FYSS5120

Efficient Numerical Programming

Department of Physics

Vesa Apaja

October 6, 2026



JYVÄSKYLÄN YLIOPISTO
UNIVERSITY OF JYVÄSKYLÄ

Contents

1 Course itinerary **9**
Relation to other courses in JYU . . . 9

2 Version control using git **10**
2.1 Some `git` commands: . . . 11
2.2 Making as change in a file and undoing it . . . 12

3 Python vs Julia vs C++ in Numerical Performance **15**

4 Python **18**
4.1 About Python . . . 18
4.2 Timing and watching memory usage . . . 18
4.2.1 Timing with the `timeit` module . . . 18
Profiling short Python codes . . . 19
Profiling long Python codes . . . 19
4.2.2 Timing and watching memory usage in iPython . . . 20
4.3 Cache decorator . . . 20
4.4 Debugging Python segmentation fault . . . 26
4.5 General advice to speed up Python . . . 26

5 NumPy **27**
5.0.1 NumPy Sorting . . . 28
5.1 Elementwise product and matrix product . . . 31
5.2 NumPy BLAS . . . 33
Intel OneAPI and MKL . . . 34
Conda Intel Python environment . . . 36
5.3 BLAS and speed . . . 37

Intel and AMD Zen Architecture . . . 38
Where did Cholesky decomposition spend time? . . . 38
5.4 Dot product calculated three ways . . . 41
5.4.1 Blis BLAS library . . . 44
5.5 View, and shallow or deep copy . . . 45
5.5.1 Copying Python lists . . . 45
5.5.2 NumPy arrays: Copying data or Changing View? . . . 48
5.5.3 NumPy: `ndarray.resize()` or `numpy.resize()`? . . . 50
NumPy method `.resize()` . . . 50
NumPy function `resize()` . . . 51
5.5.4 Elementary array slicing . . . 52
5.5.5 Detail: How to set temporary NumPy print options . . . 53
5.6 NumPy matrix operations . . . 56
5.7 NumPy broadcasting instead of `for`-loops . . . 59
Adding a dimension to an array . . . 62
5.8 NumPy `einsum` tensor operations . . . 66
5.8.1 Tensor operations in Python and in Julia . . . 66
5.8.2 Computing $D_i = \sum_j A_i B_{ij}$ . . . 67
Using NumPy broadcasting and sum . . . 68
Using NumPy `einsum` . . . 68
5.8.3 Potential energy calculation with NumPy `einsum` . . . 70
5.8.4 Optimized `einsum` . . . 71
NumPy `einsum` promotion problem . . . 73
5.9 NumPy random numbers and seeds . . . 74
Sequential code . . . 76
Parallel code . . . 77
PRNG state collision . . . 78
Why are many PRNGs no good for cryptography? . . . 78

6 SciPy **81**
6.1 SciPy robust regression . . . 82
6.2 Sparse matrices . . . 85

7 NumExpr **92**

8 Numba **93**
8.1 Numba `jit` options . . . 93
8.2 About NumPy, Numba, and NumExpr . . . 94

9 Machine learning (ML) **98**
9.1 Fully connected, dense neural network . . . 98
What fast operations can be applied to input data? . . . 98
Universal approximator theorem . . . 102
Deep NNs . . . 102
9.2 Training neural networks . . . 104
9.2.1 Math details for one-hidden layer network forward and backward propagation . . . 105
9.2.2 Making use of the gradients: Gradient descent and fiends . . . 109
9.3 Automatic Differentiation (AD) . . . 110
9.4 Terminology: Batches, epochs, and overfitting . . . 116
9.5 Learning problems . . . 118
9.6 Teaching `scikit-learn` datasets . . . 122
9.6.1 Classification . . . 123
9.6.2 Continuous target . . . 126
9.7 DeepONet: Operators as neural networks . . . 128
9.8 Graph Neural Network (GNN) . . . 132
9.9 Odds and ins of ML: pattern recognition . . . 135
9.10 Probabilistic Programming and Bayesian Inference . . . 139
Bayesian inference in physics . . . 140
9.10.1 Credibility Interval (Bayesian) vs. Confidence Interval (Frequentist) . . . 141

9.10.2 Ordinary Least Squares vs. Bayesian linear regression . . . 142
Priors for (a, b, σ) . . . 143
Bayes' theorem and the posterior we want . . . 144
Hamiltonian Monte Carlo (HMC) . . . 145
9.10.3 Gaussian Process Regression (GPR) . . . 149
What is a smooth function? . . . 149
How to compute the probability of a function in the presence of data? . . . 154
9.11 JAX . . . 158

10 Parallel Python **160**
10.1 Threads, Multiprocessing, and Free-Threaded Python . . . 160
10.1.1 Start Methods . . . 165
10.1.2 `concurrent.futures` . . . 168
10.1.3 Parallelization and what you should *not* parallelize . . . 171
10.1.4 Examples of `concurrent.futures` . . . 173
10.2 `Multiprocessing` and `Pool` . . . 177
10.2.1 Safe locking with a context . . . 178
10.3 `Subprocess`: embarrasingly parallel jobs . . . 181
10.4 MPI Parallelism with `mpi4py` (*MPI for Python*) . . . 185
Send 'Hello World' to all processes . . . 188
Parallel Monte Carlo estimate of π . . . 190
Collective calls . . . 192
10.4.1 `send`/`recv` vs. `Send`/`Recv` . . . 192
Broadcasting a NumPy array . . . 196
10.4.2 Shutting down MPI jobs after an exception . . . 196
Aborting `mpi4py` . . . 197
10.4.3 Non-blocking communication . . . 198
10.5 When parallel code gives the wrong result . . . 199

