueError

      print(f'adding new measurement: name={measurement.name} frequency={frequency}')
      self.measurements.append((measurement._measure, frequency, args, kwargs))

  def do_measurements(self):
      self.steps +=1
      for measure, frequency, args, kwargs in self.measurements:
          if self.steps%frequency == 0:
              measure(self,*args,**kwargs)
```

The `callable` test makes sure the measument can be called. I ***inherit*** the `MeasurementBase` to all measurement classes and to the simulation class. The list of measurements and their frequencies is `measurements`. The method `do_measurements` is called in the actual simulation, and a particular measurement is performed if `steps` is a multiple of that measurements `frequency`.

A measurement class has two methods, `_measure` that does the measurement, and `get`, which return the result upon request. Prototypically, a measurement looks like this:

```
class Measure_Something(MeasurementBase):
   def __init__(self, verbose = False):
      self.verbose = verbose
      self.name='measure_something'
      if(self.verbose): print('Measure_Something initialized')

  def _measure(self,simulation,*args,**kwargs):
      if(self.verbose): print(f'  {self.name} using {simulation}')
      # get data from object ``simulation'' and analyze it.
      # Something like
      # self.result += simulation.data

  def get(self):
      return self.result
```

Prototype of the `Simulation` class:

```
class Simulation(MeasurementBase):
  def __init__(self, verbose=False):
    super().__init__()
    self.verbose = verbose
    # add here initializations of simulation data

  def run(self,N):
    for _ in range(N):
        ... run the simulation for one step
        self.do_measurements()
```

Here `super()` refers to the base class, and `super().__init__()` calls `MeasurementBase.__init__()`. Without this `self.measurements.append` fails, because the list `measurements` and the counter `steps` are missing.

In the main function I create a simulation instance, a measurement instance, and add measurements:

```
if __name__ == '__main__':
    simu = Simulation()
    something = Measure_Something(frequency=10) # you may add verbose=True
    simu.add_measurement(something)
    # test run
    for i in range(5):
       print(u'\u2500'*80)  # thick horizontal line
       print('iter ',i)
       simu.run(500)
       result = something.get()
       print(f' something = {result}')
```

This runs the simulation for 500 steps, measures `something` every 10 steps, and prints out the `result`. This is repeated 5 times. A working test code is in measurement_example.py. It computes random numbers and measures their average, and lower and upper limits.

5 Python Serialization

Serialization means converting an object to a byte stream, a form which can be stored or transmitted. Serialization is very useful if you want to store data between calculations. Simple serialization can be done using `repr()`. Here are some common serialization modules:

- CSV (Comma Separated values, flat data)

write_and_read_csv.py

```
import csv
with open('file.csv','w') as f:
    f_writer = csv.writer(f, delimiter=',', quotechar='"')
    f_writer.writerow(['name','score','date'])
    f_writer.writerow(['Vesa','1', '1.1.1980'])
    f_writer.writerow(['Minttu','5', '1.1.2021'])

with open('file.csv','r') as f:
    lines = csv.reader(f)
    for line in lines:
        print(line)
```

- JSON (Javascript String Object Notation, nested data)

write_and_read_json.py

```
# write data using dump method
import json
# test data
data = [['name','score','date'],['Vesa',1,'1.1.1980'],['Minttu',5,'1.1.2021']]
with open('file.json', 'w') as f:
    json.dump(data, f, sort_keys=True)

# read data
with open('file.json', 'r') as f:
    data = json.load(f)
    for d in data:
        print(d)
```

- ***Pickle*** is Python's native data serialization module. The process is ***pickling***.

pickle_example.py

```
import pickle

def store():
    Vesa  = {'key' : 'Vesa', 'name' : 'Vesa Apaja', 'age' : 25, 'middle name' : 'A'}
    Jack = {'key' : 'Jack', 'name' : 'Jack Ripper', 'age' : 10**6, 'middle name' : 'the'}
    dic = {}  # dictionary
    dic['Vesa'] = Vesa
    dic['Jack'] = Jack
    print('dumping data')
    with open('data.pkl','wb') as f:  # write, *binary format*
        pickle.dump(dic, f)

def load():
    print('loading data')
    with open('data.pkl','rb') as f:  # read, *binary format*
        doc = pickle.load(f)
        for keys in doc:
            print(keys, '=>', doc[keys])

if __name__ == '__main__':
    store()
    load()
```

6 Matplotlib

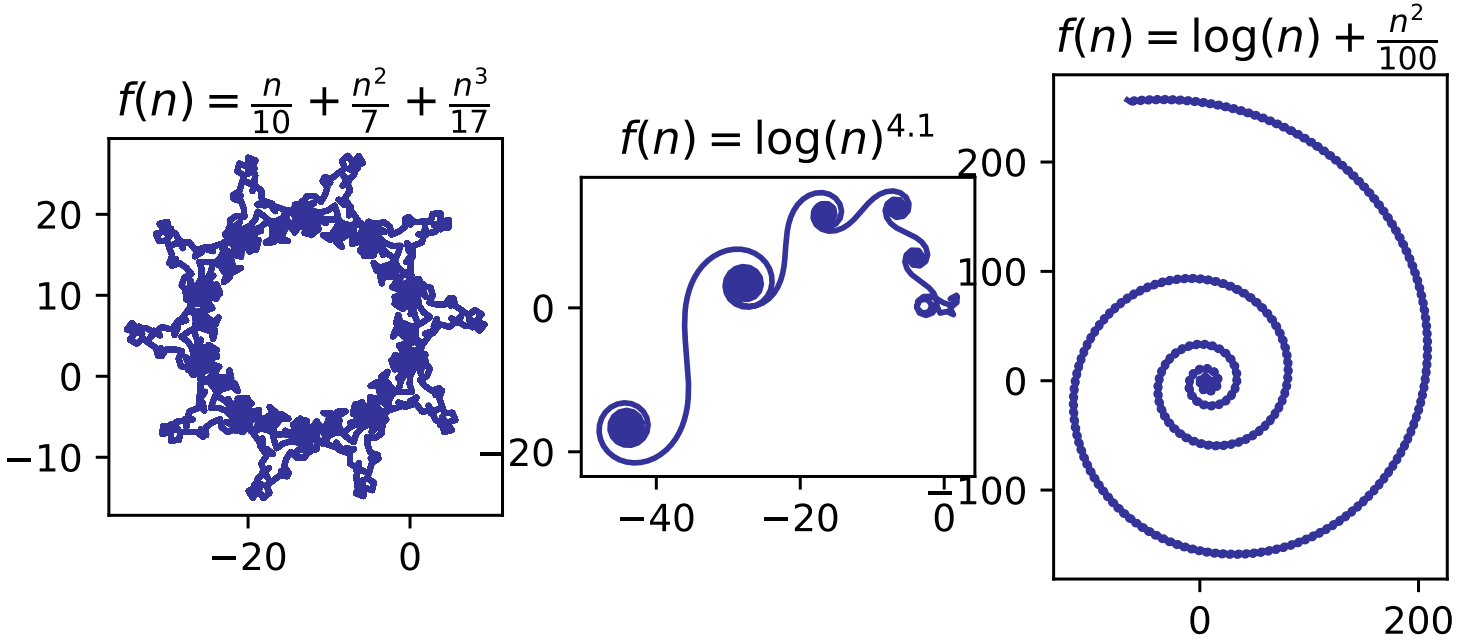
Matplotlib is *the* graphical Python toolkit.

John D. Cook plots exponential sums, and they make pretty pictures:

exposum_picture.py

```
import matplotlib.pyplot as plt
from numpy import array, pi, exp, log

def f1(n):
    return n/10 + n**2/7 + n**3/17

def f2(n):
    return  log(n)**4.1

def f3(n):
    return log(n) + n**2/100.

Ns = [15000,1200,12000] # number of points
fs = [f1,f2,f3]         # functions f(n)
titles = [r'$f(n)=\frac{n}{10} + \frac{n^2}{7} + \frac{n^3}{17}$',
          r'$f(n)=\log(n)^{4.1}$',r'$f(n)=\log(n) + \frac{n^2}{100}$']

for i,(N,f,title) in enumerate(zip(Ns,fs,titles)):
    z = array( [exp( 2*pi*1j*f(n) ) for n in range(3, N+3)] )
    z = z.cumsum()

    # plot:
    plt.subplot(1,3,i+1)
    plt.title(title)
    plt.plot(z.real, z.imag, color='#333399')
    plt.gca().set_aspect(1)

fig = plt.gcf()
fig.suptitle(r'Complex cumulative sums of $\exp(2\pi i f(n))$ from n=3...')
plt.show()
```

Complex cumulative sums of exp(2$\pi i f(n)$) from n=3...



6.0.1 Updating a plot by clicking it

One trick I recently added to my bag is updating a plot by clicking it. I usually let a Monte Carlo simulation run, and only now and then check its progress. I used to run a Python analyzer code again and again, with manually killing the plot window. This is boring, so now I just click the plot and it reads new data from the disk and updates the plot. Here's a prototype of `matplotlib`'s click-the-canvas idea:

matplotlib_click.py

```
import matplotlib.pyplot as plt

x = []
y = []
xlims = [0,1]
ylims = [0,1]

def onclick(event):
    if event.button == 1:
         x.append(event.xdata)
         y.append(event.ydata)
         plotit()

def plotit():
    plt.clf()
    plt.scatter(x,y)
    plt.xlim(xlims)
    plt.ylim(ylims)
    plt.draw()


if __name__ == '__main__':

    fig = plt.figure(figsize=(10, 8), dpi=80)
    fig.canvas.mpl_connect('button_press_event',onclick)
    plotit()
    plt.show()
```

6.0.2 Matplotlib backends and how plots are viewed

Jupyter users can skip this discussion, they've already made their choice.

Matplotlib supports a number of ***backends*** for viewing plots (Tk-window, browser,...), or what output format (PS, PDF, ...) to use. One of them is `webagg`, which sends the plot to browser:

matplotlib_backend.py

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(1,10,0.1)
y = np.sin(x)

# plot to Tk window
plt.switch_backend('TkAgg')
plt.plot(x,y,'bo-')
plt.show()

# plot to browser
plt.switch_backend('WebAgg')
plt.plot(x,y,'bo-')
plt.show()
```

I mostly save matplotlib figures by clicking the save-button on the plot window, hardly ever save plots inside a script.

7 NumPy

For numerical work `NumPy` is the most important Python package.

NumPy @numpy.org is for handling numerical data, and it brings efficicient matrix operations to python. The dedicated matrix data type is `numpy.matrix` (same as `numpy.mat`), a specialized 2D `numpy.array`:

```
>>> A = np.array([[1,2,3],[4,5,6]])
>>> type(A)
<class 'numpy.ndarray'>
>>> B = np.matrix([[1,2,3],[4,5,6]])
>>> type(B)
<class 'numpy.matrixlib.defmatrix.matrix'>
```

While `numpy.matrix` is always 2D, `numpy.array` can have any dimension, and 2D arrays won't remain 2D in all operations. See the discussion about differences of numpy array and matrix @Stackoverflow. Both matrix and array has the `@`-operator for matrix multiplication and dot product. Let's take a closer look at what `@` does.

7.1 Matrix product and elementwise product

Be careful with operators `*` and `@`:

`numpy.matrix * numpy.matrix` computes matrix product
`numpy.array * numpy.array` computes elementwise product

and (since Python 3.5)

`numpy.matrix @ numpy.matrix` computes matrix product
`numpy.array @ numpy.array` computes matrix product

For example,

```
>>> l1 = [[1,2],[3,4]]              # 2x2 list
>>> l2 = [[2,2],[2,2]]              # 2x2 list
>>> a = np.matrix(l1)               # 2x2 matrix
>>> b = np.matrix(l2)               # 2x2 matrix
>>> c = np.array(l1)                # 2x2 array
>>> d = np.array(l2)                # 2x2 array
>>> a*b                             # * is here matrix product, same as a@b
matrix([[ 6,  6],
        [14, 14]])
>>> c*d                             # * is here elementwise multiplication
array([[2, 4],
       [6, 8]])
>>> c@d                             # @ is matrix product
array([[ 6,  6],
       [14, 14]])
```

Also the operator ** has two meanings,

with `numpy.matrix a` the operation `a**2` computes matrix square, same as `a*a` and same as `a@a`
with `numpy.array a` the operation `a**2` squares all elements in `a`, same as `a*a`, *not* same as `a@a`

7.2 Dot product calculated three ways

A small speed test. In math, vectors x and y, and the result is $(x-y)^T.(x-y)$. I used `diff=x-y` and

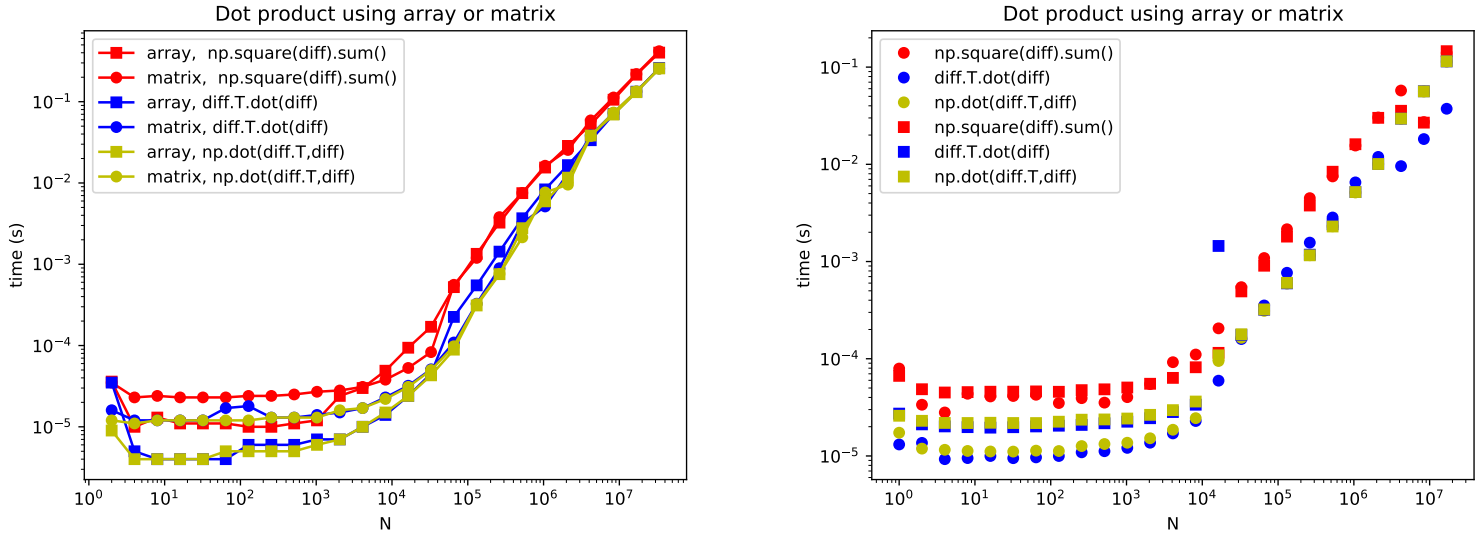
1. `diff.T.dot(diff)`
2. `np.dot(diff.T,diff)`
3. `np.square(diff).sum()`

and wanted also to see which is faster, vectors as 1D `numpy.array` or as 1D `numpy.matrix`.

Careful with shapes:
I set up data using `x = np.random.random(N)`, an array with shape (N,). converting this in-place to matrix using `x=np.asmatrix(x)` gives a shape (1,N) matrix - that's a column vector! If you feed column vectors to the functions, you compute (in math) $(1,N)^T(1,N) = (N,1)(N,1) = (N,N)$ and you end up with a matrix. With large N this eats lot's of memory! [29]

> Shape (N,) is not the same as shape (N,1), and they convert differently.

[29] I got `Memory Error` with 23 GB of RAM and only after that realized my mistake.



Left results are for AMD Ryzen 1700x, right ones are for AMD Ryzen 3900x. Conclusions:

- Dot product with `numpy.dot` is always faster than the array method `.dot` or squaring+summing
- For N$< 10^5$, 1D NumPy arrays with `np.dot` are ***always fastest*** ← BEST BET
- For N$> 10^5$, 1D NumPy matrices and `np.dot` are ***marginally fastest*** - but may be ***dead slow*** for N$< 10^5$!
- For N$> 10^7$ arrays and matrices are about equally fast
- NumPy arrays and method `.dot` speed fluctuates, and is slow for very small N

dot_product_speed.py

```
import numpy as np
import mytimer

@mytimer.timer
def dot_function(x,y):
    diff = x - y
    return diff.T.dot(diff)

@mytimer.timer
def npdot_function(x,y):
    diff = x - y
    return np.dot(diff.T,diff)

@mytimer.timer
def sqsum_function(x,y):
    diff = x - y
    return np.square(diff).sum()

if __name__=='__main__':
    arr_res = []
    mat_res = []
    funs = [sqsum_function,dot_function, npdot_function]
    labs = ['np.square(diff).sum()','diff.T.dot(diff)','np.dot(diff.T,diff)']
    cols  = ['r','b','y']

    for i in range(25):
        N  = 2**i
        x  = np.random.random((N,1))
        y  = np.random.random((N,1))
        xx = np.asmatrix(x)
        yy = np.asmatrix(y)
        for f, lab, col in zip(funs,labs,cols):
            r,t1 = f(x,y)
            r,t2 = f(xx,yy)
            lab = lab if i==0 else ''  # ternary, label only for first value in the data set
            arr_res.append([N,t1,lab,col])
            mat_res.append([N,t2,lab,col])

    import matplotlib.pyplot as plt
    plt.switch_backend('TkAgg')
    for r in arr_res:
        N,t,lab,col = r
        plt.loglog(N,t,'o',color=col, label=lab)

    for r in mat_res:
        N,t,lab,col = r
        plt.loglog(N,t,'s',color=col, label=lab)

    plt.title('Dot product using array or matrix')
    plt.xlabel('N')
    plt.ylabel('time (s)')
    plt.legend()
    plt.show()
```

Above I used the timer function as a decorator,

mytimer.py

```
from  time import process_time as T
def timer(fun):
    def wrapper(*args, **kwargs):
        tic = T()
        res = fun(*args, **kwargs)
        toc = T()-tic
        #print("mytimer: {:20}".format(fun.__name__),"took {:10.8f} seconds".format(toc))
        print(f"mytimer: {fun.__name__:20} took {toc:10.8f} seconds")
        return res,toc
    return wrapper
```

This returns the tuple `(res,toc)`, `res` is what the function returns and `toc` is the timing.

7.3 NumPy BLAS

The speed of NumPy matrix and vector operations depends on which *compiled* BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) it uses. One way to get info is

```
>>> import numpy
>>> numpy.show_config()
blas_mkl_info:
  NOT AVAILABLE
blis_info:
  NOT AVAILABLE
openblas_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
```

```
    runtime_library_dirs = ['/usr/local/lib']
blas_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
    runtime_library_dirs = ['/usr/local/lib']
lapack_mkl_info:
  NOT AVAILABLE
openblas_lapack_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
    runtime_library_dirs = ['/usr/local/lib']
lapack_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    language = c
    define_macros = [('HAVE_CBLAS', None)]
    runtime_library_dirs = ['/usr/local/lib']
%Supported SIMD extensions in this NumPy install:
%    baseline = SSE,SSE2,SSE3
%    found = SSSE3,SSE41,POPCNT,SSE42,AVX,F16C,FMA3,AVX2
%    not found = AVX512F,AVX512CD,AVX512_KNL,AVX512_KNM,AVX512_SKX,AVX512_CLX,AVX512_CNL,AVX512_ICL
```

Apparently I have (had) OpenBLAS package installed and ready for use. From command line the same is

```
$ python  -c "import numpy ; numpy.show_config()"
```

Is there support for SSE3 and AVX2? My CPU has them, type `cat /proc/cpuinfo|grep avx2`. [30] Grepping avx, I found that there are AVX2 instructions in, e.g., `numpy/core/_multiarray_umath.cpython-38-x86_64-linux-gnu.so`.

FlexiBLAS

My laptop shows config entries

[30] AMD and Intel CPUs have 256-bit AVX2, new Intel CPUs have also AVX-512, a 512-bit register.

```
libraries = ['flexiblas', 'flexiblas']
```

FlexiBLAS (cf. flexiblas @Magdeburg) was introduced to make switching BLAS libraries easier. It supports all common BLAS libraries, such as NETLIB-BLAS, OpenBLAS, ATLAS, Intel MKL, and BLIS. The point is that the program interface is the same for them all, and you can switch backends without recompiling. [31] The admin tool is `flexiblas`,

```
$ flexiblas list
System-wide:
System-wide (config directory):
OPENBLAS-OPENMP
library = libflexiblas_openblas-openmp.so
comment =
NETLIB
library = libflexiblas_netlib.so
comment =
User config:
Host config:
Enviroment config:
```

The active backend is chosen either as

```
$ flexiblas default BACKENDNAME
```

or

[31] Since Fedora Linux distro version 33 FlexiBLAS is the default.

```
$ export flexiblas=path_to_BLAS_library
```

In my case the default backend was set by the installed `flexiblas-netlib` package,

```
$ cat /etc/flexiblasrc
default = openblas-openmp
```

Intel OneAPI and MKL

According to intel-oneapi-product-fact-sheet.pdf, Intel OneAPI is "free for developers to use locally or in the Intel DevCloud and ships in December 2020." I downloaded it from Intel OneAPI and installed it in the default `/opt/intel`, activation of OneAPI is done by sourcing

```
$ source /opt/intel/oneapi/setvars.sh
```

Now NumPy has a access to the Intel MKL (Math Kernel Library),

```
>>> import numpy
>>> numpy.show_config()
blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/intel/oneapi/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/opt/intel/oneapi/intelpython/latest/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/intel/oneapi/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/opt/intel/oneapi/intelpython/latest/include']
lapack_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/intel/oneapi/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/opt/intel/oneapi/intelpython/latest/include']
lapack_opt_info:
```

```
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/opt/intel/oneapi/intelpython/latest/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/opt/intel/oneapi/intelpython/latest/include']
```

Conda Intel Python environment

Installation of `intelpython3_full` fails for some Python 3.x versions (incompatible glibc). Apparently Python 3.9 works, I did all in one step.

```
$ conda create -n conda_intel_3.9 intelpython3_full python=3.9
... long list, takes awhile
...
  Installed package of scikit-learn can be accelerated using scikit-learn-intelex.
  More details are available here: https://intel.github.io/scikit-learn-intelex
  For example:
      $ conda install scikit-learn-intelex
      $ python -m sklearnex my_application.py
```

7.4 BLAS and speed

Here's a small speed comparison of a few NumPy operations with various BLAS libraries. Operations are:

1. 2 times `np.dot(A,B)` for 4096x4096 random matrices `A,B`
2. 100 times `np.dot(A,x)` for 4096x4096 random matrix `A` and vector `x`
3. `np.linalg.svd(A)` for 2048x2048 random matrix `A`
4. `np.linalg.cholesky(A)` for 4096x4096 symmetric random matrix `A`
5. `np.linalg.eig(A)` for 1024x1024 random matrix `A`

I got these timings on AMD Ryzen 3900x (times are seconds) and `OMP_NUM_THREADS=4`.

	1	2	3	4	5
OpenBlas	5.584	1.189	8.751	12.896	4.323
MKL	6.255	0.762	6.621	13.551	3.624

AMD CPUs suffer from "Performance regressions may occur on non-Intel x86-compatible processors." [32]

[32] There used to be a way to fool MKL to unlock AVX2 optimizations, namely `export MKL_DEBUG_CPU_TYPE=5`, but Intel took that away in 2020. Thanks Intel.

Intel and AMD Zen Architecture

An interesting study of how Intel performs in an AMD Zen architecture (Ryzen CPU family) was done by Daniël de Kok, 2020-08-31-MKL-Zen. I copied the benchmark from benchmarks-performance-analysis @LANL and repeated some of Daniëls analysis.

```
$ mt-dgemm  4000 |grep GF    # gcc and OpenBlas
$ mt-dgemm-icc 4000 |grep GF  # icc and MKL
```

gcc and OpenBlas	57.314954 GFlops/s	used one thread
icc and MKL	206.648378 GFlops/s	used multiple thread

There are Zen -optimizations[33]

```
92.39% libmkl_def.so.1 [.]  mkl_blas_def_dgemm_kernel_zen
```

Where did Cholesky decomposition spend time?

```
Overhead  Shared Object                                    Symbol
  50.72%  libmkl_def.so.1                                  [.] mkl_blas_def_xdcopy
  23.51%  libmkl_def.so.1                                  [.] mkl_blas_def_dgemm_kernel_zen
   4.92%  libmkl_def.so.1                                  [.] mkl_blas_def_dgemm_pst
   3.40%  libmkl_def.so.1                                  [.] mkl_blas_def_dgemm_copyat_bdz
   2.84%  _multiarray_umath.cpython-37m-x86_64-linux-gnu.so [.] syrk
   2.62%  libmkl_def.so.1                                  [.] mkl_blas_def_xdgemv
   1.28%  libmkl_def.so.1                                  [.] mkl_blas_def_dgemm_copyan_bdz
   1.00%  libmkl_core.so.1                                 [.] mkl_lapack_dlarfb
```

Mostly copying, apparently NumPy array needs some reorganization to work in MKL. There's zen, but what about AVX2? Following Daniël's suggestion to fake Intel CPU. See instructions in Using MKL efficiently @sigma2.no. 2023: in C, add code

[33] I used linux `perf` tool with `perf top -F1000 -d10 -p PID_OF_PROCESS`.

```c
// Fake intel
int mkl_serv_intel_cpu_true() {
        return 1;
}
```

I get

```
Overhead  Shared Object                                       Symbol
  61.68%  _multiarray_umath.cpython-37m-x86_64-linux-gnu.so  [.] _aligned_strided_to_contig_size8.A
  16.48%  libiomp5.so                                        [.] _INTERNAL01e60a8a::__kmp_wait_template<kmp_flag_64<false, true>, true, false, true>
  15.54%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_dgemm_kernel_0
   1.25%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_dtrsm_kernel_ll_0
   0.80%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_dgemm_dcopy_right12_ea
```

Now AVX2 was utilized, and the Cholesky decomp took about 11 seconds. Glad you asked! The time comsuming `_aligned_strided_to_contig_size8.A` is related to memory latency, so there are lot's of memory accesses involved. Maybe threaded execution wasn't a good idea? This can be tested setting `export OMP_NUM_THREADS=1`, now I get

```
Overhead  Shared Object                                       Symbol
  38.77%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_xdcopy
  32.74%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_dgemm_kernel_0
   6.31%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_xdgemv_t
   4.12%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_xdgemv_n
   2.54%  _multiarray_umath.cpython-37m-x86_64-linux-gnu.so  [.] syrk
   2.37%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_dgemm_kernel_nocopy_NT_b1
   1.59%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_dgemm_dcopy_down4_ea
   1.29%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_dgemm_kernel_0_b0
   1.00%  libmkl_core.so.1                                   [.] mkl_lapack_dlarfb
   0.84%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_dgemm_dcopy_down12_ea
   0.72%  libmkl_avx2.so.1                                   [.] mkl_lapack_ps_avx2_dlarfx
   0.69%  libmkl_avx2.so.1                                   [.] mkl_blas_avx2_xdrotm
```

Intel compiler 2023 and AMD Ryzen 9 5950X

gcc and OpenBlas	70.293325 GFlops/s	used one thread
icx and MKL	66.083919 GFlops/s	used one thread
icx and MKL	234.707818 GFlops/s	multiple threads, not fake Intel
icx and MKL	539.644187 GFlops/s	multiple threads, faked intel (see later)

Some timings with just one thread:

	1	2	3	4	5
OpenBlas	5.346	0.389	4.848	11.083	1.195
MKL	5.208	0.467	5.281	11.885	1.159
MKL (faked Intel)	4.749	0.373	4.018	8.809	0.990

7.4.1 Blis BLAS library

Blis is a very competitive BLAS alternative. From channel `conda-forge`,

```
$ conda create -n p3.10  python=3.10
$ conda config --add channels conda-forge
$ conda config --get channels
# --add channels 'defaults'   # lowest priority
# --add channels 'conda-forge'   # highest priority
$ conda activate p3.10
(p3.10) $ conda install -c conda-forge blis
```

Installation using Python pip is

```
$ python -m pip install blis
```

If you want a more optimized blis library, get it from blis @github. The Linux installation proceeds along these lines (AMD zen3 architecture):

```
$ mkdir blis_install
$ cd blis_install
$ git clone https://github.com/flame/blis
$ cd blis
$ ./configure CC=gcc CFLAGS='-Ofast' --prefix=. --enable-cblas=auto zen3
$ make -j 5
$ make install
```

If `blis.pc` is in your `pkg-cfg` path you can find the link instructions

```
$ pkg-cfg blis --libs
```

7.5 View, and deep or shallow copy

7.5.1 Copying Python lists

It's good to know when objects are copied (slow operation) and when only alias names are assigned (fast operation). Putting an "=" sign to equate Python objects causes surprises if you jump back and forth between C++ and Python. From the Python course example,

```
>>>  x = 3
>>>  y = x
>>>  id(x),id(y)
(140128514287328, 140128514287328)
>>>  y = 4
>>>  id(x),id(y)
(140128514287328, 140128514287360)
>>>  x,y
(3, 4)
```

so `x` and `y` are the same thing until you change the contents of either.

> Python creates a copy only on demand or when it has to.

Mutable objects, lists and dictionaries, are more quirky. Take a look how ***shallow lists*** (unnested lists) behave,

```
>>> x = ['a','b']
>>> y = x
>>> id(x),id(y)
(140128370669640, 140128370669640)
>>> y = ['c','d']   # assign the name y for a new list
>>> x,y
```

```
['a', 'b'] ['c', 'd']
>>> id(x),id(y)
(140128370669640, 140128380172808)
```

With the line `y = ['c','d']` I told Python to create a new list, and to "recycle" the old name `y`. The object `y` got a completely new life in Python, all past forgotten.

But suppose I change only *one* value,

```
>>> x = ['a','b']
>>> y = x
>>> id(x),id(y)
(140128380172808, 140128380172808)
>>> y[0]='c'  # set value in-place, use old y
>>> x,y
(['c', 'b'] ['c', 'b'])
>>> id(x),id(y)
(140128380172808, 140128380172808)
```

With the line `y[0]='c'` changed the 0^{th} element of the old `y`, ***in-place***. Here `x` and `y` are still two names for the same thing and also `x` is updated when `y` changed. ***Slicing*** creates a real copy of a shallow list,

```
>>> x = ['a','b']
>>> y = x[:]  # slice of whole x to new list y
>>> id(x),id(y)
(140128370669640, 140128370775368)
```

and they are two separate lists. With sublist things get interesting. Modify the shallow list element,

```
>>> x = ['a',['b','c']]
>>> y = x[:]
>>> id(x),id(y)
(140128380170760, 140128370669640)
>>> y[0] = 'new'
>>> x,y
(['a', ['b', 'c']], ['new', ['b', 'c']])
```

This was expected, but what if we modify the sublist?

```
>>> x = ['a',['b','c']]
>>> y = x[:]
>>> id(x),id(y)
(140128380170760, 140128370669640)
>>> y[1][0]='new'
>>> x,y
(['a', ['new', 'c']], ['a', ['new', 'c']])
```

and you see that ***slicing y=x[:] copies only the references of the sublists in x***, and while the shallow lists in `x` and `y` are separate, they share the sublists. Very economical, but subject to human error. There is a method `deepcopy()` that copies the sublists,

```
>>> from copy import deepcopy
>>> x = ['a',['b','c']]
>>> y = deepcopy(x)
>>> x,y
(['a', ['b', 'c']], ['a', ['b', 'c']])
>>> y[1][0]='new'
```

```
>>> x,y
(['a', ['b', 'c']],  ['a', ['new', 'c']])
```

7.5.2 Converting 2D data: numpy.matrix ↔ numpy.array without copying

Avoid unnecessary copying of data. Let's examine a few cases:

```
>>> import numpy as np
>>> data_array = np.random.random((500,600))
>>> A = np.matrix(data_array)                  # creates a copy
>>> np.shares_memory(data_array,A)
False
>>> B = np.asmatrix(data_array)                # no copying
>>> np.shares_memory(data_array,B)
True
>>> C = np.matrix(data_array,copy=False)       # no copying
>>> np.shares_memory(data_array,C)
True
# now data_array, B, and C share data, modifying any of them changes all
```

Even in the no-copying cases tests `B is data_array` gives `False`. There data bound checking inquiry `numpy.may_share_memory()` is a bit cheeper than `numpy.shares_memory()`.

Conversion of a Python `list` to `numpy.array` can't be done in-place, (a `numpy.array` needs a bit more memory for the metadata),

```
>>> import numpy as np
>>> mylist = [1,2,3,4]
>>> arr1 = np.asarray(mylist)              # creates a copy
```

```
>>> arr2 = np.array(mylist, copy=False)    # creates a copy
```

7.5.3 NumPy arrays: Copying data or Changing View?

Exactly when is data copied and when not? NumPy arrays have a `base` attribute,

```
>>> import numpy as np
>>> a = np.arange(6)
>>> b = a.reshape((2, 3)) #  *new view* to a
>>> b.base is a
True
>>> b[1,1]=100
>>> a
[  0   1   2   3 100   5]
>>> b
[[  0   1   2]
[  3 100   5]]
```

so as the similar `base` suggested, they share the same memory and modifying either `a` or `b` will change both. A direct NumPy method `may_share_memory` agrees,

```
>>> import numpy as np
>>> a = np.arange(6)
>>> b = a.reshape((2, 3))
>>> np.may_share_memory(a,b)
True
```

A NumPy array is ***raw data*** + ***view***, raw data and information how to interpret it. Some operations copy the raw data (slow), some just create a new view (fast).

NumPy arrays have an attribute flags,

```
C_CONTIGUOUS (C)    The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)    The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)         The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)       The data area can be written to. Setting this to False locks the data, making it read-only.
                        A view (slice, etc.) inherits WRITEABLE from its base array at creation time,
                        but a view of a writeable array may be subsequently locked while the base array
                        remains writeable. (The opposite is not true, in that
                        a view of a locked array may not be made writeable.
                        However, currently, locking a base object does not lock
                        any views that already reference it, so under that circumstance
                        it is possible to alter the contents of a locked array via a previously
                        created writeable view onto it.) Attempting to change a non-writeable array
                        raises a RuntimeError exception.
ALIGNED (A)          The data and all elements are aligned appropriately for the hardware.
UPDATEIFCOPY (U)     This array is a copy of some other array.
                        When this array is deallocated, the base array will be
                        updated with the contents of this array.
FNC                  F_CONTIGUOUS and not C_CONTIGUOUS.
FORC                 F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)          ALIGNED and WRITEABLE.
CARRAY (CA)          BEHAVED and C_CONTIGUOUS.
FARRAY (FA)          BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.
```

```
>>> a = np.arange(6)
>>> b = a.reshape(2, 3)
>>> print(a.flags)
```

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> print(b.flags)
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

A 1D array is always contiguous both in C and in fortran, because there's only one index running through memory locations. In memory, `a` is stored as

`something else ...  , a[0], a[1], a[2], a[3], a[4], a[5] , ...  something else`

A 2D arrays is by default always contiguous in C, `b` is stored as

`something else ..., b[0,0], b[0,1], b[0,2], b[1,0], b[1,1], b[1,2], ...  something else`

This is not contiguous in fortran, because fortran would've used the order

`something else ..., b[0,0], b[1,0], b[0,1], b[1,1], b[0,2], b[1,2], ...  something else.`

7.5.4 NumPy: `ndarray.resize()` or `numpy.resize()`?

NumPy arrays have a method `.resize()`, but there is also a NumPy function `resize()`. What's the difference?

NumPy method `.resize()`

Resizing an object whose data is viewed by another object fails, and you can't resize an object that doesn't own it's data.

```
>>> a = np.arange(6)   # a owns the data
>>> b = a.reshape(2,3) # b is a view of a, b doesn't own the data
>>> a.resize(4,2)
Traceback (most recent call last):
ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
>>> b.resize(4,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot resize this array: it does not own its data
```

The method `a.resize()` failed because it tries to resize the array in-place, but that would change `b` as well. Setting `refcheck=False` (as in `a.resize(4,2,refcheck=False)`) skips the reference checks and you get the speed boost of the `resize()` method.

NumPy function `resize()`

```
>>> a = np.arange(6)
>>> b = a.reshape(2,3)
>>> a = np.resize(a,(4,2))
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [0, 1]])
>>> b
[[0 1 2]
```

```
 [3 4 5]]
```

The line `a = np.resize(a,(4,2))` stores the created 8-element array back to `a`; for arrays `a` and `b` this is the parting of the ways. The extra data occupying `a` is made up according to the rule "... repeated if necessary to fill out the required number of elements. The data are repeated in the order that they are stored in memory." Hence the recycled values 0, 1 in the end.

Remark: A funny way of creating a 50 element array [0,1,0,1,0,1....1] would be (btw, this works with any pattern)

```
>>> a = np.resize([0,1],50)
>>> a
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
 1 0 1 0 1 0 1 0 1 0 1 0 1]
```

7.5.5 More array slicing

In Python ***negative indices count from the end***. For example, `b[0:-3]` is all but the last three elements.

```
>>> import numpy as np
>>> a = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a
array([[1 2 3]
       [4 5 6]
      [7 8 9]])
>>> a[:,:-1] # all rows, all but the last column
array([[1 2]
       [4 5]
      [7 8]])
>>> b = np.array([1,2,3,4,5,6])
>>> b
```

```
array([1 2 3 4 5 6])
>>> b[::-1]  # reverse, ''read from the end with step -1''
array([6 5 4 3 2 1])
>>> b[::2]  # every second element, step 2
array([1 3 5])
>>> b[1:2]  # element 1, stop before element 2
array([2])
```

Reversing `b[::-1]` and picking every second element `b[::2]` work also for Python lists. However, `a[:,:-1]` doesn't, because for lists `:,:-1` is interpreted as a tuple, and list indices can't be tuples. For list of lists, such as `a=[[1,2,3],[4,5,6]]`, element 6 is adressed as `a[2][1]`, not as `a[2,1]`.

7.5.6 Curiosity: How to set temporary NumPy print options

The function `np.set_printoptions()` changes NumPy print options from that point on. Sometimes it's preferable to just temporarily change options. This can be achieved using a ***context manager***[34].

[34]This elegant context solution is by unutbu @Stackoverflow .

numpy_temporary_print_options.py

```python
# setting temporary numpy print options
import numpy as np
import contextlib

class MyPrintOptions:
    @contextlib.contextmanager
    def __printoptions(self,*args, **kwargs):
        original = np.get_printoptions()
        np.set_printoptions(*args, **kwargs)
        try:
            yield
        finally:
            np.set_printoptions(**original)

    def is_numerical(self,arg):
        try:
            1+arg
            return True
        except:
            return False

    def print(self,*args,precision=3):
        if self.is_numerical(args[0]):
            tit=''
        else:
            tit,*args= args

        if np.isscalar(*args):
            form = "{:<."+str(precision)+"f}"
            print(tit,form.format(*args))
        else:
            with self.__printoptions(precision):
                print(tit,*args)

if __name__=='__main__':

    A = np.random.random((3,2)) # try A=1234.567890123456
    print("NumPy default output:\n",A)
    mypops = MyPrintOptions() # instantiate class
    print("MyPrintOptions default 3 decimal output:")
    mypops.print(A)
    print("MyPrintOptions 12 decimal output:")
    mypops.print(A,precision=12)
    print("back to NumPy default output:\n",A)
```

The `np.isscalar()` test is there because

> `numpy.set_printoptions()` affects only NumPy arrays, NumPy passes a scalar back to Python

so all formatting has to be done there. Here I used the `format` method.

A hackish way to find out if `arg` is a number:

```
class My:
    def is_numerical(self,arg):
          try:
                1+arg
                return True
          except:
                return False
```

This relies on the fact that Python raises an exception (`TypeError`) if `arg` can't be added to a number. You could, of course, check `arg`'s type against known numerical types - What a bore! Using `try-except` is close to Python's EAFP heart:

> ***EAFP***: ***E***asier to ***A***sk for ***F***orgiveness than ***P***ermission.

Perhaps this is close to the idea,

NON-EAFP:

If you are a pilot or a bird or an aeroplane, then please fly.

EAFP:

Forgive me, could you please fly?

You see the power of EAFP when you have added a kite, a frisbee, and a flying squirrel.

7.6 NumPy matrix operations

Basic matrix operations are very fast in NumPy:

matrixops.py

```
import numpy as np
A = np.random.randint(10, size=(4,4))
A = A + np.transpose(A)
print('A =\n',A)
print('Rank(A) =', np.linalg.matrix_rank(A))
print('Tr(A)   =', np.trace(A))
print('Det(A)  =', np.linalg.det(A))
print('Inv(A)  =', np.linalg.inv(A))
print('   A^3  =', np.linalg.matrix_power(A, 3))
# eigenvalues:
lam, eigv = np.linalg.eig(A)
print('\n')

for i,l in enumerate(lam):
    print('i =',i,' l =',l)
    Av = A@eigv[:,i]
    lv = l*eigv[:,i]
    print('   Av =',Av)
    print('  l v =',lv)
```

matmul_example.py

```
# matrix multiplication operator @
import numpy as np
A = np.random.random((3, 3)) # random 3x3 matrix
B = np.random.random((3, 3))
C = A@B
print(C)
np.set_printoptions(precision=3)
print(C)
'''
[[ 0.90608775  1.21181521  1.11203345]
 [ 0.91196897  1.32365173  1.15275434]
 [ 0.85486192  1.25971373  1.1405692 ]]
[[ 0.906  1.212  1.112]
 [ 0.912  1.324  1.153]
 [ 0.855  1.26   1.141]]
'''
```

More about `numpy.set_printoptions()`

NumPy stores arrays in ***row-major order, meaning the right-most index changes fastest***. This is the same was as in C/C++, whereas fortran and Matlab use column-major order. Neither is better than the other. Which ever you choose, accessing elements consequtive in memory is fast, accessing elements stored far apart in memory is slow.

indexorder.py

```
import numpy as np
from time import process_time as T

N=10000
a = np.random.rand(N,N,3)
# values are stored  a[0,0,0] a[0,0,1] a[0,0,2] ... a[0,0,N] a[0,1,0] a[0,1,1] ...
tic = T()
a[:,:,0] = a[:,:,1] # values are far in memory, collecting them is slow
a[:,:,2] = a[:,:,0]
a[:,:,1] = a[:,:,2] # ahem, this is *not* a swapping operation
toc = T() - tic
print(toc,"seconds using (N,N,3) in default row major order")

# change index order to fortran's column major order
# values are stored  a[0,0,0] a[1,0,0] a[2,0,0] ... a[N,0,0] a[0,1,0] a[1,1,0] ...
a = np.random.rand(N,N,3)
tic = T()
a = np.asfortranarray(a) # this takes time
a[:,:,0] = a[:,:,1] # values are consequtive in memory, collecting them is fast
a[:,:,2] = a[:,:,0]
a[:,:,1] = a[:,:,2]
toc = T() - tic
print(toc,"seconds using (N,N,3) in column major order")

# same size task, but define the array in different order to begin with
a = np.random.rand(3,N,N)
tic = T()
a[0,:,:] = a[1,:,:] # values are consequtive in memory, collecting them is fast
a[2,:,:] = a[0,:,:]
a[1,:,:] = a[2,:,:]
toc = T() - tic
print(toc,"seconds using (3,N,N) in default row major order")
#0.9668405870000001 seconds using (N,N,3) in default row major order
#0.9624588530000002 seconds using (N,N,3) in column major order
#0.12479739399999978 seconds using (3,N,N) in default row major order
```

The difference in second and third is entirely due to the slow operation `a = np.asfortranarray(a)`. After that conversion is done, the remaining tasks are similar. This underlines how important it's to choose an efficient data representation right

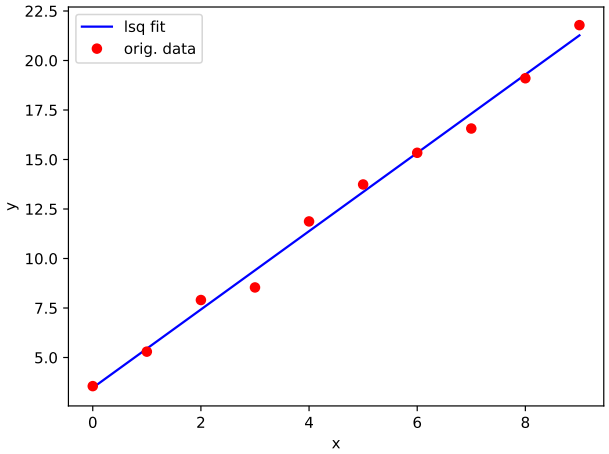
from the beginning. You can ask what storage order is use with `print(a.flags)`.