11 Python as a glue language **201**
11.1 Python extensions and embedding Python . . . 201
11.2 pybind11 and nanobind . . . 203
11.3 Cython . . . 203
11.3.1 Creating a standalone executable with Cython . . . 207
Cython and C++ . . . 209

12 GPU computations **210**
12.1 CuPy and Numba cuda . . . 210

13 Julia **213**
13.1 Julia: language highlights . . . 214
13.2 Julia command prompt . . . 215
13.3 Julia IDEs . . . 216
13.3.1 Julia for Visual Studio Code . . . 216
13.3.2 Adding Julia to Jupyter notebook or Jupyter lab . . . 217
13.3.3 Pluto . . . 217
13.4 Calling Julia from Python . . . 219
13.5 Calling Python from Julia . . . 220
13.6 Julia arrays, matrices, references, and copies . . . 221
13.7 Julia broadcasting . . . 222
13.8 Julia array loop . . . 223
13.9 Julia Automatic Differentiation (AD) . . . 224
13.10Julia Differential Equations . . . 226
13.11Julia Macros . . . 228
13.12Julia StaticArrays . . . 229
13.13Julia Metaprogramming . . . 231
13.14Multiple dispatch . . . 236
Use concrete types, avoid `Any` and avoid boxing arguments . . . 241
FFT . . . 243

The Expression Problem . . . 245
Adding a type and a method in Python . . . 246
Adding a type and a method in C++ . . . 247
Adding a type and a method in Julia . . . 248
What can Julia multiple dispatch do in science? . . . 249

14 Appendix: Python basics **250**
14.1 Python package managers and IDEs . . . 250
14.2 Python Ecosystem Overview . . . 250
14.2.1 `pip` package manager . . . 251
14.2.2 Where does Python search for modules? . . . 252
How to exclude user-installed packages . . . 252
14.2.3 Python virtual environments . . . 253
Basic Python virtual environment . . . 253
`pip freeze` . . . 253
Conda environments . . . 254
14.2.4 VS Code, Spyder, iPython, Jupyter Notebook, and Jupyter Lab . . . 255
Jupyter Lab . . . 256
14.3 Updating Python packages using `pip` package manager . . . 257
14.4 Arbitrary precision calculations . . . 258
14.4.1 Large integers . . . 258
14.4.2 Long floats with `mpmath` . . . 258
14.5 List comprehensions . . . 261
14.6 String concatenation . . . 263
14.7 Advanced unpacking . . . 264
14.7.1 Unpacking of function arguments . . . 264
14.7.2 The beauty of the extended call syntax: `*args` and `**kwargs` . . . 265
Using `*args` to absord extra arguments . . . 267
14.8 Decorators . . . 269
14.8.1 Python preprocessing and adding code for debugging . . . 271

14.8.2 Turning a decorator on/off using `python -O` . . . 273
Decorator classes . . . 274
14.8.3 Decorators with arguments . . . 275
14.9 Iterables, generators and `yield` . . . 276
Example: Generator for watching a file . . . 280
14.9.1 Generator pipelines . . . 281
14.10 Underscores . . . 283
14.10.1 Underscores in Python names . . . 284
14.10.2 Magic methods . . . 285
Context managers . . . 291
Generators in context managers . . . 291
Example of a context manager . . . 293
14.11 Coroutines . . . 294
14.12 Delegating work to subgenerators with `yield from` . . . 298
14.13 Pattern matching with `match-case` - and a warning . . . 301
14.14 Python `class` . . . 303
14.15 `@classmethod` and `@staticmethod` decorators . . . 303
14.15.1 `@dataclass` decorator . . . 305
14.15.2 `@property` decorator . . . 306
14.15.3 Changing behaviour of a library class method . . . 312
14.15.4 Poking a method to a class . . . 312
14.15.5 Poking an attribute to an object . . . 314
14.15.6 Simulation and Measurements in Python . . . 315
14.15.7 Saving and reading HDF5 data . . . 319
14.15.8 Python Serialization . . . 321