Linear regression

Here's a NumPy take on linear regression.

linregression.py

```
# Linear regression a.k.a. least square fit to a linear function
import numpy as np
import matplotlib.pyplot as plt

# test data
x = np.arange(10)
y = 2*x + 3 + (np.random.random(x.size)-0.5)*2  # y = 2x+3+noise

A = np.array([x, np.ones(x.size)])
print(A)
# linear regression
c = np.linalg.lstsq(A.T, y)[0]
print("linear regression coeffs:",c)
plt.plot(x, c@A, 'b-',label='lsq fit') # c@A is c[0]*x+c[1]
plt.plot(x, y, 'ro',label='orig. data')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



7.7 NumPy broadcasting instead of `for`-loops

Nested `for`-loops,

```
for i in a:
    for j in b:
        for k in c:
```

may be performance bottlenecks because Python is an interpreted language. This is the only reason, there's nothing wrong with loops *per se*. People have come up with a few ways to avoid `for`-loops in Python - something we never need to do in compiled languages.

As we saw earlier, list comprehensions are sometimes useful. For example

```
>>> import numpy as np
>>> a = range(100)
>>> s = []
>>> for i in a:
>>>    for j in a:
>>>        for k in a:
>>>            s.append([i,j,k])
>>> s = np.array(s)
```

can be written in a more readable form using list comprehension,

```
>>> import numpy as np
>>> a = range(100)
>>> s = [[i,j,k] for i in a for j in a for k in a]
>>> s = np.array(s)
```

and the code is a bit faster, but we're after a bigger fish.

Before continuing, I give a general advice:

> If you have a nested `for`-loop ***and*** it's a performance bootleneck, keep the loops and try Numba `jit` or `njit`, see chapter 11.

If Numba doesn't do what you need, keep reading.

I recommend talks given by Jake VanderPlas, one is about losing loops. NumPy has some very nice rules that does what loops would but a lot faster. ***Broadcasting rules*** tell how NumPy does an operation with two arrays with non-matching shapes. For example, subtracting a number from *every* element - that *every* would be the loop index we want to avoid:

```
>>> import numpy as np
>>> x = np.array([1,2,3,4,5])
>>> x.shape  # shape (5,)
>>> x-2      # array x and number 2 have different shapes
```

where NumPy "clones" number 2 to match the array `x`, something like

```
"x-2 = (1,2,3,4,5) - (2,2,2,2,2) = (-1,0,1,2,3)"
```

Suppose you have a 1D array `x` with elements x_i and you want to create a difference matrix with

$$d_{ij} = x_i - x_j \ . \tag{1}$$

If you want speed, the indices i, j should not appear in the Python program. Obviously `d = x - x` won't do (why ?). You want to trick NumPy to *clone* every element x_j in `x` and subtract that from all elements x_i. This is something NumPy broadcasting is good at.

NumPy broadcasting rules:
From right to left, look for a matching dimension or 1 in the two arrays and clone the arrays to match in shape.

The final shape is not necessarily the shape of either of the two input arrays.
Looking only at array shapes and some operator `"op"`, this is how broadcasting re-shapes arrays:

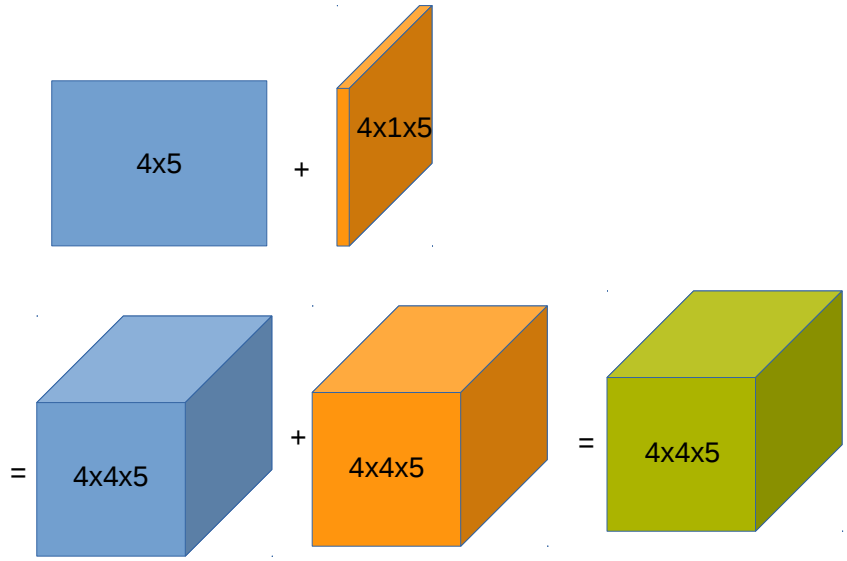
```
(5,) op (6,)                Error, no broadcasting rule
(5,1) op (6,) = (5,6)
(4,5) op (5,) = (4,5)
(4,5,6) op (5,6) = (4,5,6)
```

```
(4,5) op (4,1,5) = (4,4,5)   Note: not (4,5,5)
```

Pictorially, the "(4,5)+(5,) = (4,5)" looks like

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

I added the parenthesis for clarity, but, as you notice, ***NumPy broadcasting rules are not math rules***. Rather, think about them as rules that are plausible and do a well defined, unique operation. The 3D arrays are even more interesting, in the following example both arrays need cloning:



I leave the pictorial representation of 4 and higher dimensional cases for you! Let's get back to computing distances $d_{ij} = x_i - x_j$, for N=5 particles in D=1 dimensional space. The coordinate array is the NumPy array `x` of shape (5,). If you try `x-x`, that's (5,)-(5,)=(5,) and you get just five zeroes. But ***you can make NumPy broadcasting to clone the second x***,

if you compute the difference for array shapes (5,1)-(5,) = (5,5). That (5,5) is the shape of the difference matrix in 1D. All you need is to reshape `x` to (5,1) and subtract the original `x`. Notice that we don't change the data, just take two different views of it.

Adding a dimension to an array

Adding an extra dimension - or an ***axis*** - is easy and cheap,

```
>>> import numpy as np
>>> x = np.array([1,2,3,4,5])  # (5,)
>>> d = x[:,np.newaxis] - x    # (5,1)-(5,) => NumPy broadcasting => (5,5)
>>> d
[[ 0 -1 -2 -3 -4]
[ 1  0 -1 -2 -3]
[ 2  1  0 -1 -2]
[ 3  2  1  0 -1]
[ 4  3  2  1  0]]
```

If you prefer, you can also use the method `.reshape()`, in the example `x.reshape(5,1)` would be the same as `x[:,np.newaxis]`. You can add dimensions anywhere,

```
>>> import numpy as np
>>> y = np.ones((5,3,2), int)
>>> y.shape
(5,3,2)
>>> y[:,:,np.newaxis,:].shape
(5,3,1,2)
```

Let's compute the potential energy of particles,

$$V_{\text{pot}} = \sum_{i<j,i,j=1}^{N} V(|\boldsymbol{x}_i - \boldsymbol{x}_j|) \ ,$$

where the pair interaction $V(r)$ depends only on the interparticle distance r, the i,j-distance is $|\boldsymbol{x}_i - \boldsymbol{x}_j|$. The position vectors $\boldsymbol{x}$ are given as NumPy arrays `x` of shape (N,D) for N particles in D dimensions. Notice the chosen index order: I'm assuming the coordinate data is mostly accessed so that the dimension index runs fastest, so Python's column-major order (particle index, dimension index) is efficient. For three dimensions, the data in memory would be

other data	x(0,0)	x(0,1)	x(0,2)	x(1,0)	x(1,1)	...	x(N,0)	x(N,1)	x(N,2)	other data

To summarize, the coordinates are in an array of shape (N,D), and the distance vector $\boldsymbol{x}_i - \boldsymbol{x}_j$ is a three-index array $(i, j, \textit{dimension})$ of shape (N,N,D). We can use the broadcasting rule that takes two (N,D) arrays and gives an (N,N,D) array, schematically[35]

```
(N,1,D) - (N,D) = (N,N,D)
```

Remark: If you like, you can first write the operation for particles i and j for dimension k. The coordinates are (i,k) and (j,k), so the distance in dimension k is (i,1,k) - (j,k) = (i,j,k), and *vectorize* the operation for i and j in all N coordinates and k in all D dimensions.

The resulting distance vector array is symmetric with zero diagonal, so there's redundant information. All we need is the vector lengths, so we take the lower (or upper) triangle, and compute the vector lengths, in math $|\boldsymbol{x}_i - \boldsymbol{x}_j| = \sqrt{\sum_k (x_{i,k} - x_{j,k})^2}$. This function does the job:

[35] It's easier to get an (N,N,D) array from two (N,D) arrays with broadcasting, than an (D,N,N) array from two (D,N) arrays.

potential_simple.py

```python
import numpy as np

def V(r):
    return 2*r   # just for testing

def PotentialEnergy(x):
    d  = x[:,np.newaxis,:]-x          # broadcasting, d.shape is (N,N,D)
    r  = np.sqrt((d**2).sum(2))       # aggregation, sum of squares over dimensions (3rd index) , r.shape is (N,N)
    rs = r[np.triu_indices_from(r,1)] # r is now upper triangle without diagonals, rs.shape is (N*(N-1)/2,)
    return rs.min(),np.sum(V(rs))

if __name__=='__main__':
    # 3 particles in 2 dimensions
    x = np.array([[3,1],[4,3],[2,2]])   # shape (3,2)
    rmin,Vpot = PotentialEnergy(x)
    print(f"minimum distance = {rmin:.5f}")
    print(f"potential energy = {Vpot:.8f}")
```

The next code compares this with a nested for-loop.

potential_energy.py

```python
# potential energy of N particles in 3D space:  sum_i<j V(r_ij)
import numpy as np

# pair potential  (Lennard-Jones 6-12)
def V(r):
    return r**-12 - r**-6

# potential energy
def PotentialEnergy(x):
    d  = x[:,np.newaxis,:]-x     # broadcasting, d.shape is (N,N,D)
    r  = np.sqrt((d**2).sum(2))  # aggregation, sum of squares over dimensions (3rd index) , r.shape is (N,N)
    rs = r[np.triu_indices_from(r,1)] # collect upper triangle without diagonals, rs.shape is (N*(N-1)/2,)
    return rs.min(),np.sum(V(rs))

def PotentialEnergyLoop(x):
    pot = 0
    rmin = np.inf
    N = x.shape[0]
    for i in np.arange(0,N-1):
        for j in np.arange(i+1,N):
            r = np.sqrt( ((x[i]-x[j])**2).sum() )
            if r<rmin: rmin = r
            pot += V(r)
    return rmin,pot

def output(title,m,v,timing):
    print('\n',title)
    print(f'    min distance {m:.5f}')
    print(f'potential energy {v:.5f}')
    print(f'calculation took {timing:.3f} seconds')

if __name__=='__main__':
    x = 100*np.random.random((1000,3))  # 1000 points in 3D space
    from time import process_time as T

    tic = T()
    m,v  = PotentialEnergy(x)
    t1 = T()-tic
    output('Broadcasting and aggregation',m,v,t1)

    tic = T()
    m,v  = PotentialEnergyLoop(x)
    t2 = T()-tic
    output('Explicit for-loops',m,v,t2)
    print(f'\n Speedup: Broadcasting was {t2/t1:.1f} times faster than nested for loops')
```

7.8 NumPy `einsum` tensor operations

The name `einsum` refers to the Einstein summation convention. If you are comfortable with multiple indices, then `einsum` can do `transpose`, `sum`, `multiply`, and much mode. Even better, `einsum` is memory efficient and fast - these two qualities often go hand in hand. `einsum` is for tensor operations, so no wonder it's also built into PyTorch and Tensorflow.

7.8.1 Computing $D_i = \sum_j A_i B_{ij}$

Alex Riley has a nice blog post about the power of `einsum`. He demonstrates how array manipulation of array shapes (3,) and (3,4) can be done. In math, the example is about computing

$$D_i = \sum_j A_i B_{ij} \ .$$

Using NumPy broadcasting and sum
Let's split the task to two parts,

$$D_i = \sum_j C_{ij} \quad , \quad \text{where } C_{ij} = A_i B_{ij} \ ,$$

and do the latter with broadcasting. Shapes are, from left to right, (3,4) = (3,)*(3,4), so for broadcasting you'd want to multiply shapes (3,1)*(3,4),

```
>>> import numpy as np
>>> A = np.array([0,1,2])
>>> B = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
>>> C = A[:,np.newaxis]*B  # shapes  (3,1) and (3,4) => broadcasts to (3,4)
>>> D = C.sum(1) # same as C.sum(axis=1), D is [0 22 76]
```

Broadcasting and sum can be calculated together,

```
D = (A[:,np.newaxis]*B).sum(1)
```

The only problem is that for `A[:,np.newaxis]*B` NumPy creates a ***temporary array*** before doing the sum.

Using NumPy `einsum`

Read the math expression $D_i = \sum_j A_i B_{ij}$ as

$$\sum_j A_i B_{ij} \to D_i \tag{2}$$

and j is summed over, so with the Einstein summation convention we get

$$A_i B_{ij} \to D_i \ , \tag{3}$$

which is `D = np.einsum('i,ij->i',A,B)`:

```
>>> import numpy as np
>>> A = np.array([0,1,2])
>>> B = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
>>> D = np.einsum('i,ij->i',A,B)
>>> D
[0 22 76]
```

You know that j is summed over because it doesn't appear in the result after `->`.

Remark: Element-wise products without any summations are also simple to code using `einsum`. For example, matrix C with elements $C_{ij} = A_i B_{ij}$ can be cast as $A_i B_{ij} \to C_{ij}$, and the `einsum` code is

```
C = np.einsum('i,ij->ij',A,B)
```

Notice how simple `einsum` made this look!

All basic operations that can be done using NumPy methods, such as `transpose()`, `trace()`, and `inner_product`, have their `einsum` equivalents.

```
import numpy as np
A   = np.array(range(9)).reshape(3,3)  # shape is (3,3)
AT  = np.einsum('ij->ji',A)            # transpose
TrA = np.einsum('ii->',A)              # trace
B   = np.array(range(12)).reshape(3,4) # shape is (3,4)
AB  = np.einsum('ik,kj->ij',A,B)       # matrix multiplication, same as AB=A@B
ABe = np.einsum('ij,ij->ij',A,B)       # elementwise multiplication, same as AB=A*B
C   = np.einsum('ij,jk->ij',A,B)       # inner product, same as C=np.inner(A,B)
```

In math, the last operation is $C_{ij} = \sum_k A_{ij} B_{jk}$.

7.8.2 Potential energy calculation with NumPy `einsum`

Returning to the potential energy calculation, the task was

$$V = \sum_{i<j,=1}^{N} V(r_{ij}) \quad , \; r_{ij} = \sqrt{\sum_k (d_{ijk})^2} \quad , \quad d_{ijk} = x_{ik} - x_{jk} \; .$$

`einsum` can compute r_{ij} a bit faster,

```
def PotentialEnergy(x):
    d  = x[:,np.newaxis,:]-x
    r  = np.sqrt(np.einsum('ijk,ijk->ij',d,d))  # array with elements rij
```

```
    rs = r[np.triu_indices_from(r,1)]
    return rs.min(),np.sum(V(rs))
```

This broadcasting + einsum turned out to be almost twice as fast as broadcasting + sum accregation.

7.8.3 `einsum` optimization

For example NumPy `inner` and `dot` can link to external libraries that do the operations very fast. IMHO, there should be no reason why `numpy.einsum('ij,jk->ij',A,B)` shouldn't produce as fast code as `numpy.inner(A,B)`.

The project `opt_einsum` provides a drop-in replacement to `einsum` and claims to be faster. Installation:

```
$ python -m pip install opt_einsum --user
```

The project page provides an example of tensor contraction, and true, `opt_einsum` was 1000 times faster than `einsum`. But there's more to it, namely NumPy `einsum` has the option `optimize`. A snipped from the code:

```
optimize : {bool, list, tuple, 'greedy', 'optimal'}
        Choose the type of path. If a tuple is provided, the second argument is
        assumed to be the maximum intermediate size created. If only a single
        argument is provided the largest input or output array size is used
        as a maximum intermediate size.

        * if a list is given that starts with "einsum_path", uses this as the
          contraction path
        * if False no optimization is taken
        * if True defaults to the 'greedy' algorithm
        * 'optimal' An algorithm that combinatorially explores all possible
          ways of contracting the listed tensors and choosest the least costly
          path. Scales exponentially with the number of terms in the
          contraction.
        * 'greedy' An algorithm that chooses the best pair contraction
          at each step. Effectively, this algorithm searches the largest inner,
          Hadamard, and then outer products at each step. Scales cubically with
          the number of terms in the contraction. Equivalent to the 'optimal'
          path for most contractions.

        Default is 'greedy'.
```

So let's add this iPython code to the game,[36]

tensor_contraction.ipy

```
import numpy as np
from opt_einsum import contract

N = 10
C = np.random.rand(N, N)
I = np.random.rand(N, N, N, N)

print(f"{'einsum no optimization:':30}",end =" ")
%timeit np.einsum('pi,qj,ijkl,rk,sl->pqrs', C, C, I, C, C)
print(f"{'einsum optimization optimal:':30}",end =" ")
%timeit np.einsum('pi,qj,ijkl,rk,sl->pqrs', C, C, I, C, C, optimize='optimal')
print(f"{'einsum optimization greedy:':30}",end =" ")
%timeit np.einsum('pi,qj,ijkl,rk,sl->pqrs', C, C, I, C, C, optimize='greedy')
print(f'{"opt_einsum.contract:":30}',end =" ")
%timeit contract('pi,qj,ijkl,rk,sl->pqrs', C, C, I, C, C)
```

The line

```
%timeit np.einsum('pi,qj,ijkl,rk,sl->pqrs', C, C, I, C, C)
```

is ***iPython line magic***; the % character makes `timeit` take as an argument the rest of the line.

IPython line magic needs the iPython shell

```
$ ipython
In[1]: %load tensor_contraction
```

[36] Can you reverse engineer what the code computes? For comparision, the Python script `tensor_contraction.py` shows how to use `timeit.timeit()`.

and press enter. The `%load` is also line magic.[37]
Timings for $N = 10$

```
einsum no optimization:        516 ms ± 383 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
einsum optimization optimal:   974 µs ± 3.42 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
einsum optimization greedy:    237 µs ± 749 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
opt_einsum.contract:           528 µs ± 857 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

To make the differences more pronounced try $N = 12$

```
einsum no optimization:        2.21 s ± 18.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
einsum optimization optimal:   1.03 ms ± 3.49 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
einsum optimization greedy:    301 µs ± 2.26 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
opt_einsum.contract:           594 µs ± 752 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The speed boost from optimization is significant in the human time scale.

NumPy `einsum` promotion problem

As Alex Riley (see below) points out, be aware that data types are not promoted in `einsum`, in other words, make sure the result fits to the original data type. He gives the following example:

```python
import numpy as np
a = np.ones(300, dtype=np.int8)
print(np.sum(a))
# 300,  correct result
print(np.einsum('i->', a))
# 44, wrong result
```

The problem was that the correct result 300 won't fit to the original `np.int8` integer type.

[37] Cell magic `%%timeit` times the whole cell multiple lines.

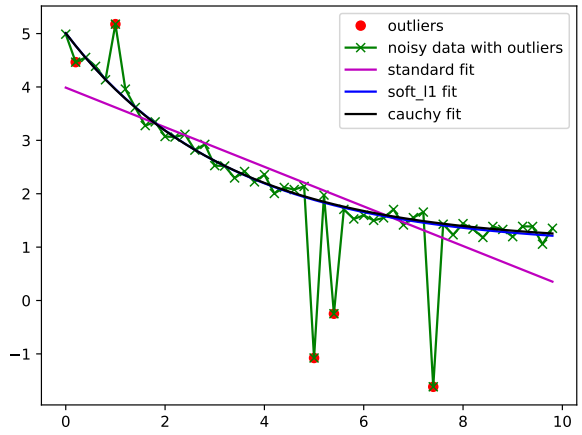
8 SciPy

SciPy adds scientific tools, such as linear algebra, integration, ordinary differential equation solving, and signal processing, to the Python toolbox. I'm not going to dive in, just say that if you use SciPy you are already doing well. Demo 2 uses SciPy to compute inter-particle distances, and I have a few sample codes that use SciPy, `numerics/gaussian_process_intro.py`, `numerics/mpi4py_eigvals.py`, and `numerics/scipy_robust_regression.py`, which I discuss here.

8.1 SciPy robust regression

We did linear regression in NumPy (7.6). SciPy robust regression can take care of outliers (badly deviating data points) and other problems. The fitting module is scipy.optimize.least_squares, and knows, e.g., trust-region, dog-leg, and Levenberg-Marguardt algorithms. Fitting to a bounded function, such as $a\sin(bx)/x$ with parameters a and b, can be done with the standard least squares method, but fitting to exponents is more volatile, as this example shows:

function	$a + b\exp(cx)$		
input data	$a = 1$	$b = 4$	$c = -0.3$
Fitting to 50 points, with some noise and a few outliers:			
standard	16482.6791514	-16478.4656682	2.21543558952e-05
soft_l1	0.958414361752	4.11960272585	-0.303014525135
cauchy	0.860903786064	4.15712082527	-0.285476235032



The fit used three functions (most examples in the net lump them together to one `generdata()` function, but I opted to keep them separate)

```
# input data
def fun(x):
    return 1.0 + 4.0*np.exp(-0.3*x)

# parametrized model function
def fitfun(x,*par):
    a,b,c,*par = par
    return a+b*np.exp(c*x)

# residual of model-data
def resfun(par, x, y):
    return fitfun(x,*par) - y
```

A technical detail: functions such as $2\sin(3x)/x$ give numerically a division-by-zero at $x = 0$, so let's see how to fix that. Actually there's a built-in function that computes $\sin(x)/x$ correcty also at $x = 0$, but that's besides the point.

```
# First attempt to avoid 1/0
def fun(x):
    x[x == 0] = 1e-10 # handle division-by-zero; still one pitfall, see later
    return 2.0*np.sin(3.0*x)/x
```

This vectorizes with a NumPy array `x`. The line `x[x == 0] = 1e-10` sets all zeros in the input to a small value, but you can't do that for a scalar `x`, so trying to patch

```
# Second attempt to avoid 1/0
# BAD, non-EAFP
def fun(x):
    if isinstance(x,(list, tuple, np.ndarray)) :
        x[x == 0] = 1e-10
    else:
        x = 1e-10
    return 2.0*np.sin(3.0*x)/x
```

A `try-except` is more pythonic,

```
# A fairly good solution, also EAFP
def fun(x):
    try:
        x = x.astype(float,copy=False)  # ugly but necessary
        x[x == 0] = 1e-10
    except:
        x = 1e-10
```

```
    return 2.0*np.sin(3.0*x)/x
```

The line `x = x.astype(float,copy=False)` makes sure the array is floats; if not, make a *copy* to a float array. This is a precaution against an interesting mistake, in Python console

```
>>> import numpy as np
>>> x = np.array([1,0,2,3])
>>> x[x == 0] = 1e-10 # detects correctly zeroes in the array, but ...
>>> x
array([1, 0, 2, 3])
```

The problem is that *an integer array can't hold floats* and now this works against our good intentions to avoid zeros.

You can, of course, sacrifice some accuracy - and add a hard-to-spot bug -

```
# A potentially dangerous way to avoid 1/0
def fun(x):
    return 2.0*np.sin(3.0*x)/(x+1e-10)  # ok, unless you have x=-1e-10 :^)
```

You may choose just to ignore the division-by-zero warnings and make sure the rest of the code is not choking if it gets a few NaNs (NaN means "Not a Number"). SciPy gets very upset about NaNs. Yet another, post-compute remedy: `numpy.nan_to_num()` replaces infinities with a finite huge value and NaNs with zeroes.

The actual fitting and extracting the fitted parameters was done like this:

```
# (x,y) is the input data we try to fit, par0 is the initial guess (1,1,1)
fit = least_squares(resfun, par0, loss='soft_l1', f_scale=0.1, args=(x,y))
a,b,c = fit.x   # extract parameters a,b,c
```

In addition to the fitted parameters, the object `fit` contains lots of status information, such as the cost function and the Jacobian.

Let's say you have found options in `optimize.least_squares` that almost always works for your problems. You may consider hiding details, there's no need to tell everywhere that you are using Scipy least squares with this loss function and that noise scale if it's your default: see the sample code `numerics/fit_adapter.py`.

8.1.1 Simplified Function Interface with `functools.partial`

It may be an overkill to write a simplified caller as a class method. Suppose you are always calling `scipy.least_squares()` with `loss='soft_l1',f_scale=0.1` and just don't want to repeat parts of the call? The `functools.partial` does a bit what `std::bind` does in C++, it creates a function with some arguments frozen, see `fit_adapter_partial.py`. The way `functools.partial` works is simple,

```
from functools import partial

def f(x,y):
   return(x+y)

add1  = partial(f,1)
add11 = partial(f,11)
print(add1(5))
print(add11(5))
# output:
# 6
# 16
```

9 Pandas

Pandas @pydata.org is a software library written for data manipulation and analysis. It adds ***DataFrames*** to python, a structure that made R data analysts favorite. In short, Pandas makes accessing, indexing, merging, and grouping data easy.

Pandas is build on NumPy, it's all about representation of data. You may probably do without Pandas and live with "plain NumPy", but Pandas is a very attractive data framework. Often Pandas users are less worried about raw speed than fluent data management, but anyhow speed comparisons don't flatter Pandas. A recent speed comparison between NumPy and Pandas was done by Goutham Balaraman, reported here. The outcome was that NumPy takes less memory and is faster for less than 50K rows, while Pandas is better at above 500K rows. Between that, it depends. Some say Pandas takes 20x longer than NumPy for very large data sets.

What is a DataFrame? From Pandas web page, the `pandas.DataFrame` is

> *Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for* `Series` *objects.*

A `Series` is a one-dimensional labeled array capable of holding any data type,

```
import pandas as pd
s = pd.Series([1, 3, 5, 7],  index=['a',  'b',  'c',  'd'])
```

a	1
b	3
c	5
d	7

A `DataFrame` can be created from `data`,

```
import pandas as pd
import numpy as np
data = {'Programming task': ['easy','usual','difficult','insanely difficult'],
'Time taken': ['5 min','20 min','1 hour','finish elsewhere'],
'Credits': ['approving nod',np.random.random(),'30','supervisor takes the credit']}
df = pd.DataFrame(data,columns=['Programming task','Time taken','Credits']) # default index
df.to_excel('tmp.xlsx',  sheet_name='Sheet1') # Excel 2010 format, uses openpyxl
df.to_csv('tmp.csv',sep='\t') # CSV format, separator is tab
```

```
print(df)
```

screen output:

```
     Programming task        Time taken                       Credits
0                easy             5 min                             1
1              usual             20 min                            10
2           difficult             1 hour                            30
3  insanely difficult  finish elsewhere  supervisor takes the credit
```

More examples on data exploration in blog post by Sunil Ray .

10 NumExrp

The package NumExpr @github.com uses a vector-based virtual machine to perform elementwise operations very fast. From introduction to NumExpr , *The virtual machine uses "vector registers": each register is many elements wide (by default 4096 elements). The key to NumExpr's speed is handling chunks of elements at a time.*

The interface can't get any simpler, a pair potential calculation could be

```
import NumExpr as ne

# pair potential  (Lennard-Jones 6-12)
def V(r):
    return ne.evaluate("r**-12 - r**-6") # uses NumExpr to evaluate the potential energy
```

and your potential energy code suddenly may get about two times faster! Sometimes `NumExpr` performs fabulously, just don't expect miracles for small arrays where NumPy excels and the overhead of compilation is significant.

11 Numba

Just-in-time compilation (JIT @Wikipedia) means compilation at run time, during execution. One JIT compiler is provided by Numba @pydata.org . Numba `jit` tries to convert the Python code to C and compiles it and runs in C speed. ***The simplest way to get all benefits of a C extension!*** Using `Numba` you can ***decorate*** the function that was a speed bottleneck with `@jit`:

```
from Numba import jit
@jit
def bottleneck():
    ...function body...
```

Typing variables may give a speed boost,

```
from Numba import jit, double
@jit(double(double[:],double[:]))
def bottlenect(x,y):
   ...function body...
   return res
```

where `x,y` are (e.g. NumPy) arrays of 64-bit floats and `res` is a 64-bit float - `double` is the same as `float64`, see Numba types @pydata.org . Typing creates the one and only ***specialization***[38] of the function. Try `Numba.typeof(a)` to find how Numba types an object `a`.

[38] Means just that there is no longer `bottleneck()` that has e.g. integers or characters as arguments. More about specializations in the C++ section.

11.1 Numba `jit` options

It's usually a good idea to try and force a function to be compiled so that it doensn't use the CPython interpreter and raise an error if this is not possible. This is done by the decorator `@njit`, which is an alias for `@jit(nopython=True)`,

```
from Numba import njit
@njit
def f(x):
    ...function body...
```

Remark: A few other `@jit` options:
`parallel=True` tries to auto-parallelize loops
`cache=True` use a cache file to shorten compilation times when the function was already compiled in a previous invocation.
`nogil=True` tries to release the global interpreter lock (GIL); possible only if Numba can compile the function in `nopython` mode (warns if not).
Multiple signatures can be defined simultanenously (this one creates two specializations),

```
from Numba import jit, int32, float32
@jit(["int32(int32)", "float32(float32)"])
def f(x):
```

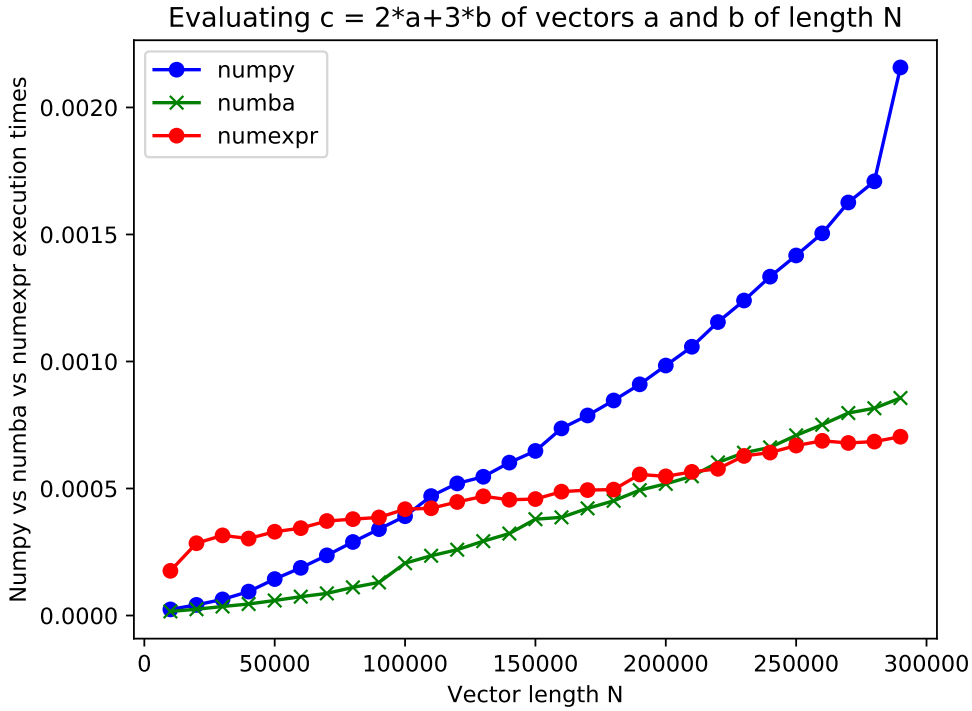
11.2 About NumPy, Numba, and NumExpr

Compilation has some overhead, so Numba or NumExpr won't help on jobs that are executed only once and take a very short time. Sometimes you can "prime" the jit compilation before the actual job by calling the jitted function once.

Remark: `Numba` and `NumExpr`, as well as `NumPy`, may autoparallelize execution, without you doing nothing.

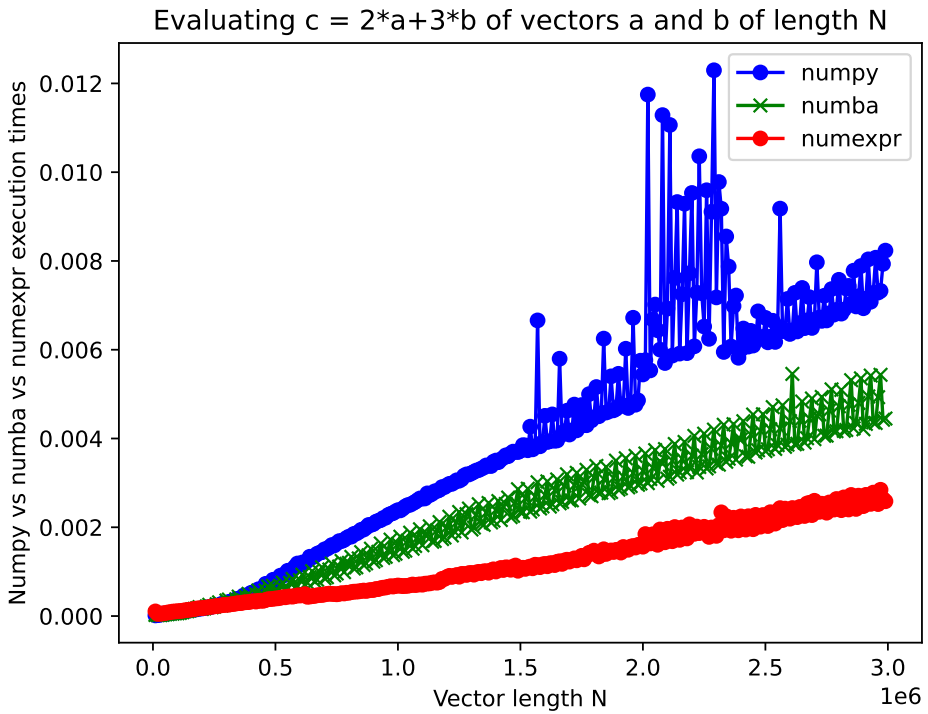
Running an internally parallel code in a cluster will upset fellow researchers!

Below you find timing results of computing $c = 2a + 3b$ for large 1D arrays a. This may not be a fair comparison but gives some hint what to expect. Looking at CPU utilization, `NumExpr` could autoparallelize a bit, using 130-170% CPU.



The jump in execution time indicates L3 cache (2 MiB) overflow. That equals 2097152 bytes and arrays a,b, and c take that at around N=87000 (from `import sys; sys.getsizeof(c)`). `NumExpr` does best with large arrays, but there is significant overhead in small array operations. Operations on unaligned or strided data benefit the most. If you have Intel MKL available NumExpr can use it.

Here's a longer test, run in a different machine:



The core code was

```
import numpy as np
import NumExpr as ne
from Numba import jit, double
import time

nrep = 100
Nmax = 300000
step = 10000
res = []

def add_fac(f1,f2,a,b):
    return f1*a+f2*b

Numba_add_fac=jit(double[:](double,double,double[:],double[:]))(add_fac)

def loop():
    for N in np.arange(step,Nmax,step=step):
        toc = 0
        c = np.empty(N)
        for rep in np.arange(nrep):
            a = np.random.random(N)
            b = np.random.random(N)
            tic = time.time()
            c = Numba_add_fac(2,3,a,b)    # Numba
            # c = add_fac(2,3,a,b)        # NumPy
            # c = ne.evaluate("2*a+3*b") # NumExpr
            toc += time.time()-tic
        res.append((N,toc/nrep))
```

12 Machine learning with Python

Python has become a very popular language in machine learning and data mining, which supposedly is behind the increasing popularity of Python. [39] I'll introduce machine learning because I assume many of you are using Python in that context.

The bread and butter in machine learning is computation of gradients, Jacobians, and Hessians, and handling of tensors. There are several Python packages to choose from, [40]

- `TensorFlow @github.com` , originally developed in Google, is *an open source software library for numerical computation using data flow graphs.*

```
$ pip install tensorflow
```

- PyTorch @pytorch.org , originally developed by Facebook, is

  an open source machine learning framework that accelerates the path from research prototyping to production deployment.

```
$ pip install torch
$ pip install torchvision
```

- `Scikit-learn @scikit-learn.org` ; Python module `sklearn`

  Scikit-learn is a ***user-friendly*** machine learning toolkit

  - Simple and efficient tools for data mining and data analysis
  - Accessible to everybody, and reusable in various contexts
  - Built on NumPy, SciPy, and matplotlib
  - Open source, commercially usable - BSD license

[39] "Machine learning" means fitting but we call it machine learning for sales purposes.
[40] Theano (2007-2017) died after version 1.0, but it's legacy lives in Keras, Lasagne and Blocks.

```
$ pip install sklearn
```

- Keras @keras.io

```
$ pip install keras
```

Keras is a ***user-friendly*** machine learning toolkit

> *high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.*

At the moment of writing (standalone) Keras appears to be *de facto* on top of Tensorflow. Since 2019 Keras is built in to Tensorflow. Keras `Sequential` network model can be defined using two idioms,

```
import tensorflow as tf
model  = tf.keras.Sequential()
```

or equivalently

```
from tensorflow.keras import Sequential()
model  = Sequential()
```

I should mention also Apache MXnet @Wikipedia , `CNTK @github.com` (Microsoft Cognitive Toolkit), and Deeplearning4j @Wikipedia .

Both Keras and PyTorch are high level abstractions. For a recent comparison of PyTorch and Tensorflow, see link @dezyre.com . An August 2021 article about Keras, Tensorflow and Pytorch is here @simplilearn.com I'm talking mostly about Keras, but also PyTorch is a quite comfortable, high level API.

As a physicist, you might appreciate SciANN: Neural Networks for Scientific Computations, "SciANN is a Keras wrapper for scientific computations and physics-informed deep learning."

12.1 Fully connected, dense neural network

View a neural network as a function,

```
y = neural_network(x)
```

You'd like to have a neural network that approximately reproduces a ***training set***, known points to fit:
⇒ `neural_network()` should be very flexible
⇒ many parameters to optimize.

We can assume `x` is numerical data stored in a NumPy array of shape, say, `(n,)`. A simple and fast operation on an array is matrix multiplication. There's a subtle difference how `@`, which uses the `__matmul__()` method, and `np.dot()` deal with multidimensional arrays, and it's safer to use here the latter,

```
y = np.dot(w, x) # w.shape is (m,n)
```

Frequently one does it the other way round,

```
y = np.dot(x, w) # w.shape is (n,m)
```

The array `w` is ***weight***, which already has `n*m` parameters. Often `y` is not located where `x` is, so we should be able to shift the output of `np.dot(w,x)` by adding a ***bias*** array `b` to the output. We get a neural network model

```
y = np.dot(w, x) + b
```

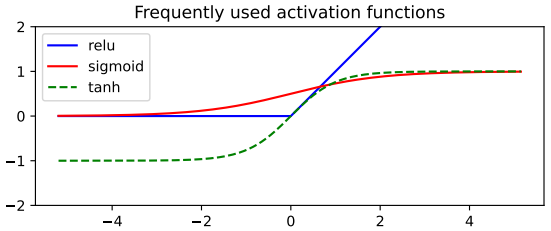
The problem is that this is a *linear* function in `x`, and the output is always a linear combination of inputs. This seriously limits the possible `{x,y}` data this function can reproduce ⇒ add ***nonlinearity*** in the form of a nonlinear function `activation()`, which could be, for example,

```
import numpy as np

# ReLU
def activation(x):
    return np.where(x>0, x, 0) # or np.maximum(0,x)
```

This cuts off any negative signals and returns the rest as they are, expressed in another way, $max(0, x)$. The $max(0, x)$ activation is known as rectifier or ***ReLU*** (rectified linear unit) activation function.

All built-in activations in Keras can be listed and plotted using the code keras_activations.py, some frequently used activations are $\tanh(x)$ and

$$\text{relu}(x) = \max(0, x) \quad \text{ReLU (Rectifying Linear Unit)}$$
$$\text{sigmoid}(x) = \frac{1}{1+\exp(-x)} \ .$$



So far we have a model[41]

```
y = activation(np.dot(w,x) + b)
```

It's best to add one more weight-bias modification, which adds scaling and shifting of activation output, and the network model

```
y = np.dot(w2,activation(np.dot(w1,x) + b1) + b2)
```

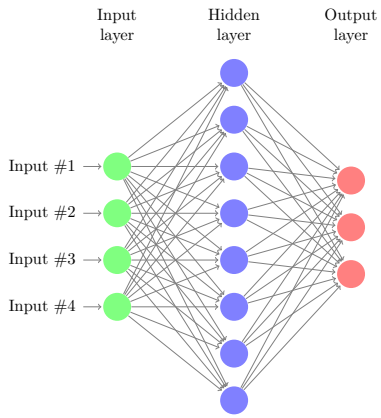
where weights and biases are different from layer to layer. In math, (`activation()` is now $\sigma()$)

$$\boldsymbol{y} = \boldsymbol{w}^2\sigma(\boldsymbol{w}^1\boldsymbol{x} + \boldsymbol{b}^1) + \boldsymbol{b}^2 \ .$$

[41] If `activation(x` is `np.where(x>0,1,0)`, the model `y = activation(np.dot(w,x) + b)` with binary (1 or 0) output is called a ***perceptron***.

We can split the operations and define a ***hidden layer*** between the input layer and the output layer,

```
x                       # input layer input
z = np.dot(w1,x) + b1 # hidden layer input
h = activation(z)       # hidden layer output is now output layer input
y = np.dot(w2,h) + b2 # output layer output
```



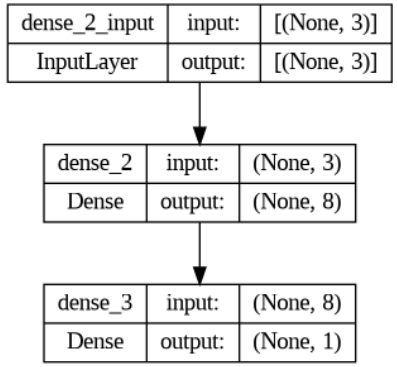
so the same principle as in

> The toe bone's connected to the foot bone,
> The foot bone's connected to the ankle bone,
> The ankle bone's connected to the leg bone,
> Now shake dem skeleton bones!

The one-hidden layer neural network is already a universal function approximator for certain continuous functions, meaning most functions $y = f(x)$ can be arbitrarily well approximated by it - given large enough width in the hidden layer.

In Keras,

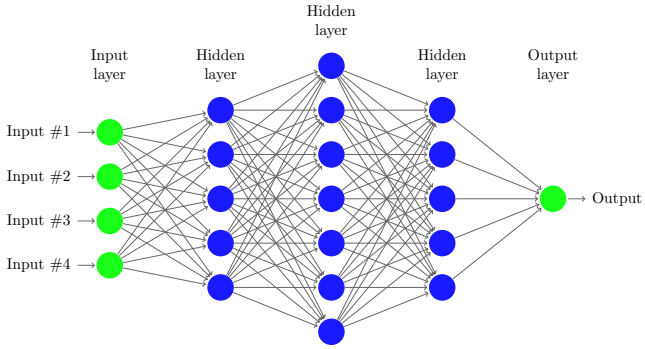
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import plot_model

model = Sequential()
# define the topology of the model
model.add(Dense(8, input_dim=3, kernel_initializer='uniform', activation='relu'))
model.add(Dense(1, kernel_initializer='uniform'))

print(model.summary())
plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)
```



No activation in the output layer; weights and biases are initialized to uniformly distributed random numbers. Summary gives parameters 32+9 = 41; from weight (8,3) + bias (8,) = 3*8 + 8 = 32, and weight (8,1) + bias (1) = 8*1+1 = 9.

We can make the network deeper by forwarding the hidden layer output to another hidden layer - that's a ***sequential*** model. Narrow but deep networks are more capable than wide but shallow ones. If every node in a layer is connected to every node in the next layer you have a ***fully connected*** network. Evaluating the output `y` from the input `x` is done via forward propagation of signal through the network, which is why this is a ***feed-forward network***.

The `keras` model of a fully connected, dense neural network in the figure could be




```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
# define the topology of the model
model.add(Dense(5, input_dim=4, kernel_initializer='uniform', activation='relu'))
model.add(Dense(7, kernel_initializer='uniform', activation='softmax'))
model.add(Dense(5, kernel_initializer='uniform', activation='tanh'))
model.add(Dense(1, kernel_initializer='uniform'))
```

See options for a dense network. For more keras examples here, and tips for debugging here. Kernel means weight, and `kernel_initializer` choses the random number distribution used to compute initial weight matrix (tensor) elements.

So far we have only built a neural network model, and set ***hyperparameters***, just a few, which define the model structure and some more parameters you fix from the start. But there are also ***model parameters***, weights and biases, which you manipulate to fit the training data better, and so far we haven't figured out how to train the network. Knowing nothing about the optimal values the optimal weight and bias values we may as well guess them,

Weights and biases are initialized to random values

12.2 Training a neural network

Training a neural network is nontrivial but in principle a straigthforward task. There are very many parameters to tweak, and only a finite set of data for training. The more training data you have the better.

Let's assume we know that for input $\boldsymbol{x}$ the output should be $\boldsymbol{y}$. This is a limited data set of known results. For example,

- function fit: a single variable function, $\boldsymbol{x} = \{-4, -3, -1, 0, 1, 2, 3, 4\}$ the values are known to be $\boldsymbol{y} = \{4, 3, 1, 4, 3, 7, 8\}$.
- cathegorize: canine-feline pictures $\boldsymbol{x} = \{fig1, fig2, ..., fig500\}$ are known to show $\boldsymbol{y} = \{cat, dog, ..., cat, cat, dog\}$.

Before training you need to fix a few things:

- Divide the data set to a ***training set*** $(\boldsymbol{x}_{tr}, \boldsymbol{y}_{tr})$, and set aside about 1/3 of the data as a ***validation set*** $(\boldsymbol{x}_{val}, \boldsymbol{y}_{val})$. The training data set consists of ***samples***, a sample is one input vector. That is, a sample is one piece of data that gives a known output.
- ***Loss function*** or ***Cost function*** ($\mathcal{L}$ or C) is the function whose value you try to minimize. Zero loss means a perfect fit to training data. You have the training set and you know the answer for that set, so you construct a function that is positive, and increases when the model result is further from the known answer. A simple one is $\mathcal{L} = ||\boldsymbol{y} - \boldsymbol{y}_{tr}||^2$ for output $\boldsymbol{y}$ and training output $\boldsymbol{y}_{tr}$.

Now you are ready to start minimizing the loss function. First, compute the ***error signal***

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} , \tag{4}$$

which tells which direction the last layer output $\boldsymbol{y}$ should change to get smaller loss $\mathcal{L}$. The error signal is the final outcry you hear in the end of the network telling the output is not quite correct. The big question is how to get from the error signal the information which direction should the layer weights and biases change? The answer is given by ***backpropagation***, (see backpropagation @Wikipedia), which lies in the essense of training neural networks.

First, an input is sent through the network - a feed forward - and evaluate the error signal from of output. Next, take the error signal and *send it back to the network from the end*, and read the gradients of weights and biases when the backpropagating error signals reaches them.

A neural network is like a radio with hundreds of adjustable knobs. The input is radio waves, and you know your favorite song is playing nonstop but sounds horribly distorted. You turn a knob a bit and it sounds somewhat better (or worse), then turn another knob, and so on. Then you realize that finding the optimal knob settings using this method would take ages. Luckily, your radio can be taught with backpropagation - it's not a common radio. The distorted sound you hear deviates from you favorite song, and that deviation is the error signal. You send the error signal back to the radio in backpropagation, and while going through the radio it attaches to every knob a mark indicating which way it should be turned (but not how much, just how bad the current setting is). You do small knob adjustments and immediately your song plays a lot better. After a few cycles your song plays as nicely as it can with that box of a radio, and you sit back and enjoy.

12.2.1 Math details for one-hidden layer network forward and backward propagation

Forward propagation in a one-hidden layer network written in Python is

```
z = np.dot(w1,x) + b1
h = activation(z)
y = np.dot(w2,h) + b2
```

which makes clear that there are two different weights and biases.

Let's write this in math, the component notation is on the first column, and the vector notation is on the second column. Here $\boldsymbol{w}^n$ and $\boldsymbol{b}^n$ indicate n:th layer weights and biases.

FORWARD PROPAGATION

$$
\begin{aligned}
z_i &= \sum_k w^1_{ik} x_k + b^1_i && \boldsymbol{z} = \boldsymbol{w}^1 \boldsymbol{x} + \boldsymbol{b}^1 && \text{(layer 1 weighted input)} \\
h_i &= \sigma(z_i) && \boldsymbol{h} = \sigma(\boldsymbol{z}) && \text{(layer 1 output)} \\
y_j &= \sum_i w^2_{ji} h_i + b^2_j && \boldsymbol{y} = \boldsymbol{w}^2 \boldsymbol{h} + \boldsymbol{b}^2 && \text{(layer 2 output)} \\
\mathcal{L} &= \frac{1}{2} \sum_j (y_j - y_{tr,j})^2 && \mathcal{L} = \frac{1}{2} ||\boldsymbol{y} - \boldsymbol{y}_{tr}||^2 && \text{(a simple loss function)} \ .
\end{aligned} \tag{5}
$$

The backward error propagation is best expressed using an ***adjoint*** and a bar notation for gradients of the loss function. (see *e.g.* automatic differentiation @Wikipedia) For any quantity $\boldsymbol{q}$ the adjoint is defined

$$
\bar{\boldsymbol{q}} := \frac{\partial \mathcal{L}}{\partial \boldsymbol{q}} \ . \tag{6}
$$

I emphasize that ***all adjoints are just numerical quantities we compute***.[42] The error signal backprobagation is initiated with the trivial

$$
\bar{\mathcal{L}} := \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1 \ . \tag{7}
$$

The key is to apply ***the chain rule*** of differentiation,

$$
\frac{\partial f(g(x,y))}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} \ , \tag{8}
$$

The arrows show how the error signal backpropagates:

[42] One reason to use the bar notation is that adjoints are a special case of gradients,

$$
\bar{\boldsymbol{q}} := \boldsymbol{\nabla}_q \mathcal{L} \ ,
$$

but it's always the gradient of $\mathcal{L}$ so why repeat it.

BACKWARD PROPAGATION

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathcal{L}} &:= \bar{\mathcal{L}} = 1 && \bar{\mathcal{L}} = 1 \\
\frac{\partial \mathcal{L}}{\partial y_j} &:= \bar{y}_j = \bar{\mathcal{L}}(y_j - y_{tr,j}) && \bar{\boldsymbol{y}} = \bar{\mathcal{L}}(\boldsymbol{y} - \boldsymbol{y}_{tr}) && \text{Error signal} \\
\frac{\partial \mathcal{L}}{\partial w^2_{ji}} &:= \bar{w}^2_{ji} = \frac{\partial \mathcal{L}}{\partial y_j}\underbrace{\frac{\partial y_j}{\partial w^2_{ji}}}_{h_i} = \bar{y}_j h_i && \bar{\boldsymbol{w}}^2 = \bar{\boldsymbol{y}}\,\boldsymbol{h}^T && \text{(used in updating } \boldsymbol{w}^2\text{)} \\
\frac{\partial \mathcal{L}}{\partial b^2_j} &:= \bar{b}^2_j = \frac{\partial \mathcal{L}}{\partial y_j}\underbrace{\frac{\partial y_j}{\partial b^2_j}}_{1} = \bar{y}_j && \bar{\boldsymbol{b}}^2 = \bar{\boldsymbol{y}} && \text{(used in updating } \boldsymbol{b}^2\text{)} \\
\frac{\partial \mathcal{L}}{\partial h_i} &:= \bar{h}_i = \sum_j \frac{\partial \mathcal{L}}{\partial y_j}\underbrace{\frac{\partial y_j}{\partial h_i}}_{w^2_{ji}} = \sum_j \bar{y}_j w^2_{ji} && \bar{\boldsymbol{h}} = (\boldsymbol{w}^2)^T\,\bar{\boldsymbol{y}} \\
\frac{\partial \mathcal{L}}{\partial z_i} &:= \bar{z}_i = \sum_j \frac{\partial \mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial h_i}\frac{\partial h_i}{\partial z_i} = \frac{\partial \mathcal{L}}{\partial h_i}\underbrace{\frac{\partial h_i}{\partial z_i}}_{\sigma'(z_i)} = \bar{h}_i \sigma'(z_i) && \bar{\boldsymbol{z}} = \bar{\boldsymbol{h}} \circ \sigma'(\boldsymbol{z}) \\
\frac{\partial \mathcal{L}}{\partial w^1_{ik}} &:= \bar{w}^1_{ik} = \underbrace{\sum_j \frac{\partial \mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial h_i}\frac{\partial h_i}{\partial z_i}}_{\bar{z}_i}\underbrace{\frac{\partial z_i}{\partial w^1_{ik}}}_{x_k} = \bar{z}_i x_k && \bar{\boldsymbol{w}}^1 = \bar{\boldsymbol{z}}\,\boldsymbol{x}^T && \text{(used in updating } \boldsymbol{w}^1\text{)} \\
\frac{\partial \mathcal{L}}{\partial b^1_i} &:= \bar{b}^1_i = \underbrace{\frac{\partial \mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial h_i}\frac{\partial h_i}{\partial z_i}}_{\bar{z}_i}\underbrace{\frac{\partial z_i}{\partial b^1_i}}_{1} = \bar{z}_i && \bar{\boldsymbol{b}}^1 = \bar{\boldsymbol{z}} && \text{(used in updating } \boldsymbol{b}^1\text{) .} \qquad (9)
\end{aligned}
$$

Notice that quantities $\boldsymbol{y}$, $\boldsymbol{h}$, and $\boldsymbol{z}$ were computed and stored in forward propagation. The operator $\circ$ denotes a Hadamard product, an elementwise product.

In some occations you may find that your network won't learn. If, for example, $\sigma'(\boldsymbol{z})$ happens to be too small, which means the weighted layer input is in a region where the activation is almost flat, you get $\bar{\boldsymbol{z}} \approx 0$, and subsequent weights and biases have zero gradient. Such a no-learning situation happens easily with the sigmoid activation, but not so easily with ReLU (unless the layer input becomes negative).

12.2.2 Gradient descent

Once backward propagation is finished you know the gradients of weight and biases, $\bar{\boldsymbol{w}} := \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$ and $\bar{\boldsymbol{b}} := \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}}$. The aim is to reduce the loss, so proceed to the direction of *negative gradients*. Backward propagation only reveals the gradients, but not how far to move in the negative gradient direction. One way is to guess a (positive) ***learning rate*** η, a small, adjustable hyperparameter:

WEIGHT AND BIAS UPDATE

$$
\begin{aligned}
\boldsymbol{w}^1 &\to \boldsymbol{w}^1 - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^1} = \boldsymbol{w}^1 - \eta \bar{\boldsymbol{w}}^1 \\
\boldsymbol{b}^1 &\to \boldsymbol{b}^1 - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^1} = \boldsymbol{b}^1 - \eta \bar{\boldsymbol{b}}^1 \\
\boldsymbol{w}^2 &\to \boldsymbol{w}^2 - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^2} = \boldsymbol{w}^2 - \eta \bar{\boldsymbol{w}}^2 \\
\boldsymbol{b}^2 &\to \boldsymbol{b}^2 - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^2} = \boldsymbol{b}^2 - \eta \bar{\boldsymbol{b}}^2 \ .
\end{aligned}
\tag{10}
$$

This approach is not particularly effective for locating the minimum of the loss function. This is primarily due to the fact that the terrain of the function can contain numerous local minima, and the algorithm described above runs the risk of becoming trapped in one of them.

Adjusting weight and biases is just like any other optimization problem with known gradients, and there are multiple algorithms to choose from. The improved algorithms remember how succesfull the previous gradient updates were and keep updating their optimization strategy. polular ones are Adam, Adagrad, Momentum, AMSGrad, RMSProp, AdaMax and Nesterov. See Sebastian Ruder: An overview of gradient descent optimization algorithms. Most are readily available in Keras/Tensorflow/PyTorch.

12.2.3 Automatic Differentiation (AD)

We saw how to get from forward propagation through a neural network to backward propagation for a single hidden layer network. Imagine adding more hidden layers and trying to repeat the same calculation. You soon realize that the pen-and-paper algebra gets messy, and it's ever so easy to make mistakes.

Can one automatically generate the backward propagation from a given forward propagation? The answer is yes, with Automatic Differentiation (AD) (see AD @Wikipedia).

Error backpropagation through a multilayer network relies on repeated application of the chain rule. I turns out that *backpropagation is a special case of AD*. Once you have defined a neural network, Tensorflow and other clever software can backpropagate the error signal automatically. [43]

Automatic differentiation is not numerical differentiation. The derivative of function $f(x)$ is

$$\frac{df(x)}{dx} = \lim_{dx\to 0} \frac{f(x+dx) - f(x)}{dx} \ , \tag{11}$$

but the ratio numerically unstable in the limit $dx \to 0$, because for dx close to machine precision $x+dx = x$ and $f(x+dx) = f(x)$. That's how you actually can *find* the machine precision:

```
>>> x = 1.0
>>> x+1e-15 == x
False
>>> x+1e-16 == x
True
```

Note added: don't test `x+1e-16 is x`, that's always `False` (hint: immutable object).

Automatic differentiation is not symbolic differentiation. All that's moving about is numbers, not symbols. You start from the right with the gradient 1, and let it backpropagate. All adjoints are just numerical arrays.

[43] One of the first to actually program backpropagation was Seppo Linnainmaa, in his Master's thesis in Univ. Helsinki (1970), without mentioning neural networks. He also made use of automatic differentiation! In 1974 Linnainmaa was awarded the first doctorate ever in computer science at the University of Helsinki.

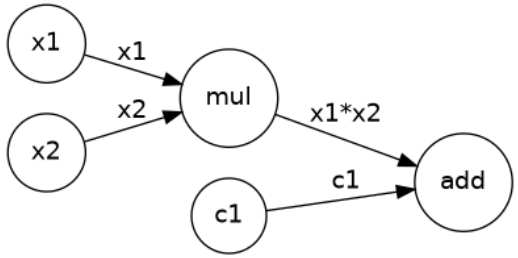
Automatic differentiation is a numerical method to compute gradients of arbitrary functions in machine precision accuracy.

I recommend Jonathan Kernes' blog about automatic differentiation. Here's a short appetizer.

Every function can be expressed as a combination of a few elementary functions, such as sin(), cos(), exp(), and all operators are also functions: $+$ is $add()$ and $*$ is $mul()$. In Python, $+$ means the magic method `__add__()` and $*$ means `__mul__()`:

```
>>> x = 3.0
>>> y = 4.0
>>> x.__mul__(y)
12
```

These are ***elementary operations***, and all computations can be written as a ***computational graph***. For example, the function `y=f(x1,x2)=x1*x2 +c1` is the directed graph shown below:



Circles are nodes with either variable (x1,x2), constant (c1), operator (mul,add) or a placeholder (variable or constant to be set in the future). Once $x1, x2, c1$ have numerical values, you can travel the graph from left to right and evaluate the function. That's not a big deal but this is: you can as easily compute *derivatives* and evaluate them. What makes calculation of derivatives possible is the fact that you know how each elementary operation (add, mul, sin, exp, ...) behaves in differentiation.

Algebraically,

$$f(x1, x2) = x1 * x2 + c1 = add(mul(x1, x2), c1) \tag{12}$$

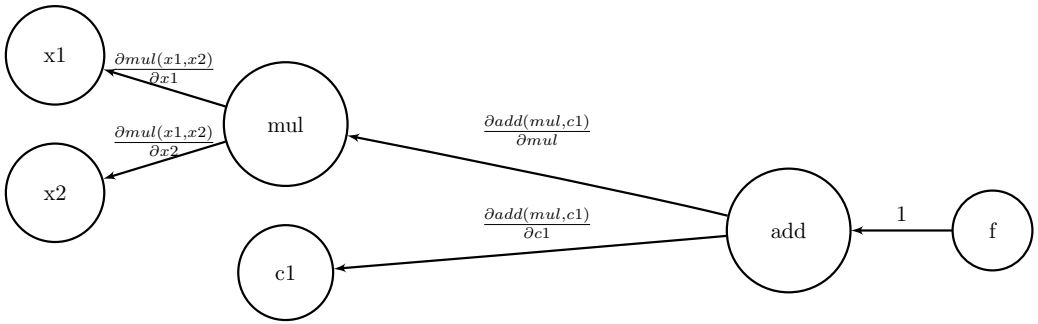
$$\frac{\partial f}{\partial x_1} = \frac{\partial\ add(mul(x1, x2), c1)}{\partial x1}\ . \tag{13}$$

But $add()$ and $mul()$ are functions, so the chain rule applies,

$$\frac{\partial f}{\partial x_1} = \frac{\partial add(mul(x1, x2), c1)}{\partial x1} = \underbrace{\frac{\partial add(mul, c1)}{\partial mul}}_{1}\underbrace{\frac{\partial mul(x1, x2)}{\partial x1}}_{x2} + \frac{\partial\ add(mul, c1)}{\partial c1}\underbrace{\frac{\partial c1}{\partial x1}}_{0} = x2\ . \tag{14}$$

The next revelation is to realize that also *derivatives can be written as a computational graph*. What's more, the derivative graph backpropagates the same computational graph we already draw, only the information in the arrows change. To kickstart backpropagation, set (just like we set $\frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$)

$$\frac{\partial f}{\partial f} = 1\ . \tag{15}$$

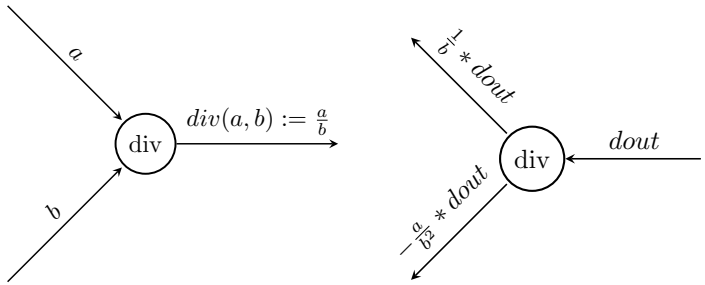


Jonathan gives a working Python code that can handle operators addition, multiplication, power, and matrix product. Each of them contains two methods, one for forward and one for backward propagation.

Remark: For example, the division operator can be handled like this:

```python
# From Jonathan Kernes' blog; Sign error in the last output corrected
class divide(Operator):
  count = 0
  """Binary division operation."""
  def __init__(self, a, b, name=None):
      super().__init__(name)
      self.inputs=[a, b]
      self.name = f'div/{divide.count}' if name is None else name
      divide.count += 1

  def forward(self, a, b):
      return a/b

  def backward(self, a, b, dout):
  return dout/b, -dout*a/np.power(b, 2)
```

The Python `class divide` methods `forward()` and `backward()` are these graphs:



Remark: The backward graph expresses the math (mark with suffix what is kept fixed)

$$d(div(a,b)) = \left(\frac{\partial div(a,b)}{\partial a}\right)_b da + \left(\frac{\partial div(a,b)}{\partial b}\right)_a db = \frac{1}{b} da - \frac{a}{b^2} db \ . \tag{16}$$

From this we need only the slopes,

$$\left(\frac{\partial div(a,b)}{\partial a}\right)_b = \frac{1}{b} \tag{17}$$

$$\left(\frac{\partial div(a,b)}{\partial b}\right)_a = -\frac{a}{b^2} \ . \tag{18}$$

The gradient output of the node is *dout*. In the first case b is fixed (no gradient in b, $db = 0$), so the gradient backpropagates along the a-branch as

$$\left(\frac{\partial div(a,b)}{\partial a}\right)_b dout = \frac{1}{b} * dout \ . \tag{19}$$

In the second case a is fixed (no gradient in a, $da = 0$), so the gradient backpropagates along the b-branch as

$$\left(\frac{\partial div(a,b)}{\partial b}\right)_a dout = -\frac{a}{b^2} * dout \ . \tag{20}$$

12.3 Batches, epochs, and overfitting

Only thing that remains to be done is to deside two more hyperparameters, the batch size and the number of epochs.

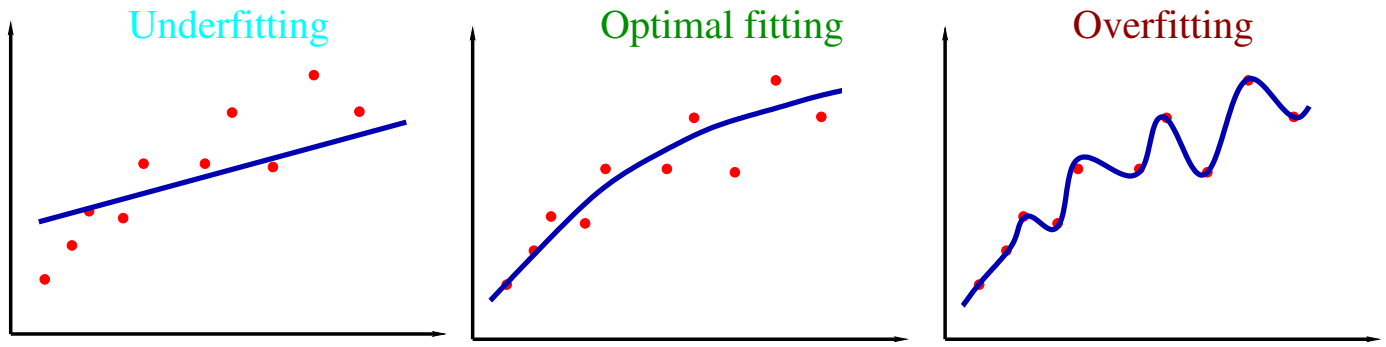
- ***A batch***: The training data set is divided into batches,

$$\begin{aligned} &\text{batch } 1: \quad \{x_{tr}\}_{1,\ldots,Nb} \to \{y_{tr}\}_{1,\ldots,Nb} \\ &\text{batch } 2: \quad \{x_{tr}\}_{Nb+1,\ldots,2Nb} \to \{y_{tr}\}_{Nb+1,\ldots,2Nb} \\ &\ldots \end{aligned} \tag{21}$$

  The model is updated only after a full batch is run through. Suppose you make a gradient descent to minimize the loss function. If you update the model after every sample you rely on just one piece of data. Following the gradient of that single input is shooting your model parameters to some direction. It's probably not the best of directions and the length of your update vector is probably not optimal. This ***stochastic gradient*** with some randomness in vectors may find a good minimum. If you choose a bigger batch and use the average gradient direction you get a ***batch gradient*** with a bit less random vectors.

- ***Epoch***: One epoch is a run through all batches, that is, the full training data set. One epoch is usually not enough because models can't learn that fast.

Schematically, three things can happen. The red dots represent the training data, and the curve the model output.



With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.

John von Neumann to Enrico Fermi

An overfitted model can perfectly reproduce the training data, but for new input the predictions are usually very poor, so such a model has no predictive power and can't generalize to new input. To detect overfitting, one puts aside some known results as ***validation data***, typically 30 percent of available data. During training, once model predictions with validation data start to deteriorate it's time to stop training. A very powerful trick to avoid overfitting is to ***randomly turn off some neurons*** so that the remaining neurons have to compensate and adapt.

12.4 Learning diabetes factors among Pima indians

The Pima data contains the following information:[44]

1. Number of times pregnant
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-Hour serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)
9. Class variable (0 or 1)

In the last entry class value 1 is interpreted as "tested positive for diabetes". CSV data (`pima-indians-diabetes.csv`)

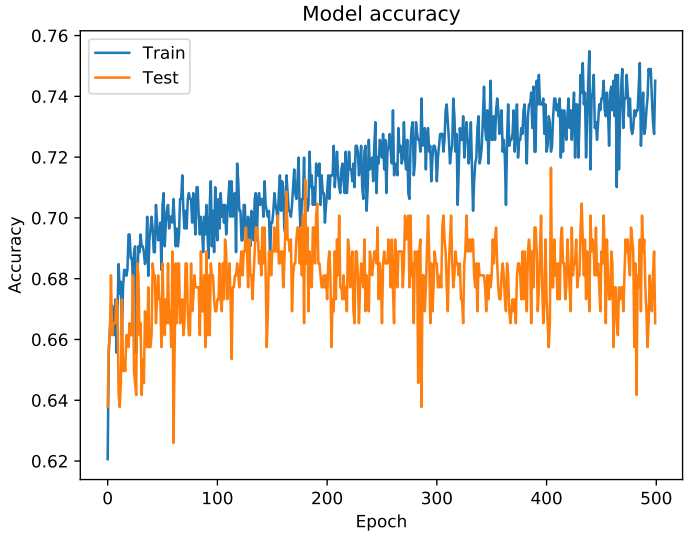
```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
...
```

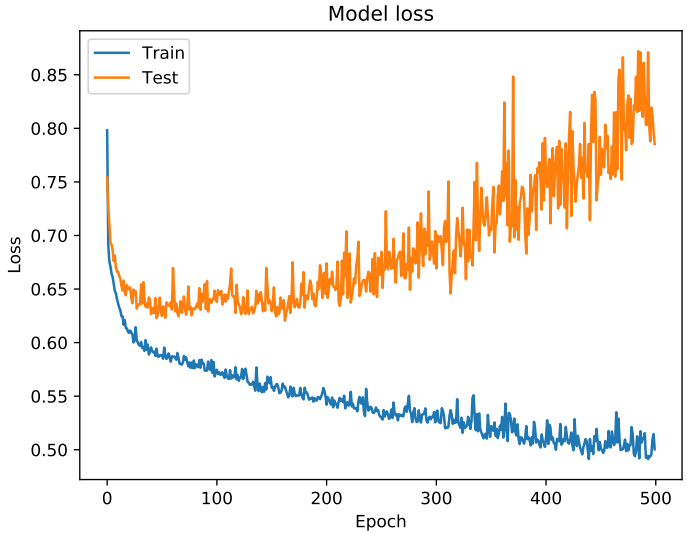
[44]The data set is no longer publicly available.

Let's examine the data using Pandas,

```
import pandas as pd

# read CSV data to DataFrame
#  header=None prevent using the first line as headers
df=pd.read_csv('pima-indians-diabetes.csv',header=None) # data is in this directory
# add column names
df.columns=['pregnancies','plasma glucose','blood pressure (mm Hg)','triceps skin (mm)',
'insulin (mm U/ml)','bmi','diab. pedigree function','age (y)','diab. diagnosed']
print(df)
# select column to fit/predict
col = 'diab. diagnosed'
y = df[col]
# use the rest as model input
X = df.drop(col, axis=1)
print('number of studied persons',len(y))
print('number of diagnosed diabetic cases in the study',y.sum())
```

If the code pima_keras.py is run for 500 epochs the results look like this:



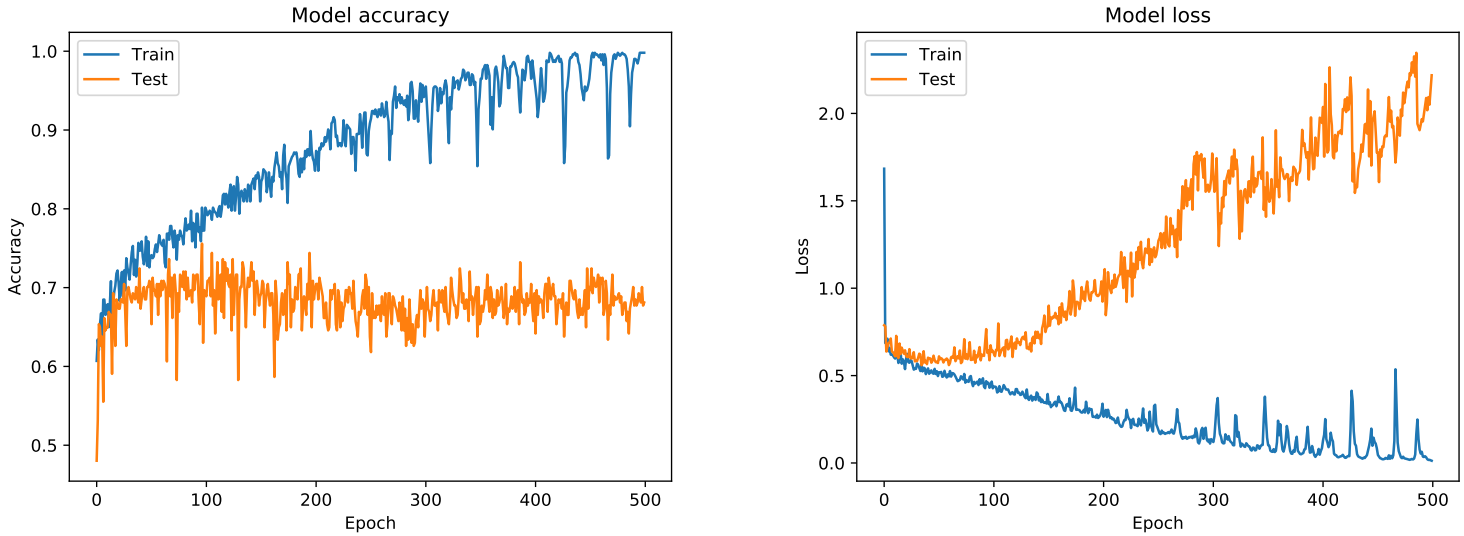


The training session tries to minimize a certain loss function, and it does well for at least during these 500 epochs. However, the loss of the test (validation) starts to increase at about epoch 100, so the predicting power of the model is no longer improving. According to `model.report()`, there are 221 parameters to fit, so no wonder the small training set can be easily overfitted.

Let's make things worse. Add more and larger hidden layers,

```
model.add(Dense(20, input_dim=8, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Whopping 8371 parameters!



As expected, the training set is almost perfectly reproduced, and the training loss is closing on zero. The test set doesn't follow predictions at all. The model parameters are useless garbage.

12.5 US Space Shuttle Data

This is just a Keras play, but the data is serious, US Space Shuttle data @archive.ics.uci.edu. It contains data from 23 shuttle flights, all launched at temperatures between 53 °F and 81 °F (12 °C and 27 °C). The solid rocket boosters were rated to be flown at temperatures of 39 °F and higher. Then came the Challenger launch on January 28, 1986. At 7 a.m. the temperature was only 24 °F, and by the time of the lauch at 11:38 a.m. the temperature was 36 °F, just above freezing. Even worse, the right solid rocket booster was still at about 28 °F: high off the ground and not getting any sunlight. The question is, having only high temperature data, can one extrapolate to find how many O-rings sealing the solid rocket boosters are under "thermal distress" at temperatures near freezing?

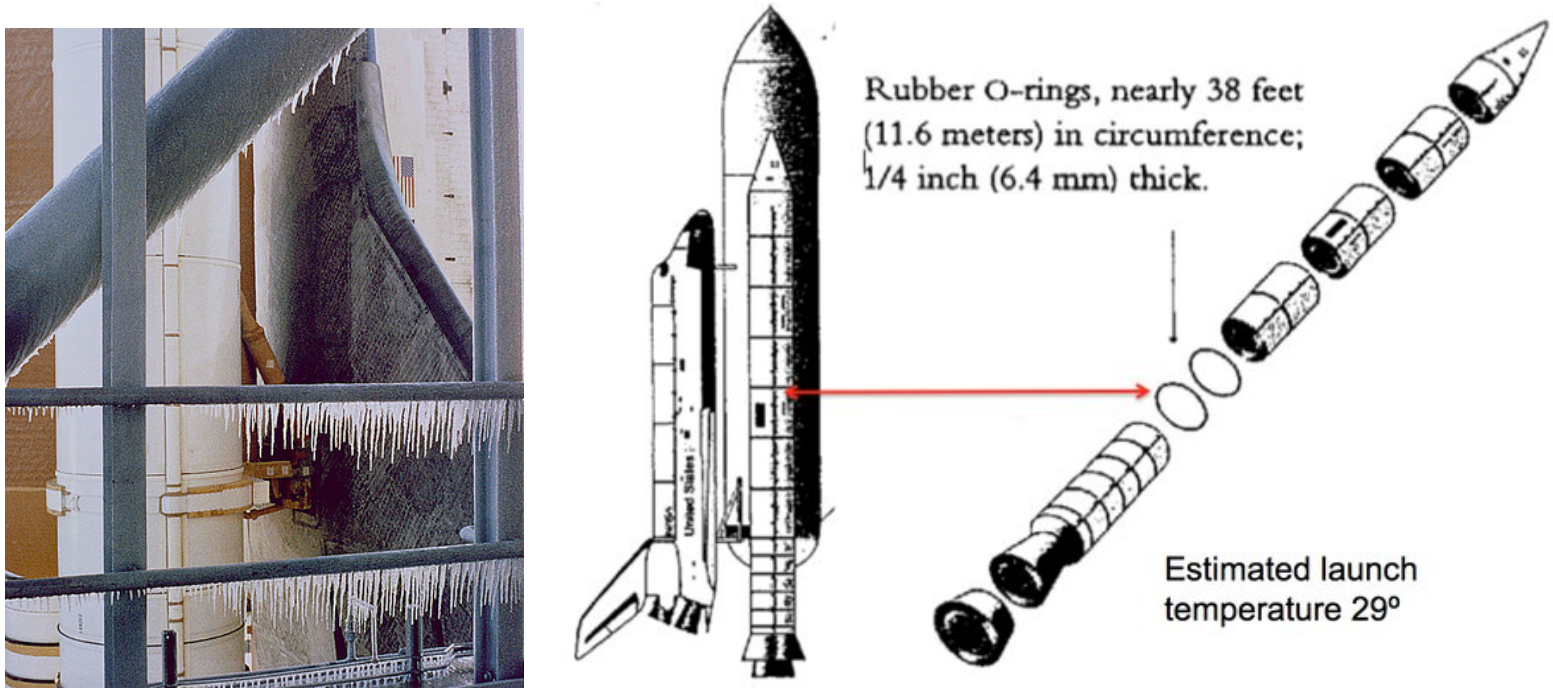


Figure 1: Challenger launch tower and the O-rings. Source: NASA

I tried the following model (full code `keras_oring.py`)

```
model = Sequential()
model.add(Dense(150,input_dim=1))
model.add(layers.Activation(activations.relu))
model.add(Dense(1))
model.add(layers.Activation(activations.sigmoid))
```

ReLU is faster and more popular than sigmoid. The changes in weights and biases are done based on gradients, computed from the loss function, and backpropagated through the neural network. The two dreaded situations in neural networks are

1. Vanishing gradient: Backpropagated to hidden layers, a layer may get very small or exactly zero gradients. This means the layer weights and biases are not changing $\Rightarrow$ part of the network is not learning. Gradients vanish easily in ***deep learning*** (deep=multiple hidden layers). The problem was analyzed by Hochreiter in his diploma thesis in 1991,[45] and as a remedy he also developed the long short-term memory (LSTM) network.
   Example: If a layer has $xw + b < 0$, then ReLU$(xw + b)$=0. The network unit dies, and in many cases stays dead. This is known as the ***dying-ReLU problem***.

2. Exploding gradient: Backpropagated to hidden layers, a layer may get huge gradients. The layer weights and biases blow up.

[45] Some Diploma thesis! The fourth chapter "Konstanter Fehlerrückfluß" does is all. Josef "Sepp" Hochreiter leads the Institute for Machine Learning at the Johannes Kepler University of Linz. I mention this from personal reasons, as someone who spent a few years as a post-doc in the Uni.

Remark: Details: Let's mark, as usual, weight matrices with w, and bias vectors with b. ***These contain all the dense network model parameters***. Often one writes a layer output vector as $y = \sigma(wx + b)$ but Keras chooses to define it as $y = \sigma(xw + b)$. Here x is layer input vector. In math notation, the Keras code above fits (x, y)-data to the function (I leave the batch size as 1)

$$y_1 = \sigma_2(\sigma_1(x_1 w_{1j}^{(1)} + b_j^{(1)}) w_{j1}^{(2)} + b_1^{(2)})) \ . \tag{22}$$

$j = 1, ..., 150$ is summed over, the activation functions are the Rectified Linear Unit (ReLU) and sigmoid (also called logistic function),

$$\sigma_1(x) = \text{relu}(x) = max(0, x) \tag{23}$$

$$\sigma_2(x) = \text{sigmoid}(x) = (1 + \exp(-x))^{-1} \ . \tag{24}$$

The number of parameters is: 150 in $w_{1j}^{(1)}$, 150 in $w_{j1}^{(2)}$, 150 in $b_j^{(1)}$, and 1 (in $b_1^{(2)}$). This makes 451 model parameters, also reported by `model.summary()`. Keras shapes are given in the form x = (None,1), y=(None,1), w1 = (1,150), b1=(None,150), w2 = (150,1), and b2=(None,1). My output should be in range [0,6], but sigmoid$(x) \in [0, 1]$. In the code I fix this issue by scaling y in the training data, $y = y/6$. Finally, I descale model output back to range [0,6].

From my part, this is just a 5 min game, but David Draper from University of Bath, UK, examined the shuttle data (and oil price data) more thoroughly. [46]

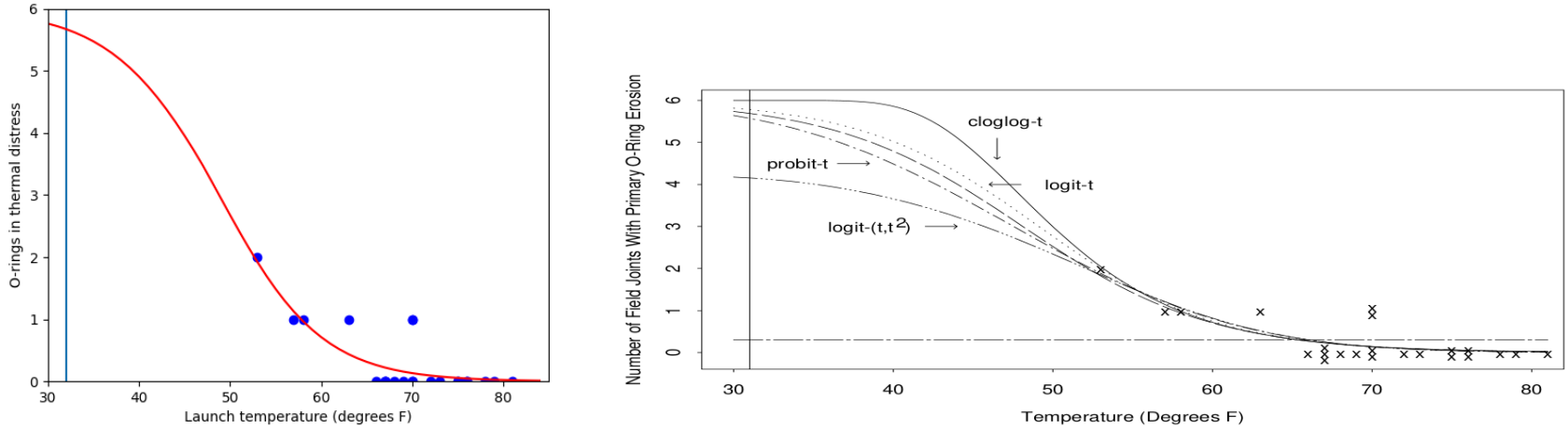
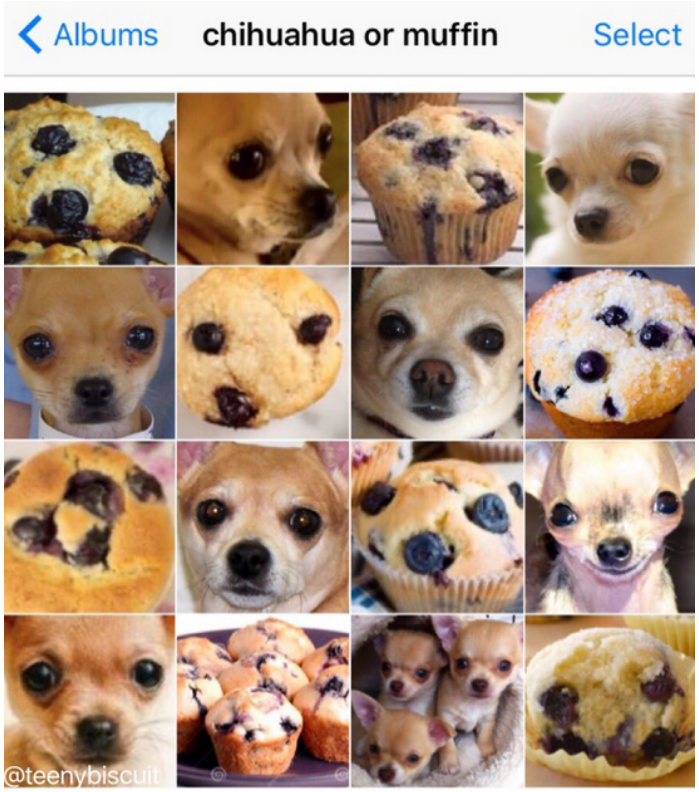
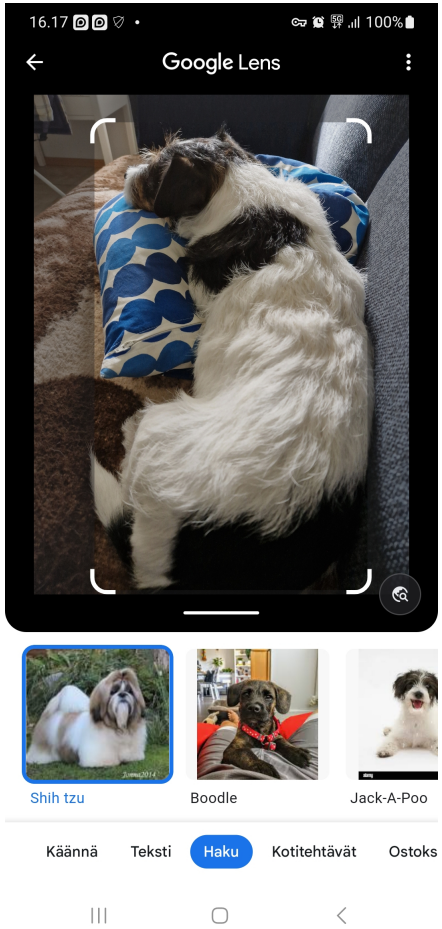


Figure 2: Left: Measured (blue) and model data (red). Right: Draper's result. The vertical lines are at the freezing temperature. There were a total of six O-rings, so this spells disaster.

[46] D. Draper, *Assessment and Propagation of Model Uncertainty*, J. R. Stat. Soc. B, 57 No 1, p. 45 (1995).

AI is not infallible, see Karen Zack @teenybiscuit





Jackrussel terrier Nuksu taking it easy.

...and can be fooled intentionally, too, see Adam Geitgey's blog machine-learning-is-fun. A 2019 Nature article Why deep-learning AIs are so easy to fool @nature.com addresses the same issue. You may be interested in viewing adversarial-machine-learning @viso.ai.

Classification of data can be done with a Support Vector Machine (SVM). I recomment An Idiot's guide to SVM's, by R.

Berwick @MIT. How to classify flowers with `sklearn` is well explained in sklearn video @youtube, the code is available as a Jupyter notebook @github. You'll see the benefits of using a Pandas DataFrame, in a Jupyter notebook with inline Matplotlib plots.

I warmly recommend Florian Marquardt: Machine Learning for Physicists (2015-2021).

A current trend in ML is obvious in the abstract from Smith *et al* (2019) [47]
(emphasis mine):

> *Computational modeling of chemical and biological systems at atomic resolution is a crucial tool in the chemist's toolset. The use of computer simulations requires a balance betweencost and accuracy: quantum-mechanical methods provide high accuracy but are computationally expensive and scale poorly to large systems, while classical forcefields are cheap and scalable, but lack transferability to new systems. Machine learning can be used to achieve the best of both approaches. Here we train a general-purpose neural network potential (ANI-1ccx) that approaches CCSD(T)/CBS accuracy on benchmarks for reaction thermochemistry, isomerization, and drug-like molecular torsions. This is achieved by training a network to DFT data then using transfer learning techniques to retrain on a dataset of gold standard QM calculations (CCSD(T)/CBS) that optimally spans chemical space. The resulting potential is broadly applicable to materials science, biology, and chemistry, and billions of times faster than CCSD(T)/CBS calculations.*

Along the same lines Hannu Häkkinen's group here in the Jyväskylä University, Nanoscience Center, demonstrated that it is possible to model the structure and dynamics of realistic monolayer-protected metal clusters using the so-called ***(Extreme) Minimal Learning Machine (E)MLM***. [48] Using data gathered in lengthy DFT (density functional) computations to train (E)MLM, they demonstrated that one can reproduce the structural parameters and energetics at several temperatures with a reasonable accuracy. The speedup was several orders of magnitude. One thing learned in this work was that it's not enough

[47] Smith *et al*, *Approaching coupled cluster accuracy with a general-purpose neural network potential through transfer learning* pdf (Nature Communications) .

[48] Antti Pihlajamäki *et al.*, *Monte Carlo simulations of $Au_{38}(SR)_{24}$ nanocluster using distance-based machine learning method*, J. Phys. Chem. A124,4827-4836 (2020), and Sami Malola and Hannu Häkkinen, *Prospects and challenges for computer simulations of monolayer-protected metal clusters*, Nature Communications. 12 (2021), (link to free article), de Souza Junior *et al.*, *Minimal Learning Machine: A New Distance-Based Method for Supervised Learning*, link to article (not freely downloadable).

to train on data of fairly good, low energy clusters because then MLM easily goes astrays and probes completely impossible structures.

In Summer 2021 two groups reported highly accurate predictions of ***protein structures***. The deep-learning models, AlphaFold2[49] and RoseTTAFold[50] could predict, given the amino acid sequence in a protein, most 3D atomic positions correct to within an angstrom. One aspect of the structure is that mutations don't occur in just single amino acids. Mutations among related amino acids are necessary to maintain stability and folding energy of the protein, and deep learning can reliably identify those correlated amino acids. A future challenge is to predict structures of multiprotein assemblies.

12.6 Gaussian process regression

I'm trying to give here an introduction to gaussian process regression (GPR). GPR is quite popular:

- If one has no idea what basis functions to use, gaussian processes offer a possibility for ***non-parametric fitting***. Non-parametric means effectively infinite number of parameters.
- Unlike other methods, GPR gives confidence limits. The standard deviation of a gaussian distribution measures the width of the distribution.
- The Central Limit Theorem says that the distribution of a *sum of random variables* approaches a gaussian for many distributions of the random variables.

Gaussian processes are implemented in the Tensorflow-Keras framework as GPflow, and in Pytorch as gpytorch.

You can implement GPR as a black box, but I'd like to give some flesh over the bones. Let $P(A|B)$ be the ***conditional probability*** that A happens if B has happened. Bayes' theorem is[51]

[49] J. Jumper *et al.*, Nature, Vol 596, 26 (2021).
[50] M. Baek *et al.*, Science 373, 871-876 (2021).
[51] I'm using the notation of Toussaint, *Bayesian Inference in Physics*, Rev. Mod. Phys., Vol. 83, (2011).

Likelihood Prior

$$P(H|D,I) = \frac{P(D|H,I)\,P(H|I)}{P(D|I)} \tag{25}$$

Posterior Marginal likelihood or Evidence

The conditional probabilities relate prior information I, new data D, and proposition or hypothesis H. The theorem tells how the ***prior (probability)*** $P(H|I)$, a probability distribution we somehow know *before* any new data is acquired, can be combined with ***likelihood*** $P(D|H,I)$ and ***marginal likelihood or evidence*** $P(D|I)$ to obtain the ***posterior likelihood*** $P(H|D,I)$. In short, Bayes' theorem tells how to add new data to prior knowledge to get a more informed probability distribution.

I've adobted the notation that emphasizes the fact that that the prior $P(H|I)$ is based on pre-acquired information I. If you leave I implicit, then the prior is $P(H)$ without bothering to write down where it came from, and the Bayes' theorem is[52]

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} \ . \tag{26}$$

After all, the pre-acquired information I hangs around as the right-most quantity in the equations given earlier, so it's quite alright to leave it unmarked. Humorously, I is a stubborn opinion that remains unchallenged.

Maybe a useful way is to think that from the start you have no idea what the result H is. Then you gain some data on the subject, mark it D_1. You continue, and new observations give you new data D_2. Repeat the experiment, and you obtain data

[52]The short notation is used in Bayes' theorem @Wikipedia.

D_3 and so on. Bayes' theorem accumulates knowledge like this,

$$
\begin{aligned}
P(H) &= ? && \text{no idea what the result is} \\
P(H|D_1) &= \frac{P(D_1|H)P(H)}{P(D_1)} && \text{result based on data } D_1 \\
P(H|D_2, D_1) &= \frac{P(D_2|H, D_1)P(H|D_1)}{P(D_2|D_1)} && \text{result based on data } D_1 \text{ and } D_2 \\
P(H|D_3, D_2, D_1) &= \frac{P(D_3|H, D_2, D_1)P(H|D_2, D_1)}{P(D_3|D_2, D_1)} && \text{result based on data } D_1,\ D_2, \text{ and } D_3 \\
&\dots . && (27)
\end{aligned}
$$

Priming data accumulation with "$P(H) = ?$" is inacceptable, so in Bayesian inference there must be *some* prior $P(H)$ to begin with. Any reasonable guess will do. The second line is the short form of Bayes' theorem, while the third line is the longer notation.

A "marginal" probability distribution indicates that some variables have been integrated (summed) away. Marginal likelihood (evidence) is marginal in the sense that it's the probability distribution for any proposition or hypothesis,

$$P(D|I) = \int dH \underbrace{P(D|H, I)}_{\text{likelihood}} \underbrace{P(H|I)}_{\text{prior}} \tag{28}$$

$$, \tag{29}$$

so it's also the normalization of the product distribution $P(D|H, I)P(H|I)$. In GPR $P(D|H, I)$ and $P(H|I)$ are gaussians, therefore $P(D|I)$ is also a gaussian. This is the whole point of using gaussians.

Predictions can now be made based on the posterior, which was the probabily distribution we got after obtaining new data. The posterior tells us how probable the hypothesis H is based on data D and the old, stubborn opinion I. The hypothesis H tells us what y_{pred} is at some chosen input x_{pred}, and we are - miraculously - able to take into account ***an infinite number of***

hypotheses and compute the probability distribution of predictions $y_{\rm pred}$! The predictive distribution is

$$P(y_{\rm pred}|x_{\rm pred}, D, I) = \int_H dH P(y_{\rm pred}|x_{\rm pred}, H) \underbrace{P(H|D, I)}_{\rm posterior} , \tag{30}$$

and the integration is simple since all probability distributions are gaussians.

Maybe it's now instructive to translate this talk about conditional probabilities and Bayesian inference into curves on the $x - y$ plane. You can think of I as a broad idea where the points (x, y) should lie, and data D is a set of a few measured points, $D = \{(x_1, y_1), (x_2, y_2), ..., (x_{N_D}, y_{N_D})\}$. The task is to construct all curves that go through points D and obey also the knowledge I. Each H gives the answer, not as a rock-solid "it's $y_{\rm pred}$", but as a probability distribution $P(y_{\rm pred}|x_{\rm pred}, H)$ that appears in the formula (30). You might now think that blah, I'd rather take a straight answer and not a distribution, but a distribution is better: From a distribution you can compute the ***mean value*** and the ***error bar*** (the width of the distribution of $y_{\rm pred}$). This makes GPR very different from ordinary curve fitting: While you can easily find, say, a spline curve that goes through points D, you can't get any estimate on how reliable that spline curve is between known points D. GPR gives both the mean curve *and* an error estimate.

The code gaussian_process_intro.py is an introduction to how the infinite number of basis functions are created. A smooth function has some correlation between points (x_1, y_1) and (x_2, y_2) if the values x_1 and x_2 are close.

In GPR we use a multivariate gaussian (aka normal) distribution, so that leaves the choice of the mean values μ and especially the choice of the ***covariance matrix*** Σ in the multivariate distribution

$$N(\boldsymbol{x}|\mu, \Sigma) \sim \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu)^T \Sigma^{-1} (\boldsymbol{x} - \mu)\right) . \tag{31}$$

This is an extension of the simple

$$N(x|\mu, \sigma) = \sim \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) , \tag{32}$$

and the covariance matrix is positive definite extension of the variance σ^2 to multiple dimensions. For example, two variables $\boldsymbol{x} = (x_1, x_2)$ could have

$$\Sigma = \begin{pmatrix} 1 & 0.4 \\ 0.4 & 1 \end{pmatrix} , \tag{33}$$

meaning that points x_1 and x_2 are slightly correlated with each other, $\Sigma(x_1, x_2) = 0.4$. For 5 points you'd have a 5x5 matrix Σ. In continous space it's better to choose a correlation function, a ***kernel function***,

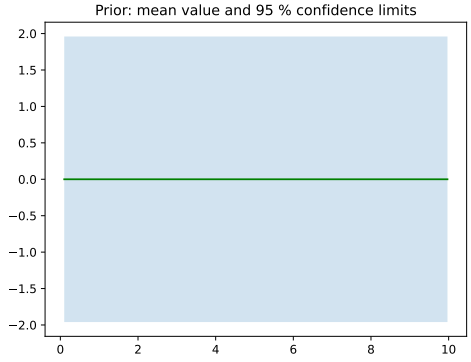
$$\Sigma(\boldsymbol{x}, \boldsymbol{x}) = k(\boldsymbol{x}, \boldsymbol{x}) \ . \tag{34}$$

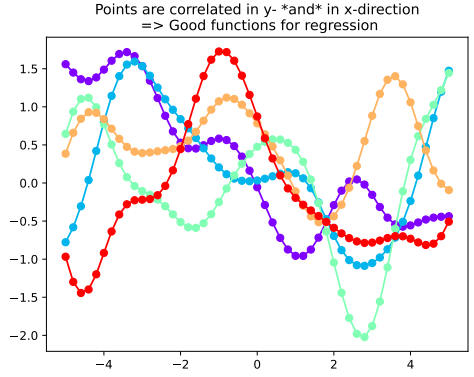
which gives higher correlation for points near each other. A popular choice is to compute the elements of Σ from the exponentiated quadratic kernel,

$$k(\boldsymbol{x}_1, \boldsymbol{x}_2) = \exp\left(-\frac{||x_1 - x_2||^2}{2\sigma^2}\right) \ , \tag{35}$$

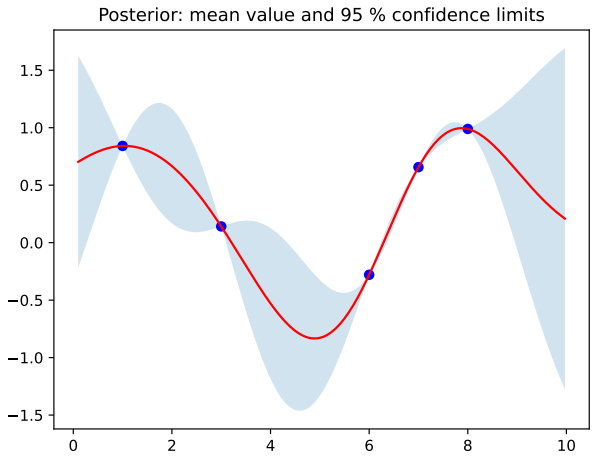
which gives 1 in the diagonal and the correlation falls off rapidly if the points are further apart. That's a recipe for nice, continuous basis functions (see figures below). I used this kernel in gaussian_process_intro.py.

Left: Prior knowledge of the result, representing $P(H|I)$. Right: A few representative gaussian process basis functions are show in the figure.





Now suppose you make a measurement and obtain five results $(x_1, y_1)...(x_5, y_5)$. These represent D, which you add to your prior knowledge. Generating new basis functions you notice they fluctuate between the measured points, but the basis functions are pinched to go through the measured points. This is done using Scikit-learn package `sklearn.gaussian_process` in the code sklearn_GPR.py.



All basis functions put together gives a probability distribution, with mean (red curve) and the 95 % confidence limits (shaded area).

12.7 JAX

It was difficult to deside where to put ***jax***, see jax @github.com. You've seen how automatic differentiation makes teaching neural networks easier. First there was ***autograd***. The page autograd @github.com says

> Note: Autograd is still being maintained but is no longer actively developed. The main developers (Dougal Maclaurin, David Duvenaud, Matt Johnson, and Jamie Townsend) are now working on JAX, with Dougal and Matt working on it full-time. JAX combines a new version of Autograd with extra features such as jit compilation.

JAX can do automatic differentiation and jit compilation. One asset may be a game changer, JAX can offload computations to GPU. JAX competes directly with NumPy and Numba, and neural network software can grow on top of JAX. JAX can do both forward-mode and reverse-mode automatic differentiation. The latter is backpropagation.

Forward-mode automatic differentiation, invented by R. E. Wenger 1964, computes derivatives by iteratively applying elementary operations to both the input values and their corresponding derivatives. It efficient if the number of inputs is small, and you want to compute the derivatives of a function with respect to those inputs. The iteration starts with an initial seed vector representing the derivatives of the input values. It then applies the elementary operations of the function to both the input values and their derivatives in a forward pass, updating the derivatives at each step.

The seed vector in forward mode is $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{x}} = 1$, similar to backward mode $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{y}} = 1$. Neural network backpropagation has seed $\frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$ for loss function $\mathcal{L}$. Forward mode uses the chain rule of differentiation to the function, such as $f(g(h(x)))$, from inside to outside, while backward mode traverses from outside to inside. Forward mode is more efficient if the function maps a small-dimensional input space to a much larget output space, $\boldsymbol{R}^n \to \boldsymbol{R}^m$ with $n \ll m$, since the number of sweeps is the input dimension. Similarly, backward mode is more efficient if $n \gg m$.[53]

[53] See AD @wikipedia.

JAX can take advantage of fused multiply–add (FMA) instructions. Notice how Numba needs an initial call to the jitted function while JAX doesn't.

jax_jit_test.py

```
import jax.numpy as jnp
from jax import jit
from time import process_time as T
import numpy as np
from numba import njit

def f(x):
  return x*s + 5*x

jax_f = jit(f) # compilation outside timing
numba_f = njit(f)
numba_f(np.ones(10))# compilation outside timing

print('Using JAX array')
x = jnp.ones((5000, 5000))
tic = T()
f(x)
toc = T()
print(f'{"f":>8} {toc-tic:<.6f} s')

tic = T()
jax_f(x)
toc = T()
print(f'{"jax_f":>8} {toc-tic:<.6f} s')

print('Using NumPy array')
x = np.ones((5000, 5000))
tic = T()
f(x)
toc = T()
print(f'{"f":>8} {toc-tic:<.6f} s')

tic = T()
numba_f(x)
toc = T()
print(f'{"numba_f":>8} {toc-tic:<.6f} s')
```

13 Parallel Python

There are almost too many avenues to take, see wiki about Python parallel processing. I'm reasoning that you have a multicore computer, so what you'd want is ***shared-memory parallel programming***. For that Python has `multiprocessing` or `concurrect.futures`.

If memory is not shared, like nodes in a computer cluster don't share memory, you need ***message passing***. For that we have ***MPI***, the message passing interface. Popular ones are OpenMPI, MPICH, Microsoft MPI (MS MPI), and Intel MPI. Python uses MPI in modules such as `mpi4py`. There are other parallel paradigms, too, such as the Hadoop framework for processing large volumes of data.

Let's get those cores working first.

13.1 Python Threads

With multicore CPU's, parallelization with Python Threads is easy. I was excited, until I watched David Beazley's Youtube talk about Python GIL (**G**lobal **I**nterpreter **L**ock) You may also visit David Beazley about GIL. [54]

[54] There are no-GIL interpreters, too, such as PyPy and Jython.

threads.py

```
import threading as th
from time import process_time as T

def count(n):
    while n>0:
        n -= 1

N = 100000000
# serial
tic = T()
count(N)
count(N)
toc = T()-tic
print("  serial: ",toc,"seconds")

#threaded
tic = T()
t1 = th.Thread(target=count,args=(N,))
t1.start()
t2 = th.Thread(target=count,args=(N,))
t2.start()
t1.join()
t2.join()
toc = T()-tic
print("threaded: ",toc,"seconds")
#   serial:  8.782534607 seconds
# threaded:  8.617532933000001 seconds
```

Python threading is no good for CPU-based tasks. Threads may be useful for watching IO-based, slow tasks.

13.1.1 PyPy - a user-friendly no-GIL interpreter

PyPy @pypy.org is a user-friendly JIT interpreter - almost a drop-in replacement for CPython - and gives reasonable speed improvements. The installation follows the familiar route. As root/admin, install pypy3 (pypy is pypy2, it's for Python2) using

a package manager, such as `apt-get` or `dnf`. Installation instruction may change, so it's better search for "install pypy3" and add your OS to the search line. Something like this:

```
$ sudo apt-get install pypy3
$ pypy3 -m ensurepip --user # install pip
$ pypy3 -m pip install numpy scipy matplotlib --user
```

These took some time. Now most of the Python examples should run. Faster, too, because `pypy` does JIT (see also Numba 11).

13.2 Python Multiprocessing

Obviously you can't turn off GIL in CPython, so CPython can't make full use of Python threads. But you *can* spawn multiple CPython interpreters for multiple tasks. Two standard library modules do exactly that:

- `multiprocessing` offers detailed control of parallel execution.
- `concurrent.futures` is an interface to `multiprocessing`, available since Python 3.2:
  - `ThreadPoolExecutor` uses a pool of *threads* to execute calls asynchronously - remember GIL.
  - `ProcessPoolExecutor` uses a pool of *processes* to execute calls asynchronously.

Since the API's of `thread` and `multiprocessing` are practically identical, let's modify threads.py,

multiprocesses.py

```
import multiprocessing as mp
from time import time as T
# don't use process_time, it's meaningless in multiprocessing

def count(n):
    while n>0:
        n -= 1

N = 100000000
# serial
tic = T()
count(N)
count(N)
toc = T()-tic
print("         serial: ",toc,"seconds")

#multiprocessing
tic = T()
t1 = mp.Process(target=count,args=(N,))
t1.start()
t2 = mp.Process(target=count,args=(N,))
t2.start()
t1.join()
t2.join()
toc = T()-tic
print("multiprocessing: ",toc,"seconds")
#         serial:  8.877176761627197 seconds
#multiprocessing:  4.42932391166687 seconds
```

Double the speed with two processes. Now we are getting somewhere!

13.2.1 How and what *not* to parallelize

I don't want to play down the importance of parallel execution, but you should realize that not all tasks should be parallelized. The code below is a textbook example of ***how to use a `Pool` of workers*** - and at the same time it is a textbook example of a ***completely idiotic thing to do***.

parallel_numpy_sqrt.py

```
# computes the  sqrt of a very many integers in parallel
import multiprocessing as mp
import numpy as np
from time import time as T



def main():
    res = []
    arg = np.arange(1000000)
    for nproc in range(1,13):
        if nproc==1:
            tic = T()
            root = np.sqrt(arg)
            toc = T()-tic
        else:
            tic = T()
            with mp.Pool(nproc) as pool:
                root = pool.map(np.sqrt,arg)
            toc = T()-tic
        print(nproc,toc)

if __name__ == '__main__':
    main()
```

```
  PID USER      %CPU %MEM     TIME+ COMMAND
20348 vap       91.4  0.7   0:36.00 python3
20403 vap        7.3  0.7   0:00.22 python3
20404 vap        7.3  0.7   0:00.22 python3
20402 vap        6.6  0.7   0:00.20 python3
20405 vap        6.6  0.7   0:00.20 python3
20406 vap        4.0  0.7   0:00.12 python3
```

Observations on Python 3.10 in an 12-core workstation:

- Serial execution: 0.004 seconds. Parallel execution: 1.95 seconds. Parallel is almost 500 times ***slower***.
- NumPy is very fast, it's compiled C-code
- Starting a process has a large overhead (handing work to a running process has an overhead, too).

The parallel speed in the example was so miserable that there must be something else than overheads in it. First, ***prosesses don't share memory space***, so `multiprocessing.pool.map` copies the whole, large list `arg` to every process. Sharing data is someting I've never done, so I leave it be. But even copying the arrays around can't be *that* slow, there is still something else going on.

parallel_numpy_sqrt2.py

```
# Testing how multiprocessing map passes arguments
import multiprocessing as mp
import numpy as np
from time import time as T

def fun(x):
    print("x=",x)
    return x

def main():
    res = []
    arg = range(10)
    for nproc in [1,5]:
        if nproc==1:
            print("A single process")
            tic = T()
            root = fun(arg)
            toc = T()-tic
        else:
            print(nproc,"processes using pool.map")
            tic = T()
            with mp.Pool(nproc) as pool:
                root = pool.map(fun,arg)
            toc = T()-tic
        print(nproc,'took time',toc)

if __name__ == '__main__':
    main()
```

```
A single process
x= range(0, 10)
1 took time 3.814697265625e-06
5 processes using pool.map
x= 1
x= 0
x= 4
x= 2
x= 5
x= 3
x= 7
x= 6
x= 8
x= 9
5 took time 0.01030731201171875
```

`multiprocess.pool.map` parallelizes execution but ***doesn't vectorize anything.***

This is why I got such a miserable parallel speed: serial NumPy vectorized and applied `numpy.sqrt` to the whole NumPy array,

but `multiprocess.pool.map` executed `numpy.sqrt` to every number in `arg` *one by one*.

I expected that a list of arguments would be chunked, and each chunk would be executed, if possible, vectorized in different processes [55]. It didn't, `multiprocessing` actually de-vectorized everything! Very well, I can do this manually, I just expected Python built-in's to handle this easy scenario.

Remark: There is a module `numpy.vectorize` but as the doc says,

> *The vectorize function is provided primarily for convenience, not for performance. The implementation is essentially a for loop.*

No need to use it.

13.2.2 Examples of `concurrent.futures`

Use a context manager to execute `function(x)` for `x` in an iterable object, e.g. a list:

```
import concurrent.futures as cf
def function(x):
    return x**3

with cf.ProcessPoolExecutor() as executor:
    result = executor.map(function, range(10))

print(list(result))
```

Again, computing cubes of array elements is not a clever parallel task, in reality you should use NumPy,

[55] CPU cores commonly have SSE, FMA, and AVX (AVX2) instruction sets to vector process at extreme speed.