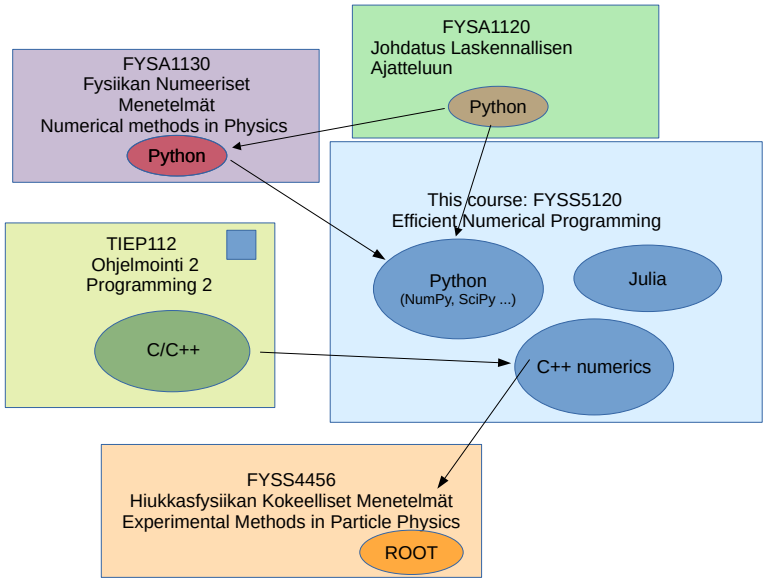
15 Pandas **324**

16 Polars **325**

1 Course itinerary

See the course web page (link).

Relation to other courses in JYU



2 Version control using git

motivation Don't rely on your memory! Many times I've made changes to a code just to see some other part of the program stopped working. Small code "improvements" are the most dangerous ones. They seem benign, but can introduce a major flaw, often surfacing after a few days when you've happily forgotten what you did. You need a way to get back to an older version, or at least view what you've done recently.

Using git. Let's assume the code you want to track is in the current directory and subdirectory `pythoncodes/`:

```
$ git init
Initialized empty Git repository in .../.git/
$ git add lectures
$ git status # list of new files "not staged for commit"
$ git commit -a -m "Initial commit" # commit git database with a comment
```

Files or directories listed as "Untracked files" won't be tracker by git, unless you `git add` them.

2.1 Some `git` commands:

`git init` Initialize a new Git repository in the current directory.

`git clone <url>` Clone an existing repository from a remote server.

`git status` Show the current state of the working directory and staging area.

`git add <file>` Stage changes (add file to the next commit).

`git add .` Stage all modified and new files.

`git commit -m "message"` Commit staged changes with a descriptive message.

`git log` Show commit history.

`git diff` Show changes not yet staged.

`git diff --staged` Show changes staged for commit.

`git branch` List branches.

`git branch <name>` Create a new branch.

`git checkout <branch>` Switch to another branch.

`git checkout -b <branch>` Create and switch to a new branch.

`git merge <branch>` Merge another branch into the current one.

`git pull` Fetch and merge changes from remote.

`git push` Push local commits to remote repository.

`git remote -v` Show configured remote repositories.

`git rm <file>` Remove a file from the repository.

`git reset <file>` Unstage a file.

`git restore <file>` Discard local changes in a file.

2.2 Making as change in a file and undoing it

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 stores the changes for later stash apply/pop.

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, but it's a delicate task that often needs manual intervention (search "git merge gif" to see what merge may do).

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 shown by `git log`. Knowing the commit hash, you can always come back to an old commit with `git checkout <commit hash>`.

3 Python vs Julia vs C++ in Numerical Performance

- **Execution model**
  - Python: Interpreted (with optional JIT via external tools like Numba)
  - Julia: Just-In-Time (JIT) compiled with LLVM
  - C++: Ahead-of-time (AOT) compiled
- **Type system**
  - Python: Dynamically typed (runtime type resolution)
  - Julia: Dynamically typed, but specializes on types at runtime
  - C++: Statically typed (compile-time type resolution)
- **Type stability (critical for speed)**
  - Python: Weak — types often change at runtime
  - Julia: Strong if code is written type-stably (performance depends on this)
  - C++: Strong — enforced by compiler
- **Compilation and optimization**
  - Python: Minimal optimization in core language
  - Julia: Aggressive specialization and LLVM optimization
  - C++: Highly optimized by mature compilers (GCC, Clang)
- **Loops and vectorization**
  - Python: Slow loops; relies on NumPy for vectorized operations

  - Julia: Fast loops comparable to C
  - C++: Fast loops with full control over optimization

- **Memory management**
  - Python: Automatic garbage collection (GC), high overhead
  - Julia: Garbage collected, but optimized for numerical workloads
  - C++: Manual (or RAII); no GC overhead

- **Array storage and access**
  - Python: NumPy arrays efficient, but Python-level access is slow
  - Julia: Native arrays are fast and contiguous
  - C++: Full control over memory layout

- **Function call overhead**
  - Python: High
  - Julia: Low after compilation
  - C++: Very low, frequent inlining

- **Parallelism and threading**
  - Python: Slightly limited by Global Interpreter Lock (GIL)
  - Julia: Built-in multithreading and distributed computing
  - C++: Full control via threads/OpenMP/MPI

- **Metaprogramming and specialization**
  - Python: Limited runtime flexibility

  - Julia: Multiple dispatch enables powerful specialization
  - C++: Templates allow compile-time specialization

- **Typical numerical performance**
  - Python: Slow unless using optimized libraries
  - Julia: Near C/C++ speed if you avoid traps that prevent optimization
  - C++: Baseline for maximum performance

Common quidelines to speed up code, either Python, Julia or C++:

- NumPy: Vectorize array operations
- Avoid repeated allocation and deallocation of memory – especially in Julia, see GC
- Parallelize (depends on problem and hardware)
- Parallelize massively using GPU (depends on problem and hardware)

4 Python

The Python basics can be found in the Appendix **??**.