```
import numpy as np

x = np.array(range(100))
result = x**3
```

Take the parallel example only as a proof of concept. The `concurrent.futures` code is pretty much the same as with `multiprocessing.Pool`,

```
import multiprocessing as mp
def function(x):
    return x**3

x = range(100)
with mp.Pool() as pool:
    result = pool.map(function, x)

print(list(result))
```

Both use as many cores as there is, which may be a good thing in a desktop computer but not necessarily in a cluster.

A better task to parallelize is to find text in files. The `glob` module offers "Unix style pathname pattern expansion",

concurrent_findtxt.py

```
# test if files *py contain word "numpy"
import glob
from time import sleep
import concurrent.futures
from itertools import repeat

def is_txt_in_file(args):
    string, infile = args
    res =  string in open(infile).read()
    sleep(1.) # delay added to make parallel execution observable
    return res

if __name__ == '__main__':
    files = glob.glob('*py')
    search_pattern = 'numpy'
    # one search pattern, multiple files; use repeat()
    args = zip(repeat(search_pattern),files)
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for pyfile, status in zip(files, executor.map(is_txt_in_file, args)):
            print(f"{pyfile} contains string {search_pattern}: {status}")
```

Notice that `executor.map(function, item1, item2)` would call `function(item1)` and `function(item2)`, not `function(item1,i`
To map arguments `('numpy', file1)`, `('numpy', file2)` ... as inputs, I used `itertools.repeat()` to get as many 'numpy's
I need to pair with each file name. Try `list(args)` to see how `zip(repeat(search_pattern),files)` works.

An interesting aspect is to use generators:

concurrent_findtxt_generator.py

```
# test if files *py contain word "numpy"
import glob
import concurrent.futures
from time import sleep
from itertools import tee, repeat

def is_txt_in_file(args):
    string, infile = args
    res =  string in open(infile).read()
    sleep(1.) # delay added to make parallel execution observable
    return res

if __name__ == '__main__':

    files = glob.iglob('*py') # generator
    files1, files2 = tee(files,2) # two independent iterators
    search_pattern = 'numpy'
    # one search pattern, multiple files; use repeat()
    args = zip(repeat(search_pattern),files1)
    with concurrent.futures.ProcessPoolExecutor() as executor:
        result = executor.map(is_txt_in_file, args) # generator

    for f, h in zip(files2, result):
        print(f"{f} contains {search_pattern}: {h}")
```

Generator `files1` will become exhausted - not in `args = zip(repeat(search_pattern),files1)` - but in `is_txt_in_file()` that uses it to generate file names. `itertools.tee()` creates two generators, `files1` and `files2`, so that I can use the latter to generate me again the list of files.

Instead of map, you can submit tasks and get their results as_completed.

concurrent_submit.py

```
import concurrent.futures
import numpy as np
from time import sleep
from time import process_time as T
import os

print(f'Your machine has {os.cpu_count()} cpus/cores')

def cube(arg):
    x,sl = arg
    sleep(sl)
    return x**3

if __name__ == '__main__':
    xsl = [(x,np.random.random()) for x in range(10)] # (x,sleep time)

    print('using map:')
    with concurrent.futures.ProcessPoolExecutor() as executor:
           for (x,sl), res in zip(xsl, executor.map(cube,xsl)):
               print(f'{x}^3 = {res:5d}, sleep was {sl:.3f}')

    print('using submit and as_completed:')
    futures = {}
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for arg in xsl:
            future = executor.submit(cube, arg)
            futures[future] = arg
        for f in concurrent.futures.as_completed(futures):
            res  = f.result()
            x,sl = futures[f]
            print(f'{x}^3 = {res:5d}, sleep was {sl:.3f}')
```

The output could be

```
Your machine has 8 cpus
using map:
0^3 =     0, sleep was 0.961
1^3 =     1, sleep was 0.150
2^3 =     8, sleep was 0.062
3^3 =    27, sleep was 0.888
4^3 =    64, sleep was 0.760
5^3 =   125, sleep was 0.850
6^3 =   216, sleep was 0.208
7^3 =   343, sleep was 0.697
8^3 =   512, sleep was 0.864
9^3 =   729, sleep was 0.082
using submit and as_completed:
2^3 =     8, sleep was 0.062
1^3 =     1, sleep was 0.150
6^3 =   216, sleep was 0.208
9^3 =   729, sleep was 0.082
7^3 =   343, sleep was 0.697
4^3 =    64, sleep was 0.760
5^3 =   125, sleep was 0.850
3^3 =    27, sleep was 0.888
8^3 =   512, sleep was 0.864
0^3 =     0, sleep was 0.961
```

- `map` branches execution to multiple "futures", and returns the *results of the futures in the order you started the tasks.*
- `submit` branches execution to multiple "futures", and `as_completed` picks up the *futures in the order they finish.*

Printing a future gives outputs like
`<Future at 0x7f0bacd38390 state=running>`
or if there is no worker free for this future,
`<Future at 0x7f0bacd38390 state=pending>`
and when finished, the state changes to
`<Future at 0x7f0bacd38390 state=finished returned int>`

A practicality: I could have shortened the `submit` part to

```
futures = [executor.submit(cube, arg) for arg in xsl]
```

This is fine if result is all I need, but since tasks complete in an inorderly fashion I would have no idea which `arg` gave what result. In the longer example I used a ***dictionary*** to hold `future:arg` pairs. Since `as_completed` tells which future finished, I can now go back and see what `arg` did it correspond to.

Another practicality: After the job is done, the context manager automatically calls `Executor.shutdown`; process cleanup is automatic.

13.3 Multiprocessing and Pool

`Pool` is a powerful feature in the `multiprocessing` module.

multi_pool.py

```
from multiprocessing import Pool
from time import sleep

def f(x):
    sleep(2/(x+1))
    print(x)
    return x*x

if __name__ == '__main__':
    x = range(101)
    with Pool(4) as p:
        res = p.map(f,x)
        print(res)
```

This creates a pool of 4 workers; looking at the process list you see it's mostly doing nothing:

```
0 S vap       3590 26052  3  80   0 - 105488 futex_ 16:50 pts/0   00:00:00 python3 multi_pool.py
1 S vap       3591  3590  0  80   0 - 50125 poll_s 16:50 pts/0    00:00:00 python3 multi_pool.py
1 S vap       3592  3590  0  80   0 - 50125 futex_ 16:50 pts/0    00:00:00 python3 multi_pool.py
1 S vap       3593  3590  0  80   0 - 50125 futex_ 16:50 pts/0    00:00:00 python3 multi_pool.py
1 S vap       3594  3590  0  80   0 - 50125 pipe_w 16:50 pts/0    00:00:00 python3 multi_pool.py
```

Notice that there are ***five*** processes: one running the show and four in the pool.

13.3.1 Safe locking with a context

Workers, threads or processes, may share resources but shouldn't access them simultaneously: ***locking*** sets access limitations. The non-simultaneity is guaranteed by a ***mutex***, MUTual EXclusion. The idea is that each worker locks the resource while accessing it.

However, it's all too easy to lock the resource and fail to release is due to an exception. As always, if you want to be sure a certain task is properly finished, you should use a ***context manager***. Luckily, `multiprocessing` and `threading` has a context `Lock()`, and in the function `good_way()` the context `with lock` makes sure the `lock` is always released, no matter how the task exits.

Remark: Some sources claim there's a method to inquire lock status of `lock=Lock()`, namely `lock.locked()`, but there's no such thing. Try `dir(lock)` or `dir(Lock())`, and you see the only methods are `acquire()` and `release()`.

---

contextlock.py

```
#from threading import Lock
from multiprocessing import Lock

# Set a lock and make sure it's properly released
def good_way(lock):
    print('called good_way')
    with lock:
        raise Exception('god_way raised an exception')

# Set a lock and release it in the end if no exception
def bad_way(lock):
    print('called bad_way')
    lock.acquire()
    raise Exception('bad_way raised an exception')
    lock.release()


if __name__=='__main__':
    lock = Lock()  # Lock is a context
    funs = [good_way, bad_way]
    for fun in funs:
        print(u'\u2500'*80)  # horizontal line
        try:
            fun(lock)
        except Exception as exc:
            print('Exception: ',exc)
            print('Trying to acquire and release the lock again')
            lock.acquire()
            lock.release()

        print('Got here')
```

Remark: Locking is one way to avoid the so-called ***Producer-Consumer problem*** (Wikipedia) shows what problems concurrency faces. The Producer manufactures items and the Consumer consumes them but all this happens asyncronously. As the Wiki page explains, a poor solution would be to have a `consumer` and a `producer` routine running syncronously. In addision, there are `sleep` and `wake_up` library routines. While awake, the consumer consumes and falls asleep when items run out. If there are less than max number of items the producer wakes up and falls asleep once the stock is full. This solution easily creates a ***race condition*** between producing and consuming, and may lead to a ***deadlock***: nothing happens.

> *The consumer eats the last item and is about to fall asleep. Right at that moment food is out and the producer happens to wake up and makes an item. The ensuing discussion "hey, here is some food, wake up!" - "I can't, I'm not sleeping yet" has no end.*

13.3.2 Bohrium

Bohrium (source code @github.com) is a parallel replacement for NumPy written in Niels Bohr Institut, Copenhagen.

> *Bohrium provides automatic acceleration of array operations in Python/NumPy, C, and C++ targeting multi-core CPUs and GP-GPUs. Forget handcrafting CUDA/OpenCL to utilize your GPU and forget threading, mutexes and locks to utilize your multi-core CPU just use Bohrium!*

If you manage to install `bohrium`, then yes, usage is that simple. Either replace `import numpy as np` with `import bohrium as np` or just run the code as (8 core machine)

```
$ OMP_NUM_THREADS=8 python bohrium test.py
```

A quick test showed it really does run in parallel. If there are parts that Bohrium can't handle, such as Matplotlib, it just falls back to NumPy and you code still runs.

13.4 Subprocess: easy parallelism

There is a simple way to use the expressive power of Python and the speed of an external program. Start the external program from within Python and use it's output in Python. The Python module Python `subprocess` is made for that. First, to execute shell command `ls -la` [56],

```
import subprocess
subprocess.run(["ls", "-la"])
```

A more general usage is also simple (you should provide `command`)

[56] `subprocess` module is overtaking modules `os` and `spawn`.

```
import subprocess
proc = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE)
proc.wait()  # wait for command to finish
print(proc.returncode)
```

The doc hints how to get the shell command+options right (see shlex and wildcards @Stackoverlow). If in bash shell the command is

```
$ du -sh *
```

then in Python this could be run as

subprocess_ex1.py

```
import subprocess as sp
import shlex
import glob

cmd = shlex.split('du -sh *')
cmd = cmd[:-1] + glob.glob(cmd[-1])
with sp.Popen(cmd, stdout=sp.PIPE) as proc:
    print(proc.stdout.read().decode('ascii'))
```

The motivation to use the lastter is that now I can do all inside Python and add extra functionality as I please. The `glob` module was used here to handle the wildcard `*`; usually this is done by the shell but now we have none; `stdout` is a byte sequence (`b'...'`), decoded here to ascii characters. Notice that adding `shell=True`, for example

```
sp.Popen(cmd, stdout=sp.PIPE, shell=True)})
```

would run the command `cmd` through the shell defined in the `SHELL` environment variable (in my case, bash).

The reason why I'm talking about the `subprocess` module in the parallel section is that

You can trivially start multiple `subprocess`'s that run in parallel and control the output in Python

Note that this code starts multiple bash shell `sleep` processes, it's not calling Python `sleep()` function. [57]

subprocess_sleep.py

```
# start multiple subprocesses *in parallel*
import subprocess as sp
from time import time as T

def sleepy(snoozetime):
    proc = sp.Popen(['sleep',str(snoozetime)])
    return proc

tic = T()
procs = []
nproc = 10
for i in range(nproc):
    proc = sleepy(1.0)
    procs.append(proc)
    print(f'process {i} put to sleep for 1 second')

# wait till they finish
for proc in procs:
    proc.communicate()
toc = T()
print('all done in',toc-tic,'seconds')
```

This program will finish in about 1 second. It started ten **embarrassingly parallel** one-second jobs.

This was nothing fabulous, I could've done it on a one-liner in bash,

```
$ for i in {1..10}; do sleep 1 & done
```

or, as a bit more readable bash script,

```
for i in {1..10}
do
   sleep 1 &
done
```

These run ten sleep processes on the background and return almost immediately. My point is that with `subprocess` I can control all within Python, and have the full Python machinery available to manipulate the results.

[57] This code uses `time.time` to better reflect the result.

13.5 MPI Parallelism with `mpi4py` (*MPI for Python*)

MPI stands for Message Passing Interface, an interface for intercommunication of processes. This communication can happen in-node or between nodes in a ***cluster***. MPI is a reliable partner, it's standardized and portable. The module `mpi4py` targets High Performance Computing (HPC), and in 2009 it was rewritten in ***Cython*** from scratch.

Before communication takes place, Python objects are serialized using Python's native `pickle` protocol (see section 5), and afterwards data is deserialized. The small overhead of using the CPU and memory is tolerable, considering pickling is simple and convenient.

Remark: ***Installing MPI on Windows 10***
Instead of OpenMPI you'd probably want to choose Microsoft MPI now that you're using their system. Good instruction how to install are here Nickson Joram's blog @Medium. This will get you going:

- Install the Visual Studio community version: link
- Install Microsoft MPI, download files `msmpisetup.exe` and `msmpisdk.msi` from link and execute.
- Launch the command prompt and type `python`. It will start the interpreter, or redirect you to MS-Store, and asks if you want to install Python 3.9. You want.
- Start the Python interpreter from the command prompt to see that it works.

The rest is simple. In the command prompt,

```
C:\users\vap> pip install upgrade pip
C:\users\vap> pip install mpi4py
C:\users\vap> pip install numpy scipy matplotlib mpi4py
```

Test that `mpiexec` works and run, for example, my `mpi4py` sample `MC_pi_mpi4pi.py`

```
C:\Users\vap>
C:\Users\vap>mpiexec
Microsoft MPI Startup Program [Version 10.0.12498.5]

Launches an application on multiple hosts.

Usage:

    mpiexec [options] executable [args] [ : [options] exe [args] : ... ]
    mpiexec -configfile <file name>

Common options:

-n <num_processes>
-env <env_var_name> <env_var_value>
-wdir <working_directory>
-hosts n host1 [m1] host2 [m2] ... hostn [mn]
-cores <num_cores_per_host>
-lines
-debug [0-3]
-logfile <log file>

Examples:

    mpiexec -n 4 pi.exe
    mpiexec -hosts 1 server1 master : -n 8 worker

For a complete list of options, run mpiexec -help2
For a list of environment variables, run mpiexec -help3

You can reach the Microsoft MPI team via email at askmpi@microsoft.com

C:\Users\vap>
```

```
C:\Users\vap>mpiexec -n 4 python -O  MC_pi_mpi4py.py
4 processes
N = 1024        pi = 3.2013846153846153731          +/- 0.31539433380898995729          timing 0.0051 s
N = 4096        pi = 3.1318675042832664345          +/- 0.16961504439362071195          timing 0.0009 s
N = 16384       pi = 3.1462424712266683357          +/- 0.09237895931963992224          timing 0.0010 s
N = 65536       pi = 3.1394067900779307756          +/- 0.03694071173172774891          timing 0.0000 s
N = 262144      pi = 3.1452336715425839308          +/- 0.01974957044445822124          timing 0.0032 s
N = 1048576     pi = 3.1424331782241972099          +/- 0.00970412703124970931          timing 0.0138 s
N = 4194304     pi = 3.1419563256853502153          +/- 0.00478874175434744215          timing 0.0710 s
N = 16777216    pi = 3.1413269036099444875          +/- 0.00319547779405307556          timing 0.3150 s
N = 67108864    pi = 3.1418529748487507014          +/- 0.00092408159204234060          timing 1.2740 s
N = 268435456   pi = 3.1414847671898522563          +/- 0.00075627191128939603          timing 4.9130 s
saving pi estimate figure to file pi_4.png
saving timings figure to file timings_4.png

C:\Users\vap>
```

Programs using `mpi4py` commonly contain the following lines:

mpi4py_1.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD  # communicator
size = comm.Get_size() # world size, that is number of MPI workers are around
rank = comm.Get_rank() # index of the process running this code
print(f'comm = {comm}')
print(f'size = {size}')
print(f'rank = {rank}')
```

Run through the Python interpreter and you won't get anything exciting,

```
$ python mpi4py_1.py
comm = <mpi4py.MPI.Intracomm object at 0x153f339b1ab0>
size = 1
rank = 0
```

However, execute 4 processes in parallel,

```
$ mpiexec -n 4 python mpi4py_1.py
comm = <mpi4py.MPI.Intracomm object at 0x1478ee5aeab0>
size = 4
rank = 0
comm = <mpi4py.MPI.Intracomm object at 0x14bb0e2dbab0>
size = 4
rank = 3
comm = <mpi4py.MPI.Intracomm object at 0x154b599f3ab0>
size = 4
rank = 1
comm = <mpi4py.MPI.Intracomm object at 0x147349dc7ab0>
size = 4
rank = 2
```

Similar outcome is with the command

```
$ mpirun -n 4 python mpi4py_1.py # start 4 processes in parallel
```

The purpose of `mpiexec` or `mpirun` is to give each process within a communicator a distinct ***rank***. In fact only `mpiexec` is defined in the MPI standard, but I've never come across a system without `mpirun`. In the widely used OpenMPI package `mpirun` is defined to be `mpiexec`. As the FAQ page explains,

> "Specifically, they are symbolic links to a common back-end launcher command named orterun (Open MPI's run-time environment interaction layer is named the Open Run-Time Environment, or ORTE - hence orterun)."

Remark: Intel has it's own MPI. If you use conda, then you can create a virtual environment using `intel` channel,

```
$ conda create -n my_intel intelpython3_full python=3.7
$ conda activate my_intel
(my_intel) $
```

This installs `mpi4py` *and* the command `mpirun` from package `impi_rt`. I'm telling all this because if you now install OpenMPI on top of that with ***WARNING! DON'T!*** `(my_intel) $conda install openmpi` it probably does install, with only a mild warning about inconsistent environment. The OpenMPI package has overwritten `mpirun`, so your `mpi4py` installation and `mpirun` won't work together any more. To cure this situation, do

```
(my_intel) $ conda remove openmpi
(my_intel) $ conda install --force-reinstall impi_rt
```

I've grown used to `mpirun`. In a SLURM batch queue system your resource reservation is automatically passed on to `mpirun`, and you can send the job to the execution queue with just

```
$ mpirun python mpi4py_1.py  # in SLURM, this starts as many processes as the SLURM script reserves
```

If you execute this command in a computer with no batch queue system it'll consume all available resources.

Send 'Hello World' to all processes

mpi4py_2.py

```
from mpi4py import MPI


comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

print(comm)
print(f'size is {size}')
print(f'rank is {rank}')

if rank == 0:
    msg = 'Hello, world'
    dst = 1
    print(f'rank {rank} sends message {msg} to rank {dst}')
    comm.send(msg, dest=dst)
elif rank == 1:
    mess = comm.recv()
    print (f'rank {rank} got message {mess}')
```

The code will fail for a single process but it works for two or more,

```
$ mpirun -n 2 python mpi4py_1.py
rank 0 sends message Hello, world to rank 1
```

```
rank 1 got message Hello, world
```

The communicator and it's size are common to all processes but they have individual ranks. Based on the rank we can divide tasks.

Parallel Monte Carlo estimate of π

Monte Carlo computation of π is a canonical example. Pick random points in a 2x2 square centered at the origin, and count how many hit the unit circle. The probability of hitting the unit circle is $hits/tries = \pi 1^2/2^2 = \pi/4$, so there you have it. The code is too long to show here, see MC_pi_mpi4py.py. Here are some ideas used in the code.

- Kick up the parallel `comm`, and get `nproc = comm.Get_size()` as shown earlier. I could've used the name `size` but somehow find `nproc` more appropriate.
- You want to use all `nproc` processes to pick different sets of random points.
- Choose how many random points `N` you want to use in total.
- Calculate how many points each process should sample. Since `N` is usually not divisible by `nproc`, divide the odd jobs evenly to processes. ***Jobs divided to processes and batches*** is the array `ns`, returned from `taskdivider()`. The purpose of batches is given below.
- Distribute entries in `ns` to processes. This is done now using `mpi.scatter()`.
- Each process calls `compute_pi(ns)`. If, for example, `ns=[101,100,100,100]`, then compute 4 numerical estimates of π, and return them to calling routine.

  Details of `compute_pi()`:
  `np.random.random()` returns random values in range [0,1], so `2*np.random.random()` returns random values in range [0,2], and shift those `2*np.random.random()-1` to get random values in range [-1,1]. I want `n` 2-dimensional points, so I call `2*np.random.random((n,2))-1`.

```
def compute_pi(ns):
"""
input:  ns   list of block sizes
output: pis  len(ns) estimated pi values
method: pick n x n random points in a square area [-1,1] surrounding a circle radius 1
```

```
        probability of hitting the circle is
        P = (area of circle)/(area of square) = pi/(2*2) = hits/n => pi = 4*hits/n
    """
    pis = np.zeros(len(ns))
    for i,n in enumerate(ns):
        points = 2*np.random.random((n,2)) - 1 # random points in square
        # count points inside the unit circle; 1 (True) for hits, 0 (False) for misses
        hits = np.sum(np.linalg.norm(points,axis=-1) < 1)
        pis[i] = 4*hits/n
    return pis
```

The `np.linalg.norm(points,axis=-1) < 1` is an array of booleans `True` and `False`, but in Python 3.x `True` *is* 1 and `False` *is* 0, so I can compute their sum using `np.sum()`.

- From the array of estimates for π, I can calculate the mean values and an error estimate using NumPy functions `np.mean()` and `np.std()`. The latter computes the standard deviation, which is the statistical error in this case. For a single value `np.std()` is 0, therefore I block tasks to `nb` batches.

`comm.scatter()` scatters data to workers,

```
ns = comm.scatter(ns, root=0)
```

scatters `ns` from rank 0 to the rest, and `ns = ...` makes them pick it up and call the received data `ns`. The opposite action is `comm.gather()` which collects values from all workers. After calculations, call

```
pis = comm.gather(pi_est, root=0)
```

to collect the list `pi_est` of calculated estimates from all workers. Notice that only rank 0 has the list, other ranks have `pis` equal `None`. [58]

[58] The type of `None` is `<class 'NoneType'>`. Clever Python.

Collective calls

Logically, `scatter()` and `bcast()`, `allgather`, and `alltoall` are known as ***collective calls***. If you want all workers to reveive the same `data`, broadcast it using `bcast(data, root=0)`. The call syntax is

```
comm.Scatter(data, recvbuf, root=0)                # scatter data to workers
comm.Gather(sendbuf, recvbuf, root=0)              # collect from workers
data= comm.bcast(data, root=0)                     # broadcast to all workers
comm.Reduce(value, value_sum, op=MPI.SUM, root=0)  # sum up values from workers
```

Here `MPI.SUM` tells to reduce the sum, product would be reduced with `MPI.PROD`.

13.5.1 send/recv or Send/Recv

`send/recv` are made for sending/receiving general Python objects, therefore they are slow. `Send/Recv` communicate contiguous NumPy arrays much faster.

```
if rank = SOURCE:
    comm.send(data, dest=DEST, tag=TAG)
else:
    data = comm.recv(source=SOURCE, tag=TAG)
```

`TAG` adds an extra identification layer on top of source and destination ranks. The `recv` function will wait for `data` that comes from `SOURCE` *and* is tagged with `TAG`.

NumPy array MPI communication is an example of a so-called ***buffer-like object***. In calls to `Recv` the receiving buffer must be there in advance, so that it can receive data immediately,

```
if rank == SOURCE:
    comm.Send(data_sent, dest=DEST, tag=TAG)
else:
    data_reveived = np.array(...)  # pre-define the array
    comm.Recv(data_received, source=SOURCE, tag=TAG)
```

I used distinct names `data_sent` and `data_received` to emphasize that they aren't the same arrays, although their sizes must match:

> The sending and receiving buffers must have the same size

The `data_sent` you `Send` must match in size the `data_received` in the `Recv`. A mismatch is easy to spot in short examples.

mpi4py_Send_Recv.py

```
#
# run with
# mpirun -np 2 python mpi4py_Send_Recv.py
#
from mpi4py import MPI
import numpy as np
from  time import process_time as T

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

Nrep = 1000  # number of repeats
shape = (100,5) # shape of NumPy array to communicate

# send/recv Python objects
if rank== 0: tic = T()
for _ in range(Nrep):
    if rank == 0:
        data_sent = np.random.random(shape)
        comm.send(data_sent, dest=1, tag=11)
    elif rank == 1:
        data_received = comm.recv(source=0, tag=11)

if rank== 0: timing1 = T() - tic

# Send/Recv NumPy array
if rank== 0:  tic = T()
for _ in range(Nrep):
    if rank == 0:
        data_sent = np.random.random(shape)
        comm.Send(data_sent, dest=1, tag=12)
    elif rank == 1:
        data_received = np.empty(shape, float)
        comm.Recv(data_received, source=0, tag=12)

if rank==0:
    timing2 = T()-tic
    print(f'{Nrep} times MPI communicate NumPy array {data_sent.shape}' )
    print(f'send/recv took {timing1:.5f} seconds')
    print(f'Send/Recv took {timing2:.5f} seconds')

#1000 times MPI communicate NumPy array (100, 5)
#send/recv took 0.00923 seconds
#Send/Recv took 0.00335 seconds
```

The example uses automatic MPI datatype discovery, and you can write simply

```
data_sent = np.random.random(shape)
comm.send(data_sent, dest=1, tag=11)
```

without telling that `data_sent` has type `MPI.DOUBLE`. A too small receive buffer, for example

```
if rank == 0:
      data_sent = np.random.random((10,10))
      comm.Send(data_sent, dest=1, tag=12)
elif rank == 1:
      data_received = np.empty((5,5), float)
      comm.Recv(data_received, source=0, tag=12)
```

will cause an exception `mpi4py.MPI.Exception: MPI_ERR_TRUNCATE: message truncated`. However, a too big receive buffer is legitimate,

```
if rank == 0:
      data_sent = np.random.random((10,10))
      comm.Send(data_sent, dest=1, tag=12)
elif rank == 1:
      data_received = np.empty((50,50), dtype=float)
      comm.Recv(data_received, source=0, tag=12)
```

but the array `data_received` will contain carbage filled with `np.empty()` and no warning is issued. To avoid array size mismatches `mpi4py_Send_Recv.py` had `shape = (100,5)` visible to all ranks.

Broadcasting a NumPy array

mpi4py_numpy_broadcast.py

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = np.arange(10, dtype=int)
    print(f'data to be broadcasted: {data}')
else:
    data = np.empty(10, dtype=int)

comm.Bcast(data, root=0)

# check all got the data:
print(f'rank  {rank} got {data}')
```

13.5.2 Shutting down MPI jobs after an exception

In case an exception is raised the MPI runtime should kill all child processes and shutdown with an error code. Exit is handled by a call to `MPI.COMM_WORLD.Abort` or `MPI_ABORT`. However, in some MPI implementations, notably in `mpi4py`, a process may be left behind, waiting for a communication from a dead process.

If you run the previous example with just one process, there's no-one to receive the data. The failure occurs in a single process case, so exiting that process is enough and no processes will be left behind. This is but a happy coincidence, exceptions happen in multiprocess jobs, too.

Consider an exception in a two-process job. I could make a process call `raise Exception('error here')`, but it's simpler

to just compute 1/0,

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
assert size > 1
if rank == 0:
  1/0
  comm.send(None, dest=1, tag=5)
elif rank == 1:
  comm.recv(source=0, tag=5)
```

If you run this code (call it `test.py`),

```
  $ mpirun -np 3 python test.py
  Traceback (most recent call last):
    File "test.py", line 8, in <module>
      1/0
  ZeroDivisionError: division by zero
```

`Abort()` is never called, the code hangs and must be killed manually. This is especially annoying and harmful in a cluster environment, because jobs left running in compute nodes after the main process has died take up resources the batch job system thinks are free.

Aborting `mpi4py`

There's a way to abort `mpi4py` processes on exceptions. Run the code with the ***-m mpi4py*** option,[59]

[59] See mpi4py.run.

```
$ mpirun -np 3 python -m mpi4py  test.py
Traceback (most recent call last):
File  ....
File "test.py", line 8, in <module>
  1/0
ZeroDivisionError: division by zero
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode 1.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------
```

and the failed code graceful exited with `MPI_ABORT`.

A side note on Python. You must run the code as `mpirun -np 3 python -m mpi4py test.py`. This is *not* the same as running `mpirun -np 3 python test.py` with `import mpi4py` in the file `test.py`. The reason is that

```
python -m mod mycode.py
```

treats `mod` as a script and executes it before interpreting `mycode.py`. On the other hand, `import mod` in a script only brings everything in `mod` to your scripts namespace, without executing anything.

13.5.3 Non-blocking communication

So far all MPI communication I've talked about is ***blocking***, meaning processes can't proceed until the communication ends. The ultimate blocking call is `MPI.COMM_WORLD.Barrier()`, which waits for all processes to arrive at the barrier.

Blocking can sometimes be avoided, and for that `mpi4py` has a set of non-blocking calls, adding `i` or `I` to the name of the blocking sibling:

```
# blocking send
comm.send(data, dest=1, tag=5)
data = comm.recv(source=0, tag=5)
# non-blocking send is isend
req = comm.isend(data, dest=1, tag=5)
do_something_else()
req.wait() #
# non-blocking recv is irecv
req = comm.irecv(source=0, tag=1)
data = req.wait()
```

The `comm.isend` and `comm.irecv` methods return ***request instances*** (hence I called the variable `req`). These request instances must be completed with a blocking call to either `req.test` or `req.wait`. If you just want to inquire the status of the request `req` without stopping, then `req.test` will return `True` if the operation has completed.

14 Python as a glue language

Motivation:
For a large part, programs are not about heavy computations but housekeeping. Lines after lines reading input data, fixing it, analysis of output data, and finally saving and plotting results. While it's *possible* to do all that in C/C++, it isn't always worth the effort. Python has excellent household skills, it's simple and flexible, easy to test and to debug.

Even a Python API itself can be repetitive, boring, and unintuitive. But how low level control do you actually need? Then go on and write an easy-to-use Python frontend - yes, a Python frontend to Python. This happened to old variants of Theano and Tensorflow neural network environments. Keras could use either as a backend, and it was a lot easier and more fun to use! The Keras approach was so successful that it was built into Tensorflow 2. PyTorch had two years of Tensorflow experience to learn from, and the developers put a lot of effort to create an approachable API to begin with.

How about Combining Python and C/C++?

> From now on I assume you have a C/C++ code you want to use in Python

If not, consider skipping this section. Program in Python and import Numba.

14.1 Python extensions and embedding Python

- **Python extensions**
  are modules written in C/C++ or some other language. The modules can be imported to Python code using `import module`. The extension can also be a shared library (`library.so` in Linux, DLL in Windows), loaded into Python interpreted using extension API **Cython** or Python C API. Both NumPy and SciPy are Python extensions.

- **Embedding Python**
  You can also embedd the Python interpreter in C/C++ or some other language. Embedding Python gives the host code Pythonic flexibility and scripting capabilities.

Both extending and embedding Python can be done using the Python/C API. Writing C/C++ extensions using Python/C API is documented here. This approach is used if you see in C/C++ code lines such as

```
#include <Python.h>
static PyObject *
func(PyObject *self, PyObject *args)
{...}
```

In my opinion, this isn't a very elegant way. I don't want to marry C/C++ and Python on this level.

Remark: One reason for not going through the Python/C API is that it's so easy to program ***memory leaks***. Every Python object has a ***reference count***, a counter telling how many objects reference to it. When that counter hits zero the object can be safely deleted freeing memory, because no-one can and will use it ever again - that's ***garbage collection***. If you create another reference to an object the reference count increases, and if you fail to decrease that count twice the object can't ever be deleted. This reference counting problem induces memory leaks, extensively discussed in c-info.how-to-extend. If you want to see what the reference count is, try

```
import sys
import numpy as np

a = np.array((10,10))
print(sys.getrefcount(a))
# 2
b=a
print(sys.getrefcount(a))
# 3
```

The first reference count 2 looks odd, shouldn't it be 1? From web reference of `sys.getrefcount()`,

> *Return the reference count of the object. The count returned is generally one higher than you might expect because it includes the (temporary) reference as an argument to getrefcount().*

From the numerics point of view, Python extensions are more relevant. I will discuss calling C/C++ from Python using SWIG @swig.org (Simple Wrapper Interface Generator), which makes the task of getting C/C++ and Python to work together more enjoyable. Well, less painful.

14.2 SWIG (Simplified Wrapper and Interface Generator)

This chapter is a stub, and reflects the fact that I haven't used SWIG in real projects.

Why `SWIG @swig.org` ?[60]

- SWIG supports many languages: Python, Perl, Ruby, Tcl *etc.*.
- If I had a well-tested C++ code base, I wouldn't want to pollute it with non-C++ instructions. SWIG keeps the C++ and Python sides separated, and supports standard C++ features.
- SWIG was created to address limitations in C/C++, and the documentation mentions
  - Writing a user interface is rather painful (i.e., consider programming with MFC, X11, GTK, or any number of other libraries).
  - Testing is time consuming (the compile/debug cycle).
  - Not easy to reconfigure or customize without recompilation.
  - Modularization can be tricky. Security concerns, such as buffer overflows.
- The latest update is SWIG-4.1.1 from 2022, but it's a mature code so I'm not worried.

14.2.1 SWIG examples

SWIG depends on ***C++ header files***, not on the rest of the C++ source code

C++20 added ***modules***, and headers may be there only for backward compatibility. Should I be worried?

I'm not sure about C++20 module support in SWIG. SWIG has `%module`, but it's a completely different thing.

[60] SWIG core is by none other than David Beazley.

To tell SWIG how to read the headers you need a ***SWIG interface file***, a file containing ANSI C/C++ declarations and special SWIG directives. At this point you may want to consult the examples in the SWIG tutorial or the SWIG manual.

The tutorial part *SWIG for the truly lazy* suggest that you first try to include the C++ header file:

```
%module example
%{
/* Includes the header in the wrapper code */
#include "header.h"
%}

/* Parse the header file to generate wrappers */
%include "header.h"
```

Done that, you can use the interface file `example.i` to create the wrapper `example_wrap.cxx`,

```
$ swig -c++ -python -py3 example.i
```

Best practice: Let the Python module `distutils` do the compilation and installation of Python extensions.

Since Python 3.10 `distutils` is marked deprecated and is replaced with `setuptools`. NumPy web page says (distutils status @numpy.org "`numpy.distutils` has been deprecated in NumPy 1.23.0. It will be removed for Python 3.12; for Python $\leq$ 3.11 it will not be removed until 2 years after the Python 3.12 release (Oct 2025)." At the moment of writing Python 3.12 is being released. Should I be worried?

The module `distutils` takes care of the Python version-dependent features. In total, you need

1. The ***C/C++ code*** you want to embed (without `int main()`)
2. A SWIG ***interface file*** (usual file suffix `.i`, like in `interface.i`)
   Some benefits of *not* listing all header files in the interface:

- Limit wrapper generation only to functions that you actually need in Python
- No need to dig numerous header files to see how the SWIG Python interface is constructed

3. A `distutils` ***setup file*** (usually `setup.py`)

14.3 Cython

The web page Cython says that

> *Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language (based on Pyrex). It makes writing C extensions for Python as easy as Python itself.*

See also quickstart quide.

A canonical example is finding prime numbers (picked from cython_tutorial),

primes.pyx

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

The file extension `.pyx` reveals breaking of Python source compatibility (e.g. the `cdef int` instructions). In addition to the function `primes.pyx` you need a setup file for `setuptools`, the successor of `distutils`,

setup_primes.py

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("primes.pyx"),
)
```

Compile the function,

```
$ python setup_primes.py build_ext --inplace
```

and import the module to Python,

```
>>> import primes
>>> primes.primes(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

14.3.1 Creating a standalone executable with Cython

Consider the prime number function and a main routine,

primesmain.py

```
import sys

def primes(kmax):
    klim = 1000
    try:
        if kmax>klim: raise ValueError
        p=[None]*kmax
        result = []
        k = 0
        n = 2
        while k < kmax:
            i = 0
            while i < k and n % p[i] != 0:
                i = i + 1
            if i == k:
                p[k] = n
                k = k + 1
                result.append(n)
            n = n + 1
        return result
    except:
        print('too many primes (>%d)'%klim)
        return -1

if __name__ == "__main__":
    try:
        if sys.argv[1:][0].isnumeric():
            n = int(sys.argv[1:][0])
            print(primes(n))
        else:
            raise ValueError
    except:
        print('usage: ',sys.argv[0],' number')
```

This can be run from the command line,

```
 $  python primesmain.py 10
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

`Cython` can generate a 5729 lines of C-code from 33 lines of Python code,[61]

```
$  cython --embed -o primesmain.c primesmain.py
```

What's embedded is the CPython interpreter. Compile the C-code, add paths to the header file `Python.h` ad link the Python library,

```
$  gcc  -I/usr/include/python3.11 -o primesmain  primesmain.c -lpython3.11
```

Testing,

```
$  primesmain 10
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

`Cython` can make your Python code faster, but you need to do at least two things:

- Type variables (with `cdef int` etc.) ← ***very important for speed***
- Make sure arrays are correctly accessed from C

You may end up editing the code quite extensively, and in the end your Cython code may look nothing like Python. The good news is that with minimal changes you get *some* speedup. For Cython benchmarks, see Fibonacci speed with Cython @readthedocs (emphasizing that for Fibonacci, cacheing is far more important that cythonizing) and standard deviation with Cython @readthedocs. The pages are from 2014, so take the results with a grain of salt.

Cython and C++

This is a long story. I suggest you read, for example, wrapping C++ @readthedocs. In the heart is again a `setup.py` file for `setuptools`.

[61] 2023: 8375 lines.

15 Julia

Julia (See @Wikipedia) is a new general purpose programming language, especially suited for scientific computing. The development started 2012 by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Version 1.0 was released on 8. August 2018, version 1.9.3 was released August 24, 2023.

How to get Julia:

- Homepage https://julialang.org
- Downloads https://julialang.org/downloads
- Source code https://github.com/JuliaLang/julia
- Packages https://juliapackages.com/

A good place to start is Youtube videos from Juliacon 2022 and 2023.
Julia has lots of useful packages:

- Differential equations, Fourier transforms, iterative solvers, nonlinear dynamics *etc.*
- Visualization interface and toolset (`Plots.jl`)
- Interfaces for C, fortran, C++, Python, R, and Java
  Call Julia from python: `PyJulia`. Call Python functions (`PyCall`). Call C functions directly.
- Deep learning, machine learning, and AI.
- Parallel and distributed computing
- Tasks (Coroutines) and Channels
- CUDA support

A Deep Introduction to Julia for Data Science and Scientific Computing
contains a particularly nice summary of syntax in MATLAB, Python and Julia,
MATLAB-Python-Julia cheatsheet

The Debian benchmark game gives a hint of speed, Julia against C, and Julia against Python 3. Summary: C beats Julia (sometimes just marginally). Julia beats Python in most cases by a very large margin. Bear in mind, though, that the comparison was against *bare* Python, not using NumPy or SciPy. Nobody is supposed to use Python like that, but since NumPy is using C, one would benchmark C against Python and C. Some of the C-codes use heavily AVX/AVX2 intrinsics[62]

The source codes are also available, so it's a good place to see how common tasks can be coded in Julia.

15.1 Julia IDEs

15.1.1 Julia for Visual Studio Code

See julia-vscode.org or julia-vscode @github. Get Julia, then Visual Studio Code. and Choose `Install` in the `VS Code Marketplace` (more instructions in github).

My Linux box installed VSCode with these commands given by linux @code.visualstudio.com:

```
$ sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
$ sudo sh -c 'echo -e "[code]\nname=Visual Studio Code\nbaseurl=https://packages.microsoft.com/yumrepos/vsco
$ dnf -y install code
```

15.1.2 Adding Julia to Jupyter notebook or Jupyter lab

Once you have installed Julia, install the `IJulia` package, From Julia console,

[62] AVX stands for Advanced Vector Extensions, functions starting with `__mm256_` and vector variables starting with `_m256_`. Microsoft Teams uses AVX2 instructions to create a blurred or custom background behind video chat participants and for background noise suppression. AVX @ Wikipedia.

```
julia> using Pkg
julia> Pkg.add("IJulia")
```

and start the Jupyter Notebook on the command line,

```
  $ jupyter notebook
```

15.2 Calling Julia from Python

The project pyjulia @github provides Julia calls from Python,

```
  $ python -m pip install julia
```

and in Python console

```
>>> import julia
>>> julia.install()
```

with some testing,

```
>>> from julia import Base
>>> Base.sin(90)
0.8939966636005579
>>> Base.sind(90)
1.0
```

Remark: This may suggest that you run first

```
>>> from julia.api import Julia
>>> jl = Julia(compiled_modules=False)
```

That worked with Julia 1.8.0 and Python 3.10. Sometimes Python packages can't keep up with Julia updates.

15.3 Julia: language highlights

- Static typing.
  If you have a coded a function $f(x)$ and a function call $f(5.5)$, Julia will compile a version specialized and optimized for `Float64` type of arguments. C/C++ can do optimization only at compile time, just like Numba and JAX. The strength of Julia is JIT, which compiles *and* optimizes code in run time.
- Easy to install - for C++ you'd need the whole toolchain.
- Multiple dispatch
- Metaprogramming support, from Lisp
- LaTex-style Greek letters
- Distributed computing in the language itself: `using Distributed`
- Static arrays are fast: `using StaticArrays`
- Short examples: julia-by-example
- A whole book full of Julia by Ben Lauwens: ThinkJulia: How to Think Like a Computer Scientist

Some Julia ideas are familiar from Python, like automatic closing a file; `do block` corresponds to Python context,

```
open("output.txt", "w") do outfile
    write(outfile, data)
end
```

Julia introduces `tasks` and `channels`. Similarly to coroutines in Python, tasks are executed in a single thread. The venerable Fibonacci sequence could be [63]

fibo.jl

```
function fib(c::Channel)
    a = 0
    b = 1
    put!(c, a)
    while true
        put!(c, b)
        (a, b) = (b, a+b)
    end
end
```

Usage:

```
fib_gen = Channel(fib); # create a task
for i in 1:100
    println(i," ",take!(fib_gen))
end
# outputs Fibonacci numbers, until the 94th number is negative: too large integer.
```

[63] Example by A. Downey and B. Lauwens. ThinkJulia.jl.

The channel `fib` is waking up on demand. The call `take!(fib_gen)` takes the next number from the channel, where it was put by `put!(c,a)` and later by `put!(b,a)`. The values are updated with `(a,b) = (b,a+b)`. The Julianic way is the Pythonic way.

15.4 Julia command prompt

Greek letters look nice,

```
julia> \beta #  + press tab
julia> \pi   # + press tab  (prints greek pi and = 3.1415926535897...)
```

Trying to set $\pi = 1$ fails, `ERROR: cannot assign a value to variable MathConstants.π from module Main`

> You can copy-paste Julia console codes, Julia automatically strips away the leading prompt `julia>`. Same as `ipython` strips `>>>`'s.

Rational and complex number are there,

```
julia> a = 1//2         # typeof(a): Rational{Int64}; dig with numerator(a), denominator(a)
julia> a = 1+3im        # typeof(a): Complex{Int64}
julia> a = complex(1,3) # same
```

Variable typing needs some knowledge of Julia types, a convenient way is to ask about them in the console,

```
julia> supertype(Float64)  # AbstractFloat
julia> subtypes(Integer)   # 3-element Array{Any}:  Bool  Signed  Unsigned
julia> typeof(1.0)         # Float64
julia> typeof(1)           # Int64
```

Handy operations:

```
julia> a = 30
julia> b = 20
julia> minmax(a,b)   # tuple (20,30)
julia> c = 10
julia> muladd(a,b,c) # a*b+c, 610
julia> 3 + true      #  4
julia> x = 1.0
julia> sinpi(x)      # 0.0      ; sin(pi*x), but more accurate
julia> sin(pi*x)     # 1.2246467991473532e-16
```

I'm not sure if `muladd(a,b,c)` and `a*b+c` compile to the same assembly code, it's up to the underlying JIT compiler LLVM. Fused multiply-add (FMA) instructions are high performance.[64]

15.5 Julia arrays, matrices, references, and copies

```
julia> a = Array{Float64}(undef,2,2) # 2x2 Matrix Float64
julia> b = similar(a,Int)            # array similar to a, but integers
```

```
julia> a = [1.0 2.0;2.5 3.5]
julia> b = a           # b is a reference to a
julia> b[1,2] = 0      # changes both b and a
julia> c=copy(a)       # c and a are separate
julia> a = [1,2,3,4]   # 4-element Vector{Int64}
```

[64] Interested in digging assembly codes produced by fortran, C or Julia? Take a look at the blog post by Hendrik Ranocha Optimizing Trixi.

```
julia> b = [1;2;3;4]    # 4-element Vector{Int64}, same as [1,2,3,4]
julia> c = [1 2;3 4]    # 2x2 Matrix{Int64}
julia> d = [1 2 3]      # 1x3 Matrix{Int64}
julia> e = [3 4 5]'     # 3x1 adjoint(::Matrix{Int64}) with eltype Int64
julia> d*d'             # 1x1 Matrix{Int64}
```

Vector can be created from range using `collect`, [65]

```
julia> a = collect(1:2:10) # 5-element Vector{Int64}: 1 3 5 7 9
```

Common are also

```
julia> a=zeros(2,2)
julia> a=ones(2,2)
julia> a=rand(2,2)      # random elements in range [0,1)
julia> a=randn(2,2)     # gaussian random elements
julia> b=repeat(a,2,2) # a copied around twice in both directions
```

15.6 Julia broadcasting

Julia broadcasting rules were designed with NumPy broadcasting in mind, although JIT speed is not as dependable on vectorization. For example

```
julia> a=rand(4,5);          # 4x5 Matrix
julia> b=rand(4,1,5);        # 4x1x5 Array
```

[65] Some type names have changed, the now 5-element VectorInt64 used to be called 5-element ArrayInt64,1.

```
julia> c=broadcast(*,a,b)    # 4x5x5 Array, same as broadcast(*,b,a)
julia> c=a.*b                # 4x5x5 Array, same as broadcast(*,a,b)
```

Many users just love the dot notation. NumPy would broadcast arrays like this:

```
python> import numpy as np
python> a=np.random.random((4,5))       # 4x5 Array
python> b=np.random.random((4,1,5))     # 4x1x5 Array
python> a*b                             # 4x4x5 Array
```

The reason for these "discrepancies" (e.g. 4x5x5 vs. 4x4x5) is that

multidimensional arrays in Julia are in column-major order

meaning the left-most index changes fastest in moving from an element to the next one in memory. In contrast,

multidimensional arrays in NumPy are in row-major order

where the right-most index changes fastest. Therefore, a 4x5 array in Python "means" a 5x4 array in Julia, and broadcasting reflects this.

15.7 Julia array loop

These are the recommended ways

```
for a in A
    # Do something with the element a
end

for i in eachindex(A)
```

```
    # Do something with i and/or A[i]
end
```

15.8 Julia Automatic Differentiation (AD)

Automatic differentiation was already meantioned in discussing backpropagation in neural networks. Now we let Julia do the work, see the manual pages juliadiff and ForwardDiff.jl. Remember that AD computes derivatives with machine accuracy, something finite difference methods can't do. The code `aziz-diff.jl` reflects this fact.

aziz-diff.jl

```
# install package:
#    julia -e 'using Pkg; Pkg.add("ForwardDiff"); Pkg.add("Plots"); Pkg.add("PyPlot")'
# run from Julia console:
# include("aziz-diff.jl")

using ForwardDiff: derivative
using Printf

# definition of the function in file aziz.jl in current directory
push!(LOAD_PATH,pwd()) # add current directory to module search path
using Aziz
#  alternative: you could
# include("aziz.jl")
#  but then you'll need to use Aziz.aziz everywhere, unless you define aziz = Aziz.aziz

r  = [0.01:0.5:10.0;]
V  = aziz.(r)
Vp = derivative.(aziz, r);

@printf("%15s %15s %30s\n","r","V(r)","V'(r)")
for (a,b,c) in zip(r,V,Vp)
    @printf("%15.5f %15.5f %30.20f\n",a,b,c)
end

r0 = 7.61
println("\ncomparing numerical der=(f(r+h)-f(r))/h and present AD result at r=$r0:\n")
Vp0 = derivative.(aziz, r0);
h = 0.1
a0 = aziz(r0)
while h > 1.e-14
    global h
    der = (aziz(r0+h)-a0)/h
    @printf("h=%10.3e  der=%25.20f   AD = %25.20f   der-AD=%25.20f\n", h, der, Vp0, der-Vp0)
    h = 0.1*h
end

println("plotting...")
using Plots
pyplot() # pyplot backend
# more points for plotting
r  = [0.01:0.01:10.0;]
V  = aziz.(r)
Vp = derivative.(aziz, r);
p = plot(r,V,label="V(r)",ylims=(-20,20))
p = plot!(r,Vp,label="V'(r)") # update previous plot

# just to make the plot appear long enough to see it
display(p)
readline()
```

15.9 Julia Differential Equations

This is a tutorial solution for the Lorentz equations, Julia: solving a set of differential equations

diffeq.jl

```
#http://docs.juliadiffeq.org/latest/tutorials/ode_example.html
#Example-2:-Solving-Systems-of-Equations-1

using DifferentialEquations: ODEProblem, solve
function lorenz(du,u,p,t)
 du[1] = 10.0*(u[2]-u[1])
 du[2] = u[1]*(28.0-u[3]) - u[2]
 du[3] = u[1]*u[2] - (8/3)*u[3]
end

u0 = [1.0;0.0;0.0]
tspan = (0.0,1000.0)
println("solving...")
prob = ODEProblem(lorenz,u0,tspan)
sol = solve(prob)

println("plotting...")
using Plots
pyplot() # pyplot backend, looks nicer
plot(sol,idxs=(1,2,3),linewidth = 0.05)  # old version had deprecated vars=...
```

15.10 Julia StaticArrays

See StaticArrays.jl. Static arrays will speed up small array calculations, up to about 100 elements long. There the benefit can be quite noticeable.