4.1 About Python

Python is an interpreted language, the interpreter is the program `python`, which usually runs the `CPython` interpreter. Interpreting instructions takes longer than executing compiled code (as in C, 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, but Python programs are more flexible and testing can be done line by line. Nobody uses "pure Python", practically all numerical work relies on the compiled and optimized NumPy and SciPy stack and their derivatives.

4.2 Timing and watching memory usage

4.2.1 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`. For parallel jobs the best timer is `time.process_counter`, which measures the elapsed wall 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.

Profiling short Python codes

A native Python profiler is the module `profile`. Profiling an entire code, sorted in decreasing cumulative time, [1]

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

In Python console,

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

Profiling long Python codes

The module `cProfile` is a C extension and has smaller overhead than `profile`. Usage is similar to `profile`,

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

```
$ 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 14.8.

4.2.2 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
```

4.3 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 (means minimal invocation) to `@lru_cache(maxsize=None)`. `@lru_cache` has a few options that may improve performance in some cases.

The decorator `@staticmethod` tells that the method is bound to the class itself, not to a specific instance. In the following example method `b()` doesn't need access to instance-specific data (no `self.something`), so it's suited as a class method.

```
class MyClass:
    x = 1.1
    def a(self, y):
        print(self.x, y)
        self.x = y

    @staticmethod
    def b(y):
        print(y)

# instance my, call method a
my = MyClass()
my.a(10.11)

# call class method b "directly"
MyClass.b(22.00)
```

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))
```

4.4 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. Now you at least know what failed so keep digging.

4.5 General advice to speed up Python

Important:

- Use ***compiled libraries*** (SciPy, NumPy,...) as much as possible – don't use "pure" Python for heavy computations.
- Represent numerical data using ***NumPy***
- ***Vectorize*** loop away, don't do anything 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.
- ***Cache*** results of slow, repeated operations.
- `for` loops can be made fast using ***Numba*** (keep loops as simple as possible, don't call any non-Numba functions).

Things that may give small speed gains, some of these improve readability and/or maintainability:

- Favor ***local variables*** over global ones
- ***List comprehensions*** are usually a bit faster than `for` loops. The interpreter does a better job with such a limited list comprehension loop compared with a general `for` loop.
- 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 an entry in a list is very slow!
- Testing membership in a ***set*** is as fast as testing membership in a dictionary.
- Try ***built-in methods*** first. Some data type have their own optimized methods for common tasks, such as sorting.

5 NumPy

`NumPy @numpy.org` is for vectorized manipulation of numerical data.

5.0.1 NumPy Sorting

NumPy method `numpy.ndarray.sort` sorts the array in place, while `numpy.sort()` returns a sorted copy of an array.

```
import numpy as np
a = np.array([5,2,6,1,5,7,4,3])
b = a.sort() # b is a sorted copy of a, a is not affected
a.sort()     # a is sorted, the original ndarray np.array([5,2,6,1,5,7,4,3]) is overwritten
```

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, based on `argsort()`

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]]
print('data 0',data[:,0])
print(data[:,0].argsort()) # [2 1 0]
print(data[[2,1,0]])
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
```

Do not use the legacy NumPy matrix class `np.matrix`, use 2D `np.array` instead.

The `@`-operator does matrix multiplication or dot product.

5.1 Elementwise product and matrix product

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

```
>>> import numpy as np
>>> a = np.array( [[1,2],[3,4]])   # 2x2 array, represents a matrix
>>> b = np.array( [[2,2],[2,2]])   # 2x2 array, represents a matrix
>>> a*b                            # elementwise multiplication
array([[2, 4],
[6, 8]])
>>> a@b                            # matrix product
array([[ 6,  6],
[14, 14]])
```

The operator ** is elementwise, a**2 is the same as a*a.

Array shape (N,) ≠ array shape (N,1), they convert differently.

```
>>> import numpy as np
>>> a = np.random.random(5)
>>> a.shape
(5, )
>>> (a.T).shape # transpose and its shape <= transpose does nothing!
(5, )
>>> b = a.reshape(5, 1)
>>> b.shape
(5, 1)
>>> (b.T).shape # transpose and its shape
(1, 5)
```

5.2 NumPy BLAS

The speed of NumPy matrix and vector operations relies on the speeds of the compiled BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage). 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`. [2] Grepping avx, I found that there are AVX2 instructions in, e.g., `numpy/core/_multiarray_umath.cpython-38-x86_64-linux-gnu.so`.

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:
```

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

```
    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
```

5.3 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." [3]

[3] 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[4]

```
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

[4] 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_` 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

5.4 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.

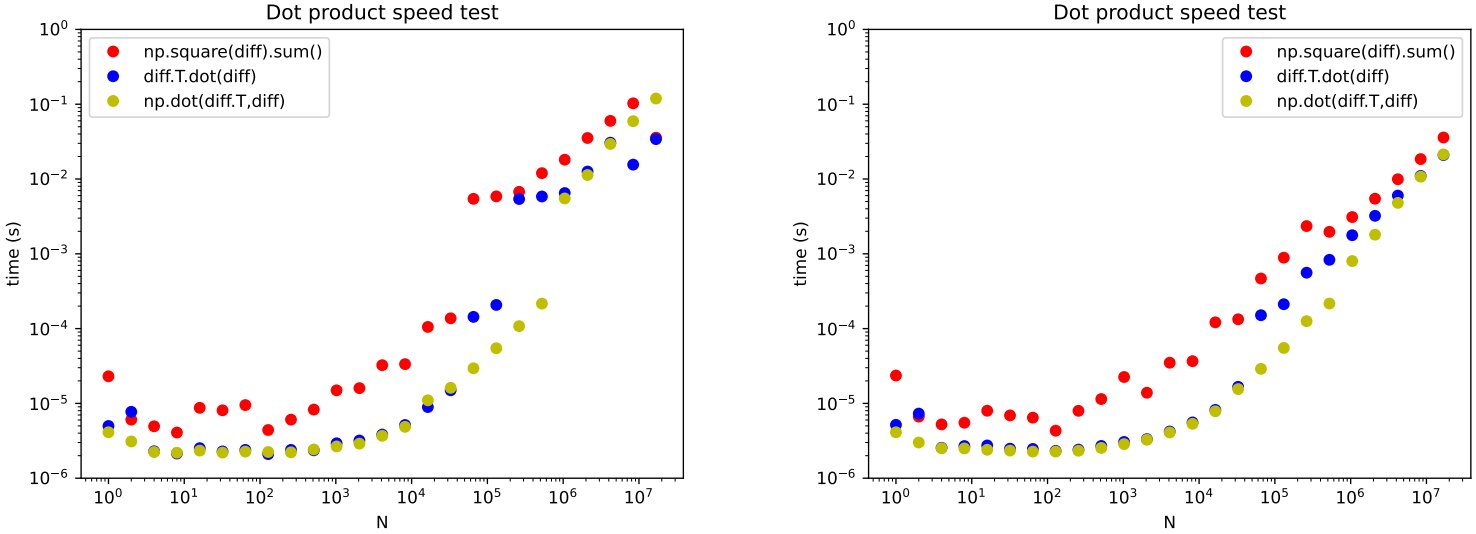


Figure 1: The timings for AMD Ryzen 3900x show that `numpy.dot()` is always fast. The figures were computed using
`OPENBLAS_NUM_THREADS=6 python dot_product_speed.py`
`OPENBLAS_NUM_THREADS=1 python dot_product_speed.py`
respectively. The sudden jump in timings around 10^5 in the left figure is caused by OpenBlas internal decision to change strategy from a single-threaded to a multi-threaded operation, and starting threads takes a while.

dot_product_speed.py

```
import numpy as np
from mytimer import timer

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

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

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

if __name__=='__main__':
    arr_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))
        for f, lab, col in zip(funs, labs, cols):
            r, t1 = f(x, y)
            lab = lab if i==0 else ''  # ternary, label only for first value in the data set
            arr_res.append([N, t1, 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)

    plt.title('Dot product speed test')
    plt.xlabel('N')
    plt.ylabel('time (s)')
    plt.ylim([1e-6, 1.0])
    plt.legend()
    plt.show()
```


Above I used the timer decorator,

mytimer.py

```
from time import perf_counter as T
def timer(fun):
```

5.4.1 Blis BLAS library

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

```
$ conda create -n p3.13  python=3.13
$ conda config --add channels conda-forge
$ conda config --get channels
# --add channels 'defaults'   # lowest priority
# --add channels 'conda-forge'   # highest priority
$ conda activate p3.13
(p3.13) $ conda install -c conda-forge numpy scipy "libblas=*=*blis"
```

This way you get NumPy and Scipy that use blis. Installation using `pip install blis` seems to be a Cython wrapper that does not automatically make NumPy or SciPy use 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
```

After that, you also need to recompile NumPy and SciPy and link them to blis, so you see it is a long journey.

5.5 View, and shallow or deep copy

5.5.1 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 next 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']])
```

5.5.2 NumPy arrays: Copying data or Changing View?

Avoid unnecessary copying of data.

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]
>>> arr = np.asarray(mylist)    # creates a copy, because list is not NumPy array
```

Using `np.asarray` is a safe bet, it make sure the output is a `np.array`.

```
>>> import numpy as np
>>> myarr = np.array([1,2,3,4])
>>> arr = np.asarray(myarr) # no copying: just sets arr=myarray, because myarr already is a np.array
```

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 attributes, for a complete list see flags (link).

```
>>> import numpy as np
>>> a = np.arange(6)
>>> b = a.reshape(2, 3)
>>> print(a.flags)
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
```

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

5.5.3 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.

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 0 1 0 1 0 1 0 1 0 1 0 1]
```

5.5.4 Elementary 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]`.

5.5.5 Detail: 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***[5].

[5]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.

5.6 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 conversion 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)`.

5.7 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 8.

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.