15.11 Julia Macros

Julia macros offer a LISP-style runtime interpreter, much more than #`define` in C++.

```
using BenchmarkTools
function f(x)
  a = 0
  for i in 1:1000
     a+=sqrt(float(i*x))
  end
end
@btime f(1.111);         # timing and memory usage
@code_native f(1.111);   # generated code in assembly, good for spotting speed traps
```

Both `@btime` and `@time` outputs are quite readable.

```
julia> @btime rand(10^6);
# 1.089 ms (2 allocations: 7.63 MiB)
julia> @time rand(10^6);
#0.001782 seconds (7 allocations: 7.630 MiB)
```

```
>julia> @which 2 + 2
# +(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8,
# UInt128, UInt16, UInt32, UInt64, UInt8} in Base at int.jl:87
julia> @which 1.0*3.4
# *(x::Float64, y::Float64) in Base at float.jl:332
```

These just tell what method would be used and where the source is (line 87 in `int.jl`). Here `T` is the type, apparently any kind of integer.

15.12 Julia Metaprogramming

See metaprogramming. A polynomial function could be

poly1.jl

```
# evaluates polynomial
# p = sum_{i=1}^n a_i x^i
function poly1(x, a...)  # ellipsis arbitrary length list
    p = zero(x) # pick type from x
    for i in eachindex(a)
        p += a[i]*x^i
        #println(a[i],"*",x,"^",i) # just to see what we sum up
    end
    return p
end
# Example:
# poly1(2.0,1,2,3)  = 1*(2.0)^1 + 2*(2.0)^2 + 3*(2.0)^3
#                   = 34.0
# poly1(2.0,-1,0,1,2,3) = -1*(2.0)^1 + 0*(2.0)^2 + 1*(2.0)^3 + 2*(2.0)^4 + 3*(2.0)^5
#                       = 134.0
```

Using Horner's method it's evaluated as

poly2.jl

```
# evaluates polynomial
# p = sum_{i=1}^n a_i x^i
# using Horner's method, see https://en.wikipedia.org/wiki/Horner%27s_method
#
function poly2(x, a...)
    p = zero(x)
    for ai in reverse(a)
        p = muladd(x,p,ai) # same as  p = x*p + ai
    end
    return x*p
end
# Example:
# p(x) = a_1*x^1 + a_2*x^2 + a_3*x^3
#      = x*(a_1 + x*(a_2 + x*a_3))
# poly2(2.0,1,2,3) = 1*(2.0)^1 + 2*(2.0)^2 + 3*(2.0)^3
#                  = 2.0*(1 + 2.0*(2 + 2.0*3))
#                  = 34
```

Writing the Horner's method as a macro expression,

poly3.jl

```
# evaluates polynomial
# p = sum_{i=1}^n a_i x^i
# using Horner's method, see https://en.wikipedia.org/wiki/Horner%27s_method
#
macro poly3(x, p...)
    ex = esc(p[end])
    #println(ex)
    for pi in reverse(p[1:end-1])
        ex = :(muladd(xx, $ex, $(esc(pi))))
        #println(ex)
    end
    ex = :(xx*$ex)
    Expr(:block, :(xx=$x), ex)
end

# Example:
# p(x) = a_1*x^1 + a_2*x^2 + a_3*x^3
#      = x*(x*(x*a_3+a_2)+a_1)
#      = x*muladd(x,muladd(x,a_3,a_2),a_1)
```

This leaves plenty of questions unanswered, but worry not. Without going to syntactic details, you can see how this works. First uncomment the `println` lines and run

```
julia> include("poly3.jl")
julia> @poly3(x,a_1,a_2,a_3)
```

Yes, that's right: don't give any values to `x`, `a_1`, `a_2`, and `a_3`! This will end up in a an error message, but only after printing what the `expression ex` is at every step. From that it's easier to see what the macro is doing. Give it a try! A great way to debug short metaprograms - and it works also in C++ template metaprogramming. More to Julia style, you can expand the macro to see what it does,

```
julia> @macroexpand @poly3(x,a1,a2,a3)
quote
    var"#525#xx" = Main.x
    var"#525#xx" * Main.muladd(var"#525#xx", Main.muladd(var"#525#xx", a3, a2), a1)
end
```

This is more readable than the `println` method. Now you have tools to assertain that the macro expression works as expected.

Time to give the polynomials a benchmark run,

```
julia> using BenchmarkTools
julia> include("poly1.jl")
julia> include("poly2.jl")
julia> include("poly3.jl")
julia> @btime poly1(2.0,1,2,3,4,5)
# 1.402 ns (0 allocations: 0 bytes)
julia> @btime poly2(2.0,1,2,3,4,5)
# 1.402 ns (0 allocations: 0 bytes)
julia> @btime @poly3(2.0,1,2,3,4,5)
# 0.020 ns (0 allocations: 0 bytes)
```

In my machine, the `poly3.jl` version was 70 times faster than the other two. The one single faster execution saves you less than a nanosecond, so unless you do it 10^9 times you won't see the difference. Consider this as a fine example of fast code without any real-life impact.

You can name the macro,

```
julia> horner5(x) =  @poly3(x,1,2,3,4,5)
julia> horner5(2.0)
#258.0
```

Later we'll see how such metaprograms are handled in C++. There are already quite a few Julia applications around. One of them is `Trixi.jl` @github, an expandable numerical simulation library. Benchmarks of real world problems agains the HPC Fortran code Fluxo show how good Julia can be. The JuliaCon 2021 video gives a nice overview, see Trixi @JuliaCon2021.

GPU programming benefits a lot from metaprogramming capabilities, see CUDA @juliahub.com.

15.13 Multiple dispatch

If you call Python method `obj.calculate(args)`, the interpreter chooses `calculate()` to execute based on the type of the object `obj`; this is ***single dispatch***. In the code

```
class MyClass:
      def calculate(self, args):
          ...

obj = MyClass()
obj.calculate(args)
```

the choice of dispatching `calculate()` is based on the first argument `self` - hence single dispatch. Some binary operations are related to magic methods, such has + is `__add__()` and * is `__mul__()`; these are ***double dispatch***. In a math expression $x + y$ it hardly makes sense to attach the operation of addition to x, neither does it naturally belong to y.

What, then, is multiple dispatch? Multiple dispatch means that a method is chosen depending on the type and number of all the arguments. For example,

```
julia> f(x::Int64, y::Int64) = 2x + y
f (generic function with 1 method)

julia> f(x::Float64, y::Int64) = 3x + 2y
f (generic function with 2 methods)

julia> @which f(1,2)
f(a::Int64, b::Int64) in Main at REPL[1]:1

julia> @which f(1.0,2)
f(a::Float64, b::Int64) in Main at REPL[1]:1
```

```
julia> methods(f)
# 2 methods for generic function "f":
[1] f(x::Int64, y::Int64) in Main at REPL[1]:1
[2] f(x::Float64, y::Int64) in Main at REPL[2]:1
```

Try `methods(+)` and you get 207 methods, all specialized to adding different kinds of data types. It only makes sense because addition makes sense for many types of data.

Many kinds of geometrical objects have an area. In Julia it could be computed like this:

julia_area_dispacth.jl

```
abstract type Shape end

struct Circle
    radius::Float64
end

struct Rectangle
    width::Float64
    height::Float64
end

# Define methods to calculate the area of different shapes
area(shape::Circle) =  * shape.radius^2
area(shape::Rectangle) = shape.width * shape.height

# Tests
circle = Circle(3.0)
rectangle = Rectangle(4.0, 5.0)

println("Area of the circle: ", area(circle))
println("Area of the rectangle: ", area(rectangle))
```

In C methods are chosen based on their name, hence in GSL (Gnu Scientific Library) you meet FFT methods such as

```
int gsl_fft_complex_radix2_forward(gsl_complex_packed_array data, size_t stride, size_t n)
int gsl_fft_real_radix2_transform (double data[], size_t stride, size_t n)
```

so you have make up a distinct function name for different situations. C++ has function overloading, which means you can write a single function

```
int fft_forward(args)
```

and let the compiler deduce what method to apply by looking at the arguments. However, C++ function overloading is done at the *compile time*.

Despite the fact that Julia does compilation, it does it in run-time, and, as Stefan Karpinsky points out,[66] a code like this is possible:

[66] See answers in link @discourse.julialang.org. There are 133 of them and growing, so relax, it's difficult to explain what multiple dispatch is, but it's simple to just use it.

julia_dispatch.jl

```
abstract type A end
struct B <: A end
struct C <: A end

f(a::A) = "it's an A"
f(b::B) = "it's a B"
f(c::C) = "it's a C"

g(a::A) = f(a)

println(g(B()))
println(g(C()))

# output:
# it's a B
# it's a C
```

The syntax `B <: A` means `B` is a subtype of `A`. The result underlines the fact that `a::A` doesn't mean `a` is type `A`, it's a hint to the compiler that variable `a` is expected to be of of type `A` or of any of `A`'s subtypes. A variable carries with it its type. In static function overloading you would *always* execute `f(a::A)`. In Julia, run-time typing is the thing, while compile-time typing doesn't happen at all. Note that you could write as well

```
f(a::A) = "it's an A"
f(a::B) = "it's a B"
f(a::C) = "it's a C"
```

As usual, there's a way to write extra code in most languages to include multiple dispatch, for example in Python as modules `multipledispatch` or `multimethod`, see example multiple_dispatch.py. These are Python library modules, whereas Julia has

multiple dispatch built in to the very core of the language. In Python `multipledispatch` or `multimethod` dispatch decisions are made dynamically at runtime, which can be less efficient compared to Julia's static dispatch. Also Python `multipledispatch` or `multimethod` make the dispatch decision based on the first argument, which limits the possibilities compared to Julia, which makes the dispatch decision based on all arguments.

The Expression Problem

The Expression Problem @wikipedia is a question about ***code reuse*** in a programming language. One would like to

- add new data types to existing methods
- add new methods to existing data types

without modifying or duplicating existing code. Leave out the word "data" if you wish.

Let's write a table with data types as columns and methods as rows,

	data type1	data type2
method1()	calc_11	calc_12
method2()	calc_21	calc_22
method3()	calc_31	calc_32

Here `calc_xx` is existing, well-tested code base you don't want to edit. Adding a new type means adding a column, adding a new method means adding a row.

Adding a type and a method in Python

In object-oriented languages, such as Python and C++, you could have classes with methods. In Python, you could have an abstract base class `Calc`

```
from abc import ABC, abstractmethod

class Calc(ABC):
    pass
class Type1(Calc):
    def method1(self):
        print('calc_11')
    def method2(self):
       print('calc_12')
    def method3(self):
       print('calc_13')
class Type2(Calc):
    def method1(self):
        print('calc_21')
    def method2(self):
        print('calc_22')
    def method3(self):
        print('calc_23')
```

Adding a new type is simple,

```
class Type3(Calc):
    def method1(self):
        print('calc_31')
    def method2(self):
        print('calc_32')
    def method3(self):
        print('calc_33')
```

How about adding a new `method4()`? You could inherit types and extend them,

```
class ExtType1(Type1):
    def method4(self):
        print('calc_14')
class ExtType2(Type2):
    def method4(self):
        print('calc_24')
```

We solved the expression problem after a fashion. The price is high, because from this point on we'd have to keep in mind that `ExtType1` supercedes `Type1`. There's no method `method4()` in `Type1` objects,

```
obj1 = Type1()
ob1.method4() # FAILS
```

A better solution would be to add the new method to the existing class by defining methods and injecting them to classes `Type1` and `Type2`,

```
def method4_for_1(self):
    print('calc_14')
def method4_for_2(self):
    print('calc_24')

Type1.method4=method4_for_1
Type2.method4=method4_for_2
```

and now also `ob1.method4()` works, even if `ob1=Type1()` was created before adding `method4()`! Some call this process affectionately *monkey patching*. You could also overwrite any of the old methods, but such a liberal attitude to overruling previous decisions can be hazardous. Injecting a new method to an existing class has the price that relevant code may be scattered in multiple files.

Adding a type and a method in C++

C++ is statically typed, so adding a new method to an existing class is harder. It can be done to classes that allow it using the so-called visitor design pattern.[67] Visitor pattern is double dispatch, the method to execute depents on what object defines the visitor and on the object that calls the visitor.

Adding a type and a method in Julia

Julia multiple dispatch solves the expression problem. Julia functions have methods,

```
julia> f(x::Int) = 2x
f (generic function with 1 method)
julia> f(x::Float64) = 3x
f (generic function with 2 methods)
```

Adding a new type is simple,

```
julia> f(x::String) println(``a string it is'')
```

and so is adding a new method to existing type (`String` in this example),

```
julia> g(x::String) = println("g-string")
g (generic function with 1 method)
```

For example, measurements have usually error bars, $x = 1.45 \pm 0.01$, and the package `Measurement` adds that data type. You can propagate data with error bars through an ODE solver (`DifferentialEquations`) and plot it (`Plots`). See julia-errorbars.jl. It just works.

> Julia shines in code reusability and there's less need for boilerplate code.
> To summarize, in Julia you have a language for very generic code.

[67] Visitor patterns can be coded in Python as well, but they aren't needed: see injecting a method to existing class.

16 C++

16.1 A brief history of C++

- 70's and 80's:
  - FORTRAN-77 fixed-sized tables annoy programmers
  - Pascal becomes popular after Borland introduces a lightning fast compiler
  - Dennis Ritchie creates C language with dynamic tables

- From 1979 on: Bjarne Stroustrup creates C++ as a sort of "improved C" - but is it? The jury is still out. (The name "C++" was coined in 1983)

- Characteristic to C++:
  - strong typing of variables (supposedly) reduces programming mistakes
  - Object-oriented programming possible, though not forced
  data belonging together is collected together to an "object", that can be moved around as one big chunk.
  - Compiled language, for speed.

- Early 90's: No C++ standard, compiler manufacturers made their own decisions. C++ portability: None.

- 1998 : C++ Standard! A second birthday of C++
  -An extensive standard library

- 2011 : C++11 Standard (working name was C++0x)
  - Easier ways to do very common tasks
  - C++ books written before about 2010 are better left to collect dust

A timeline of C++ can be found on the page ISO C++ committee.

16.2 About these C++ lectures

First, I apologize that these C++ notes contain so many topics unrelated to fast numerics. Maybe some ideas will help you get a job done, and that's a good start. C++ is a multi-purpose language and was never intended to specialize on numerics, that's why it's so easy to get lost in the jungle and find yourself tasting interesting mangos while looking for some bamboo.

I'm going to introduce some C++11 and C++17 features and it's a whole lot different to what old C++ used to be. C++20 is now fully supported, and I should update these notes (again). ***The truth is, I'm falling behind in C++.*** As s general goal, I try to avoid (naked) pointers and use references instead. I hope this course gives you self-confidence to write your own C++ program that solves a numerical task using a chosen library and read and modify C++ programs.

I'm always impressed by the fact that the C++ `std` namespace zoo has over 1800 `std::thing` animals.[68] According to news @bbc.com,

> Prof Webb found that people who have been studying languages in a traditional setting - say French in Britain or English in Japan - often struggle to learn more than 2,000 to 3,000 words, even after years of study.
>
> In fact, a study in Taiwan showed that after nine years of learning a foreign language half of the students failed to learn the most frequently-used 1,000 words.

I take this as a proof that if you can learn the C++ `std` namespace, you can learn any language!

[68] See many of them in symbol_index @cppreference.com.

16.3 Easy tasks

Yilun Zhang, Learning data science
70 upvotes by Chiraz Ben A, Gunnar Oehlschlägel, Jake Merchant, (more)

Hello world in:

Python:

```
print "Hello, world!"
```

Java:

```
public class HelloWorld {
    public static void main(String[] args) {
System.out.println("Hello, world!");    }
}
```

C++:

```
#include <iostream>
 int main(){    std::cout << "Hello, world!\n";
}
```

C:

```
#include <stdio.h>
int main(void){
printf("Hello, world!\n");
}
```

JS:

```
//myfile.js
console.log("Hello, World!");

***command line***
node myfile.js
```

MySQL:

```
DELIMITER $$CREATE FUNCTION hello_world() RETURNS TEXT COMMENT
DELIMITER ;
 SELECT hello_world();
```

Python 3:

```
print('Hello, world!')
```

Python + NumPy + SciPy
is a very powerfull combination

Eigenvalues and eigenvectors of a random 5x5 matrix:

```
import numpy as np
import scipy.linalg as la
A = np.random.randint(0, 10, (5, 5))
print("matrix A = \n",A)
e_vals, e_vecs = la.eig(A)
print ("eigenvalues : \n",e_vals)
print ("eigenvectors : \n",e_vecs)
```

how-can-scipy-be-fast-if-it-is-written-in-an-interpreted-language-like-python

16.4 Online sources for C++ programmers

- In Finnish: www.ohjelmointiputka.net/oppaat.php
  Kattaa myös Python, C, ja PHP kielet
  (*Mureakuha* kaatui kilpailuun. Sivustojen suhteista kirjoitettiin mehevästi Putkan 10v sivuilla)
- www.cplusplus.com
- www.learncpp.com
- www.greenteapress.com/thinkcpp/ (Book, 2012)
- Online courses at www.edx.org, free and non-free

Don't read old C++ books. Anything written before 2011 should be taken with a grain of salt.
Be systematic with your style. When to use capital letters and when to use underscores is up to you. Some coders stick to the so-called Hungarian style, where `m_xxx` signify member function called `xxx`, `m_ixxx` add this to return an `int` *etc.* And some coders call this the idiotic style.

Think carefully what to put in *header files*. About style and good C++ practices, I recommend Bjarne Stroustrups home page www.stroustrup.com. After all, he created the language, so he may have a solid opinion.
Be sceptic about benchmarks. Especially so if the tests were run by the code developers themself. They may have run the benchmarks using an outdated CPU architecture to hide that their code cannot use AVX or FMA, and they don't want to compare with codes that can. Comparisons that show how a library is faster than MKL may have prevented multithreaded execution, because that's where MKL excels. Run your own benchmarks.

16.5 C++ in Matlab or Octave

You can call a compiled C or C++ programs from Matlab using MEX files (Matlab executable files). Matlab can access C/C++ library routines and may work through a numerical bottleneck faster. See instructions in write-cc-mex-files.html.

Octave is a free Matlab-lookalike, with an almost identical syntax but different internals. Octave has interface to several languages, including C++. The compiled files used in Octave are called oct-files, see instruction in oct-files.

16.6 C or C++ in Python 3

See part 1 of these lecture notes: Extending Python 3 with C++, Cython, ctypes.

17 A really brief introduction to C++

Example: Main program

main1.cpp

```
#include <iostream>       // here cout is defined

int main()
{                         // ALL BEGINS
  int i,j;                // integers i and j
  i=5;                    // set values to i and j
  j=20;
  int k=i+j;              // define k on the fly
  std::cout<<"k="<<k<<"\n";    // output to screen
}                         // ALL ENDS
```

`#include <iostream>`	using the standard input/output streams library
`using namespace std;`	make visible all keywords defined in `std` namespace
`int main()`	the main program is a function (and can have arguments), returns an integer

The function `main()` can end with

```
return 0;
```

or

```
return (0);
```

so that the function `main()` gives out the integer 0. Out where? Hmm, to the operating system. Safe to leave out entirely.

17.1 The meaning of `#include <iostream>`

The ***preprosessor directive*** `#include <iostream>` tells the compiler to make available all that's defined in the C++ standard library header `iostream`. [69].

Two ways to include headers:

```
#include<iostream>  // part of standard library
#include "myheader.h" // a home-made header
```

The difference is the header search path, the latter uses the current path first, after that the include path. To see what headers were included, try

```
$ g++ -M code.cpp
```

and to include a `PATH` using

```
$ g++  code.cpp -I PATH
```

[69] Take a peek at what headers exist and what's in them : www.cplusplus.com/reference

You get pretty far with these headers:

- `#include <iostream>`: input and output
  `cout`, `cin`, `cerr`, `clog` - c for screen console.
- `#include <iomanip>`: formatted output; if you like `printf()` forget this.
- `#include <cmath>` (C++) or `#include math.h` (C or C++) : mathematical functions
  `sin, cos, log, sqrt, pow`... x^2 is coded `pow(x,2)`
- `#include <complex>`: complex numbers
- `#include <fstream>`: file handling (also `ofstream`, `ifstream`,...)
- `#include <string>` : character strings
- `#include <random>` : (pseudo)random numbers
- `#include <functional>` : function objects

Three styles to write a `double` precision function:

fun1.cpp

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

fun2.cpp

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

fun3.cpp

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

Here the argument `x` was chosen to be a `double`.

Example: Function defined before `main()`

fun_before.cpp

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

double f(double x) {
  return sin(x)*cos(x*x);
}
int main()
{
  double x,y;
  x=2.333;
  y=f(x);
  std::cout<<"f("<<x<<")="<<y<<"\n";
}
```

```
f(2.333)=0.482627
```

The compiler knows all it needs to know about the function at the point when `f()` is met in `main()`.

Example: Function defined after `main()`

fun_after.cpp

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

double f(double);
int main()
{
  double x,y;
  x=2.333;
  y=f(x);
  std::cout<<"f("<<x<<")="<<y<<"\n";
}
double f(double x)
{
  return sin(x)*cos(x*x);
}
```

When the compiler meets `f()` inside `main()`, it wants to know what it takes as arguments and what it returns. Therefore you need to provide it with a sneak peak

```
double f(double);
```

called **prototype** or **function declaration**; You have to give the prototype also if the function is defined in ***another file***. Failing to declare a function results the error message `fun_after.cpp:9: error: 'f' was not declared in this scope`. Here we encounter the concept of `scope`.

C++11 standard made an addition, the return type of a function can be declared like this:

```
auto f(double) -> double; // trailing return type
```

Pretty useless in this example. You may appreciate how this helps the compiler, if you think of how the return types of class methods become clear only after going through all the types of the arguments. There is another use of `->`: `foo->bar` means `(*foo).bar`.

17.2 Scope

The scope tells the range of visibility. In which parts of the program is an object defined, and also who may know the thing even exists[70]. The basic ideas are global objects (visible everywhere), and local objects (visible in a limited scope). The scope limits are curly brackets, boundaries of a ***block***:

```
double z; // I'm visible to anyone - mess me up at will
int main()
{
 double x; // I'm visible between {...}
 ...
}
double function f(double k)
{
        double x; // I'm not the same x as in main()!
        z=10;     // but I'm the one and only global z
{
```

A local loop variable exist only in the loop

[70] Sounds modern, invented in the 50's.

```
...
unsigned j = 5;
for(unsigned i=0;i<10;++i){if(i==j) break;}
cout<<"i=j when i="<<i<<endl;   // WON'T WORK
```

When the program leaves the loop, the variable `i` ceases to exist. The last line is not allowed.

17.3 Simple file operations

C++ idea: create a *stream*, either for writing (`ofstream`) or for reading (`ifstream`)

`o` as out, `i` as in, and `f` as file.

To a stream we stuff things using the operator << and read with the operator >>.
The console output `cout` and the console input `cin` are streams.

Example: Write to a file

fileio.cpp

```
#include <iostream>
#include <fstream>

int main () {
  std::ofstream myfile("output");
  myfile << "Text to file output"<<"\n";
  myfile.close();
}
```

This declared and opened a stream in the one line with `ofstream`

```
ofstream myfile("output");
```

The file `output` contains now the text "Text to file output", and the file is closed with

```
myfile.close();
```

The object (instance) `myfile` of the class `ofstream` has a method (member function) `close()`.

18 C++ Classes

A class is a collection of data and methods to manipulate the data that somehow belong together. It's just an abstract type, a bit like `int` only tells what an integer is. You need an ***object (same as: instance of a class)***, a manifestation of the class. A class `Bird` may have an object `crow`, which tells the compiler that "a crow is a bird".

18.1 Private and public data, methods

The method `comp_energy()` is conveniently a ***class method, a member function***.

```
class ClassName{
   // private things
public:
   // public things
   // and member functions, used to manipulate and show contents of private date
};  // this line may already contain objects of this class
```

For example,

```
class SystemState{
  double E; // energy
public:
  void comp_energy(); // compute energy
  double energy()const {return E;};    // read energy
} state1,state2;
```

This defines the class and immediately constructs two objects `state1` and `state2`. You can do this later as well, `SystemState state1;`

To work, the `class SystemState` still needs the definition of the method `comp_energy()`. If it's short, it can be defined *in situ*, on the spot in side the class (as we did for `energy()`), but you are free to define it later:

```
// member function of class SystemState
void SystemState::comp_energy(){
  E=3.12515e-10; // just anything for testing
}
```

The variable `E` used by the method is exactly the same `E` as in the object, not just any `double E` ! Now the energy of the state can be computed and read,

```
int main()
{
  state1.comp_energy();        // compute state1 energy
  cout<<state1.energy()<<endl; //   print state1 energy
  return 0;
}
```

This introduced one benefit of a class, ***data encapsulation***, which means that data can be protected from inadvertent changes or hackers. Just make it `private` and give only carefully chosen methods the permit to touch the data. Secondly, upon improving the code you can improve the methods without disrupting the workings of the rest of the code.

A class has properties that can be ***inherited***, meaning another class can inherit the properties and methods of a parent class. Smart base classes can be inherited and you can recycle code. In the C++ language itself a lot of things are inherited, for example `ofstream` header file inherits this chain of mre general purpose headers:

`ios_base` ← `ios` ← `ostream` ← `ofstream`

On the bottom there is a general-purpose class `ios_base`.

Example: Define a Car class and instantiate a few cars

car.cpp

```
// How to create a Car class with member function get_color()
#include <iostream>

class Car
{
  std::string color;  // private member
public:
  std::string get_color() const {return color;};
  Car(std::string col) {color=col;}     // constructor
  ~Car(){std::cout<<"Car demolished\n";}  // destructor
};

int main()
{
  Car cadillac("black_and_white");  // create a Car with a color
  Car ferrari("red");
  std::cout<<"cadillac is ";
  std::cout<<cadillac.get_color()<<"\n"; // get cadillac's color
  std::cout<<"ferrari is ";
  std::cout<<ferrari.get_color()<<"\n";
  Car buick(cadillac);              // copy cadillac to buick: all properties
  std::cout<<"buick is ";
  std::cout<<buick.get_color()<<"\n";
}
```

```
cadillac is black_and_white
ferrari is red
buick is black_and_white
Car demolished
Car demolished
Car demolished
```

`color` is a member of `Car`; It's not declared `public`, so it's `private`. `get_color()` is *only* means to read the car color,

```
  cout<<buick.color<<endl; // error
```

```
car.cpp:5:10:  error:  'std::string Car::color' is private
```

Notice, how the class `Car` has a function with the very same name as the class itself:

```
Car()
```

is a class constructor, used to create an object (a Car) and to give it some properties. This constructor is used if and only if the color is given in parenthesis. The compiler looks how we try to create a Car and find a match from the list of possible ways to constructor a Car.

Object is a manifestation of a class, a thing that has all the properties defined in the class:

```
Car lada("black"); // Car is a class, lada is an object
```

Meaning "lada is a Car and it's black". When the compiler comes to this line, it fetches a constructor in the class `Car` answering the needs, a function that looks like this:

```
Car(string)
```

Such a function exists, so the compiler will create a black lada. It may become as a surprise, that constructor of a colorless Car fails:

```
Car volvo;
```

fails, because `Car` has no constructor

```
Car()
```

This would be the ***default constructor***, one made by the compiler if no other constructor is specified - after all, a class must have *some* constructor to be useful at all. The only thing a default constructor does, is to reserve some space for an object, nothing else. C++ language specifies that

> If you give a constructor for a class, the default constructor vanishes

This is happened in the example: there was no longer any colorless Car constructor.

Objects can be copied,

```
Car buick(cadillac);
```

and `buick` picks the color from the `cadillac`.
Wait a moment, what function did the copying? It was the ***copy constructor***, provided by the compiler automatically. As you guessed, there is a method called ***destructor***, which deletes an object. It has the tilde sign,

```
~Car(){cout<<"Car demolished\n";}
```

Compilers create this automatically, then it's called a ***default destructor***. You can also provide one yourself, and include a clear message when an object is destroyed.

To summarize, a class has a set of building methods:
(add the prefix "default" if you let the compiler create a method)

constructor	how an object would be created
copy constructor	how an object is copied (resulting two similar objects with different name)
copy assignment	(=) makes a copy from right to left
move constructor	tells how the information of an object is transfered to another the object essentially steals the resources of the other object, leaving it in default state
move assignment	pilfer the resources on the right and gives them to the left
destructor	tells how an object should be destroyed

C++ has a ***rule of five***, saying that if you **need** to write your own destructor, chances are you need to provide all five,

destructor
copy constructor
move constructor
copy assignment operator
move assignment operator

because none of the default version generated by the compiler meet the needs of your class.

[70] A better constructor would use an ***initialization list***.

18.2 Member function qualifiers `const` and `noexcept`

C++ has "compiler directives", ***qualifiers*** that facilitate code optimization and help to catch object misusage during compile time.

```
Class MyClass {
  ::
public:
  void member1() const; // does not change any data member of the class (except mutable)
  void member2() noexcept; // does not throw any exceptions
  void memder3() const noexcept; // neither throws nor changes data members (unless mutable)
  :::
}
```

Here "throw" is a mechanism of making exceptions, errors or warnings, later discussed in section 33. The qualifier `noexcept` potentially leads to faster code (section 27). If a function cannot fail it's safe to use `noexcept`.

18.3 Example of a data structure

If you need to write serious simulation code with coordinates and distances, use a library that provides vectors and fast linear algebra. This example is for light-duty use only.

Example: A class for a 3D point and distance between points

class_points.cpp

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

namespace co{
  // Class for 3D coordinates
  class Point{
  public:
    double x,y,z;
  };

  double distance(const Point &, const Point &);
}
int main()
{
  co::Point point1 = {0.0,0.0,0.0}, point2={1.0,2.0,1.0};
  // move point1 a bit
  point1.x = 1.0;  // public data member accessed directly
  std::cout<<"distance = "<< co::distance(point1, point2) <<"\n";
}
// Euclidian distance between two points
double co::distance(const co::Point &c1, const co::Point &c2){
    return ( sqrt(pow(c1.x-c2.x,2) + pow(c1.y-c2.y,2) + pow(c1.z-c2.z,2)  ) );
}
```

The function `distance()` has a very common name, therefore it was embedded to the ***namespace*** `co`. This makes clear that `co::distance()` clearly calls that particular function. Namespaces were invented to prevent name conflicts, in a larger code base there could be another `distance()` defined elsewhere.

Before you start writing a simple data structure, check first the C++ Standard Library. If the structure is there use it, don't re-invent the wheel. For example, a pair of practically anything is `std::pair<type1,type2>`.

Example: Creating pairs `std::pair` using `std::make_pair`

pairs_of_anything.cpp

```
// using std::pair to make pairs of almost anything
#include <iostream>
#include <vector>
int main()
{
  std::pair<int,int> iipair;
  iipair= std::make_pair(10,12);
  std::cout<<iipair.first<<" "<<iipair.second<<std::endl;

  std::pair<int,double> idpair(1,120.324);
  std::cout<<idpair.first<<" "<<idpair.second<<std::endl;

  // vector of pairs (int,vector)
  std::vector< std::pair<int,std::vector<double>> > ivpairs;
  std::vector<double> v{10.1,20.2,30.3,40.4}; // testing vector
  for(int i=0; i<10;++i)  ivpairs.push_back(std::make_pair(i,v));
  for(auto i:ivpairs) {
    std::cout<<"i="<<i.first<<" vector=";
    for(auto j:i.second) std::cout<<j<<" ";
    std::cout<<std::endl;
  }
}
```

Remember `tuple` from Python? `std::tuple` (tuple @cplusplus.com) is a generalization of `std::pair`.

19 Templates - Generic instructions and algorithms

Templates are one of the best things C++ has to offer. C++ templates do the same as Python's universal functions - and more.[71]

> A template is an abstract model of what to do

C++ libraries use templates to achieve excellent performance and multi-purpose applicability. Some methods are so common, that you hardly think they are templates:

- `std::sort()` is a template to sort arguments - many kinds of arguments, not just numbers
- `std::pair()` is a template to combine two things - many kinds of things
- `std::swap()` is in a template to swap two things of the same type - many kinds of things

The vector contained of integers is `vector<int>`. The angular bracket is for ***template parameters***, they define what type of objects the container holds. `std::vector<int> x` tells the compiler to search the `std` namespace and find a model for a `vector`, and bring one to life - ***instantiate*** - using `int` as data type and call the container `x`.

Basic templates are simple. Suppose you have a function that is written for `double`'s,

```
void f(double x){
  // do something with x
}
```

Then you realize this procedure should work also for `int`'s and `string`'. You can overload f (section 24),

```
void f(int x){
  // do something with x
}
```

```
void f(string x){
  // do something with x
}
```

[71] A series of articles An-Idiots-Guide-to-Cplusplus-Templates give more insight to templates.

This is fine until you realize the algorithm in `f()` is so good it can be applied for many more data types. At this point the function overloading must give way to some kind of "general function overloading", a ***function template***.

```
template <typename T>
void f(T x){
  // do something with x
}
```

Here `T` is the type of `x`. A template is never omnipotent, for example there are type `T` objects that cannot be printed with `std::cout`; upon a failed attempt you will be greeted with a long compiler error message.

For a more serious example, see `basic/simple_template.cpp` and `basic/static_assert.cpp`. The latter shows how to check types in compile time and to detect if a wrong template is used by mistake.

19.1 Variadic functions and templates

Some functions should accept a variable number of arguments. Logically, there should be a ***variadic template*** with any number of template parameters,

```
  template <typename ...Ts>
```

Here `...` is called ***ellipsis***. Variadic templates were designed by Douglas Gregor and Jaakko Järvi. The standard library has the variadic template `std::tuple`.

Remark: There are several attempts to code a type-safe `printf` in C++, for example by Andrei Alexandrescu (see type-safe printf discussion @stackoverflow). Boost has worked on the issue quite a while, see on choices to be made in printf.

Example: Variadic template to sum squares of numbers.

variadic_template.cpp

```cpp
// Any number of arguments to sum_of_squares()
#include <iostream>
#include <cmath>
using std::cout;
using std::pow;

auto sum_of_squares() {return 0; }// end of recursion

template <typename T,typename ...Ts>
auto sum_of_squares(T first, Ts ... rest)
{
  return pow(first,2) + sum_of_squares(rest ...);   // recursive
}

int main()
{
  cout<<sum_of_squares(1,2)<<"\n";
  cout<<sum_of_squares(1,2,3,4,5)<<"\n";
  cout<<sum_of_squares(1.1,2.2,3.3)<<"\n";
  cout<<sum_of_squares(1,2.1,3.1)<<"\n";
}
```

For more examples, see https://en.wikipedia.org/wiki/Variadic_template. Remarkably, there's absolutely no run-time overhead: variadic templates are expanded at compile-time. In contast, in C language variadic functions are resolved at run time.

20 C++ Standard Library

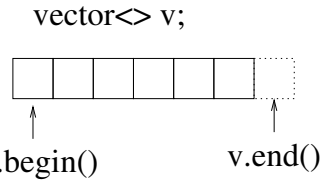
C++ Standard Library comes with all C++ compiler suites. What, then is the Standard Template Library (STL) many people talk about? I rise my hands here. I've figured out this much: The STL existed before the Standard Library, and parts of it seem to be now in the Standard Library. People talk about STL and mean the Standard Library and also talk about the Standard Library when they mean STL. As an ***end user*** of the C++ products, I couldn't care less, and have taken the pragmatic approach to accept it's a library that comes with the compiler suite. I call them both as *the C++ Standard Library*. Period.

On late hours I simply think that

> STL means Standard Template Library (official)
> STL means STandard Library (inofficial, my own idea to save my boiling brain)

The Standard Library contains also the C Standard Library, so you are able to use some C features as well. But not all of C, notably C++ programmers avoid naked pointers to the extreme, and they are never really *needed* in C++. Some parts are templates (multi-purpose models), such as

- ***containers*** are collections of objects
  `vector`, `stack`, `deque`, `list`, `map`, ...
- `iterators` are for going through elements of containers
  `begin()`, `end()`



> `v.begin()` points to the begin of the container `v`
> `v.end()` points to *one step past* the end of the container `v`.

Why one step past the end? It was chosen like that because of cleaner loops:

```
for(auto it=v.begin();it<v.end()) {...}
```

- `algorithms` are frequently needed ways to process data
  `sort`, `find`, `min_element`, `max_element`, `reverse`, ...

Example: Swap two numbers using `std::swap`

std_swap.cpp

```
#include <iostream>
#include <utility>
int main()
{
  using namespace std;
  double a = 5;
  double b = 10;
  cout <<"before "<< a<<" "<<b<<endl;
  swap(a,b);
  cout <<"after  "<< a<<" "<<b<<endl;
}
```

```
before 5 10
after  10 5
```

Someone familiar with C might be concerned about what values `a` and `b` have after the call to `swap`. How are the arguments passed to the function and what comes back?

21 C++ References

I'll start with the so-called "`lvalue` references" - we'll come to that detail later.
The three *basic* ways to pass data to functions are:

- ***Pass by value***

```
void dothis(int);  // function prototype
...
c=15;
dothis(c); // work with number 15, not with c; cannot change contents of c
```

- ***(C++) Pass by reference***

```
void dothat(int&);  //  function prototype
...
c=15;
dothat(c); // work with c, can change contents of c
```

- ***Pass a pointer***

```
void doodd(int *);  // function prototype
...
c=15;
doodd(&c);  // works with the pointer to c; can change c at will
```

By looking at just the function call there is no way to tell if the argument is passed-by-value or passed-by-reference! Only the function prototype (declaration) reveals which it is. In practice you end up guessing whether a function can change the contents of variable `c`: `dothis(c)` doesn't, but `dothat(c)` does!

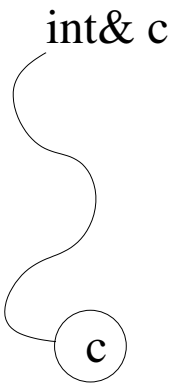
21.1 Why would a reference be safer than a pointer?

Imagine that the memory of a computer is a stack of boxes.
A pointer `int *c` means "take integer c from upmost box". Mistakes can happen:

- The box may have a wrong number or not an integer but a mouse trap (real number)
- The box can be bottomless (contains a pointer to the box below)
- The box doesn't exist (points outside the allowed memory space)

A programmer is let to do all this without the compiler ever noticing any bad deeds. Running the program gives odd results. I once wrote a program that was supposed to do a simple calculation - and it printed "Full Moon tonight".

C++ reference to variable `c`,
`int& c`
is like a cord or a thin rope tied to the integer `c`. There is no place for mistake, the end of the cord always has `c`, because there is no way to detach the cord and tie it to anything else. In other words,a C++ reference cannot be detached from its variable.



Amuse yourself with these almost relevant examples:

Pass by value (C or C++):
Write the numbers 5 and 10 on a piece of paper, show it to you college and tell him/her to copy the numbers to *his/her own piece of paper in reverse order*. Swap failed: your paper still holds 5 and 10, not 10 and 5

Pass by reference (C++):
Write the numbers 5 and 10 on a piece of paper, hand it to you college *without letting go of the paper*. Tell him/her to swap

the numbers on your paper, then pull the piece of paper back. Swap achieved! Robustness: Your college immediately sees if your hand is empty or there are no numbers on the paper.

Pass by pointer (C or C++):
Write the numbers 5 and 10 on a piece of paper and tell your college the paper is in a specific drawer. Tell him/her to swap the numbers written on the paper and to put it back to the drawer, Swap achieved! Robustness: your college picks a wrong piece of paper from the drawer and wonders why you want to swap two telephone numbers.
Remarks:

1. If you have many colleges and/or 10000 numbers, copying them around may take a lot of time and paper.

   Prefer passing references, it's fast and economical.

2. For the `swap` to function properly, you can't pass numbers 5 and 10 by value; that is, copies of the values.
   Two working ways:

   - C++ style: `swap` is called with references, `swap(a,b);` (This is ineffective code due to copying)

     swap_reference.cpp

     ```
     void swap(int& a,int& b){
       int c;
       c=a; a=b; b=c;
     }
     ```

   - C style: `swap` is called with pointers, `swap(&a,&b);`

swap_pointer.cpp

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

One benefit of passing by value is that the original value is safe, only a copy is sent out. In C++ you can give the reference `const` qualifier, to protect it from changes. Caveat: I've heard some libraries ignore this qualifier.

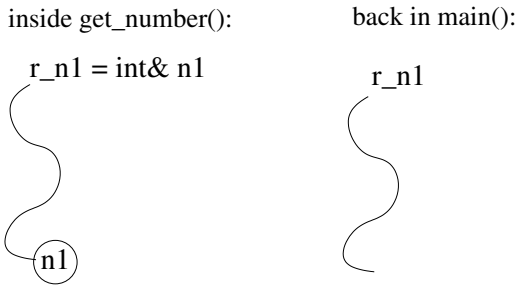
21.1.1 Unsafe references

Alas, a reference is not completely safe. It's still possible to shoot yourself in the leg.
A ***dangling reference*** is one way to (mis)use an address of an object after the object has ceased to exist.

Example: Dangling reference

dangling_reference.cpp

```
// Unsafe code example
// run with valgrind -q a.out
#include<iostream>
int & get_number()
{
  int n1 = 100;
  int & r_n1 = n1;
  return r_n1;
}
int main()
{
  int number = get_number();
  std::cout<<"number is "<<number<<"\n";
}
```



Think where the reference `r_n1` is referring to. The line

```
int& r_n1 = n1;
```

tells `r_n1` is a reference to an integer, which is `n1`. Nothing wrong with that, but `n1` is a ***local variable***, so it's no longer alive after you exit the function `get_number()`. After returning to `main()`, the reference `r_n1` points to where ***n1 used to be!*** What makes dangling references perilous, is that sometimes the reference happens to dangle over the right spot.

Example: A dangling reference that the g++ compiler will notice

dangling_reference3.cpp

```cpp
// Unsafe code example
// g++ warns at complile time
#include <iostream>
int & get_number()
{
  int n1 = 100;
  return n1;
}
int main()
{
  int number = get_number();
  std::cout<<"number is "<<number<<"\n";
}
```

You don't necessarily need a function call to get a dangling reference. Example `basic/dangling_reference2.cpp` shows what happens if you store the reference to a vector element and then resize the vector. There is no guarantee the stored reference is pointing to a valid location any more.

valgrind -q a.out often detects such memory problems.

My own valgrind user advice:

Any message from `valgrind -q a.out` means your code has a bug.

21.2 `lvalue` and `rvalue`

In 2010 things changed a lot. As Kornel Kisielewicz @Stackoverflow aptly puts it,

> *The whole massacre began with the move semantics. Once we have expressions that can be moved and not copied, suddenly easy to grasp rules demanded distinction between expressions that can be moved, and in which direction.*

Move semantics is about moving objects, as opposed by copying them. Forgive me for cutting corners, read the previous post or c11-tutorial-explaining-the-ever-elusive-lvalues-and-rvalues by Danny Kalev. The question to ask is: can the thing be ***moved*** from one memory location to another, and does it have an ***identity***?
Expressions, program statements that have a value, have long or short lifetimes. Long-living expressions have an identity (named entities), while short-living expressions die soon and have no identity.

- `lvalue` is locator value, a name of a memory location.
  In `double x = 10.0;` `x` is the name of a memory location holding a `double`, so it's an `lvalue` . More technically, an `lvalue` is either a non-movable expression or one with identity. Either one that lives long, or one that cannot be moved.

- `rvalue` is a value in a memory location.
  In `double x = 10.0;` the number `10.0` is an `rvalue` . More technically, an `rvalue` is a movable expression or one without identity. It may be also a temporary object about to die: evaluating `x=(a*c)+b` the compiler may create a temporary to hold the result of `a*c`.

Obviously it doesn't make sense to try something like `10.0 = x;`. `lvalue` is like a drawer, and `rvalue` is like a solid object. You can put a solid object into a drawer, but you can't put a drawer into a solid object.

21.3 rvalue references and rvalue references

The next question is, how to refer to an lvalue or to an rvalue ? You need rvalue references and rvalue references.

- lvalue reference
  Marked with a single ampersand &, for example int& a, same as int &a and int & a. [72]

- rvalue reference
  Marked with a double ampersand &&, for example int&&. But some &&'s aren't rvalue references; keep reading.

Why rvalue references? Take a look at the Robin-Hood code on the next page. It has the function rich() that returns a plain number 500, an rvalue . [73] I overloaded print_ref(), so that you can see what type of expression the argument is,

```
argument int&
500
argument int&&
500
```

The latter is an rvalue reference, it's the type of rich(), which returned

```
   return 500;
```

[72] Which one to use, int &a or int& a?. A matter of taste. In int& x, y it would appear as if there's a type int& that applies to both x and y, which is not the case. In this case int &x, y looks better. However, it's confusing to declare mixed types on the same line.

[73] It's a bit odd Robin Hood code, because the rich has an inexhaustible amount of money.

robin_hood.cpp

```cpp
// Robin Hood code in C++
//g++ -Wextra -std=c++14 -Wpedantic -g robin_hood.cpp
#include <iostream>

int rich(){
  return 500; // "rich gives out 500 gold coins"
}

void print_ref(int &x){
  std::cout<<"argument int &\n";
  std::cout<<x<<"\n";
}
void print_ref(int &&x){
  std::cout<<"argument int && \n";
  std::cout<<x<<"\n";
}


int main()
{
  int poor = rich();
  print_ref(poor);
  print_ref(rich());
}
```

21.4 One-liners of `lvalue` and `rvalue` references

I recite the excellent examples in what-is-a-rvalue-reference by Varun and let the compiler have it's say:

lvalue_rvalue_reference.cpp

```cpp
#include <iostream>

int getData(){
  return 9;
}

int main(){
  int x = 10;
  int & lvalueRef = x;   // lvalueRef is a lvalue reference
  std::cout<<lvalueRef<<"\n"; // output: 10

  //int & lvalueRef2 = (x+1); // Error - lvalue Reference Can't point to rvalue
  //invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'

  int && rvalueRef = (x+1); // rvalueRef is rvalue reference
  std::cout<<rvalueRef<<"\n";    // output: 11

  //int & lvalueRef3 = getData(); // Error - lvalue Reference Can't point to rvalue
  //invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'

  const int & lvalueRef4 = getData(); // OK but its const
  int && rvalueRef5 = getData();
  std::cout<<rvalueRef5<<"\n";  // output: 9
}
```

21.5 The strange `T&&` and the *Perfect Forwarding Problem*

Let's walk the road to the `T&&` notation paved by Scott Meyers and Eli Bendersky.

Task: Write a wrapper function `wrapper()` that *perfectly forwards* all arguments to any function `f()`.

In Python it was simple, just pass arguments as `(*args, **kwargs)`. In C++ everything has to be typed, and it's passing the types correctly that is the tricky bit. [74] Try a simple wrapper,

```
// try one
template <typename T1, typename T1, typename T3>
void wrapper(T1 x, T2 y, T3 y){
  f(x,y,z);
}
```

Then you realize that this will always call `f` by value, so the arguments `x,y,z` are copied over. This is no good if your function is `f(int& x, int& y, int& z)`. Clever you are and modify you code,

```
// try two
template <typename T1, typename T1, typename T3>
void wrapper(T1& x, T2& y, T3& y){
  f(x,y,z);
}
```

However, this won't work if you call `f` with `rvalues`, for example `f(1.0,3.0);`. The compiler complains about an invalid initialization of non-`const` reference from an `rvalue`. It cannot make a reference required by `T1& x` that points to number 1.0, because 1.0 doesn't provide a memory location to point to.

[74] C++ is considered to be weakly typed, because in strongly typed languages conversions between different types are forbidden.

You could cure the problem by adding a `const` to an argument. To *any* argument. At this stage you smell something burning. You would have to write overloaded wrapper functions for all possible combinations, in this case

```
wrapper(      T1& x,       T2& y,       T3& z)
wrapper(const T1& x,       T2& y,       T3& z)
wrapper(      T1& x, const T2& y,       T3& z)
wrapper(      T1& x,       T2& y, const T3& z)
wrapper(const T1& x, const T2& y,       T3& z)
wrapper(const T1& x,       T2& y, const T3& z)
wrapper(      T1& x, const T2& y, const T3& z)
wrapper(const T1& x, const T2& y, const T3& z)
```

This gets impossible with increasing number of arguments, and things only gets worse once you add the possibility of `rvalue` references, more qualifiers.

Solution
Introduce `std::move` and `std::forward`.

Before getting carried away, Scott Meyers reminds that

> `std::move` moves nothing and `std::forward` forwards nothing. They do a ***type cast***.

C++ solves the *perfect forwarding problem* with the type cast `std::forward`:

```
template <typename T1, typename T2, typename T3>
void wrapper(T1&& x, T2&& y, T3&& z) {
    f(std::forward<T1>(x), std::forward<T2>(y), std::forward<T2>(z));
```

```
}
```

But *how* does `std::forward` solve the perfect forwarding problem? First, `T&&` is not always an `rvalue` reference. There are cases where one does ***type deduction***. This recycling of `&&` is widely considered unfortunate, but we have to live with it. Scott Meyers calls `T&&` an ***universal reference***, some call it a ***forwarding reference***. To make sense out of a thing like `&&&` one has a rule:

> ***Reference Collapsing rule***: `&` always wins.
> `T` is `T`
> `T&` is `T&`
> `T&& &` is `T&`
> `T& &&` is `T&`
> `T&& &&` is `T&&`

These rules seem odd, but they work:

```
wrapper(1.0,int& y,int&& z); // rvalue, reference, rvalue reference
// will call
f(std::forward<double &&>(1.0), std::forward<int& &&>(y), std::forward<int && &&>(z));
// will reference-collapse to
f(double&& 1.0, int& y, int&& z); // rvalue reference, reference, rvalue reference
```

`std::move` is a single-minded, Robin-Hood cast:

```
poor = std::move(rich);  //rich becomes an rvalue
                         // ''movable'', property ready to be stolen.
                         // std::move does this always.
```

```
poor = std::move(another_poor); // really *always* , a single-minded cast
```

Here `std::move(another_poor)` makes everything in `another_poor` movable, and `poor = ...` does the moving.

22 C++ Smart pointers

Sometimes numerical libraries require pointer arguments. C++ has no automatic carnage collection, therefore naked C-style pointers lead easily to memory leaks. C++11 added ***smart pointers***, pointers wrapped with information about their expected lifetime.

A `unique_ptr<T>` is a non-copiable pointer to type `T` object, and there can be only one pointing to that object. The pointer gets automatically deleted once it's scope ends, and memory is released.

unique_pointer.cpp

```cpp
#include <iostream>
#include <memory>

class Resource
{
public:
        Resource() { std::cout << "Resource acquired\n"; }
        ~Resource() { std::cout << "Resource deleted\n"; }
    void use() { std::cout<<"Resource use\n"; }
};

void test()
{
    // allocate memory from the heap:
    auto res = std::make_unique<Resource>(); // One way
         // std::unique_ptr<Resource> res{new Resource()}; // Another way
    std::cout<<"test using Resourse\n";
    res->use();
    std::cout<<"about to exit test()\n";
}// smart pointer automatically deleted here

int main()
{
    std::cout<<"calling test()\n";
    test();
    std::cout<<"returned from test()\n";
}
```

If you need to pass a `unique_ptr` to a function, you need to move it using `std::move()`, which also gives away the ownership of the `unique_ptr` to the function.

unique_pointer_moving.cpp

```cpp
#include <iostream>
#include <memory>

class Resource
{
public:
        Resource() { std::cout << "Resource acquired\n"; }
        ~Resource() { std::cout << "Resource deleted\n"; }
    void use() { std::cout<<"Resource use\n"; }
};

void test2(std::unique_ptr<Resource> res)
{
    std::cout<<"test2 using Resourse\n";
    res->use();
     std::cout<<"about to exit test2()\n";
} // smart pointer automatically deleted here

void test()
{
    // allocate memory from the heap:
    auto res = std::make_unique<Resource>(); // One way
         // std::unique_ptr<Resource> res{new Resource()}; // Another way
    std::cout<<"test using Resourse\n";
    res->use();
    // pass pointer to test2
    // test2(res); // error: use of deleted function ... <= there's no copy constructor for unique_ptr
    test2(std::move(res));
    // smart pointer already deleted in test2
    std::cout<<"about to exit test()\n";
}

int main()
{
    std::cout<<"calling test()\n";
    test();
    std::cout<<"returned from test()\n";
}
```

A `shared_ptr<T>` can be copied, and it keeps a reference count (remember the reference count in Python?) `use_count()` of how many copies are around. Once the last copy leaves it's scope the pointer is deleted and memory is released. A `shared_ptr<T>` passed as a function argument (`test2()`) limits the ways how the function can be called; it also lends the ownership of the `shared_ptr<T>` to the function. There's no point of passing a `shared_ptr<T>` to a function unless it will manipulate the smart pointer. The function `test2()` misuses the shared pointer, instead you can pass a reference as in `test3()`.

shared_pointer.cpp

```
#include <iostream>
#include <memory>

class Resource
{
public:
    int value;
        Resource() { std::cout << "Resource acquired\n"; }
        ~Resource() { std::cout << "Resource deleted\n"; }
    void use() { std::cout<<"Resource use; value "<<value<<"\n"; }
};
std::shared_ptr<Resource> test1()
{
    // allocate memory from the heap:
    // std::shared_ptr<Resource> res(new Resource); // one way
        auto res = std::make_shared<Resource>(); // another way
    std::cout<<"test1() using Resource\n";
    res->use();
    res->value = 1;
    std::cout<<"about to exit test1()\n";
    return res;
}
void test2(std::shared_ptr<Resource> res)
{
    std::cout<<"test2() using Resource res\n";
    res->value = 2;
    res->use();
    std::cout<<"about to exit test2()\n";
}
void test3( Resource& res)
{
    std::cout<<"test3() using Resource res\n";
    res.value = 3;
    res.use();
    std::cout<<"about to exit test3()\n";
}

int main()
{
    std::shared_ptr<Resource> shared_res2;
    {
        auto shared_res1 = test1();
        std::cout<<"returned from test1()\n";
        std::cout<<"shared_res1 use_count = "<<shared_res1.use_count()<<"\n";
        std::cout<<"copying shared resource\n";
        shared_res2 = shared_res1;
        std::cout<<"shared_res1 use_count = "<<shared_res1.use_count()<<"\n";
    } // pointer shared_res1 is not destroyed, because shared_res2 needs it
    {
        test2(shared_res2);
        std::cout<<"returned from test2()\n";
    }
    std::cout<<"shared_res2 use_count = "<<shared_res2.use_count()<<"\n";
    test3(*shared_res2);
}
```

A `weak_ptr<T>` is rarely used. It is like a `shared_ptr<T>`, except it doesn't increase the reference count and may therefore point to a deleted resource. It's can be passed to a function that can create a `shared_ptr<T>` from it by locking the resource,

thus adding the reference count by one.

23 C++ Standard Library: A closer look

23.1 `std::vector` container

`std::vector` containers are flexible and minimize the risks of dynamic memory allocations. But they are not math vectors.

Example: Create an empty container and push a few elements to it.

vector.cpp

```cpp
// Create empty vector container and push data in
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int main()
{
  vector<double> x;
  for(unsigned i=0;i<6;i++){
    x.push_back(pow(i,2)); // push i^2 to vector x
    cout<<"i="<<i<<" x="<<x[i];
    cout<<" size of x="<<x.size()<<endl;
  }
  cout<<"final     size ="<<x.size()<<"\n";
  cout<<"final capacity ="<<x.capacity()<<"\n";
}
```

```
final     size =6              size of the container increased automatically
final capacity =8              more space reserved than needed, just in case you extend the vector
```

23.1.1 Iterators

Iterators point to elements of a container.

Example: Print all elements of a `std::vector` using iterator

vector2.cpp

```cpp
// print all but 2 elements of
// a vector container using iterator (it)
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int main()
{
  vector<double> x;
  // push i^2
  for(unsigned i=0;i<6;i++){
    x.push_back(pow(i,2));
  }
  // all but the last two elements: iterator
  for(auto it=x.begin(); it!=x.end() - 2; it++){
    cout<<*it<<' ' ;
  }
  cout<<endl;
  // all elements: range-for loop
  for(auto elem:x) cout<<elem<<' ';
  cout<<endl;
}
// output:
// 0 1 4 9
// 0 1 4 9 16 25
```

Same in Python:

```python
if __name__=="__main__":
    x = [i**2 for i in range(6)]
    print(x[:-2])
    print(x)
```

`auto` is a message to the compiler: "deduce the type yourself". Use often.

In the example `auto` spared you from typing the ugly type

```
vector<double>:: iterator it;  // type of an iterator
```

One way to fill a `std::vector` is `std::fill`; it's very fast if you need to reset an existing `std::vector` [75],

```
std::vector<double> y(5);
std::fill(y.begin(), y.end(), 1.0);     // math: y=(1.0,1.0,1.0,1.0,1.0)
std::fill (y.begin()+1,y.end()-2,2.0); // math: y=(1.0,2.0,2.0,1.0,1.0)
```

To create a `std::vector` and set values the cleanest ways are

```
std::vector<double> y(5,1.0);  // math: y=(1.0,1.0,1.0,1.0,1.0)
std::vector<double> y{1.0,1.0,1.0,1.0,1.0} // Universal initialization
```

C++ lets you initialize almost anything with the ***universal initialization*** *\{\}*.

C++ has also a ***range-for*** loop:

```
std::vector<double> y(100);
for( auto & elem:y) {elem=1.0}; // notice the & : use reference to elements!
```

The loop goes through all elements of `y` without you worrying about how many there are.

Be careful with the ampersand `&`:

[75] Can you do algebra with the length of a `std::vector`?
`x.resize(0)` sets the length to zero, so `x.size()` is 0. OK, but `x.size()-1` is 18446744073709551615, not -1 !
Lengths have type `unsigned` and those can't store negative numbers.

```
for( auto elem:y) {elem=1.0};  // this does nothing at all!
for( auto elem:y) {cout<< elem<<" ";}; // this works (but doesn't try to change y)
```

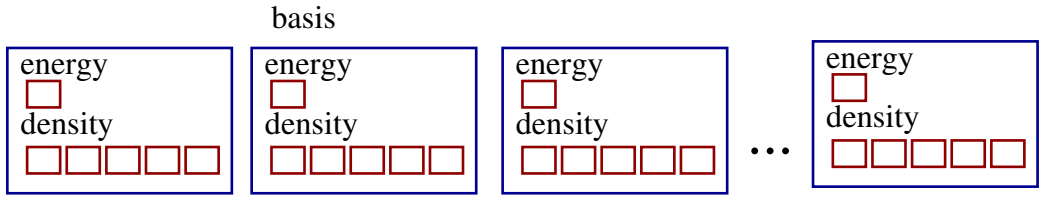
23.1.2 Storing objects into std::vector

Example: std::vector storing objects

vector_of_class_objects.cpp

```
// g++ -std=c++11 vector_of_class_objects.cpp
#include <iostream>
#include <vector>
#include <cmath>

class WaveFunction{
public:
  double energy;
  std::vector<double> density;
};
int main()
{
  std::vector<WaveFunction> basis; // a vector of WaveFunctions
  WaveFunction wf;
  for (int i=0;i<10;++i){ // make a 10 wavefunction basis
    wf.energy = i*i;
    for (int j=0;j<5;++j) wf.density.push_back(sqrt(j)*i);
    basis.push_back(wf);
    wf.density.clear();// REMEMBER THIS or wf.density keeps growing
  }
  // output for testing
  for (auto wf: basis) {   // wf goes through elements of basis
    std::cout<<" energy = "<<wf.energy<<"\n";
    std::cout<<"density = ";
    for (auto den: wf.density) std::cout<<den<<" "; // den goes through a density in wf
    std::cout<<"\n";
  }
}
```

A `std::vector` can hold with almost any type of data. Here the container contains object of the self-made type WaveFunction (blue boxes), and a number (energy) and a `std::vector` (density).



These are exactly the same thing:

```
class WaveFunction{
 public:          // class: all is private by  default
  double energy;
  vector<double> density;
};
```

```
struct WaveFunction{
 double energy;  // struct: all is public by default
 vector<double> density;
};
```

23.1.3 Sneak peak: overloading operator <<

If you have made up your mind about how you want objects printed, you can ***overload the << operator***.

Example: Clean output of an object using overloaded <<

better_vector_of_class_objects.cpp

```
// g++ -std=c++11 better_vector_of_class_objects.cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <iomanip>

class WaveFunction{
  double energy;
  std::vector<double> density;
public:
  WaveFunction(double energy_, std::vector<double> density_): energy{energy_},density{density_}{};
  friend std::ostream& operator<< (std::ostream& os, const WaveFunction& rhs)
  {
    os<<std::fixed<<std::setprecision(8);
    os<<"energy = "<<std::setw(15)<<rhs.energy<<" density = ";
    for (auto den: rhs.density) os <<std::setw(15)<<den<<" ";
    os<<"\n";
    return os;
  }
};
int main()
{
  std::vector<WaveFunction> basis; // a vector of WaveFunctions
  std::vector<double> dens; // to fill density data
  for (int i=0;i<10;++i){ // 10 wavefunction basis (any number is ok)
    for(int j=0;j<5;++j) dens.emplace_back(sqrt(j)*i); // 5 density values (any number is ok)
    basis.emplace_back(i*i,std::move(dens));
  }
  // output for testing

}
```

We'll come back to this later, just note the line

```
friend std::ostream& operator<< (std::ostream& os, const WaveFunction& rhs)
```

23.2 Heterogeneous types stored in `std::vector`

Old C++ coders say `std::vector` containers can't hold but one type of data, because C++ is a statically typed language. People found a way around this limitation by hiding types inside types. This was done in `Boost/variant.hpp`, and since C++17 we have `std::variant`.

Here's an example from Andy G's Blog gieseanw@wordpress.com. Define a variable that can hold three types of data:[76]

```
std::variant<int, double, std::string> myVariant;
myVariant = 1; // initially it's an integer
```

and a ***visitor*** handler for each of the data types (this is called a *visitor pattern*),

```
struct MyVisitor {
  void operator()(int& _in){_in += _in;}
  void operator()(double& _in){_in += _in;}
  void operator()(std::string& _in){_in += _in;}
};
```

Finally, invoke visitors using `std::visit`,

```
std::visit(MyVisitor{}, myVariant);
```

and you're done.

Another example of `std::variant` by Filipek @cppstories.com. Again, a visitor is defined for each type, [77]

[76] The blog shows how to define an easy-to-use heterogeneous container which is faster than the `std::vector` implementation.

[77] Interesting applications: ***state machines***, computing roots of a function when there are one, two or none, and more.

vector_variant.cpp

```cpp
#include <iostream>
#include <vector>
#include <variant>

class Triangle{
public:
    void Render()  { std::cout << "Drawing a triangle!\n"; }
};

class Polygon{
public:
    void Render() { std::cout << "Drawing a polygon!\n"; }
};

class Sphere{
public:
    void Render() { std::cout << "Drawing a sphere!\n"; }
};

int main(){
    std::vector<std::variant<Triangle, Polygon, Sphere>> objects {
        Polygon(), Triangle(), Sphere(), Triangle()
    };

    auto CallRender = [](auto& obj) { obj.Render(); };

    for (auto& obj : objects)
        std::visit(CallRender, obj);

}
```

Visitors that do a similar operation are most welcome, since they all collapse to a template.

C++ shows it's verbal talent in explaining the compiler what we want the code to do,

```
std::any                // type that can be anything (656 lines of code in libstdc++)
std::monostate          // empty state type for std::variant
std::holds_alternative // test what the currently active type is
```

I have no idea when and how to use `std::any`, so I can't help you there.

23.3 Moving, not copying

Moving objects instead of copying them is a big thing in C++. ***Moving*** is a Robin Hood operation,

```
poor=std::move(rich);
```

pilfers `rich` from it's resources and hands them to `poor`. Way more effective than copying money or gold!

As mentioned earlier, `std::move(rich)` does not actually move `rich`, it's a ***cast*** that makes it *movable*. With `std::move` you tell the compiler that now an object has resources that can be robbed. In other words, `std::move` is for turning lvalues to rvalues (see section 21.2) so that you can call ***the move constructor***. The move constructor is the "Robin Hood code".

A move constructor and a move assignment look like this:

```
X::X(X&& other);            //  move constructor
X& X::operator=(X&& other); //  move assignment operator
```

You can be even more dramatic. If objects of a class should never be copied, you can forbid copying by ***deleting the copy constructor and the copy assignment***:

```
class NoCopy {
        // ...
        NoCopy(const NoCopy&) = delete;// forbid copying
        NoCopy& operator=(const NoCopy&) = delete;
};
```

The next lengthy example tries to elucidate situations when an object is copied and when moved. For transparency, the copy and move constructors as well as the copy and move assignments print out a message, so you can see which one is invoked.

Example: Copying and moving

move_constructor.cpp

```cpp
#include <iostream>
#include <vector>

class X{
public:
  int count;
  std::vector<double> vec;
  X() = default ; // constructor
  X(int count_, std::vector<double> vec_): count{count_},vec{vec_} {} // constructor
  ~X() noexcept = default; // destructor
  auto operator = (X& rhs) & -> X&  {
    vec = rhs.vec;
    count = rhs.count;
    std::cout << "-- copy assignment called --\n";
    return *this;
  }
  X (const X& rhs) : vec(rhs.vec), count(rhs.count) {
    std::cout << "-- copy constructor called --\n";
  }
  auto operator = (X&& rhs) & noexcept -> X& {
    vec       = std::move(rhs.vec);
    count     = std::move(rhs.count);
    rhs.count = 0;   // no way to delete integer
    // rhs.vec   = {};  // resizes x to 0 length, but done already
    std::cout << "-- move assignment called --\n";
    return *this;
  }
  X(X&& rhs) noexcept : vec(std::move(rhs.vec)), count(std::move(rhs.count)) {
    std::cout << "-- move constructor called --\n";
  }
};

int main() {
  X x1{3,{1,2,3}}; // initialize count to 3, vec to (1,2,3)
  std::cout<<"line: X x2 = x1;\n";
  X x2 = x1;
  std::cout<<"line: X x3(x1);\n";
  X x3(x1);
  std::cout<<"line: X x4 = std::move(x1);\n";
  X x4 = std::move(x1);
  std::cout<<"line: X x5(std::move(x2));\n";
  X x5(std::move(x2));
  X y1{99,{100,200,300}}, y2;
  std::cout<<"line: y2 = y1;\n";
  y2 = y1;
  std::cout<<"line: y2 = std::move(y1);\n";
  y2 = std::move(y1);
}
```

```
$ a.out
line: X x2 = x1;
-- copy constructor called --
line: X x3(x1);
-- copy constructor called --
line: X x4 = std::move(x1);
-- move constructor called --
line: X x5(std::move(x2));
-- move constructor called --
line: y2 = y1;
-- copy assignment called --
line: y2 = std::move(y1);
-- move assignment called --
```

Try adding some tests after the last move assignment:

```
std::cout<<y2.vec[0]<<std::endl; // OK
std::cout<<y1.vec.size()<<std::endl; // 0, OK
std::cout<<x1.vec.size()<<std::endl; // 0, OK
std::cout<<y1.vec[0]<<std::endl; // Segmentation fault, as expected
std::cout<<x1.vec[0]<<std::endl; // Segmentation fault, as expected
```

Move assignment has the line `rhs.vec = {};`, which made `y1.vec` zero length. We really *moved* resources.

`std::move` makes an object movable, but it doesn't guarantee it will be moved.

`std::move` sets a move permission, but it's not forcing anything.

23.4 std::valarray and std::array

std::vector is not a mathematical vector, it's an expandable list. None of the common math vector operations are defined for std::vector. This led one to introduce to C++ the class std::valarray (not a container!). Recent compilers should produce as fast code with std::valarray as with std::vector, because they use the very efficient ***expression template*** technique. The closest relative to std::valarray is in the Boost library, std::valarray ≈ boost::ublas::vector. Use std::array if you know the (maximum) size at compile time.

A speed comparison of a few containers and how to fill them is in file bigarray_speedtest.cpp. Compiled with

```
$ g++  -Ofast -march=native -mtune=native bigarray_speedtest.cpp
```

and using N=10^7. It's convenient to define a macro,

```
#define MAXN 10000000
:::
std::array<int,MAXN> bigarray;  // FIXED SIZE
```

data type	space reservation	setting k:th element	timing (s)
raw array	new arr[N]	arr[k] = k	0.972
std::vector	init to size N	arr[k] = k	1.32
std::vector	init to size 0	push_back(k)	4.97
std::vector	init to size 0, reserve(N)	push_back(k)	3.18
std::vector	init to size 0, reserve(N)	emplace_back(k)	3.18
std::valarray	init to size N	arr[k] = k	3.19
std::array	fixed size N	arr[k] = k	0.009

The fixed-size std::array was 100 x faster than the fastest dynamically allocated array! What you loose in flexibility you gain in speed.

Be aware, that `std::array` may be stored in stack. You may need to increase the `stacksize`, in bash [78]

```
$ ulimit -s unlimited
```

The `std::vector` method `reserve()` can be used with the methods `puch_back()` and `emplace_back()`. However, don't make space for the `std::vector` in the constructor and then `push_back()`, you're extending the `std::vector`! This is fine,

```
std::vector<int> v;
v.reserve(2);
v.push_back(11);
v.push_back(22);
// OK, v is (11,22)
```

but this isn't,

```
std::vector<int> v(2);
v.push_back(11);
v.push_back(22);
// NOT OK, v is (0,0,11,22)
```

[78] I had stacksize 8192, way too small! Segmentation Fault was imminent.

23.5 Give an alias to a type with `using`

IMHO the lack of built-in numerical data structures, such as vectors and matrices, prevents C++ to be a perfect language for numerics. I face a dilemma: How to code vectors and matrices? A few options:

1 Should I stick to `std::valarray`, `std::array`, or `std::vector` and live with the deficiences?
In a small project, that needs to be more portable than numerically less ambitious: Yes.
In a serious numerical project: No.
Go to option 3.

2 Should I write a class for a math vector myself?

NEVER write your own classes for math vectors or matrices.

You may see C++ numerics cources in the web and books telling you how to make a matrix class. Don't fall to the trap, you'll make mistakes, waste time and get a slow program. Sure, these are deceivingly simple tasks at a first glance, but once you hit the speed bumps you can hear your tailpipe clank to the floor.
Go to option 3.

3 Should I use a "vector" or "matrix" from an external (external to C++ standard) library?

YES. But not directly, utilize `using`

With `using` (or `typedef`) you can give a name to your own data structure.
Write the code so that the decision about the data type is in one place, in case you change your mind.

```
using new_name = existing_type;
```

23.6 Heavier usage of aliases

Let's assume the namespace `lib_one` defines vectors and matrices from a library. You can define

```
namespace my{
  namespace lib  = lib_one;  // namespace alias; library_one provides vector and matrix
  using d_data_t = double;
  using i_data_t = int;

  template<typename T>        // template alias
  using vector = lib::vector<T>;
  template<typename T>        // template alias
  using matrix = lib::matrix<T>;

  using d_vec  = vector<d_data_t>;
  using d_mat  = matrix<d_data_t>;
  using i_vec  = vector<i_data_t>;
  using i_mat  = matrix<i_data_t>;
}
```

and your code uses, for example,

```
d_vec x;
d_mat m;
i_vec iv;
```

If you change your mind and want to use `long` data types and `lib_two` instead, then just edit two lines,

```
namespace lib  = lib_two;
using i_data_t = long;
```

without touching anything else. Be cautious, `lib_one::vector` and `lib_two::vector` may be incompatible.

23.7 Stream iterators *(read on spare time)*

Stream is an object representing an I/O *(input/output)* channel. Streams are considered one of the best qualities of C++ by C++ programmers - and one of the worst by C programmers. A stream iterator is something that goes through a stream. [79]

Think of the following task: Write a program that reads an arbitrary string of characters from the keyboard. Sounds simple, but in many programming languages this is very lengthy to do safely. Remember, user may hit any key and and keep on hitting for a month. Below is a C++ suggestion.

Example: Read characters to a `std::vector`

streamiter1.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{
  vector<string> s;
  // input from standard input (cin)
  copy(istream_iterator<string>(cin), //from
       istream_iterator<string>(),    //end
       back_inserter(s));             //to
  // output to standard output (cout)
  copy(s.begin(),s.end(),                    // from
       ostream_iterator<string>(cout," ")); // to
  cout<<endl;
}
```

Use the keyboard to give characters and press ctrl-d when you are done. Here `std::copy` is an algorithm and I'm going to tell more about algorithms next.

[79] istream = stream in, ostream= stream out

23.8 Algorithms and utilities

The C++ Standard Library has many useful methods, search algorithms, sorting algorithms etc. The list is long, please visit the pages

www.cplusplus.com/reference/algorithm
www.cplusplus.com/reference/utility

Examples:
If `v` and `w` are `std::vector` containers, and `it` is an iterator of that container type, then

```
it = std::max_element(v.begin(),v.end());
```

returns the iterator `it` that points to the largest element, the largest element is `*it`.

```
std::sort(v.begin(),v.end());
```

sorts the container

```
std::swap(v,w);
```

swaps the contents of containers `v` and `w`. C++ has this in the header `<utility>`.

23.8.1 About std: min_element, max_element, find, sort, reverse

Example: C++ Standard Library algorithms: minimum, maximum and search of an element

algo_minmax.cpp

```
// Finding elements from a vector container
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

namespace my{   // utility to print out a vector
  template<typename T>
  void vector_out(const std::vector<T> v){
    for(auto x:v) std::cout<<x<<" ";
    std::cout<<"\n";
  }
}
int main(){
  std::vector<double> v{1.4,1.6,0.2,1.8,0.1,1.5}; // or do many push_back's
  std::cout<<"original vector\n";
  my::vector_out(v);
  std::cout<<"minimum element = "<<*min_element(v.begin(),v.end())<<"\n";
  std::cout<<"maximum element = "<<*max_element(v.begin(),v.end())<<"\n";

  // find element with some value
  auto it = find(v.begin(),v.end(),0.2);  // I'd prefer find(v,0.2)
  if(it==v.end())
    std::cout<<"failed to find value\n";
  else {
    std::cout<<"found value "<<*it<<"\n";
    // reverse some elements
    std::cout<<"reverse starting from "<<*it<<"\n";
    std::reverse(it,v.end()); // I'd prefer std::reverse(it)
    my::vector_out(v);
  }
  std::cout<<"sorting...\n";
  std::sort(v.begin(),v.end());
  my::vector_out(v);
}
```

Iterators may look a bit messy, but they are easily hidden from view. Especially, if all you want is the number where the iterator points at.

Example: C++ Standard Library algorithms: another version of minimum search

easyusemin.cpp

```
// Finding min element; trying to simplify usage
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

namespace my{
  double min_element(const std::vector<double> & v){
    // no explicit iterator! We need only *(iterator)
    return *(std::min_element(v.begin(),v.end()));
  }
}
// calling routine is clean and simple:
int main(){
  std::vector<double> v{1.7,1.3,2.8,4.1};
  std::cout<<"minimum element = "<<my::min_element(v)<<std::endl;
}
```

Be cautious when coding specialized versions of C++ Standard Library algorithms. The namespace `my` protects my own version of `min_element`.

23.8.2 `std::swap` is a template

Previously given `swap` using reference variables is for swapping one type of things only. What if we want to swap two real numbers or two other type of variables? Terribly boring to write each variable type it's own version of `swap`.

Write a ***template***. Actually it's been done already, `std::swap` looks like this. Like many standard library codes, this too were refurnished in C++11. The user sees only the improved performance.

Example: `std::swap` template in `<utility>`

swap_template.cpp

```
#include <utility>
template <typename T> void swap(T& a, T& b)
{
  T c(std::move(a)); a=std::move(b); b=std::move(c);
}
template <class T, std::size_t N> void swap(T (&a)[N], T (&b)[N])
{
  for (std::size_t i = 0; i<N; ++i) swap(a[i],b[i]);
}
// C++98 version was
//{
//  T c(a); a=b; b=c;
//}
```

You can check that this compiles,

```
$ g++ -c swap_template.cpp
```

`typename T`	is equivalent to `class T`, the C++ standard defines this
`void swap`	swap returns nothing; no point in writing `result = std::swap(a,b); // nonsense`
`T c(std::move(a))`	create type `T` object `c` and `move` the contents of `a` to object `c`
	`std::move` make sure you use the correct `move` function from the `std` namespace
	`std::move(a)` fix `a` to a movable object (one whose contents can be stolen).

What can be swapped with this template:
(C++11): "Type `T` shall be move-constructible and move-assignable (or have swap defined for it)"
C++11 introduced ***move semantics***, and replaced copy (`T c(a)`) for move (`T c(std::move(a))`).

Many C++ Standard Library algorithms are compact and efficient.

Example: C++ Standard Library permutation algorithm

algo_permutations.cpp

```cpp
// Using C++ Standard Library algorithm next_permutation
//
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

namespace my {
  template<typename T>
  struct State{
    T i;
    std::vector<T> v;
    State(T ind,std::vector<T> vec) {i=ind; v=std::move(vec);}
    void print(){
      std::cout<<"state # "<<i<<" ( ";
      for(auto x:v) std::cout<<x<<" ";
      std::cout<<")"<<std::endl;
    }
  };
}

int main () {
  std::vector<my::State<int>> states;
  std::vector<int> vec={0,1,2,3,4};
  std::sort(vec.begin(),vec.end()); // make sure next_permutation starts ok
  int i = 1;
  do {
    states.push_back(my::State(i,vec));
    i++;
  } while (next_permutation (vec.begin(),vec.end()));
  for (auto s:states) s.print();
}
// state # 1 ( 0 1 2 3 4 )
// ...
// state # 120 ( 4 3 2 1 0 )
```

The loop

```
do {
 ...
 } while(next_permutation (vec.begin(),vec.end())
```

goes on until the test `while (true)` changes to `while (false)`.

A physics example: Generate all many-body states with a fixed number of spin-1/2 fermions.
Each spin state can hold 0 or 1 fermions (Pauli rule). If you have 4 spin states and 2 fermions, possible states are (0011),(0101),(0110),(100
making 6 many-body states.

Example: Fermion states by permutation

algo_fermion_states.cpp

```cpp
// Finding fermion basis states using next_permutation
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>

namespace my {
  template<typename T>
  struct State{
    T i;
    std::vector<T> v;
    State(T ind,std::vector<T> vec) {i=ind; v=std::move(vec);}
    void print(){
      std::cout<<"state # "<<std::setw(10)<<std::left<<i<<std::left<<" = ";
      for(auto x:v) std::cout<<x;
      std::cout<<std::endl;
    }
  };
}

int main () {
  const int Nstates=8; // single-particle states
  const int N=5; // number of fermions
  std::cout<<"All "<<N<<"-fermion states for a system of "<<Nstates<<" single-particle states\n";
  std::vector<my::State<int>> states;
  std::vector<int> vec(Nstates,0); // 0 occupation by default
  for (int i=0; i!=N; ++i) {vec[i]=1;} // occupy single-fermion states
  std::sort(vec.begin(),vec.end()); // make sure next_permutation starts OK
  int i = 1;
  do {
    states.push_back(my::State(i,vec));
    ++i;
  } while (next_permutation (vec.begin(),vec.end()));
  for (auto s:states) s.print();
}
```

```
All 5-fermion states for a system of 8 single-particle states
state # 1           = 00011111
state # 2           = 00101111
state # 3           = 00110111
...
state # 56          = 11111000
```

Electrons can have spin up (↑) or down (↓). For example 110010 could stand for (11)(00)(10) as occupations of up-down pairs, (↑↓)(00)(↑). Each state has only zeroes and ones, so binary coded states could also be used. The line

```
  std::sort(vec.begin(),vec.end()); // make sure next_permutation starts OK
```

is a safety precaution to make sure *all* permutations are found. Starting from an unsorted `std::vector`, such as `10101011`, wouldn't permute through all possible states.

23.9 Function returning a tuple

I got so used to tuples in Python that I now find it hard to keep from using them.
Example: Function returns a tuple

function_returns_tuple.cpp

```cpp
#include <iostream>
#include <tuple>
#include <type_traits>

template <typename  T>
auto f(T x){
  auto posneg = (x>0)?" is positive":" is negative";  // ternary
  auto isint =  (std::is_integral<T>::value)?" and an integer":", but not an integer";
  return std::tuple{x,posneg,isint}; // or  return std::make_tuple(i,posneg,isint);
}

int main(){
  for(int i=-5; i<6; ++i){
    auto [a,b,c] = f(i);  // C++17 structured bindings
    std::cout<<a<<b<<c<<std::endl;
  }
  double x;
  for(int i=-5; i<6; ++i){
    x = i*1.234;
    auto [a,b,c] = f(x);
    std::cout<<a<<b<<c<<std::endl;
  }
}
```

C++17 made handling of tuples so much easier, that C++ begins to look like Python! For example,

Example: Tuples since C++17

tuples.cpp

```
#include <iostream>
#include <tuple>
#include <string>

int main(){
  std::tuple<int,float,std::string> mytuple={1,3.1415,"one pie"};
  // structured bindings
  auto & [a,b,c] = mytuple; // get references to tuple members
  std::cout<<a<<" "<<b<<" "<<c<<std::endl;
  // std::tie with std::ignore
  float j;
  std::string k;
  std::tie(std::ignore,j,k) = mytuple;
  std::cout<<j<<" "<<k<<std::endl;
  // same with structured bindings
  auto [dummy,a2,a3] = mytuple; // no warning about unused variable
  std::cout<<a2<<" "<<a3<<std::endl;
  // testing ...
  auto [aa,_,cc] = mytuple;
  std::cout<<"_="<<_<<std::endl;
}
```

To my knowledge, C++ committees have not decided how to mark an ignored tuple element in structured bindings. The last lines demonstrate how using a lone underscore _ as a dummy variable in structured bindings may lead to interesting behaviour; now _ equals 3.1415.

There are several ways to define the output type of a function `f()`:

1. Use `decltype`

```
decltype(f(1)) res;
```

   This does not make a function call `f(1)` at run-time, the type is deduced at compile-time. In the sample code this will fix `res` to tuple `(int, string, string)`.

2. Set the return type yourself

```
std::tuple<int, string, string> res;
```

3. Use `auto` with a dummy call to `f()`,

```
auto res = f(1);
```

   This makes a function call `f(1)` at run-time.

4. Define `res` on the fly,

```
  auto res = f(i);
```

23.10 Header guards and namespace encapsulation

This section demonstrates how to write a function for home-made statistics, with C++ Standard Library algorithms and without. This is just for demonstration, there are much better statistical library routines than this. Here we push numbers to a `std::vector` and compute the statistical mean and standard deviation of the data in the function `get_stats()`.

Let's start with a header. Headers are for the compiler, with some information to you.

Example: First attempt as a header for `get_stats()` The ***preprosessor directives***

```
#ifndef MYHEADER_HPP
#define MYHEADER_HPP
...
#endif
```

make up a ***header guard***. They make sure this piece of code is not processed more than once.

- Header guards avoid circular inclusions, leading to a "too many include files" error.
- Header guards lead to shorter compile times.

C++20 added ***modules***, which don't need anything like this.

To use this header, stored in the file `myheader.hpp`, put in the beginning of the program the line

```
#include "myheader.hpp"
```

This line may well be in many program units, hence the header guard: `ifndef` stands for "if not defined". If `MYHEADER_HPP` is not defined, define `MYHEADER_HPP` and process the rest. The next time the compiler tries to `#include "myheader.hpp"`, it already has `MYHEADER_HPP` set and doesn't process the file.

A non-standard, but widely supported way to avoid the lengthy guard is to put this to ***source code*** use

```
// non-standard header guard IN SOURCE CODE, NOT IN HEADER
#pragma once
// more source code
```

This is polular, because the naming of macros in `\ifndef` has to be unique to work properly.

The file suffix `\textbf {.hpp}` is one way to tell that this is a header file, not to be processed unless `include`'d. In C one has the suffix `\textbf {.h}`, and it will also do in C++. ***In C++ the common practise is to put to headers only function declarations***.

What if `get_stats()` is part of a huge pile of code, where another `get_stats()` happens to exist, with the same type of arguments? You get an ***ambiquity error*** also called ***name collision***. Let's use ***namespace encapsulation***. Define our own namespace, where `get_stats()` lives, so that we can be sure which of the many `get_stats()` should be invoked.

A remark about style and good habits: refrain from taking the whole `std` namespace unnecessarily,

```
using namespace std; // don't do this
namespace mydefs
{
...
}
```

This is *impolite*, because if you ever give this header to a college to be used in his/her code, the poor fellow gets the whole `std` namespace, wanted or not. This easily leads to to name collisions, if the college was not carefully protecting his/her cute and short function names, such as `get()`.

Namespace encapsulation saves the day, something like `myclasses::vector` is clearly not a `std::vector`.

23.11 Formatted output with <iomanip>

Keep the numerical output readable, avoid mixing columns like this,

```
x y z
1.542234 12.4234 0.1213
13.0 4.234 1.00
```

In science, the underlying method itself may limit the accuracy of the result, so it's a bad practice to publish 6 decimals if the method is reliable only up to 2 decimals.

Typically, you'd want to set the width of the field (say, 10), the output form (say, `fixed`), and the number of decimals (say, 6):

```
std::cout<<std::fixed<<std::setprecision(6);  // 6 decimals
std::cout<<std::setw(10)<<"x"<<std::setw(10)<<"y"<<std::setw(10)<<"z"<<'\n'; //field width is 10
std::cout<<std::setw(10)<<x<<std::setw(10)<<y<<std::setw(10)<<z<<'\n';
```

This is awful, but at least the same formatting works also with writing to a file; it's a stream just as `std::cout`.

```
std::ofstream output("data.out");
output<<std::fixed<<std::setprecision(6);
output<<std::setw(10)<<x<<std::setw(10)<<y<<std::setw(10)<<z<<"\n";
```

```
data.out:
  1.542234 12.423400  0.121300
 13.000000  4.234000  1.000000
```

See `numerics/output_formatting.cpp` for more examples. Beware, that after formatting some setting are still on (`fixed`), while some are immediately forgotten (`setw`)!
C has the famous `printf` function, which is quite readable, but not considered a good C++ practise. I suggest that if you are already good with `printf` use it. [80]

23.12 `std::complex`: complex numbers and arithmetics

Example: Basic operations with complex numbers

[80] Pros of a `stream` object is type safety, the compiler can (always?) tell if a data type cannot be sensibly dealt with, and flexibility, it works the same way on screen and on file.

complex_ex.cpp

```
#include <iostream>
#include <complex>

int main()
{
  std::complex<double> c1,c2;
  c1 = std::complex<double>(1.5,2.2);
  c2 = std::complex<double>(1.0,3.3);
  std::cout<<"c1="<<c1<<"\n";
  std::cout<<"c2="<<c2<<"\n";
  // real(c2) or c2.real()
  std::cout<<"real(c2)="<<real(c2)<<"\n";
  std::cout<<"imag(c2)="<<imag(c2)<<"\n";
  std::cout<<"c1+c2="<<c1+c2<<"\n";
  std::cout<<"c1*c2="<<c1*c2<<"\n";
  std::cout<<"conj(c1)="<<conj(c1)<<"\n";
  std::cout<<"c1/c2="<<c1/c2<<"\n";
}
```

In math, the multiplication of complex numbers is

$$\begin{aligned} x &= a + i\,b \qquad y = c + i\,d \\ xy &= ac - bd + i(ad + bc)\ . \end{aligned}$$

So how does the compiler know that if `x` and `y` are type `std::complex`, then `x*y` means this operation? It's called ***operator overloading***, but let's first take a look at the simpler ***function overloading***.

24 Function Overloading, Optional Arguments and Default Arguments

Function overloading in C++ means you can assign ***different, but related tasks under one function name***. This is nothing new, in math the exponent of a real number x is $\exp(x)$ and the exponent of a complex number c is $\exp(c)$. Even $\exp(M)$ of matrix M is under the same name exp().

C-language has no function overloading, only math functions have been overloaded. Designers of C++ have apparently a different opinion of what's good practise.

> Which function to execute is determined by the argument types and number.
> Reason: The compiler must be able to tell which function version to compile.

For example, `f(int i)`, `f(double x)` and `f(int i, int j)` can be used for overloading.

Function overloading makes two things possible:

- ***Optional arguments*** may or may not be set in the function call.
  For example, `estimate(x)` may do a slightly different calculation than `estimate(x,a)`. No need to call them `estimate1(x)` and `estimate2(x,a)`.

- ***Default arguments***: unless given, the argument has its default value.
  Imagine the boredom and messy program if you always have to call the function
  `myfun(x,y,alpha,beta,gamma);`
  with all five arguments even though you know you in most cases have `alpha=1;beta=5;gamma=14.5;`. In C++ you can set these values as defaults, and call the function simply
  `myfun(x,y);`

Important difference:

- Optional arguments *are used in the function only if they are present.*
  Their presence or absence causes different code to be executed

- Default arguments *are always used in the function* and they must have some values, default or given.

Example: Function arguments a and b are optional

function_overload.cpp

```
// Function integ() can do two different things, depending on arguments
#include <iostream>

double integ(void) {
  std::cout << "no args to integ, integrating from 0 to 1"<<"\n";
  return (1.0); // just test
}

double integ(double & a,double & b) {
  std::cout << "two args to integ, integrating from "<<a<<" to "<<b<<"\n";
  return (2.0);// just test
}


int main(){
  double a=5,b=10;
  integ();
  integ(a,b);
}
```

```
no args to integ, integrating from 0 to 1
two args to integ, integrating from 5 to 10
```

Example: Functions second argument is by default 1.0

function_overload2.cpp

```cpp
// Function fun has a default value 1.0 for the second parameter b
#include <iostream>

double fun(double a, double b= 1.0); //IMPORTANT LINE
int main(){
  double a=5,b=10;
  fun(a);
  fun(a,b);
}
double fun(double a,double b) {
  if(b==1) {
    std::cout <<"a="<< a<<" default case b=1"<<"\n";
  }
  else {
    std::cout <<"a="<< a<<" not default case, b="<<b<<"\n";
  }
  return (1.0);// just test
}
```

```
a=5 default case b=1
a=5 not default case, b=10
```

Default argument is given a value *only* in the function declaration.

This can make the default value hard to find! [81]

[81] Technically, it's possible to set default values in function *definition*, but I strongly advice you not to. Your code will not be.

25 Operator overloading

In section 23.12 we learned, that the complex number multiplication is done correctly by the `*` operator. The way this was achieved is ***operator overloading***: an operator can be told to do a slightly different operation depending on the data type.

> Operator overloading can greatly improve code readability

Operator overloading is something you don't necessarily have to learn yourself, but you will appreciate it if someone has done a good job overloading operators. As an example, without overloading, adding two complex numbers `c1` and `c2` would read something like this:

```
c3 = add(c1,c2);
```

With overloaded `+` operation it reads

```
c3 = c1 + c2;
```

The compiler *does* change the `+` to a function call, but it's out of sight. The code is readable and just like math.

> If you overload an operator, make sure it works as expected

This is related to

> ***The law of least astonishment***: The program should behave in a way that least astonishes the user. [82]

Here is a story of a code that didn't obey that law:

> One day I was surfing the net to find examples on how operators can be overloaded. I came across one that overloaded the `+` operator for complex numbers. Upon testing, I found that the sum `c3 = c1 + c2` really is computed correctly. I thought the implementation was ok and used it in my code. Then I *was astonished* to get wrong results! No, not because `c3` was computed wrong, no-hou. It was because I didn't come to think that the operation `c3 = c1 + c2` was changing also the value of `c1`! It did, and in the code that followed `c1` had a wrong value.

A classic piece of bad code is this attempt to use a macro for squaring:

```
// BAD CODE, *Never* use #define, this is just for educational purposes
#define SQUARE(x) ((x)*(x))
int a=2;
int b=SQUARE(a++); // you would think this squares 2 and *then* increments it by one
                   // No. The result is (2)*(3)=6.
```

There are some rules and limitations to operator overloading:

- Think of operators as functions with one or two arguments, called unary and binary operators, respectively. If your operation needs two arguments, take one *existing binary operator* and overload that.
- You can't invent new operators
  ("`my_clever_new_operator`" is no good)
  Only some of the existing ones can be overloaded:

[82] Steve Oualline, *How Not To Program in C++*.

```
+  -  *  /   %   ^  &  |  ~   !
=  <  >  += -= *= /= %= ^= &=
|= << >> <<= >>= == != <= >= &&
|| ++ -- ,   -> [] () new delete
```

- The order of execution prevails ($*$ is executed before $+$)

You may find tempting to overload an operator to compute powers, because the math form x^y and the function call `pow(x,y)` look so very different. Resist the temptation and use `pow`.

fortran operator overloading
In C++ you *can't invent* your own operators. In fortran you can overload existing ones if you apply it to your own data type, but if you apply it to an existing data type you *must invent* a new operator! Working on 2D tables `double A(:,:),B(:,:)`, where the data type exists, you must invent an operator, such as `.x.` :

```
! fortran
interface operator(.x.)
   module procedure multMatrix
end interface operator(.x.)
...
function multMatrix(lhs,rhs) result(res)
```

and use it as `D=A.x.B.x.C`. It's also simple to create you own matrix type and overload * ,

```
! fortran
type matrix
   double, pointer:: data(:,:)
end type Matrix
interface operator(*)
   module procedure multMatrix
end interface operator(*)
...
function multMatrix(lhs,rhs) result(res)
```

and write `D=A*B*C`. but then you have to dig the `data` from the objects (in fortran it's percent sign, as in `A\%data`).

25.1 Overloading << to print class objects

Example: Operator << overloaded to print class objects

myclass_overload.cpp

```
// How to overload << to print a self-made class object
#include <iostream>
#include <vector>
#include <iterator>
#include <iomanip>
#include <fstream>

class MyClass
{
  double a,b;
  std::vector<double> v;
public:
  // universal initialization
  MyClass(double a_,double b_,std::vector<double> v_): a{a_},b{b_},v{v_}{}
  // overload << for MyClass objects;  make << a friend to grant access toprivate data
  friend std::ostream& operator <<(std::ostream& , const MyClass&);
};

std::ostream& operator<<(std::ostream & os, const MyClass& obj)
{
  using namespace std;
  os<<fixed<<setprecision(8); // some I/O manipulation
  os<<"a="<<obj.a<<" b="<<obj.b<<endl;
  os<<"v=";
  for(auto ele:obj.v) os<<ele<<" ";
  return os;
}

int main()
{
  using namespace std;
  MyClass obj{2.2,3.1,{1.0,2.0,3.0}}; // universal initialization
  cout<<obj<<endl;  // screen output uses overloaded <<
  ofstream out("MyClass.out");
  out<<obj<<endl;  // file output uses overloaded <<
}
```

This overloaded operator `<<` can also write to a file

Without overloading `cout<<testclass` cannot work, because there is no standard way to print the contents of a vector

container. Overloading `<<` is lengthy, but once done you have a clean output of objects of a self-made class without any explicit calls to an output function in your main program. This may sound a small achievement, but in numerics you learn to appreciate clean formulas. Assume you have a matrix A, vectors a, b and c, and hte job is to compute $c = Ab + a$. With overloading, this will at best look likes

```
c=A*b+a;
```

which is easier to decipher than a nested function call,

```
c=vec_add(matrix_multiply(A,b),a);
```

Glad you asked! The overloaded operator `<<` is compiled with a certain logic.
Operation `cout<<obj` for `MyClass obj` means:

- Compiler finds what class the object `cout` belongs (that'll be `ostream`)
- Compiler finds the method `<<` from the class `ostream` and prepares to call it with arguments `(cout,obj)`. This translates to something not completely unlike `(ostream.<<)(cout,obj)`
- Compiler goes through the methods `(ostream.<<)` to find a function with arguments `(ostream&, MyClass&)` or one without the ampersand `&`. It can find one, we just wrote such a method :

  ```
  std::ostream& operator<<(std::ostream &, const MyClass&)
  ```

  The `friend` attribute gives `<<` grants to access the private data members of `obj` (for printing).

26 C++ Standard Library: more algorithms

26.1 std::for_each

The algorithm `std::for_each` performs a given operation to all elements. As arguments you give the beginning, the end and what to do.

Example: `std::for_each` and printing (some) elements of a `std::vector`

vector_print_foreach.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

void doubleout(double y) { std::cout << " " << y; }
int main () {
  std::vector<double> x{1.1,2.2,3.3};
  std::cout << "x vector: \n";
  //  for_each (x.begin(), x.end(), doubleout);
  std::for_each (x.begin(), x.begin()+2, doubleout);
  std::cout<<"\n";
}
```

This applies `doubleout()` to all elements. In general, the applied function can be anything, as long it's declared `anything(double x)`, i.e., it must eat doubles.

26.2 When to use `std::for_each` ?

Now you may wonder what more `std::for_each` has to offer than the range-for loops. After all, you can print all elements of a container more neatly with a range-for loop:

Example: Range-for loop and printing all elements of a `std::vector`

vector_print_range_for.cpp

```
#include <iostream>
#include <vector>
using namespace std;
void doubleout(double y) { cout << " " << y; }
int main () {
  vector<double> x{1.1,2.2,3.3};
  cout << "x vector: \n";
  for(auto elem:x){
    doubleout(elem);
  }
  cout<<endl;
}
```

This is just as good as the `std::for_each` example, and compiles faster!
This is where `for_each` shines:

- Access only part of the elements

  ```
  for_each(x.begin(),x.begin()+2,doubleout()); // output two values from the beginning
  ```

- Access all elements or some elements and use anything a class can contain

  Example: `std::for_each` can do things range-for loops can't

for_each_limited.sum.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

struct LimitedSum {
  void operator()(int i) { if (i > 1) sum += i;}
  int sum{0};
};

int main() {
  std::vector<int> x{1,2,3,4};
  LimitedSum lim = std::for_each(x.begin(), x.end(), LimitedSum());
  std::cout << "Limited sum = " << lim.sum << "\n";
}
```

Notice how we access the member of the class `LimitedSum` after a call to `std::for_each`: The return value of `for_each` is the very same class object that the function object in the argument is. Function objects are discussed more in chapter 27.3.

Without storing the return value we had no access to the data member `sum`. See the details on the next page.

26.3 `std::for_each` in detail

One way `std::for_each` can be implemented is this:

foreach_template.cpp

```
// std::for_each works essentially like this
#include <utility>
template<class Iter, class Func>
Func for_each(Iter first, Iter last, Func f)
{
  while (first!=last) {
    f (*first);
    ++first;
  }
  return std::move(f);
}
```

The third argument `f` can be a class - as long as `f(*first)` is defined! This hints that the name of a class can sometimes be used the same way as a function; they are called function objects.

`for_each` argument third is `class Func f`, and the return value is the same `f`

The "pass-by-value feature"
Notice also how the third argument is passed by value, as `class Func`. This means the argument object `Func` has a one-way ticket, it does not return as an argument. Instead, the function gives it out as a return value. So in as argument, out as a return value. Example: If you send in a function object and change something in that object, you have to use the return value object. The example `numerics/foreach_functor2.cpp` shows the principle, it computes the cosines of elements (done by the function object) and collect their sum as a data member in the object.

Parallel `for_each`
The gcc has the file `.../parallel/for_each.h`, which gives away that `for_each` can be parallelized: All elements are pushed separately though the same function, so why not do it in parallel.

The page Draft of Technical Specifaction tells just how draft the parallel part was in 2015:

Working Draft, Technical Specification for C++ Extensions for Parallelism

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

This kind of put me off and I had to go for coffee. I'm really thankful that things have improved since! We'll return to parallel C++ later. Promise.

26.4 The `std::generate` algorithm

One way to generate values to a container.

Example: Fill a `std::vector` with random numbers

generate_random_vector.cpp

```
#include<iostream>
#include <algorithm>
#include <vector>

double double_random() {
  // poor random numbers 0...1
  return rand()*1.0/RAND_MAX; // avoid int/int !
}
int main() {
  std::vector<double> v(20);
  // fill vector with random numbers
  std::generate(v.begin(), v.end(), double_random);
  // output
  for(unsigned i=0; i<v.size(); ++i){
    std::cout<<i<<"  "<<v[i]<<"\n";
  }
}
```

The next chapter takes a look at why `generate` may be dangerous for random number generation.