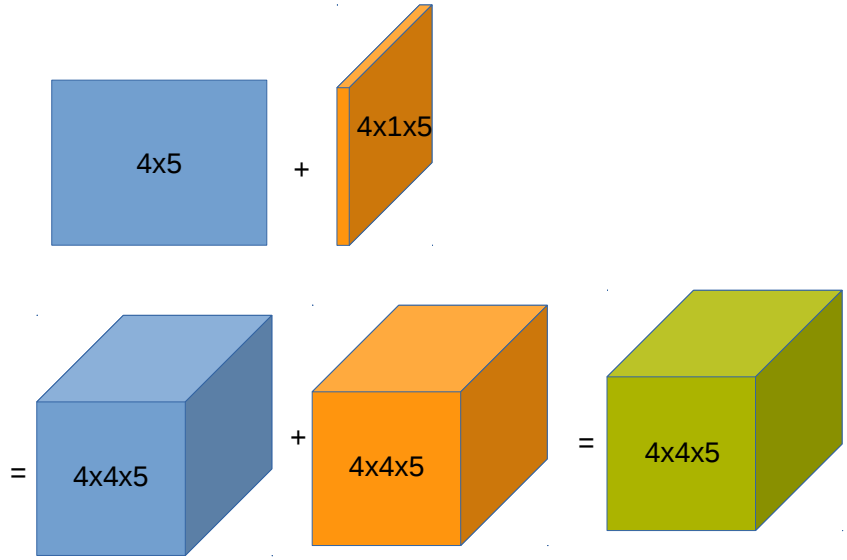
The final shape is not necessarily the shape of either of the two input arrays.
Looking only at array shapes and some operator "op", this is how broadcasting re-shapes arrays:

```
(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 $(i, j, \mathit{dimension})$ 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[6]

[6] 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.

```
(N,1,D) - (N,D) = (N,N,D)
```

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:

potential_simple.py

```
#
# Demonstrates NumPy broadcasting
# Very slow in this specific case,
# better: scipy.spatial.distance.pdist
# even better: Numba njit for loops !
#
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')
```

5.8 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.

5.8.1 Tensor operations in Python and in Julia

These two codes compute the matrix product,

$$C_{ij} = \sum_k A_{ik} B_{kj} \ ,$$

```python
# Python
import numpy as np
A = np.random.rand(2, 3)
B = np.random.rand(3, 4)
C = np.einsum('ik,kj->ij', A, B)
```

You know that `k` is summed over because it doesn't appear in the result, after `->`.
The Julia macro `@tensoropt` is a speed-optimized version of `@tensor`.

```julia
# Julia
using TensorOperations
A = rand(2, 3)
B = rand(3, 4)
@tensor C[i, j] := A[i, k] * B[k, j]
```

For example, in Hartree-Fock theory of atoms and molecules, the Fock matrix F is

$$F_{mn} = H_{mn} + 2\sum_{r,s} D_{rs}b_{mnrs} - \sum_{r,s} D_{rs}b_{mrns}$$

Computation of F can be coded as

```python
# Python
import numpy as np
np.copyto(F, H) # copy values to existing F
# slicing F[:,:] = H  does the same a bit faster, without checking array dimensions
F += 2 * np.einsum('rs,mnrs->mn', D, b) # sum over r,s
F -= np.einsum('rs,mrns->mn', D, b)     # sum over r,s
```

```julia
# Julia
using TensorOperations
F .= H  # copy values to existing F elementwise
@tensor F[m,n] += 2*D[r,s]*b[m,n,r,s] # sum over r,s
@tensor F[m,n] -= D[r,s]*b[m,r,n,s]   # sum over r,s
```

5.8.2 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 computes the sum

$$D_i = \sum_j A_i B_{ij} \ .$$

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 `A[:,np.newaxis]*B` cause NumPy to 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 `->`.

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 $C_{ij} = \sum_k A_{ij} B_{jk}$.

5.8.3 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.

5.8.4 Optimized `einsum`

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:

```
$ pip install opt_einsum
```

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`, `optimize : {bool, list, tuple, 'greedy', 'optimal'}`.

So let's add this iPython code to the game,[7]

[7] Can you reverse engineer what the code computes? For comparision, the Python script `tensor_contraction.py` shows how to use `timeit.timeit()`.

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
```

and press enter. The `%load` is also line magic.[8]

[8] Cell magic `%%timeit` times the whole cell multiple lines.

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 is that the result 300 won't fit to the original `np.int8` integer type. If you try directly `np.int8(300)`, you get the error

`OverflowError: Python integer 300 out of bounds for int8`

but `np.einsum` doesn't check this.

5.9 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:

> Although recycling is couraged for environmental reasons, never recycle random numbers.

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

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. Some popular and good ones are:

- Mersenne Twister MT19937 or MT19937-64 @Wikipedia is a well-tested, widely used generator. First version 1997. Pros: Long period ($2^{19937} - 1$), high quality. Cons: Large state buffer (2.5 KiB) and non-vectorizing design, a bit slow.
- Permuted congruential generator PCG @Wikipedia A newcomer (2014) with long period ($2^{128} \approx 3 \times 10^{38}$).
- Xoshiro256 (@Wikipedia does operations `xor`, `shift` and `rotate`).