26.5 C++ Standard Library algorithms - take care of copies

As the previous example showed, `std::generate` is a nice way to fill a container with random numbers. There is a potential risk, however. The functioning of `std::generate` is equivalent to this:

```
template<class ForwardIt, class Generator>
void generate(ForwardIt first, ForwardIt last, Generator g){
```

```
    while (first != last) {
        *first++ = g();
    }
}
```

The third argument is `Generator gen`, and `Generator` is a class. ***The generator is passed by value, so a copy of g is used***. The compiler creates a copy of `g` using the copy constructor of the class `Generator`.

`std::generate` may copy the third argument.

`std::for_each` may copy of the third argument.

⇒ ***The random number generator is copied, too***.

Why not copy a random number generator (rng)? A rng is just another program. Given a ***seed***, it can produce a nearly random number sequence. The sequence depends only on the seed, the algorithm is deterministic and the same seed gives exactly the same number sequence. That's why it's often called a *pseudo* random number generator. If you copy the rng, you have two identical "number mills". If you compare the two number sequences they produce, you find that within each sequence the numbers are (almost) random, but the numbers in the two sequences are badly correlated. Turning the crank of two similar mills in a simulation code can give you exciting, but wrong, results. [83]

Every generator, that is not giving a constant output, must have a ***state***. In other words, a generator that can give non-constant output has to remember where it is. A clock that cannot keep track of time is a stopped clock. Copying a ***stateful*** generator has to be done with care.

26.6 C++ Standard Library algorithms - stateful objects and `std::ref`

Algorithms `std::for_each()` and `std::generate()` are not useless in context of stateful objects. There is a simple remedy to the copy problem: For stateful objects, use a *reference wrapper*

[83] Ah. why not give the generator a new seed and get a new random sequence, different from the other? That way you would have several number mills with different seeds. Bad idea. The rng algorithms have been tested to produce a number sequence random only in relation to numbers within the same sequence. The basic problem is that the *seeds* that determine the sequences are not random. Ok, why not run one mill with a "mother seed" to give you seeds for many rng's? That too, has not been tested to give sufficiently random results. Algorithmic generation of (pseudo) random numbers is a tricky thing.

```
std::vector v(100);
std::generate (v.begin(), v.end(), std::ref(gen));  // gen is stateful object
```

`std::ref` is a helper function to generate a std::reference_wrapper, meaning (see `std::ref` or `std::reference_wrapper`)
`std::ref(10)` is `std::reference_wrapper<int>`

Example: Fill a `std::vector` with unique id numbers from an id generator.

generate_id_vector.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

class IdGen{
  int id;  // object's state
public:
  IdGen(): id(0) {std::cout<<"constructed an IdGen\n";}; // constructor
  int operator()(void) {return(id++);} // functor
}idgen;
int main() {
  std::vector<int> v(20),w(20);
  std::generate(v.begin(), v.end(), std::ref(idgen)); // try these *without* std::ref
  std::generate(w.begin(), w.end(), std::ref(idgen));
  std::cout<<"v = \n";
  for(auto x:v) std::cout<<x<<" ";
  std::cout<<"\n";
  std::cout<<"w = \n";
  for(auto x:w) std::cout<<x<<" ";
  std::cout<<"\n";
}
```

27 A few things that may speed up your code

You may be interested to check out Wiki: C++ Performance improving features. If you are a good programmer, find out what "Move Semantics" and "Perfect Forwarding" mean in C++, and dive into the pool of "Smart Pointers". Read books and

postings by Scott Meyers!

27.1 noexcept: no-throw quarantee

Recommendation: Use frequently in numerical code

If your function never throws an exception, the keyword noexcept may let the compiler to optimize your code more. [84] If it does throw, your code will terminate - Ha, you lied! Still, noexcept is one of the very latest features of C++, so don't believe just any blog posts, just give it a try. Usage:

```
void myfunction() noexcept
{
  //...
}
```

27.2 constexpr: compile-time constant expressions

Recommendation: Use frequently in numerical code

Not quaranteed to give any speedup, but an interesting concept. Computation of factorials recursively can be traditionally done like this:

```
// Common way to compute a factorial
int factorial ( const int n ) {
  int fact = 1 ;
  if ( n <= 1 )
    return 1 ;
  else
```

[84] We return to throw-catch in chapter 33.

```
    fact = n*factorial ( n - 1 ) ; // recursion
  return fact ;
}
```

C++ is able to perform tasks during compilation, coded as ***template metaprogramming*** or with `constexpr`.

```
long int constexpr factorial (int n)
{
    return n > 0 ? n * factorial( n - 1 ) : 1; // one return statement is allowed
}
int main()
{
   constexpr long int fact13=factorial(13);  // 13! is computed in compile time
  ...
}
```

If you look at the assembly code after compilation (`g++ -S code.cpp` and look at `code.s`) you see that indeed fact13 is set equal to 6227020800.

The `constexpr` qualifier tells the compiler, that the expression can be be evaluated at compile time.

Evaluation of numerical constants defined with the `const` qualifier may be deferred to run-time. One way to see the difference is to add a `std::cout<<...` in the `constexpr` function. If the result is stored to a `const`, the code will compile, but if the result is stored to a `constexpr` the code won't compile and gives `error: call to non-constexpr function std::basic_ostream`.

Remark: The `const` qualifier means that, once initialized, the value cannot be changed. `constexpr` is more than `const`, it's a *constant expression*. Functions declared `constexpr` can compute the allocator template parameter of e.g. `std::list<>`
(see stackoverflow:difference-between-constexpr-and-const).

If you know C++ metaprogramming and Haskell, you might be interested to read What Does Haskell Have to Do with C++?

The factorial can be computed with template metaprogramming - just for demonstrative purposes:

factorial_meta.cpp

```
// Template metaprogram
#include <iostream> // just for testing
template<int N>
struct Factorial {
  static const long value = N * Factorial<N-1>::value;
};
template<>
struct Factorial<1> {
  static const long value =1;
};
int main()
{
  const long fact13 = Factorial<13>::value;
  std::cout<<fact13<<std::endl;
}
```

This is as simple as template metaprograms get. The key is to follow how templates cause instantiation of other templates:

1. The number 13 is fed to `Factorial` as a template argument
2. The compiler ***instantiates*** the template `Factorial<13>`
3. `Factorial<13>` tells to instantiate the template `Factorial<12>`, which tells to instantiate `Factorial<11>` and so on.
4. Finally, `Factorial<1>` is instantiated, and the compiler notices that there is a ***template specialization*** corresponding to argument `<1>`: the recursive instantiation ends and the `value` is set to 1.

Compilers can't handle very deep recursive instantiation, and the data type `long` can't hold large factorials anyhow.[85]

[85] `long` is the same as `long int`. There's also `long long`.

27.3 Function objects (functors)

Recommendation: Use frequently in numerical code

Function objects, or ***functors*** for short, are popular in numerics. You can define a function in a class that is not a method (member function). This function is called when the class name is used as a function name.

Example: A function object to compute $\sin(x)$, $\cos(x)$ and $\tan(x)$

functor.cpp

```
// Function object - functor
#include <iostream>
#include <cmath>
#include <functional>
class TrigFuns{
public:
  void operator()(double x) {
    std::cout<<"x="<<x<<std::endl;
    std::cout<<"sin(x)      cos(x)     tan(x)\n";
    std::cout<<sin(x)<<" "<<cos(x)<<" "<<tan(x)<<std::endl;
  }
};
int main()
{
  TrigFuns trig;
  trig(20.0);
  std::invoke(TrigFuns(),10.0);
}
```

Here `trig` is an object, but used as if it were a function - hence it's a function object. I used the `std::invoke` (see 27.5) to execute the "bare" function object.

Why use a function object? Why not an ordinary function?

1) Function objects are not passed as pointers, so the compiler can easily *inline* them (insert to its place). It's simple to wrap the function in a class and make it a function object,

functor_to_function.cpp

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

// Functor:
class Func{
public:
  double operator()(double x) { return sin(x);}
};

// free function:
void apply(Func g,double x) {std::cout<<g(x)<<std::endl;}

int main()
{
  Func f;
  apply(f,10.0);
}
```

2) A class can contain data members, such as counters, accessible only to class methods. A function object can easily perform complicated tasks. For example, a function object can compute the cosine of all input and, simultaneously, compute the sum of these cosines - this is done in `numerics/forarch_functor2.cpp`.

Example: A function object used with `std::for_each`

foreach_functor.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

class TakeCos{
public:
  void operator()(double& x){ x=cos(x); } // function object
};

int main () {
  std::vector<double> x{1.1,2.2,3.3};
  std::cout << "vector x      : ";
  for(auto e:x) std::cout<<e<<" ";
  std::cout<<"\n";
  for_each(x.begin(), x.end(), TakeCos());
  std::cout << "vector cos(x) : ";
  for(auto e:x) std::cout<<e<<" ";
  std::cout<<"\n";
}
```

If the task is this simple, or it's supposedly used only here, it's more convenient to use a ***lambda***, introduced later in chapter 35.

27.4 Five ways to pass a function to a function

Passing a a function to a function is a deceivingly innocent task. In Python functions are just like any other objects and you can kick them around as you wish. In C++ things are lot more complicated. The goals are:

> The compiler should be able to optimize as much as possible
> The code should not be too verbose; less is better
> Preferably avoid naked pointers

Let's first consider what ***callable objects*** - things that can be used as functions - we have:

- Functions and pointers to functions
- Objects created by `std::bind` (section 28.1)
- Lambdas (section 35)
- Function objects (classes that overload the function call operator (), section 27.3)

I give a few ways and comment on each. Examples of each style is in the file `numerics/function_to_function_speed_test.cpp`.

1) Pass the function as a ***function pointer***:

```
double integrate( double (*f) (double), double a, double b){
 // integrate function f(x) from a to b
}

// usage:
res = integrate(f,1.0,2.0);
```

Function pointers have been around quite a while, but they are nevertheless ***efficient***. The ***run-time overhead is very small***, but the applicability is ***limited***; obviously you can't pass all callable objects as a pointer. You can pass a non-capturing lambda thusly:

```
auto f = [](double x) {return x*2;};  // lambda with the name f, captures nothing
double (*ptr)(double) = f;               // ptr points to the lambda
// ptr(x) is the same as f(x)
```

but you can't get a pointer to a capturing lambda,

```
int par = 1.0;
auto f = [par](double x) {return par*x*2;};  // lambda with the name f, captures par
double (*ptr)(double) = f;
// gcc gives an error:
// error: cannot convert
// 'main()::<lambda(double)>'to 'double (*)(double)' in initialization
```

[86]

2) Pass the function using the class template `std::function`

Class template `std::function` is a general-purpose polymorphic function wrapper.

Recommendation: use `std::function` only if your function is determined at run-time

```
double integrate(const std::function<double(double)> &f, double a, double b){
  // integrate function f(x) from a to b
  :::
}
```

[86] There exists code that converts capturing lambdas to function pointers, for example Viorel @wordpress.com.

```
    // usage:
    res = integrate(f,1.0,2.0);
```

If the function has been declared earlier, you can let the compiler deduce the types (now `double(double)`)

```
    double integrate(const std::function<decltype(f)> &f, double a, double b)
```

`std::function` is a general-purpose function wrapper with ***a significant run-time overhead*** and it may do dynamical allocation. It can be a bit slow, but it's omniponent. It can take also member function pointers.

3) Pass the function using a ***function template***.

Recommended method:
Use a function template if your function is known at compile-time.

```
template <typename T>
double integrate (T&& f, double a, double b) {
  ::: // use f() as usual
}

// usage:
integrate(f, 1.0, 2.0);
```

Recent compilers can ***inline*** this if it's profitable, and ***any callable object can be passed without any overhead***. The generated code is ***very fast***, because the compiler knows the passed function at compile-time.

It also works with lambdas, for example

```
integrate([](double x){return 1.0/(1.0+x);} , 0.0, 10.0);
```

also with a capturing lambda,

```
double p=1.0;
integrate([p](double x){return p/(p+x);} , 0.0, 10.0);
```

4) Pass the function as a ***template parameter***.
To my eye the code looks strange, but, nevertheless, it's valid C++:

```
template<double f(double)>
double integrate(double a, double b){
::: // use f() as usual
}

// usage:
integrate<f>(1.0, 2.0);
```

5) Pass the function using a ***function_view, function_ref or "an impossibly fast C++ delegate"***.

There are auxiliary C++ codes that try to provide a generic, callable object view. The point is that `std::function` owns the callable, meaning it has to make a copy, so a code that don't own the callable should be more efficient.

The basic idea has been re-invented and coded several times over. There is one in Yakk @Stackoverflow, in LLVM compile infrastructure project @github, and in *The impossibly fast C++ delegates* by Sergey Ryazanov and fast delegates by Don Clugston.

One implementation of a function view is by Vittorio Romeo. However, this is not without flaws (it may view an already destroyed temporary), and *Implementing function_view is harder than you might think* by Jonathan implements a safer code under the name `function_ref`.

27.5 C++17 calls with `std::invoke`

Probably `std::invoke` won't speed up you code, but it facilitates coding. It provides a coherent way to call member functions, lambdas, function objects etc. The time-honored way to call a function object is

```
struct PrintNum {
   void operator()(int i) const {
       std::cout << i << '\n';
   }

pn = PrintNum();
pn(5);
// PrintNum(5); // error, no matching function for call to PrintNum::PrintNum(int)
```

With `std::invoke`, there is no need to create a `PrintNum` object,

```
struct PrintNum {
   void operator()(int i) const {
       std::cout << i << '\n';
   }

std::invoke(PrintNum(),5);
```

You may wonder: "Why do we need `std::invoke`, we can call these callables without it?". True, but with `std::invoke` you don't have to know what kind of callable comes in. You can `std::invoke` and it will be called. Especially calling a member function is tricky.

You may also wonder: "Are `std::function` and `std::invoke` related?". The C++ reference has examples of both, I just put them to the same code to make comparison easier; see `std_invoke.cpp`; it's too long to fit here.

27.6 Cache data

Don't recompute the same data over and over again. Use *cached* values. This ***memoization*** became familiar in the Python part of these lecture notes, where it was implemented as a decorator. C++ has no decorators (at least, not yet), but surely memoization is possible. There is no standard way, but universal memoization is possible; see for example here and here.

27.7 Use `emplace_back()` instead of `push_back()`

Recommendation: Use always if available

The difference is that

- `push_back()` will contruct the container element someplace else and move it to the end of `std::vector`
- `emplace_back()` will contruct the container element *in-situ* at the end of the `std::vector`

The extra moving done by `push_back()` takes longer, so the latter simply has to be faster. The difference used to be larger, but compilers evolve: In 2017 the following test run gave timings `push_back()`:151 ms vs. `emplace_back()`:134 ms. Nothing spectacular in this case. [87]

It's important that `emplace_back()` can **forward parameters to contructor**! You don't have to construct the value of the `std::vector` element, you just tell *how* it should be made.

Notice how one aliases `std:vector`,

```
template <typename T>
using vec = std::vector<T>;
```

[87] In an older version I recycled the same container `v` after calling `v.clear()`: Not a fair comparison.

This is called ***templated alias***, works for templates like `std::vector`.

Example: `push_back()` vs. `emplace_back()` speed test

emplace_vector.cpp

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

// templated alias to vector<>
template <typename T>
using vec = std::vector<T>;

struct MyThing{
  int idat;
  vec<double> x;
  MyThing(int idat_, vec<double> x_) noexcept : idat{idat_},x{x_}{}
};

int main(){
  using clock = std::chrono::steady_clock;
  using std::chrono::milliseconds, std::chrono::duration_cast;
  const int N=1000000;
  auto t0 = clock::now();
  std::vector<MyThing> v;
  for (auto i=0;i<N;++i){
    v.push_back(MyThing(i,vec<double>{1.0,2.0,3.0,4.0,5.0,6.0}));
  }
  auto t1 = clock::now();
  auto d = duration_cast<milliseconds>(t1-t0);
  std::cout <<"vector push_back took: "<<d.count() << " ms\n";

  std::vector<MyThing> w;
  t0 = clock::now();
  for (auto i=0;i<N;++i){
    w.emplace_back(i,vec<double>{1.0,2.0,3.0,4.0,5.0,6.0});
  }
  t1 = clock::now();
  d = duration_cast<milliseconds>(t1-t0);
  std::cout <<"vector emplace_back took: "<<d.count() << " ms\n";
}
```

Example: `emplace_back()` can construct with parameters.

emplace_parameter_arguments.cpp

```cpp
//
// emplace_back with parameter arguments
//
#include <iostream>
#include <cmath>
#include <vector>

struct MyClass{
  double value;
  MyClass(const double & par,double phi) noexcept : value{par*sin(phi)} {}
};

int main(){
  const int N=1000000;
  const double par=3.866;
  std::vector<MyClass> v;
  for (auto i=0;i<N;++i){
    v.emplace_back(par,i*M_PI/N); // forward parameters to constructor
  }
  std::cout<<"first 10 values:\n";
  for (auto it=v.begin();it<v.begin()+10;++it) std::cout<<(*it).value<<" ";
  std::cout<<"\n";
}
```

27.8 Prefer the methods of containers over generic algorithms

Container-specific algorithms can take advantage of the container properties. Sometimes generic algorithms are not available at all. For example, the `list` container has no random access iterator, so `std::sort` won't work. Instead, there is a `sort` *method* in `list`:

```
# include <iostream>
# include <list>
int main(){
   std::list<int> mylist={1,5,3,2,6,4} ;
   mylist.sort() ; // calls the sort member function
   for (auto el: mylist) {cout<<el<<endl;}
}
```

27.9 Expression templates *(read on spare time)*

This chapter gives you an idea how complicated it is to master C++ and why I never will.

The take-home message is: ***use good math libraries.***

The basic idea behind expression templates is to use operator overloading to build parse trees

Expression templates were used by Todd Veldhuizen in his matrix library Blitz++ in mid 90's, ever since applied in practically all numerical C++ libraries (see Todd Veldhuizen: "Techniques for Scientific C++"). [88] Early expression templates were only able to cure one bad C++ side effect, namely that ***one should not create temporaries in every corner***. [89] *But this*

[88] Blitz++ still works behind scenes: Last time I looked, SciPy (Scientific Python) module used parts of the Blitz++ library.

[89] `Valarray` tries to avoid intermediates (temporaries) using "proxy" objects. Most libraries prefer the expression template technique due to its generality.

only taught C++ codes to behave the way any reasonable code should. Since then we have learned quite a bit and realized that avoiding temporaries is not the only essence in speed.

What's this "avoid temporaries"-fuss all about? C++ has this wonderful thing called operator overloading. The problem is that if you apply is straightforwardly, the compiler easily creates temporaries. Consider how the addition operator can be translated to function calls:

```
D = A+B+C means add(A,B)+C, so set temporary M=add(A,B), finally set D = add(M,C)
```

If `A,B,C` and `D` fill the fast ***cache memory***, then the intermediate result `M` "drops" something out of cache to slow RAM - 10x slower or more. Another example is adding three vectors, but using only one component,

```
D[0] = (A+B+C)[0];
```

You can easily expand this the most effective way,

```
D[0] = A[0]+B[0]+C[0];
```

but if you're not careful, the compiler adds up the whole million elements to make `A+B+C`, puts it to `D` and only *then* looks for `D[0]`!

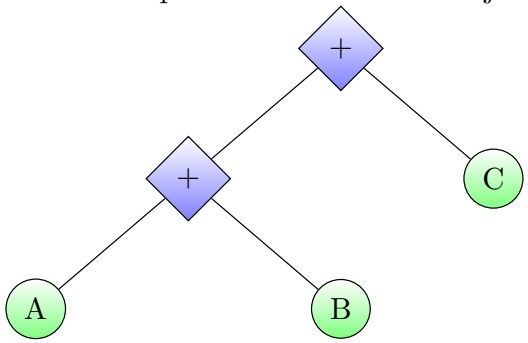
Continue reading if you the previous discussion didn't drop out of your cache.

Templates can help to avoid temporaries. If a compiler meets a line of code it can't immediately recognize, it starts looking for a suitable template ("model"). Finding one, it ***instantiates*** the template, i.e. brings it alive. This template can itself instantiate other templates and so on. Apart from instantiating another template, a template can instantiate another copy of itself (recursive template). This recursion is in the heart of many expression templates. For identification, templates have ***template parameters***, which tell exactly what kind of model to instantiate: [90]

[90] Excuse me for mixing template parameters and template arguments. I guess the former is a placeholder in the template definion and the latter is what's put there in a certain instantiation? I honestly don't know any better.

```
template <parameters>
   blaablaa(){...}
```

Letting templates instantiate new templates lets one express the addition of objects as a tree:



Let `A,B,C,D` be class `Array` objects. Then the tree could be

```
Array A, B, C, D;
D = A + B + C;
first + should translate as expression X<Array,plus,Array>() + C;
second + should translate as expression X<X<Array,plus,Array>,plus,Array>();
```

Here `X` is a vertex for objects "left" and "right" the operator "plus" in between. The code `advanced/recursive_template.cpp` shows how such a tree structure `X<X<Array,plus,Array>,plus,Array>` is created. The example `advanced/expression_template.cpp` computes the sum of elements in a `std::vector` using expression templates.

Expression templates can delay the evaluation until the "=" sign is reached (lazy evaluation)

The compiler goes through the whole tree at compilation time, instantiating vertices `X`, as many as needed. Reaching the end it knows exactly what is needed (such as only `D[0]`).

Operations should return expressions, not results.

Expressions are something that can be further manipulated during compilation.

Compiling the code with a recursive template essentially generates changes to the code, or "the code changes itself". This is the essence of ***template metaprogramming***: templates can generate new code during program compilation. Alas, compilation takes longer.

28 Generation of (Pseudo) Random Numbers

A random number generator is a program, that, given a seed (say 23525176471263), produces a sequence of numbers that appears random. It's like a number mill: In goes the seed and out comes flour of random numbers. Always the same output.

Stages to invoke C++ random number generation @ cppreference:

1) Include the headers

```
#include <random>
#include <functional>  // if you use std::function
```

2) Choose the random number algorithm

```
std::mt19937 gener; // Mersenne twister
```

Only the given name `gener` appears from now on, so editing this single line is sufficient to switch to another generator (`linear_congruential_engine`, `subtract_with_carry_engine`),

```
std::linear_congruential_engine gener;
```

3) Choose from available distributions - there are many
`uniform_real_distribution` (parameters: start and end)
`normal_distribution` (parameters: mean value and variance).
Uniform distribution `unif_dist` U[0,1) is created like this (the name `unif_dist` is my own):

```
    std::uniform_real_distribution<double> unif_dist(0,1);
```

and a normal distribution like this (again the name `normal_dist` is my own):

```
    std::normal_distribution<double> normal_dist(0,1);
```

4) Give the generator a seed **only once** (the suffix `u` means `unsigned`)

```
  gener.seed(4835267u); // same sequence every time you run the code
```

or pick the seed from the system clock

```
gener.seed(static_cast<uint_fast32_t> (std::time(0)); // at least 32 bits
// different sequence every time you run the code (if time(0) has changed)
```

or rely on `std::ramdom_device` (not without problems, see a discussion in cpps-random_device.html)

```
   gener.seed(std::random_device{}());
```

5) EITHER (i) `std::bind` the generator and the distribution together:

```
auto normal_random = std::bind(normal_dist, gener); // do this once
```

and use it like this to get a single random number `r`

```
double r = normal_random();
```

OR (ii) get the random number directly from a call to `distribution(generator)` :

```
double r = normal_dist(gener);
```

28.1 Simplify function calls with `std::bind`

Before continuing with random numbers, lets look what `auto normal_random = std::bind(normal_dist, gener)` did: It created a function `normal_random()` that means "call `normal_dist` with argument `gener`".

Example: `std::bind` is versatile

bind_example.cpp

```
// How to use bind to
// a) change the order of arguments
// b) turn a 3 argument function to a 2 argument function
//
#include<iostream>
#include <functional>
void  f(const double & x,const double & y,const double & z)
{
  std::cout<<"called f with arguments "<<x<<" "<<y<<" "<<z<<std::endl;
}
int main()
{
  using namespace std::placeholders;  // for _1, _2
  f(1.1,2.2,3.3);
  // a) change the order of arguments with bind:
  auto invf=std::bind(f,_3,_2,_1);   // bind return type is std::function
  invf(1.1,2.2,3.3);
  // b) create a two-argument function from f
  auto g=std::bind(f,_1,_2, 333.3); // bind 3rd argument to fixed value 333.3
  g(10.1,20.2);
}
```

The next examples are a set of easy-to-use helper functions to initialize a generator and get uniform distribution (`unirand()`), normal distribution (`gaussrand` and `gaussrand2()`) or exponentially distributed (`exprand()`) random numbers. The goal was to hide all inconveniences to `numerics/random.cpp`, so that the bread-and-butter code is clean and simple,

```
double random = myrandom::RNG::gauss()
```

Example: C++ random number generation

random.hpp

```
#ifndef RANDOM_HPP
#define RANDOM_HPP
#include <random>

namespace myrandom{
  class RNG{
    std::mt19937_64 gen;     // generator
    uint_fast64_t myseed;    // seed
  public:
    RNG(void);
    RNG(uint_fast64_t);
    uint_fast64_t get_seed();
    double unif();
    double gauss();
    double gauss(double, double);
    double normal();
    double normal(double, double);
    double exp();
  };
}
#endif
```

random.cpp

```cpp
#include "random.hpp"
#include <functional>
#include <iostream>

// constructors:
myrandom::RNG::RNG(): myseed{std::random_device()()} {
  std::cout<<"RNG: seed from std::random_device\n";
  std::cout<<"RNG: seed = "<<myseed<<"\n";
  gen.seed(myseed);
}

myrandom::RNG::RNG(uint_fast64_t seed_) : myseed{seed_}{
  std::cout<<"RNG: custom seed "<<seed_<<"\n";
  gen.seed(seed_);
}


// member functions:


uint_fast64_t myrandom::RNG::get_seed(){
  return myseed;
}

double myrandom::RNG::unif(){
  static std::uniform_real_distribution<double> unif_dist(0,1);
  return unif_dist(gen);
}
double myrandom::RNG::gauss(){
  static std::normal_distribution<double> norm_dist(0,1);
  return norm_dist(gen);
}

double myrandom::RNG::gauss(double a, double s){
  static std::normal_distribution<double> norm_dist(a,s);
  return norm_dist(gen);
}

double myrandom::RNG::normal() {return gauss();}
double myrandom::RNG::normal(double a, double s) {return gauss(a, s);}

double myrandom::RNG::exp(){
  static std::exponential_distribution<double> expo_dist;
  return expo_dist(gen);
}
```

28.2 Return to `std::generate`: the member function predicament

In chapter 26.4 we saw how to use the `std::generate` algorithm and in chapter 26.5 we found a way to use standard library algorithms without copying the generator function object that will be called. In the examples so far, these generator functions have all been free functions - in reality that's rarely the case. I'm going to use `IdGen` as a test class, what I have in mind is a C++ built-in random number generator in a member function of the self-made class `RNG` in files `random.hpp` and `random.cpp`.

Working with a *class*, there are several ways to use `std::generate`

- **If the class `IdGen` defines a function object)**, you can
  - Use the function object directly as a generator function,

```
std::generate(v.begin(),v.end(), IdGen());
```

  - create an object of the class and use the as a generator function,

```
IdGen idgen;
std::generate(v.begin(),v.end(), idgen);
```

    If you want to generate ***unique*** id numbers, you have to prevent `std::generate` from copying the generator. You can't do that on `IdGen()`, only to the object `idgen`:

```
std::generate(v.begin(),v.end(), std::ref(IdGen())); //Error
std::generate(v.begin(),v.end(), std::ref(idgen));  //OK
```

    The class `IdGen` does not have a unique integer `id` to keep track off.
    For that unique `id` you need an object of the class `IdGen` and prevent copying it.

This is demonstrated in the following code.

generate_functor.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

class IdGen{
  int id;
public:
  IdGen(): id{0}{std::cout<<"constructed an IdGen with initial id="<<id<<"\n";}
  int operator()(void){return (id++);};
};

int main(){
  std::vector<int> v(5),w(5);
  std::cout<<"call std::generate with class name IdGen()\n";
  std::generate(v.begin(), v.end(), IdGen());
  for (auto vv:v) {std::cout<<vv<<" ";}
  std::cout<<"\n";
  std::generate(w.begin(), w.end(), IdGen());
  for (auto ww:w) {std::cout<<ww<<" ";}
  std::cout<<"\n";

  std::cout<<"call std::generate with std::ref(idgen), class IdGen object idgen\n";
  IdGen idgen;
  std::generate(v.begin(), v.end(), std::ref(idgen));
  for (auto vv:v) {std::cout<<vv<<" ";}
  std::cout<<"\n";
  std::generate(w.begin(), w.end(), std::ref(idgen));
  for (auto ww:w) {std::cout<<ww<<" ";}
  std::cout<<"\n";
}
```

- **If you want to use a member function of the class `IdGen` as a generator function** you need to dig deeper. Consider this variant of `IdGen`:

```
  class IdGen{
     int id;
   public:
     IdGen(): id{0}{std::cout<<"constructed an IdGen with initial id="<<id<<"\n";}
     int get_id() {return id++;}
};
```

You need an object, e.g. `IdGen idgen;`, and that object has a working id generator `idgen.get_id()`. Using that in a `std:generate` should be easy,

```
  std::vector<int> v(5):
  std::generate(v.begin(),v.end(), idgen.get_id);  // Error
```

but it doesn't compile: `error: invalid use of non-static member function 'int IdGen::get_id()'`. Since `std::genera` can take in function objects, we need a way to turn the member function `get_id()` into one.

First working solution: use `std::bind` Several web pages suggest this,

```
   IdGen idgen;
   generate(v.begin(), v.end(), std::bind(std::mem_fun(&IdGen::get_id), &idgen));
```

What an ugly beast! I found that in gcc also this works, without `std::mem_fun`,

```
IdGen idgen;
generate(v.begin(), v.end(), std::bind(&IdGen::get_id, &idgen));
```

Notice also: no `std::ref()` needed. **Details - only if you are in for an adventure:**
First of all, `std::mem_fun(&IdGen::get_id)` is supposed to turn the member function `IdGen::get_id()` to a function object. Great, but it's not quite enough,

```
generate(v.begin(), v.end(), std::mem_fun(&IdGen::get_id)); // Error
```

won't compile. The algorithm wants a *directly callable* function object, one that can be called simply `f()`, but `std::mem_fun` returns a *pointer* to the member function which is not directly callable. In other words, the call that is tried is `(&IdGen::get_id)()`, which is ill-formed:

```
int id = (&IdGen::get_id)(); // Error
```

You need `std::bind`, which creates a call wrapper and binds the member function pointer made by `std::mem_fun` to the specific object:

```
int id = bind(mem_fun(&IdGen::get_id), &idgen)(); // OK
```

These problems have been recognized (Proposal: Make Pointers to Members Callable). Note added: There's also `std::mem_fn`. [91]

[91] Unwillingness to break old code makes programming language vocabulary grow in every "improvement". In spoken languages, youngsters don't give a _ (here _ is placeholder in `std::placeholders` for a four-letter word) whether oldtimers understand them or not! Result: effective word recycling.

Example: A member function as a generator function in `std::generate`

generate_member_function.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>
class IdGen{
  int id;
public:
  IdGen(): id{0}{std::cout<<"constructed an IdGen with initial id="<<id<<"\n";}
  int get_id() {return id++;}
};

int main(){
  IdGen idgen;
  std::vector<int> v(5),w(5);
  auto gen = std::bind(&IdGen::get_id, &idgen);
  std::generate(v.begin(), v.end(), gen);
  for (auto vv:v) {std::cout<<vv<<" ";}
  std::cout<<"\n";
  std::generate(w.begin(), w.end(), gen);
  for (auto ww:w) {std::cout<<ww<<" ";}
  std::cout<<"\n";
}
```

Output:

```
0 1 2 3 4
5 6 7 8 9
```

Second working solution: use a lambda

Lambdas are your best friends every time you need to use standard algorithms.

Recommended way:

generate_member_function_lambda.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

class IdGen{
  int id;
public:
  IdGen(): id{0}{std::cout<<"constructed an IdGen with initial id="<<id<<"\n";}
  int get_id() {return id++;}
};

int main(){
  IdGen idgen;
  std::vector<int> v(5),w(5);
  auto gen = [&idgen](){return idgen.get_id();};
  std::generate(v.begin(), v.end(), gen);
  for (auto vv:v) {std::cout<<vv<<" ";}
  std::cout<<"\n";
  std::generate(w.begin(), w.end(), gen);
  for (auto ww:w) {std::cout<<ww<<" ";}
  std::cout<<"\n";
}
```

Example: Using `std::generate` with class `RNG` random number generator

generate_random_vector_lambda.cpp

```cpp
// compile: g++ generate_random_vector_lambda.cpp random.cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <fstream>
#include <iomanip>
#include "random.hpp"

int main(){
  std::vector<double> v(5), w(5), big(1e5);
  myrandom::RNG rng ;//(23142566);
  std::cout<<"seed is "<<rng.get_seed()<<"\n";
  std::generate(v.begin(), v.end(), [&rng](){return rng.unif();});
  for (auto vv:v) {std::cout<<vv<<" ";}
  std::cout<<"\n";
  std::generate(w.begin(), w.end(), [&rng](){return rng.unif();});
  for (auto ww:w) {std::cout<<ww<<" ";}
  std::cout<<"\n";
  std::generate(big.begin(), big.end(), [&rng](){return rng.normal(3.0,.1);});
  std::ofstream out("big");
  for (auto b:big) {out<<std::fixed<<std::setprecision(30)<<b<<"\n";}
  out.close();
}
```

29 Boost and Ordinary Differential Equations (ODE's)

Boost library has served as a test bench for ideas, some have made their way to the C++ standard. Boost extends the C++ standard library with ***algorithms, special functions, differential equation solvers and many more.***
The documentation is

http://www.boost.org/doc/libs/

and the math part is in

http://www.boost.org/doc/libs/?view=category_Math

The examples are thorough, e.g. how to use your own vector type is shown here (version 1.65.1). As long as your data structure conforms with a standard or a Boost container, you can just throw it into Boost. Especially the library odeint for solving ordinary differential equations is comprehensive.

Boost is almost entirely a ***header-only library*** : nothing to compile. If you have admin rights, installing and using Boost is trivial using he package manager. If not, just download boost and unpack it to, say, directory `$HOME/boost`. Some ODE sample codes are now in `$HOME/boost/libs/numeric/odeint/examples/`, and can be translated like this (assume bash shell):

```
$ export BOOSTDIR=$HOME/boost
$ export SAMPLES=$BOOSTDIR/libs/numeric/odeint/examples/
$ g++ -std=c++11 -O3 -I$BOOSTDIR -I$SAMPLES $SAMPLES/chaotic_system.cpp
```

Boost has a seasoned library for solving ODEs. A typical ODE solution goes along this route:

1. If your equation is n, that is second order or higher, you first separate it to n coupled first order differential equations (details in 34.3).

2. Choose the integration algorithm, the ***stepper***, that solves the next point, given a starting point:

   - **Fixed-step-size routines**: Simple and fast, accuracy of the solution is your responsibility.
   - **Adaptive-step-size routines**: Try to reach an accuracy goal (absolute and relative error limit).

boost_ode_simple.cpp

```cpp
#include <iostream>
#include <boost/numeric/odeint.hpp>
#include <fstream>

using state_type = std::vector<double> ;
using stepper_type = boost::numeric::odeint::runge_kutta4<state_type> ;

/* Solves x''(t) = -x(t) - gam*x'(t), split to coupled
    x'(t) = y(t)
    y'(t) = -x(t) - gam*y(t)
   notation: x(t)=x[0], y(t)=x[1], x'(t) = dxdt[0], y'(t) = dxdt[1]
*/
void harmonic_oscillator(const state_type &x, state_type &dxdt, const double /*t*/)
{
  const double gam=0.15;
  dxdt[0] = x[1];
  dxdt[1] = -x[0] - gam*x[1];
}


int main()
{
  stepper_type stepper;
  state_type x ={1.0,2.0} ; // initial values, x(0) = 1, x'(0) = 2
  // integrate all the way to final time:
  //integrate_const( stepper , harmonic_oscillator , x , 0.0 , 100.0 , 0.01 );
  //cout<<"final point "<<x<<endl;

  // use method do_step to follow the solution step by step
  std::ofstream out("ode.dat");
  const double dt = 0.01;
  for( double t=0.0 ; t<100.0 ; t+= dt )
    {
      stepper.do_step( harmonic_oscillator , x , t , dt );
      std::cout<<t<<" "<<x[0]<<" "<<x[1]<<"\n";
      out<<t<<" "<<x[0]<<" "<<x[1]<<"\n";
    }
  out.close();
}
```

boost_ode_adaptive.cpp

```cpp
// solve y'(t) = -t*y ,  condition y(0)  = -2
// Adaptive integration using Boost::numeric::odeint
// compile :
// g++ --std=c++11 boost_ode_simple.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <boost/numeric/odeint.hpp>

using state_type= std::vector<double>;
using stepper_type= boost::numeric::odeint::runge_kutta_cash_karp54<state_type>;

namespace my{
  using std::cout;
  void system(const state_type& y, state_type& dydt, const double t)
  {
    dydt[0] = -t*y[0];
  }
  void output(double t, double y, double exact){
    cout<<std::fixed<<std::setprecision(16);
    static bool first=true;
    if(first) {
      cout<<std::setw(20)<<"t"<<std::setw(20)<<"Boost solution";
      cout<<std::setw(20)<<"exact solution"<<std::setw(20)<<"error\n";
      first = false;
    }
    cout<<std::setw(20)<<t<<std::setw(20)<<y<<std::setw(20)<<exact<<std::setw(20)<<y-exact<<"\n";
  }
}

int main()
{
  const double e_abs=1e-15,e_rel=1e-15; // absolute and relative error goal
  auto stepper = boost::numeric::odeint::make_controlled<stepper_type>(e_abs, e_rel) ;
  state_type y ={-2.0} ; //condition y(0)=-2 gives y(t) = -2.0*exp(-0.5*t*t)
  const int n=50; // solve 50 points
  const double t1 = 0.0, t2=10.0;
  const double dt = (t2-t1)/(n-1);

  for(int i = 0; i<n ; ++i)
    {
      auto t = t1+i*dt;
      auto exact = -2.0*exp(-0.5*t*t);
      my::output(t,y[0],exact);
      auto steps =integrate_adaptive(stepper, my::system, y, t, t+dt, 0.01);
      // cout<<"steps "<<steps<<endl; // how many sub-division steps were needed
    }
}
```

Such a simple `my::system` begs to be replaced with a lambda:

```
auto steps =integrate_adaptive(stepper,
           [](const state_type& y, state_type& dydt, const double t)// lambda
           {dydt[0] = -t*y[0];}                                      // lambda
           , y, t, t+dt, 0.01);
```

This same ODE will be solved using GSL in chapter 34.3.

30 Linear algebra - which library to use?

Basically, the answer is BLAS and LAPACK, written in Fortran and available in www.netlib.org. The C-versions (CLAPACK, CBLAS) are heavily used in GSL. Optimized BLAS and LAPACK libraries are delivered by Intel OneAPI MKL, and AMD AOCL (AMD Optimizing CPU Libraries). Linux comes by default with **OpenBlas**, which owes a lot to GotoBlas2 by Kazushige Goto. These all ***low-level linear algebra libraries***, and not very use friendly. I recommend you pick a C++ header-only library that can use optimized BLAS and LAPACK as a backend.

> Header-only libraries only need to be copied to your machine.
> Include the library path in compilation, that's all that it takes.

Header-only libraries are (arguably) platform independent, but there are some problems as well.[92] Here are a few C++ linear algebra libraries you might find useful, they use ***generic programming*** and ***expression templates***:

- **Boost ublas**. The collection `boost::ublas` has BLAS levels 1-3 functionality for dense, packed and sparse matrices. On the left is an example of a Hermitian matrix @boost.org, on the right an example how I might use it. I like a tighter layout, and hide uninteresting library choises from `main()`.

[92] See, *e.g.*, Header-only @Wikipedia).

boost_hermitian.cpp

```cpp
#include <boost/numeric/ublas/hermitian.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    hermitian_matrix<std::complex<double>, lower> ml (3, 3);
    for (unsigned i = 0; i < ml.size1 (); ++ i) {
        for (unsigned j = 0; j < i; ++ j)
            ml (i, j) = std::complex<double> (3 * i + j, 3 * i + j);
        ml (i, i) = std::complex<double> (4 * i, 0);
    }
    std::cout << ml << std::endl;
    hermitian_matrix<std::complex<double>, upper> mu (3, 3);
    for (unsigned i = 0; i < mu.size1 (); ++ i) {
        mu (i, i) = std::complex<double> (4 * i, 0);
        for (unsigned j = i + 1; j < mu.size2 (); ++ j)
            mu (i, j) = std::complex<double> (3 * i + j, 3 * i + j);
    }
    std::cout << mu << std::endl;
}
```

my_boost_hermitian.cpp

```cpp
#include <boost/numeric/ublas/hermitian.hpp>
#include <boost/numeric/ublas/io.hpp>

namespace herm{
  using namespace boost::numeric::ublas;
  template <typename T>
  using lower = hermitian_matrix<T, lower>;
  template <typename T>
  using upper = hermitian_matrix<T, upper>;
}

int main () {
  using dtype = std::complex<double>; //only complex
  herm::lower<dtype> ml(3,3);
  for (unsigned i=0; i<ml.size1();++i) {
    for (unsigned j=0; j<i; ++j)
      ml(i,j) = (3*i+j,3*i+j);
    ml(i,i) = (4*i,0);
  }
  std::cout<< ml<<std::endl;

  herm::upper<dtype> mu(3,3);
  for (unsigned i=0; i<mu.size1();++i) {
    mu(i,i) = (4*i,0);
    for (unsigned j=i+1;j<mu.size2(); ++j)
      mu(i,j) = (3*i+j,3*i+j);
  }
  std::cout<<mu<<std::endl;
}
```

- **Armadillo** (NICTA, Australia) Can utilize MKL and OpenBlas, to mention a few.
- **MTL4** (Simunova, a C++ software company in Dresden)
- **Blaze** A project initiated by Klaus Iglberger, main developers in Erlangen, Germany.
  A link to a fine article on Smart Expression Templates.
- **Eigen** The eigenvalue problem specialist. Widely used in numerics community.

A comparison of array views didn't flatter Armadillo, see speed tests @romanpoya.medium.com.[93]

30.1 Armadillo examples

Example: Armadillo: Matrix product (source: Armadillo web page)

arma_matrix_multi.cpp

```
#include <iostream>
#include <armadillo>

int main()
{
  using arma::mat, arma::randu;
  mat A = randu<mat>(4,5); // mat is double
  mat B = randu<mat>(4,5);

  std::cout << "A*trans(B) =" << "\n";
  std::cout << A*trans(B) << "\n";
}
```

Linking examples, the details vary depending on the compiler, operating system, and Armadillo installation:

```
$ g++  arma_matrix_multi.cpp -larmadillo # builtin
$ g++  arma_matrix_multi.cpp -I$ARMA_INSTALL_DIR/include -lopenblas # external
```

[93] The tests were run 2020, but I got similar results 2023.

Example: Armadillo: Eigenvalues of a symmetric matrix

arma_eigenvalues.cpp

```cpp
// g++ arma_eigenvalues.cpp  -larmadillo

#include <iostream>
#include <iomanip>
#include <armadillo>

int main(){
  using arma::mat, arma::vec, arma::randu, arma::trans;
  mat A = randu<mat>(5,5);
  vec eigval;
  mat eigvec;
  A = A+trans(A);  // a way to make a symmetric matrix
  std::cout<< "A= \n"<<A<<"\n";

  eig_sym(eigval, eigvec, A); // this does all work

  vec x(eigval);
  for (unsigned i=0; i<A.n_rows; i++){
    std::cout<<"-----------------------------------------"<<"\n";
    std::cout<<"eigenvalue "<<i<<" = "<<eigval(i)<<'\n';
    x = eigvec.col(i); //x(j) =  eigvec(j,i);
    std::cout<<"               x = "<<trans(x);
    std::cout<<"check: Ax -lambdax = "<<trans(A*x-eigval(i)*x)<<"\n";
  }
}
```

Armadillo calls the LAPACK routine `dsyev` ("double symmetric eigenvalue"). Linking examples,

```
$ g++  arma_eigenvalues.cpp -larmadillo # builtin
$ g++  arma_eigenvalues.cpp -I$ARMA_INSTALL_DIR/include -lopenblas # external
$ g++  arma_eigenvalues.cpp -I$ARMA_INSTALL_DIR/include -lblas -llapack # external
```

31 Calling C or fortran from C++

Here are some principles about mixing fortran, C and C++.

Example: Fortran subroutine `dgemm` (part of BLAS) computes the matrix-matrix product.

```
SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,
                 B,LDB,BETA,C,LDC)
DOUBLE PRECISION ALPHA,BETA
INTEGER K,LDA,LDB,LDC,M,N
CHARACTER TRANSA,TRANSB
DOUBLE PRECISION A(LDA,*),B(LDB,*),C(LDC,*)
```

Declaration in C++:

```
extern "C"
{
    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);
```

```
}
```

- `extern` "C"
  means "compile C style".
  It tells the C++ compiler to forget about function overloading and identify the function by its name only.
- The compiled fortran subroutine is in object code file `dgemm.o` or inside a library with an underscore: `dgemm()` is shown as `dgemm_()`.
- fortran is "pass by reference", so all arguments have to be pointers. Here `double* a` points to the start of a double array.

As if this were not enough, there is one more problem: matrix storage in fortran and Matlab is ***column major***, but in C/C++ it's mostly ***row major***. Armadillo is C++, but it mimics Matlab syntax and ***Armadillo is column major***. Below is a figure about the two storage habits.

$$\begin{aligned} \text{math notation:} &\quad \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \\ \text{fortran/Matlab/Armadillo is row major :} &\quad 1\ 5\ 2\ 6\ 3\ 7\ 4\ 8 \\ \text{Most C/C++ is column major :} &\quad 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ . \end{aligned}$$

Both are natural, but moving between languages you have to change indexing before and after a call to `dgemm`. For square matrices this means transpose. If you have defined a static array in C-style like this

```
const int n=100;
double a[n][n];
```

the fortran call has `const int* n` .

Finally, by default indices begin from 0 (C/C++) or from 1 (fortran)
fortran : 1 → N, meaning `V(1),V(2)...,V(N)`
C++ : 0 →(N-1), meaning `V(0),V(1)...,V(N-1)`

32 Fixed-size arrays in C++: plain array and `std::array`:

Two different things that can be called "arrays".

- Plain, C-style array. A static array is made like this:

```
double array1[10]; // memory allocation for 10 elements
int array_int[]={3,6,12}; // memory allocation and fill with values
```

  Dynamic arrays are created with keywords `new` and deleted with `delete`. I'm not telling you how to use plain arrays, because they don't conform with C++ containers. Deleting a two-dimensional, dynamic array `array2` is *not* simply `delete[][] array2`. Below are two examples.

Example: Passing an C array to function in C++ style

array_to_function.cpp

```cpp
#include <iostream>

void f(int d[],const int sized){ // C++ style
  // empty [] tells compiler d is an array
  // d is passed by reference!
  // You are dealing with the original array, not with it's copy.
  for (int i=0;i<sized;++i) std::cout<<i<<" "<<d[i]<<"\n";
}
int main(){
  int j[]={2,4,5};
  f(j,3);
}
```

Example: Passing an C array to function in C style

array_to_function2.cpp

```cpp
#include <iostream>

void f(int *d,const int sized){ // C-style, think of d as a pointer
  for (int i=0;i<sized;++i) std::cout<<i<<" "<<d[i]<<"\n";
}
int main(){
  int j[]={2,4,5};
  f(j,3);
}
```

- There is one more data type suitable for numerical data in C++, `std::array` (mentioned earlier in 23.4). `std::array` is a container. From cppreference, `std::array` can be initialized like this:

```cpp
std::array<int, 3> a1{ {1,2,3} };  // double-braces required
std::array<int, 3> a2 = {1, 2, 3}; // except after =
std::array<std::string, 2> a3 = { {std::string("a"), "b"} };
```

  Notice how `std::array` type and dimension is defined as template parameters! Since it's a container, it has many built-in methods.

  Fixed size enables checks at compile-time and optimization: in C++ also template arguments can be checked. For that C++ has `std::static_assert` (see `basic/static_assert.cpp`). `std::static_assert` can greatly benefit code development, by letting you know if you, by mistake, give contradicting definitions.

33 Exception handling with `throw` and `catch`

The C++ Standard Library has a class dedicated to exception handling, `std::exception`. One part of it is `runtime_error`. One way to make your own error handling process is to *inherit* the class,

```
class MyException : public std::exception{
 ...
}
```

and add a new property.
In numerics exceptions are often simple. I'm using this error handling (although I seldom do)

```
  try{
    my_function();
  }
  catch (char const* e) {
    cerr << e << endl;
    return 1;
  }
```

and the function has the row

```
  void my_function(void){
  {
          ...
    if(test) throw "test failed";
    ...
  }
```

Error happens if `test` is `true`, and the exception with message "test failed" is thrown and we leave the function. Later the exception is caught with `catch` [94]

[94] The function `my_function()` throws an exception and wishes some exception handling routine will take care of it properly.

If `throw` is executed, the function execution terminates; it's still more gentle than halting the *program* execution. Any code after `throw` is not executed and essentially the function does not return at all (so no need to have a `return` in a function that only throws).

In the example the message is output to stream `std::cerr`, similar to `std::cout`, but specialized for error outputs. This makes it possible to separate error output from normal output. In bash,

```
$ a.out 2> err
```

causes normal `std::cout` output to screen and `std::cerr` output to go to file `err`. There is also `std::clog` for log outputs.

34 Gnu Scientific Library (GSL)

GSL in wikipedia

GSL is free and written in C, but linkage is made easy for C++ users:

> *The library header files automatically define functions to have extern "C" linkage when included in C++ programs. This allows the functions to be called directly from C++.*

Example: Bessel function $J0$
Compilation:

```
$ g++ gsl_bessel.cpp `gsl-config --libs` # backticks: bash-evaluate gsl-config --libs
```

gsl_bessel.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <gsl/gsl_sf_bessel.h>

int main(void) {
  double x = 5.0;
  double y = gsl_sf_bessel_J0(x);
  printf("J0(%g) = %.18f\n", x, y);
  // or using iomanip:
  //cout<<fixed<<setprecision(18);
  //std::cout<<"J0("<<x<<") = "<<y<<endl;
}
// J0(5) = -0.17759677131433826 4
```

34.1 GSL: statistics

`double gsl_stats_mean` (*const double* `data[]`, *size_t* `stride`, *size_t* `n`) [Function]

This function returns the arithmetic mean of *data*, a dataset of length n with stride *stride*. The arithmetic mean, or *sample mean*, is denoted by $\hat{\mu}$ and defined as,

$$\hat{\mu} = \frac{1}{N} \sum x_i$$

where x_i are the elements of the dataset *data*. For samples drawn from a gaussian distribution the variance of $\hat{\mu}$ is σ^2/N.