Core Python relies on MT19937,

```
>>> 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. [9]

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

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

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

============================================ ===
BitGenerator Streams that work with Generator
```

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

```
------------------------------------------ ---
MT19937
PCG64
PCG64DXSM
Philox
SFC64
========================================== ===
...
>>> print(np.random.default_rng())
Generator(PCG64)
```

The NumPy PRNG has two layers.

- **Layer 1**: The ***BitGenerator*** is the algorithm that produces the (pseudo)random bits (often 64 bits)
- **Layer 2**: The random bits given by the ***BitGenerator*** are used to generate random numbers that sample the wanted distribution (uniform, normal, exponential, ...).

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 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])
```

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?

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.[10]

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:

> 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[11], and the PRNG algorithm gets stuck there. Even if the state is only partly

[10] 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.

[11] 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.

zeroes, you end up with less random output.[12]

PRNG state collision

What may happen if workers use the same PRNG algorithms with distinct yet too similar seeds? Remember a PRNG evolves its internal state deterministically and outputs results. With a touch of bad luck, the state of one copy of PRNG may coincide (collide) with a state some other PRNG has visited. From then on, the PRNG will evolve the same route as the other PRNG did, and you are recycling random numbers. Avoiding such collisions is impractical if not impossible, all we want is to make them very improbable.

Why are many PRNGs no good for cryptography?

In NumPy, the MT19937 state can be printed out,

```
>>> import numpy as np
>>> rng = np.random.Generator(np.random.MT19937())
>>> state = rng.bit_generator.state
>>> print(state['state']['key'])
>>> print(len(state['state']['key']))
```

and you find the key to be 624 64-bit integers. Every time you generate a random number, the key is updated ("tempered"), but the update is deterministic. As a result, anyone who sees 624 random numbers generated by a MT19937 can reverse-engineer the state and clone the PRGN.

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 10 child SeedSequences and pass them to child processes.
```

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

```
>>> 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

```
>>> 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 debugging, but often you need to produce fresh data the *next time* you run the code, so you need seeds that change with *time*. [13] 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

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

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

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.

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.

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)]
```

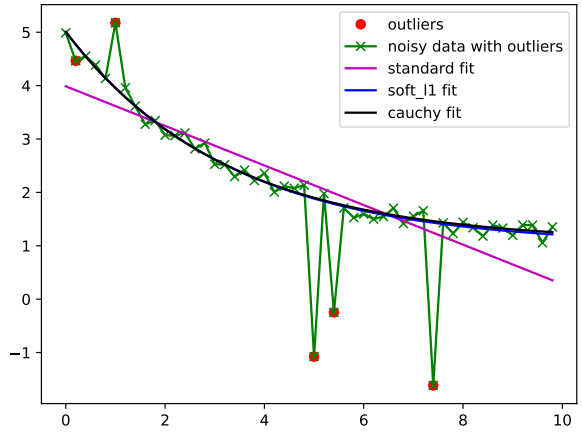
6 SciPy

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`.

6.1 SciPy robust regression

We did linear regression in NumPy (**??**). 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`.

6.2 Sparse matrices

```
from scipy.sparse import coo_array, csr_array
```

This brings in the two main sparse matrix formats, ***COO*** (COOrdinate or triplet format) and ***CSR*** (Compressed Sparse Row format).

COO is convenient for building a sparse matrix; CSR is convenient for using it.

For example,

$$A = \begin{pmatrix} 5 & 0 & 8 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 6 \end{pmatrix} \tag{4}$$

```
import numpy as np
from scipy.sparse import coo_array

row = np.array([0, 0, 1, 3, 3])
col = np.array([0, 2, 1, 0, 3])
data = np.array([5, 8, 1, 3, 6])
A = coo_array((data, (row, col))) # sparse (4,4) array
print(A.toarray()) # dense format for checking
# array([[5, 0, 8, 0],
#        [0, 1, 0, 0],
#        [0, 0, 0, 0],
#        [3, 0, 0, 6]])
```

Here the shape (4,4) was inferred, but with all-zero rows or columns you must specify the shape,

```
import numpy as np
from scipy.sparse import coo_array

row = np.array([0, 0, 1, 3, 3])
col = np.array([0, 2, 1, 0, 3])
data = np.array([5, 8, 1, 3, 6])
A = coo_array((data, (row, col)), shape = (4, 6)) # sparse (4,6) array
```

```
print(A.toarray())
#array([[5, 0, 8, 0, 0, 0],
#       [0, 1, 0, 0, 0, 0],
#       [0, 0, 0, 0, 0, 0],
#       [3, 0, 0, 6, 0, 0]])
```

COO allows ***duplicate coordinates***, which are summed over when constructing a dense or CSR format array.