`double gsl_stats_variance` (*const double* `data[]`, *size_t* `stride`, *size_t* `n`) [Function]

This function returns the estimated, or *sample*, variance of *data*, a dataset of length n with stride *stride*. The estimated variance is denoted by $\hat{\sigma}^2$ and is defined by,

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - \hat{\mu})^2$$

where x_i are the elements of the dataset *data*. Note that the normalization factor of $1/(N-1)$ results from the derivation of $\hat{\sigma}^2$ as an unbiased estimator of the population variance σ^2. For samples drawn from a gaussian distribution the variance of $\hat{\sigma}^2$ itself is $2\sigma^4/N$.

This function computes the mean via a call to `gsl_stats_mean`. If you have already computed the mean then you can pass it directly to `gsl_stats_variance_m`.

Example: Arithmetic mean and standard deviation

gsl_statistics.cpp

```cpp
#include <iostream>
#include <vector>
#include <gsl/gsl_statistics.h>

int main(void) {
  // using plain array
  double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6};
  double mean, variance;
  mean     = gsl_stats_mean(data, 1, 5);
  variance = gsl_stats_variance(data, 1, 5);
  std::cout<<"     mean = "<<mean<<"\n";
  std::cout<<" variance = "<<variance<<"\n";
  // using std::vector
  std::vector<double> v = {17.2, 18.1, 16.5, 18.3, 12.6};
  mean     = gsl_stats_mean(&v[0], 1, 5);
  variance = gsl_stats_variance(&v[0], 1, 5);
  std::cout<<"     mean = "<<mean<<"\n";
  std::cout<<" variance = "<<variance<<"\n";
  }
```

Passing a `std::vector` as a pointer `&v[0]` isn't pretty, but it works. All that the GSL function needs is the start address and the length of the data. The elements of a `std::vector` are consecutive in memory.

34.2 GSL: Fast Fourier Transform (FFT)

I want to transform complex `data` with length 2^N, $N \in \mathbb{Z}_{>0}$, like this:

```
fft(data,direction) // direction = 1 forward, -1 backward
```

Here `direction` is 1 for a Fourier transform and -1 for an inverse transform. Notice that I want to keep this syntax no matter what library does the FFT.

I wan't to postbone the decision how the data is stored, so I write the header as a template that works at least for `std::vector`, fixed-size `std::array`, `std::valarray`, armadillo vector and Blaze vector and possibly something else, too.

Example: A possible GSL FFT header

gsl_fft.hpp

```
#ifndef GSL_FFT_HPP
#define  GSL_FFT_HPP
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>

namespace my_GSL_FFT
{
  template <typename T>
  int fft(T& data, int direction){
    int status;
    const size_t stride=1;
    size_t n;
#ifdef ARMA
    // Armadillo data has no member size, use n_elem
    n = data.n_elem;
#else
    n = data.size();
#endif
    double* pdata = reinterpret_cast<double*> (&data[0]);
    if(direction>0){
      status = gsl_fft_complex_radix2_forward(pdata, stride, n);
    }
    else {
      status = gsl_fft_complex_radix2_backward(pdata, stride, n);
    }
    if(status!=GSL_SUCCESS) return 1;
    return 0;
  }
}
#endif
```

34.2.1 Passing a pointer to complex data

C++ complains if you try to take the address of a complex number like this:

```
double* pdata = &data[0].real(); // may not compile
```

Here `data[0].real()` is an ***rvalue***, a temporary whose life is about to end (see chapter 21.2). It does not have an address that lives long enough to be used. The first complex element *does* have a "good address", an `lvalue` , so I tell the compiler to use that and pretend it's pointing to a `double` using `reinterpret_cast`. The type change done by `reinterpret_cast` is in your responsibility, it basically tells the compiler "trust me, I know what I'm doing".

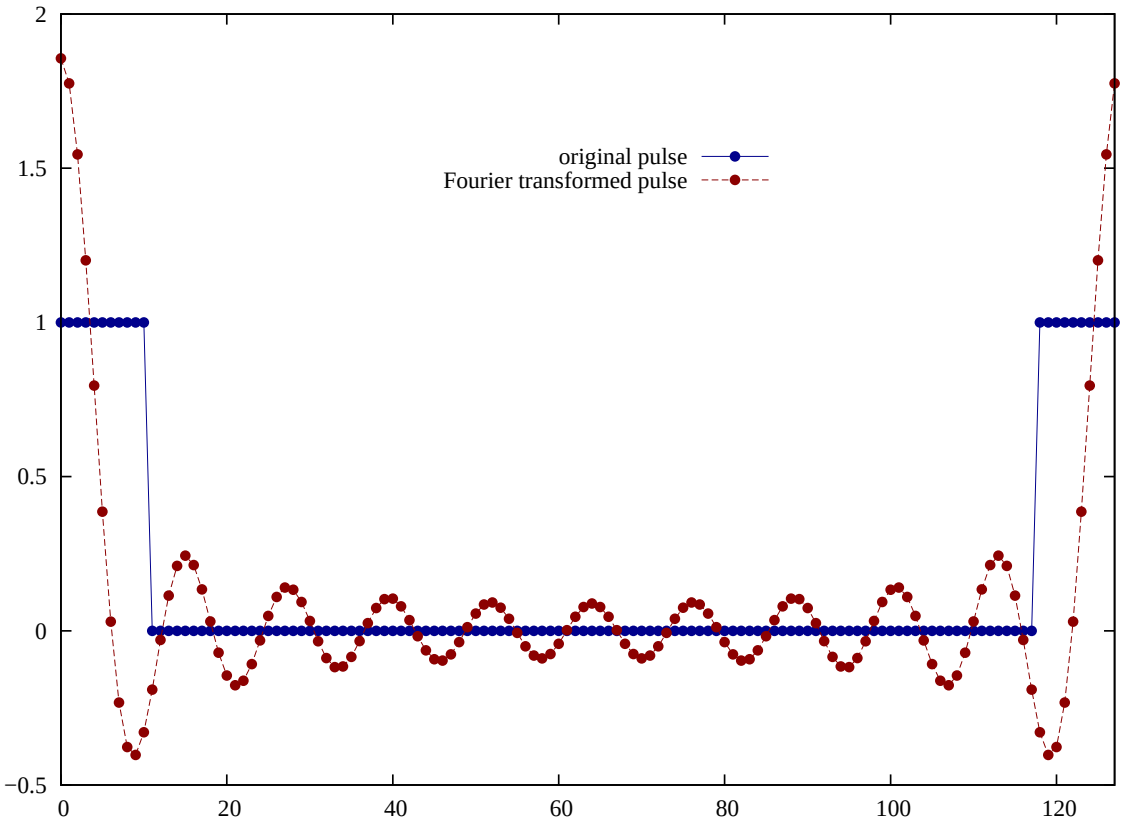
```
double* pdata = reinterpret_cast<double*> (&data[0]);
```

Armadillo vector does not conform with a standard container, specifically it has no `size()` method. The least I want is to add a third argument to the clean call `fft(data,direction)` to tell the data size! I have chosen to use the ***preprocessor***; if I'm using Armadillo vectors

```
#define ARMA
```

A test code is `numerics/gsl_fft_arma_main.cpp`, the one using `std::vector` is `numerics/gsl_fft_main.cpp`.

Result:



The data is represented, as usual in FFT, in *wrap-around order*. You can easily write utilities to do the wrapping from "natural order" and unwrapping to natural order; move data or use an index table.

34.3 GSL: differential equations

Let's solve a simple equation, one that we can solve analytically,

$$y'(t) = -ty \quad , \quad y(0) = -1 \ , \quad \text{exact solution} \quad y(t) = -2e^{-t^2/2} \ .$$

gsl_ode_simple.cpp

```cpp
// solve y'(t) = -t*y ,  condition y(0)  = -2
// g++ -std=c++11 gsl_ode_simple.cpp `gsl-config --libs`
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv2.h>

int func (double t, const double y[], double f[], void *params){
  f[0] = -t*y[0] ; // y[0] = y,  f[0] = y' = dy/dt = -t*y
  return GSL_SUCCESS;
}
void output(double t, double y, double exact){
   std::cout<<std::fixed<<std::setprecision(16);
   static bool first=true;
   if(first) {
     std::cout<<std::setw(20)<<"t"<<std::setw(20)<<"gsl solution";
     std::cout<<std::setw(20)<<"exact solution"<<std::setw(20)<<"error\n";
     first = false;
   }
   std::cout<<std::setw(20)<<t<<std::setw(20)<<y<<std::setw(20)<<exact<<std::setw(20)<<y-exact<<"\n";
}

int main ()
{
  double exact;
  const int n=50;                 // # of points
  double t0 = 0.0, t1 = 10.0;     // time start and end
  double dt=(t1-t0)/(n-1);        // time step
  double y[1] = {-2.0};           // initial value; table with one entry

  gsl_odeiv2_system sys = {func, NULL, 1, NULL}; // not using a Jacobian, hence NULL pointer
  auto driver = gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd, 1e-13, 1e-13, 0.0);

  output(t0,y[0],y[0]);
  for (int i = 0; i<n; ++i) {
    auto t = t0+i*dt; // begin of t interval
    if(gsl_odeiv2_driver_apply (driver, &t, t+dt, y) != GSL_SUCCESS)
      {
        std::cout<<"FAILED near "<<t<<"\n";return 1;
      }
    exact = -2.0*exp(-0.5*t*t);
    output(t,y[0],exact);
  }
  gsl_odeiv2_driver_free(driver);
}
```

The next one is more involved, it's directly from the GSL manual. The 2nd order, nonlinear Van Der Pol oscillator equation is

$$x''(t) + \mu x'(t)(x(t)^2 - 1) + x(t) = 0 \ .$$

The numerical solver is for *1st order* equations, so we split this 2nd order equation to two coupled 1st order equations. Define a new variable y,

$$\begin{aligned} x'(t) &= y(t) \\ y'(t) &= -x(t) - \mu y(t)(x(t)^2 - 1) \end{aligned}$$

An n:th order differential equation is solved as n coupled 1st order differential equations

The code solves two unknown functions $\{x(t), y(t)\}$ point by point, starting from initial values at $t = 0$. Once a point $\{x(t), y(t)\}$ has been solved, the derivatives in that point can be computed and the next point $\{x(t + dt), y(t + dt)\}$ can be solved, and so on.

The Van Der Pol oscillator position $x(t)$ can have very sharp turns for some values of μ. Near sharp turns we apply ***adaptive stepsize***: In order to maintain numerical accuracy, the algorithm takes shorter steps in t. [95]

The biggest challenge is bookkeeping. We have (i) the math on paper, (ii) 1st order differential equation notation in GSL and (iii) the notation in GSL solver. The GSL manual uses this general notation:

$$\frac{dy_i(t)}{dt} = f_i(t, y_1(t), ..., , y_n(t)) \quad , \quad i = 1...n \ .$$

Don't spend too much time studying this, but tabulated, the three notations are related like this:

[95] If this is neglected, the solution goes astray after every steep turn. It will follow *a* solution curve, but not the one we started with, until after the next turn it picks yet another wrong curve and so on. Since the numerical accuracy is limited anyhow, you can be sure this happens sooner or later in t.

A	B	C
$x(t)$	$y_1(t)$	`y[0]`
$y(t)$	$y_2(t)$	`y[1]`
$x'(t) = y(t)$	$\frac{dy_1(t)}{dt} = f_1(t, y_1(t), y_2(t))$	$f[0]$
$y'(t) = -x(t) - \mu y(t)(x(t)^2 - 1)$	$\frac{dy_2(t)}{dt} = f_2(t, y_1(t), y_2(t))$	$f[1]$
$\frac{\partial x'(t)}{\partial x(t)} = 0$	$\frac{\partial f_1(t,y_1(t),y_2(t))}{\partial y_1(t)}$	`m[0,0]`
$\frac{\partial x'(t)}{\partial y(t)} = 1$	$\frac{\partial f_1(t,y_1(t),y_2(t))}{\partial y_2(t)}$	`m[0,1]`
$\frac{\partial y'(t)}{\partial x(t)} = -1 - 2\mu xy$	$\frac{\partial f_2(t,y_1(t),y_2(t))}{\partial y_1(t)}$	`m[1,0]`
$\frac{\partial y'(t)}{\partial y(t)} = -\mu(x^2 - 1)$	$\frac{\partial f_2(t,y_1(t),y_2(t))}{\partial y_1(t)}$	`m[1,1]`
$x''(t)$	$\frac{df_1(t,y_1(t),y_2(t))}{dt}$	`dfdt[0]`
$y''(t)$	$\frac{df_2(t,y_1(t),y_2(t))}{dt}$	`dfdt[1]`

Implementation:

- The function `func()`(4 top rows of the table) computes $\{x'(t), y'(t)\}$ from known values $\{x(t), y(t)\}$. The results are stored to table `f`, elements {`f[0],f[1]`}.
- The function `jac()` (4 bottom rows of the table) hold the Jacobi matrix. This information is used only in higher order solvers - the more knowledge, the less function evaluations. A twist to bookkeeping: `jac()` is supposed to fill a 1-dimensional table `dfdy`, where the Jacobi matrix is stored as `m[i,j] = dfdy[i + dimensio * j]`. Hence the odd calls to `gsl_matrix_*`. If you find this hard to follow, fill `dfdy` as you please. A plain `for` loop is not a bad idea.

gsl_func_jac.h

```
int func (double t, const double y[], double f[], void *params){
  double mu = *(double *)params;
  f[0] = y[1];
  f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
  return GSL_SUCCESS;
}
int jac (double t, const double y[], double *dfdy, double dfdt[], void *params){
  double mu = *(double *)params;
  gsl_matrix_view dfdy_mat
    = gsl_matrix_view_array (dfdy, 2, 2);
  gsl_matrix * m = &dfdy_mat.matrix;
  gsl_matrix_set (m, 0, 0, 0.0);
  gsl_matrix_set (m, 0, 1, 1.0);
  gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
  gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));
  dfdt[0] = 0.0;
  dfdt[1] = 0.0;
  return GSL_SUCCESS;
}
```

GSL has nice ***driver functions*** in the header `gsl_odeiv2.h`, which are much easier to use than the basic functions in the header `gsl_odeiv.h`. Whichever, some preparatory steps need to be done.

- Choose the integration algorithm, that is, how the next point $\{x(t+dt), y(t+dt)\}$ is computed. Plenty of choice here: `rk2`, `rk4`, `rkck`, `rk8pd`, `rk2imp`, `rk4imp`, `bsimp`, `rk1imp`, `msadams` and `msbdf`.
- Choose the control criterion, that is, when are shorter steps needed. For example absol. error 10^{-6}, relative error 0
- GSL wants all information about the problem at hand in one data structure of type `gsl_odeiv2_system`:

```
gsl_odeiv2_system sys = {func, jac, 2, &mu} ;
```

This collects information about the functions, their derivatives, Jacobi matrix, order and the parameter(s). Any number of parameters are passed as a pointer
`void * params`.

GSL driver function[96]

```
auto solver =
      gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd, 1e-6, 1e-6, 0.0);
```

and it's called later like this

```
int status = gsl_odeiv2_driver_apply (solver, &t, ti, y);
```

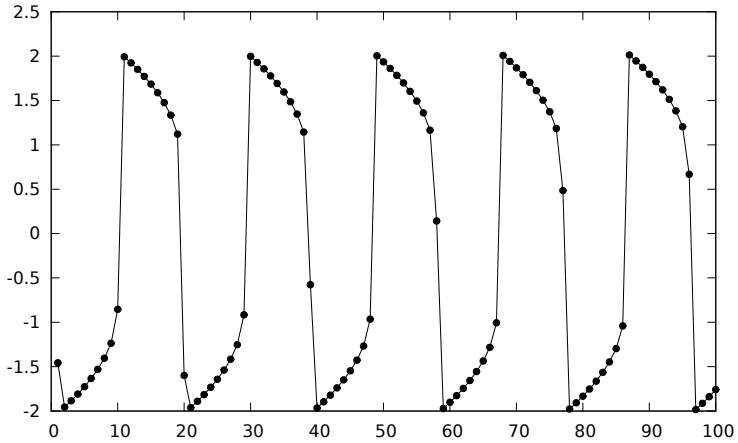
Passed values are the system `sys`, the algorithm `solver`, start point `t`, end point `ti` and initial values `y`. The solution at time `ti` comes out in `y`, as well.

Set initial values and find the solution between t=0...100.

[96] Actually there are four drivers to choose from.

gsl_ode_part.cpp

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv2.h>
#include "gsl_func_jac.h"
int main (void){
  double mu = 10;
  gsl_odeiv2_system sys = {func, jac, 2, &mu};
  auto solver =  gsl_odeiv2_driver_alloc_y_new (
                &sys, gsl_odeiv2_step_rk8pd, 1e-6, 1e-6, 0.0);
  int i;
  double t = 0.0, t1 = 100.0;
  double y[2] = { 1.0, 0.0 };
  for (i = 1; i <= 100; i++) {
    double ti = i * t1 / 100.0;
    int status = gsl_odeiv2_driver_apply (solver, &t, ti, y);
    if (status != GSL_SUCCESS) {
      printf ("error, return value=%d\n", status);
      break;
    }
    printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
  }
  gsl_odeiv2_driver_free (solver);
}
```



Van Der Pol oscillator with $\mu = 10$. The line is just a linear interpolation between solved points.

34.4 GSL: interpolation

It's convenient to hide the GSL-specific parts to a header. I chose to use `std::vector` for data. The example recognizes two spline types, not very ambitious :ˆ)

- ***"cspline" (natural cubic spline)***
  Changes in one point causes non-local changes to the spline → unstable
  "Natural" refers to setting end point derivatives to zero.

- ***"akima" or "Akima" (natural Akima spline)***
  Changes in one point causes only local changes to the spline → stable

Steps:

1) Reserve space for accelerator (speeds up searches)

```
auto accel = gsl_interp_accel_alloc() ;
```

2) Reserve space for interpolation

```
gsl_spline spline = gsl_spline_alloc(gsl_interp_cspline, ...);
```

3) Initialize the interpolation with known points (x, y) (now in a `std::vector`):

```
gsl_spline_init(spline,&x[0],&y[0],x.size()) ;
```

4) Compute the interpolated y values to `yy` (iterator `posy`) at points `xx` (iterator `posx`)

```
*posy++ = gsl_spline_eval(spline, *posx, acc);
```

If needed, the 1st and 2nd derivative could be computed:

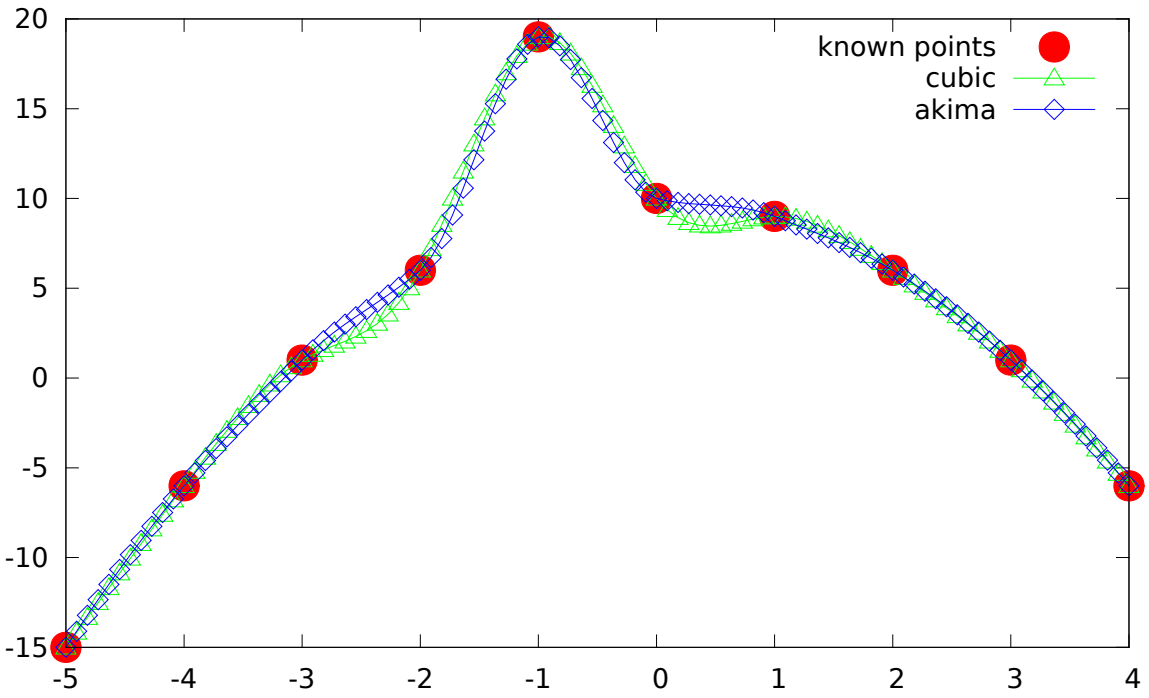
```
... = gsl_spline_eval_deriv(...);
... = gsl_spline_eval_deriv2(...);
```

gsl_spline.hpp

```cpp
// wrapper to GSL spline
// spline type "cspline" is natural cubic spline
// spline type "akima" or "Akima" is natural Akima spline
#ifndef GSL_SPLINE_HPP
#define GSL_SPLINE_HPP
#include <iostream>
#include <cstring>
#include <vector>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

namespace my
{
  using namespace std;
  using vec= vector<double>;

  void spline(const vec& x,const vec& y, vec& xx, vec& yy, const string& type){
    auto acc  = gsl_interp_accel_alloc();
    gsl_spline* spline;
    int choose=0;
    if(type.compare("akima")==0 || type.compare("Akima")==0 ) choose=1;
    switch (choose){
    case 0:
      spline = gsl_spline_alloc(gsl_interp_cspline, x.size());
      break;
    case 1:
      spline = gsl_spline_alloc(gsl_interp_akima, x.size());
      break;
    }
    gsl_spline_init(spline, &x[0], &y[0], x.size());
    auto posy = yy.begin();
    for(auto posx = xx.begin() ; posx != xx.end() ; ++posx){
      *posy++= gsl_spline_eval(spline, *posx, acc);
    }
    gsl_spline_free(spline);
    gsl_interp_accel_free(acc);
  }
}
#endif
```

gsl_spline.cpp

```cpp
// spline test   g++ -std=c++11 gsl_spline.cpp `gsl-config --libs`
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
// home made call to GSL spline
#include "gsl_spline.hpp"

typedef std::vector<double> vec;

void output(const vec& x, const vec y)
{
  std::cout<<std::fixed<<std::setprecision(8);
  for(size_t i=0;i<x.size();++i)
    std::cout<<x[i]<<" "<<y[i]<<"\n";
  std::cout<<"\n\n\n";
}

int main(){
  vec x(10),y(10); // known points (x,y)
  double t;
  t=-5.0;
  for(unsigned i=0;i<x.size();++i){  // you could use std::generate
    x[i] = t;
    y[i] = 10.0-t*t;
    t += 1.0;
  }
  y[4] += 10.0; // lift one point up to demonstrate locality/nonlocality
  output(x,y);

  // interpolated points xx
  vec xx(100),yy(100);
  double dx = (x[x.size()-1]-x[0])/(xx.size()-1);
  for(unsigned i=0;i<xx.size();++i){ xx[i] = x[0]+i*dx;}

  // interpolate using natural cubic spline
  my::spline(x,y,xx,yy,"cspline");
  output(xx,yy);

  // interpolate using natural Akima spline
  my::spline(x,y,xx,yy,"Akima");
  output(xx,yy);
}
// a.out>tt
// gnuplot
// p 'tt' i 0 w p pt 7 ps 1.5 t'known points','' i 1 w lp pt 8 t'cubic','' i 2 w lp pt 12 t'akima'
```



34.5 GSL: Monte Carlo integration

Let's compute an N dimensional integral over a hyper cube,

$$\int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 ... \int_{a_N}^{b_N} dx_N \;\; f(x_1, x_2 ... x_N) \; .$$

GSL offers three ways (see Wikipedia Monte Carlo sampling)

1) Plain - sample random points inside the hyper cube; a very crude method

2) MISER - (Press & Farrar) stratified sampling algorithm; sample points from areas that give largest error

3) VEGAS - (Lepage) combines both stratified sampling and importance sampling (sample points from areas that affect the result the most)

The example code evaluates the integral

$$\frac{1}{\pi^3}\int_0^\pi dx \int_0^\pi dy \int_0^\pi dz \frac{1}{1-\cos(x)\cos(y)\cos(z)} = 1.39320392\ldots .$$

Prerequisites:

- Random number generator
- Integrand:

```
 Data Type: gsl_monte_function
This data type defines a general function
with parameters for Monte Carlo integration.

double (* f) (double * x, size_t dim, void * params)
this function should return the value f(x,params)
for the argument x and parameters params,
where x is an array of size dim giving
the coordinates of the point where the function is to be evaluated.
size_t dim the number of dimensions for x.
void * params a pointer to the parameters of the function.
```

The integrand is passed as a function pointer. ROOT (CERN C++ package) includes, among plenty of other things, Monte Carlo integration and a wrapper to pass a function object to GSL.

(see http://www.gnu.org/software/gsl/manual/html_node/Monte-Carlo-Examples.html)

gsl_monte.carlo.cpp

```
// g++ -std=c++11 -Wall gsl_monte.carlo.cpp `gsl-config --libs`
#include <iostream>
#include <iomanip>
#include <string>

#include <gsl/gsl_math.h>
#include <gsl/gsl_monte.h>
#include <gsl/gsl_monte_plain.h>
#include <gsl/gsl_monte_miser.h>
#include <gsl/gsl_monte_vegas.h>

double func (double *x, size_t dim, void *params);
void display_results (std::string title, double result, double error);

int main ()
{
  const size_t dim=3;
  double result,error;
  std::string method;
  gsl_rng *r;
  gsl_monte_function G = { &func, dim, 0 };
  double a[dim] = { 0, 0, 0 };
  double b[dim] = { M_PI, M_PI, M_PI };
  size_t calls = 500000;
  // random number generator
  gsl_rng_env_setup ();
  r = gsl_rng_alloc (gsl_rng_default);

  {
    method="plain";
    auto s = gsl_monte_plain_alloc (dim);
    gsl_monte_plain_integrate (&G, a, b, dim, calls, r, s,
                               &result, &error);
    gsl_monte_plain_free (s);
    display_results (method , result, error);
  }
```

Output:

```
================================================
Method : plain
================================================
result = 1.41220870 +/- 0.01343586
 exact = 1.39320393
|diff| = 0.01900477
================================================
Method : MISER
================================================
result = 1.39132158 +/- 0.00346056
 exact = 1.39320393
|diff| = 0.00188235
================================================
Method : VEGAS warmup
================================================
result = 1.39267259 +/- 0.00341041
 exact = 1.39320393
|diff| = 0.00053134
================================================
Method : VEGAS
================================================
result = 1.39328139 +/- 0.00036248
 exact = 1.39320393
|diff| = 0.00007746
```

34.6 Add numbers to file names

Sometimes it's convenient to have file names contain numerical parameters. One option is to use `stringstream` objects, see `basic/stringstream_ex.cpp`. It's not convenient, because one has to dig a "C string" out of the `stringstream` object `s` using `s.srt().c_str()` [97] An easier way is shown below.

Example: Open files `res_`, suffix is a real number.

[97] A valid file name has to end with null character ("\0") like a character string in C.

string_to_filename.cpp

```cpp
// Principle: how to get a computed number to a file name
#include <iostream>
#include <fstream>
#include <math.h>
#include <string>
using namespace std;
int main()
{
  const string str = "res_";
  for (unsigned i=1;i<5;i++){
    ofstream out(str+to_string(cos(i)));
    out.close();
  }
}
/* opened files
res_0.540302
res_-0.416147
res_-0.989992
res_-0.653644
*/
```

35 Lambda Functions/Expressions

I have already mentioned lambda functions or expressions in a few examples. They are nice. They are fast. A lambda function is compiled to a function object and easily *inlined*. Boost has it's own lambda's, parts of the ideas came to be in C++11.

Usage: Define a "temporary" function *on the spot*, exactly where it's used. It can remain unnamed.

Bonus: No class needed to get a function object. The lambda remains local and you can't misuse it elsewhere.

The brackets `[]` in a lambda are for capturing things from outside. Here is a list of what is captured and how:

```
[]        capture nothing
[x, &y]   capture x by value, y by reference
[&]       capture all external variables by reference
[=]       capture all external variables by value
[&, x]    capture x by value, all else by reference
[=, &x]   capture x by reference, all else by value
```

Example: A locally applied function `func`

autolambda.cpp

```
//g++ -std=c++11 autolambda.cpp

#include <iostream>

int main()
{
  auto func = [] () { std::cout << "Hello world\n"; };
  func();
}
```

Example: Output a container and the cubes of the elements

lambda1.cpp

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

int main()
{
  std::vector<int> v{11,22,33};
   std::for_each(v.begin(), v.end(), [](int n) {std::cout << n<<" "<<pow(n,3) <<"\n";});
}
```

```
11 1331
22 10648
33 35937
```

Note: `int n` is passed by value.

lambda2.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <complex>

int main()
{
  typedef std::complex<double> cmplx ;
  std::vector<cmplx> v={cmplx(11,12),cmplx(22,21),cmplx(33,32)};
  for(auto x:v) std::cout<<x<<" ";
  std::cout<<" original data"<<"\n";
  // take complex conjugate of elements
  std::for_each(v.begin(),v.end(), [](cmplx& c) {c = conj(c);});
  for(auto x:v) std::cout<<x<<" ";
  std::cout<<" complex conjugate"<<"\n";
}
```

Tulostus:

```
(11,12) (22,21) (33,32)
(11,-12) (22,-21) (33,-32)
```

Note: The elements are passed by reference to complex conjugation.
Example: Calculate the product of `std::vector` elements.

lambda4.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
   std::vector<double> v{1.0,2.0,3.0};
   double prod=1.0;
   std::for_each(v.begin(),v.end(), [&prod] (double x) { prod *= x;});
   std::cout <<"product = "<< prod << "\n";
}
```

```
product = 6
```

Note: `prod` is captured by reference.
Example: Chop all elements below some limit to zero.

lambda5.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

typedef std::vector<double> vec;

void rand_vector(vec& v){
  generate(v.begin(),v.end(), [](){return rand()/((double)RAND_MAX);});
}

int vector_chop(vec& v, const double lim){
  int count = 0;
  for_each(v.begin(),v.end(), [&count,lim](double& x){if(x<lim) {x=0.0; count++;}});
  return count;
}

int main()
{
   vec v(7);
   rand_vector(v);
   for(auto x:v) std::cout<<x<<"\n";
   const double limit=0.4;
   std::cout<<"chopped "<<vector_chop(v,limit)<<" values below "<<limit<<"\n";
   for(auto x:v) std::cout<<x<<"\n";
}
```

36 Parallel C++

Choose from two types of parallelism:

1. MPI parallel code. The way to parallelize over multiple nodes.
   MPI is a standard, you can install, for example, OpenMPI. The MPI function calls are in principle similar to those used in `mpi4py`, but I won't go into details in these lecture notes.

2. Multihreaded code. The way to parallelize within one multi-core CPU.
   C++17 has a quite extensive support for multithreading. Some threading options:

   - Intel's OpenAPI Threading Building Blocks (TBB)
   - POSIX threads (pthreads) are low-level API
   - Windows threads

On top of these you have very high-level API's. One of them is OpenMP, which can be used in several languages (C, C++, fortran...). Another one is C++11 threads.

36.1 Intel OneAPI TBB

OneAPI TBB implements work stealing. The work load is initially evenly distributed to cores, but if a core becomes idle it can steal work load from a heavily burdened core. This dynamical redistribution is done without any intervention of the programmer.

Intel OneAPI compilers and TBB can be easily chosen using ***environment modules***, both in Linux and in Windows. I recommend using the lua-based Lmod module system (installation instructions).[98] Intel OneAPI is usually installed under `/opt/intel/oneapi`, and it provides the script `modulefiles-setup.sh`. Choose the directory for your private module files, for example

[98] Another option is to use the Environment Modules open source project.

```
$ sh  /opt/intel/oneapi/modulefiles-setup.sh --outdir= ${HOME}/privatemodules
```

Then add the line `module use ${HOME}/privatemodules` to your `${HOME}/.bashrc`. After this you can either start a new bash shell or in the current shell type

```
$ . ~/bashrc
module avail
```

This lists available modules. Intel compilers setup is

```
$ module add icc
```

which sets up the environment to `icc` (for C), `icpc` (for C++), and `ifort` (for fortran). TBB environment is set up using

```
$ module add tbb
```

Examples of compilations:

```
$ g++ -std=c++17 sort_parallel.cpp `pkg-config --libs --cflags tbb`
$ g++ -std=c++17 sort_parallel.cpp -L/opt/intel/oneapi/tbb/latest/lib/intel64/gcc4.8/ -ltbb
$ g++ -std=c++17 sort_parallel.cpp -L${LIBRARY_PATH} -ltbb
$ icpc -Ofast sort_parallel.cpp  -ltbb
```

Apparently `icpc` finds the TBB headers and the library with just the option `-ltbb`.

Remark: Why couldn't *my* `g++` do that? `module add tbb` appends to the `CPATH` environment variable the path `/opt/intel/oneapi/tbb/2021.4.0/include/`, and `g++` finds the headers. Also the compile-time path to libraries is setting `LIBRARY_PATH`, but unfortunately the compiler sees the path as (`g++ -v ...`) `LIBRARY_PATH=... :/usr/lib/../lib64/:...:/opt/intel/oneapi/tbb/2021.4.0/lib/intel64/gcc4.8/`. I have also an open-source TBB library at `/usr/lib64/libtbb.so` (use by Blender, Sagemath *etc.*), and the linker finds that wrong TBB first. So how come `g++ -std=c++17 sort_parallel.cpp -L${LIBRARY_PATH} -ltbb` works? For some reason, `${LIBRARY_PATH}` is the correct `tbb` path. These kind of quirks are very annoying.

36.2 POSIX Threads (pthread)

Basics are in cplusplus.com. Notice the extra flags `-pthread` needed in compilation

```
$ g++ -std=c++17  threads1.cpp -pthread
```

threads1.cpp

```
// compile: g++ -std=c++17 -pthread threads1.cpp
#include <iostream>
#include <chrono>
#include <thread>
using namespace std::chrono_literals; // s as second
void foo() { std::cout<<"started foo\n"; std::this_thread::sleep_for(3s); }
void bar(int x) { std::cout<<"started bar\n"; std::this_thread::sleep_for(5s); }

int main() {
  std::thread th1 (foo);     //  a thread calls foo()
  std::thread th2 (bar,0);   //  a thread calls bar(0)
  th1.join();                // wait for th1 to finish
  std::cout<<"th1 joins\n";
  th2.join();                // wait for th2 to finish
  std::cout<<"th2 joins\n";
}
```

Especially file IO gets easily garbled with concurrent read/write, so better use a ***mutex*** (MUTual EXclusion). The next example is a `Printer` class with a mutex:

threads_printer.cpp

```cpp
//
// Linux: compile using
//    g++ -std=c++17 threads_printer.cpp -pthread
//
#include <iostream>
#include <chrono>
#include <thread>
#include <mutex>
#include <string>
#include <fstream>
#include <vector>

namespace demo{
  class Printer{
  private:
    std::mutex printmtx_;
  public:
    void print(std::string file, int x) {
      std::lock_guard<std::mutex> lock(printmtx_);
      std::cout << "thread id "<<std::this_thread::get_id() << " printing to file " << file << '\n';
      std::ofstream out(file);
      out<<x<<"\n";
      out.close();
    }
  };
}

int main() {
  demo::Printer printer;
  auto job = [&printer](int x){printer.print("test",x);};
  std::vector<std::thread> ths;

  for(int i=0;  i<10; i++){
    std::thread th(job,i);
    ths.push_back(std::move(th)); // must use std::move
  }
  for(auto &th: ths) th.join();  // must use auto &
}
```

Some useful features in `threads_printer.cpp`:

- `printmtx_` is a static mutex, every instance of `Printer` shares the ***same mutex***.
- `std::lock_guard<std::mutex> lock(printmtx_)` replaces the usual unsafe lock-unlock sequence

```
// unsafe code, may leave the lock on
std::mutex mtx;
void fun(){
  mtx.lock();
    ::: do something - but this may exit before unlock!
  mtx.unlock();
}
```

  with a ***safe locking***: it releases the lock when it goes out off scope, also in case of an exception:

```
// safe code, never leaves the lock on
std::mutex mtx;
void fun(){
  std::lock_guard<std::mutex> lock(mtx);
    ::: do something
} // scope of lock() ends, and it's released automagically
```

- Threads are collected to a `std::vector` container. A thread must be ***moved*** to the container,

```
ths.push_back(th); //error
// g++ says use of deleted function 'std::thread::thread(const std::thread&)'
ths.push_back(std::move(th)); // Ok
```

- Joining all threads is now simple,

```
for(auto& th : ths) th.join();
```

where `auto& th` makes possible to actually close the thread (`auto th` would again try to copy the thread and close the copy).

You can also use `std::swap` to swap threads, just `std::swap(t1,t2);` will swap threads `t1` and `t2`.

36.3 C++17: Parallel `std` Algorithms

If your code uses std algorithms, you can trivially parallelize over multiple threads. Since C++17, many algorithms accept an ***execution policy***:

1. `std::execution::seq` is sequential
2. `std::execution::par` may be parallel
3. `std::execution::par_unseq` may be parallel and vectorized.

The algorithms are overloaded, you can add the policy:

```
std::algorithm_name(policy, /* normal arguments */);
```

The next page show how sorting 10^8 random floating point numbers is done sequentially or in parallel. I got the result[99]

```
ordinary sequential sort
 Timer: 11.043943000000 secs
parallel execution, policy std::execution::par
 Timer: 1.879833000000 secs
parallel execution, policy std::execution::par_unseq
 Timer: 1.897244000000 secs
```

[99] Using g++ 9.2 and 8 threads, set in bash `export OMP_NUM_THREADS=8`.

Compilation shows how parallel algorithms rely on TBB (`-ltbb`):

```
g++ -Ofast -std=c++17 sort_parallel.cpp timer.cpp -ltbb
```

or
`clang++ -Ofast -std=c++17 sort_parallel.cpp timer.cpp -ltbb`
Both compilers give comparable speed.

sort_parallel.cpp

```cpp
// C++17 parallel sort
// System requirements: The Threading Building Blocks C++ headers and shared development libraries (tbb)
// compile using environment module tbb
//   module add tbb
//   g++ -Ofast  -std=c++17 sort_parallel.cpp `pkg-config --libs --cflags tbb`
#include <vector>
#include <algorithm>
#include <iostream>
#include <random>
#include <execution>
#include "timer.hpp"

int main(){
  using std::cout;
  my::Timer timer;
  // a vector to sort
  const int N=1e5;
  std::vector<double> v(N);
  std::mt19937_64 rng;
  std::uniform_real_distribution<double> unif(0,1);
  for(auto& el : v) el = unif(rng);
  std::vector<double> w(v); // copy


  timer.tic();
  std::sort(v.begin(), v.end());
  timer.toc();
  cout<<timer<<" sequential sort \n";

  v=w;
  timer.tic();
  sort(std::execution::par, v.begin(), v.end());  // C++17
  timer.toc();
  cout<<timer<<" parallel sort, policy std::execution::par \n";

  v=w;
  timer.tic();
  sort(std::execution::par_unseq, v.begin(), v.end());  // C++17
  timer.toc();
  cout<<timer<<" parallel sort, policy std::execution::par_unseq \n";

}
```

36.3.1 Parallel `std::reduce` and `std::transform_reduce`

C++17 introduced some new algorithms, mainly to overcome limitations that prevent their parallelization. The old algorithm `std::accumulate` works, by definition, from left to right,

$$a_1 \ \mathbf{op} \ a_2 \ \mathbf{op} \ a_3 \ \mathbf{op} \ a_4 \ , \tag{36}$$

but in general one can't change this to

$$(a_1 \ \mathbf{op} \ a_3) \ \mathbf{op} \ (a_2 \ \mathbf{op} \ a_4) \ . \tag{37}$$

However, this is valid if the operator **op** is commutative and associative, such as when $\mathbf{op} = +$,

$$a_1 + a_2 + a_3 + a_4 = (a_1 + a_3) + (a_2 + a_4) \ , \tag{38}$$

thus allowing for parallel execution of partial sums in threads.
`std::reduce` takes the liberty to evaluate the expression in ***arbitrary order***.
`std::transform_reduce` can, for example,

- Compute inner product in parallel (parallel version of `std::inner_product`)

```
// computes x dot x
result = transform_reduce(x.begin(),x.end(),x.begin(),0.0);
```

- Apply a function to each element and ad them up, that is,[100] $result = \sum_i f(x_i)$,

```
result = transform_reduce(x.begin(),x.end(),x.begin(),0.0,std::plus<>(),f);
```

  The product $result = \prod_i f(x_i)$ can be computed with

[100] Notice the initial value 0.0, a plain 0 will give just result=0.

```
result = transform_reduce(x.begin(),x.end(),x.begin(),1.0,std::multiplies<>(),f);
```

36.4 OpenMP parallel programming

OpenMP offers an easy way to parallelize loops, see, for example, http://bisqwit.iki.fi/story/howto/openmp/

Example: OpenMP parallel loop; too small a task to get any speed-up, though.

openmp_ex.cpp

```cpp
// g++ -std=c++17 -pipe -O3 -march=native -fopenmp -mfpmath=sse -msse2 openmp_ex.cpp
#include <iostream>
#include <cmath>
#include <vector>

int main()
{
  std::vector<double> tab(200);
  const int N=tab.size();
#pragma omp parallel for
  for(int i=0; i<N; ++i) // threads have their own private i
    {
      tab[i] = sin(2*M_PI*i/N);
      //std::cout<<i<<"\n"; // will produce mess, but shows parallelism
    }
  //std::cout<<"\n";
  for(auto t:tab) std::cout<<t<<" ";
  std::cout<<"\n";

}
```

The ***preprosessor directive*** `\#pragma omp parallel for` tells what to parallelize. You need to read more from a better source, but here are some other pragmas:

```cpp
#pragma omp parallel // Somewhere later comes a parallel computation
#pragma omp parallel num_threads(3) // parallel over 3 threads
#pragma omp for // following for-loop is parallel
```

```
#pragma omp for private(k) // each thread has it's own copy of k
#pragma omp for shared (m) // all threads use the same m
#pragma omp for ordered schedule(dynamic) // parts of for must happen in due order
#pragma omp ordered // following statement must be done in ordered fashion
```

A loop variable, such as `i` in the previous example, is automatically "private", meaning each thread get its own value of `i` from the OpenMP scheduler. The scheduler decides who computes what.

In linux, the number of thread is set to four using ***environment variables***:

```
setenv OMP_NUM_THREADS 4    (csh or tcsh shell)
export OMP_NUM_THREADS=4    (bash shell)
```

These are for run-time only, you can change these as needed without re-compiling the code.

```
> export OMP_NUM_THREADS=2
> a.out
> export OMP_NUM_THREADS=16
> a.out
```

runs `a.out` first using two threads (cores) and then using 16 cores.
The function `omp_set_num_threads(16)` does the same, overriding `OMP_NUM_THREADS`. See the next example.

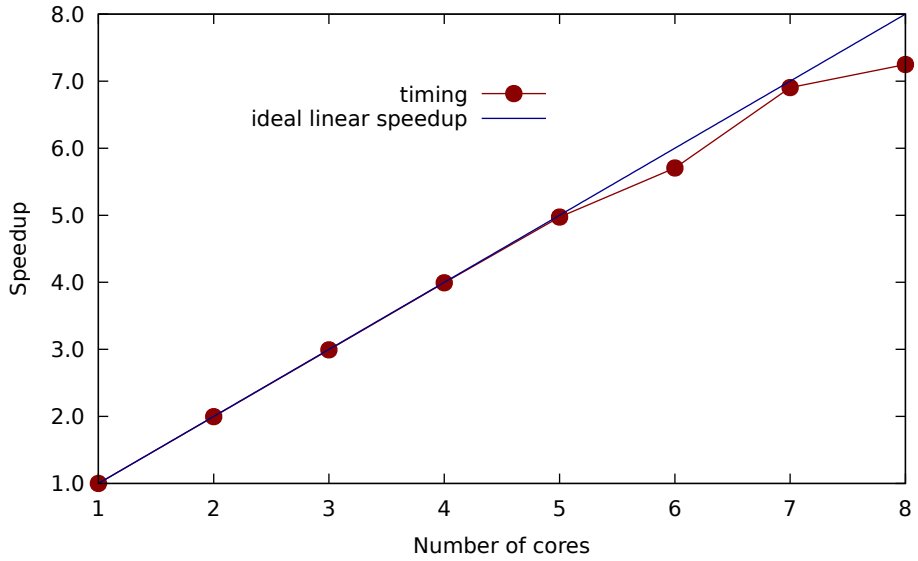
`OMP_NUM_THREADS` and `omp_set_num_threads()` are only suggested maximum number of threads.

Example: `for`-loop reduction with timing.

openmp_reduction.cpp

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include <omp.h>
#include <chrono>

using namespace std::chrono;
int main()
{
  const int n=100000000;
  std::ofstream fileout("timing");
  double time,reftime=0.0;
  const int ncores = 12;
  std::cout<<ncores<<"\n";
  for(int nc=1; nc!=ncores+1; nc++) {
    omp_set_num_threads(nc); // override OMP_NUM_THREADS
    double sum=0;
    auto t0 = high_resolution_clock::now();
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) sum+=1/cos(i);//thread sums put together in the end
    auto t1 = high_resolution_clock::now();
    auto d = duration_cast<milliseconds>(t1-t0);
    time = d.count()/1000.0;
    std::cout<<"sum = "<<sum<<"   cores "<<nc<<"  timing "<<time<<" s"<<"\n";
    if(nc==1) reftime = time;
    fileout<<nc<<" "<<time<<" "<<reftime/time<<"\n";
  }
  fileout.close();
}
```

`numerics/openmp_reduction.cpp`: Timings on AMD Ryzen 7 1700X Eight-Core processor



cores	time (seconds)	speed-up factor
1	1.457	1
2	0.730	1.996
3	0.487	2.992
4	0.365	3.992
5	0.293	4.973
6	0.244	5.705
7	0.211	6.905
8	0.201	7.249

OpenMP has some overhead, so don't expect any speed-up for short tasks.

37 Tips and tricks

Here ::: means "some lines of code". The triple dot "..." has a C++ meaning, it's an "ellipsis" :)

- How to pause execution:

```
cout<<"PRESS ENTER TO CONTINUE\n";
cin.ignore();
```

- Infinite loops can be typed as `while(true) \{:::\}`, or more to the point, as

```
#define ever (;;)
:::
for ever
{
 // break to get out
 :::
}
```

  I heartily recommend *stackoverflow.com* discussions. ⇐ ***This is THE tip!***

- (Linux) Long compiler messages can be piped to `more` (or `less`); Notice the "&" after the pipe character

```
g++ -std=c++11 program.cpp |& more
```

- If you are bored, try strangest-language-feature:

strangest_language_feature.cpp

```cpp
#include <iostream>

int main()
{
  int x[3]={1,2,3};
  std::cout<<"x[1]="<<x[1]<<"\n";
  std::cout<<"1[x]="<<1[x]<<"\n";
}
```

This one is absolutely priceless:
what-is-the-worst-real-world-macros-pre-processor-abuse-youve-ever-com .

This one is C, but works also in C++: fast inverse square root @Wikipedia. The original Quake III game really has the lines of code cited, see Quake III @githib

- If you think C++ has plenty of initialization, see Mike Lui's blog (Jan 3rd 2019)
  seriously-bonkers
  especially the Forest Gump adaptation
  c++ init_forest.gif

- (Linux) It's tedious to type `g++ -std=c++11 ...` all the time. Write a makefile, use an IDE, or set an alias,
  `alias g++='g++ -std=c++11'` (bash, put this line to $HOME/.bashrc and type ". $HOME/.bashrc")
  `alias g++ 'g++ -std=c++11'` (csh or tcsh, put this line to ${HOME}/.cshrc and type "source ${HOME}/.cshrc)
  and you only need `g++ program.cpp`

- `std::numeric_limits` (from `<limits>`) tells what traits a type has, hence they are "type_traits" For example, `std::numeric_limi`
  gives 2147483647. If you have in mind a class that can use an optimized algorithm and another related one that cannot, you can give the objects a trait. Based on that trait, the code can automatically decide if the optimized or the default algoritm is applied. See `advanced/traits.cpp`.

38 Some more C++ in the net

- Genetic algorithms are used for optimization
  Genetic-Algorithm-Library
- Bartlomiej Filipek gives examples how to use the standard algorithms instead of raw loops:
  Top-Beautiful-Cplusplus-std-Algorithms-Examples

39 Farewell words for C++ numerical programmers

- Prefer C++ Standard Library ***containers*** over dynamic arrays. Fixed size ***std::array*** is fast.
- Express math matrices and vectors using libraries, ***Eigen, Armadillo, MTL4, Blaze ...***.
- Link BLAS and LAPACK operations from ***MKL*** or ***OpenBlas***.
- Use ***C++ Standard Library algoritms***, not loops
- ***Forget type safety***, it's the least of C++ numerics problems.
- ***Move, don't copy.*** Use `std::ref()` or a lambda in standard algorithms such as `std::generate`.
- Pass functions as ***function objects***, ***lambda functions***, and only as a last resort use `std::function`. Avoid pointers to functions.
- ***Avoid verbose clutter.*** Hide ugly details to headers. Your main C++ code should express how the numerical problem is solved, not to teach the C++ programming language.

```
fft(data,1,n); // overloaded fft(), library call in a header
gsl_fft_complex_radix2_forward(data,1,n); // Same job. This is why I don't use C
```

- Use ***namespace protection***.
- ***Templates are you friends***.
- ***Use operator overloading to make cleaner code.*** Test thoroughly.

```
outfile << myobject; // overloaded <<, more readable than a function call
A = (B+M)*c ; // overloaded * and +, B,M complex matrices,c real.
A = multiply_complexmat_realvec(sum_complexmat_complexmat(B,M),c); // C? Oh no!
```

- Give data types short and descriptive names with *using*.

```
using my_cvec = lib1::part3::section4::vector<complex<double>> ;
```

- ***A fast algorithm in a slow computer beats a slow algorithm in a fast computer.***
- If you really need low-level access, use ***smart pointers*** (`std::unique_ptr`, `std::shared_ptr`). Why didn't I mention them until now? If possible, learn to live without *any* pointers. People used `std::auto_ptr` in their codes, only to find the feature was deprecated in C++11 and removed in C++17. What's the lifetime of `std::unique_ptr` and `std::shared_ptr`?

Those are my principles, and if you don't like them... well, I have others.
Groucho Marx