```
import numpy as np
from scipy.sparse import coo_array

row = np.array([0, 0, 0])
col = np.array([1, 1, 1])
data = np.array([5.0, 8.0, 1.0])
A = coo_array((data, (row, col)), shape = (3, 3))
print(A.toarray())
# [[ 0. 14.  0.]
#  [ 0.  0.  0.]
# [ 0.  0.  0.]]
```

There is a very good reason for the COO add-duplicates convention. Consider a Finite Element Method (FEM) calculation, where different elements share a node. The bookkeeping would be complicated, but the COO format lets one to define each element as if they were separate, and dump them all to a COO.

Consider three collinear nodes and two elastic spring elements that share the mid node. Write down the stiffness matrices of both elements as if they existed alone, and just append the row, col and data. Finally, let COO do the assembly.[14]

[14] Multiplied by displacements, the stiffness matrix gives the forces.

elastic_coo.py

```python
# Simple combination of FEM elastic data for overlapping nodes using SciPy'c COO format
import numpy as np
from scipy.sparse import coo_array

# Lists use in to construct the COO format array
rows = []
cols = []
data = []

# --- ELEMENT 1 ---
elem1_nodes = [0, 1] # global node indices
elem1_stiffness = np.array([
    [ 100, -100],
    [-100,  100]
])
print("Element 1 stiffness matrix:\n", elem1_stiffness)

for i in range(2):
    for j in range(2):
        rows.append(elem1_nodes[i])
        cols.append(elem1_nodes[j])
        data.append(elem1_stiffness[i, j])

# --- ELEMENT 2 ---
elem2_nodes = [1, 2] # global node indices
elem2_stiffness = np.array([
    [ 200, -200],
    [-200,  200]
])
print("Element 2 stiffness matrix:\n", elem2_stiffness)

for i in range(2):
    for j in range(2):
        rows.append(elem2_nodes[i])
        cols.append(elem2_nodes[j])
        data.append(elem2_stiffness[i, j])

# --- STIFFNESS MATRIX ASSEMBLY USING COO ---
# Create the sparse matrix for 3 nodes (3x3 shape)
stiffness = coo_array((data, (rows, cols)), shape=(3, 3))

# Convert to a dense array to see the automatic summation at overlapping Node 1
print("\nStiffness Matrix:")
print(stiffness.toarray())
```

Starting with SciPy 1.17.0, indexing and slicing support was officially added to `coo_array`. There is one obvious downside of COO: accessing data row-wise is inefficient, so computing matrix vector products Ax is slow. CSR was made for fast access of sparce matrix rows. The computation of Ax is

$$y_i = \sum_{j=0}^{n-1} A_{ij} x_j \ , \tag{5}$$

but in the sparse case each row j of A contains only a few non-zero elements. Instead of summing over all j, sum only over the non-zero elements,

$$y_i = \sum_{k=\mathrm{indprt}_i}^{\mathrm{indprt}_{i+1}} \delta_{j,\mathrm{indices}_k} \, A_{ij} x_j \ . \tag{6}$$

A sparse CSR array is made of

- data: nonzero values
- indices: column index of each value
- indptr: row ranges

The last entry will look peculiar, but it's constructed to make the computation $y = Ax$ loop trivially,

```
for i in rows:
    for k in range(indptr[i], indptr[i+1]):
        j = indices[k]
        y[i] += data[k] * x[j]
```

A typical workflow is [row, col, data] → COO → CSR,

```
# ipython
>>> import numpy as np
>>> from scipy.sparse import coo_array, csr_array
>>> row = np.array([0, 0, 1, 3, 3])
>>> col = np.array([0, 2, 1, 0, 3])
>>> data = np.array([5, 8, 1, 3, 6])
>>> A = coo_array((data, (row, col)))
>>> B = A.tocsr()
>>> x = np.array([1,2,3,4])
>>> %timeit A@x   # 3.48 \mu s ± 84.7 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
>>> %timeit B@x   # 1.91 \mu s ± 17.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Construct the 1D finite-difference Laplacian,

$$-\frac{d^2u}{dx^2} \approx -\frac{1}{h^2}\begin{pmatrix} 2 & -1 & 0 & 0 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots & 0 \\ \vdots & & & & & & \\ 0 & \dots & 0 & 0 & & -1 & 2 \end{pmatrix} \tag{7}$$

```python
import numpy as np
from scipy.sparse import diags

N = 100_000

A = diags(
    [-np.ones(N-1),
      2*np.ones(N),
     -np.ones(N-1)],
    offsets=[-1, 0, 1],
    format="csr"
)

print(A.shape)
print(A.nnz)
```

Instead of $10^5 \times 10^5 = 10^{10}$ entries taking 80 GB, we needed only 299998 entries taking 2.3 MB.

7 NumExpr

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 compilation overhead is significant.

`NumExpr` is excellent for elementwise operations, for example

```
import numpy as np
import numexpr as ne

# large random element arrays
N = 1000000
a = np.random.rand(N)
b = np.random.rand(N)
c = np.random.rand(N)

# Evaluate an expression using NumExpr
result = ne.evaluate("a + b * c")
```

8 Numba

Just-in-time compilation (JIT @Wikipedia) means compilation at run time, during execution. One JIT compiler is provided by Numba @pydata.org . Numba analyzes Python bytecode, performs type inference, creates Numba IR → LLVM IR → optimized native machine code.

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*** of the function. Try `Numba.typeof(a)` to find how Numba types an object `a`.

8.1 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 interpreter and raises 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...
```

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),

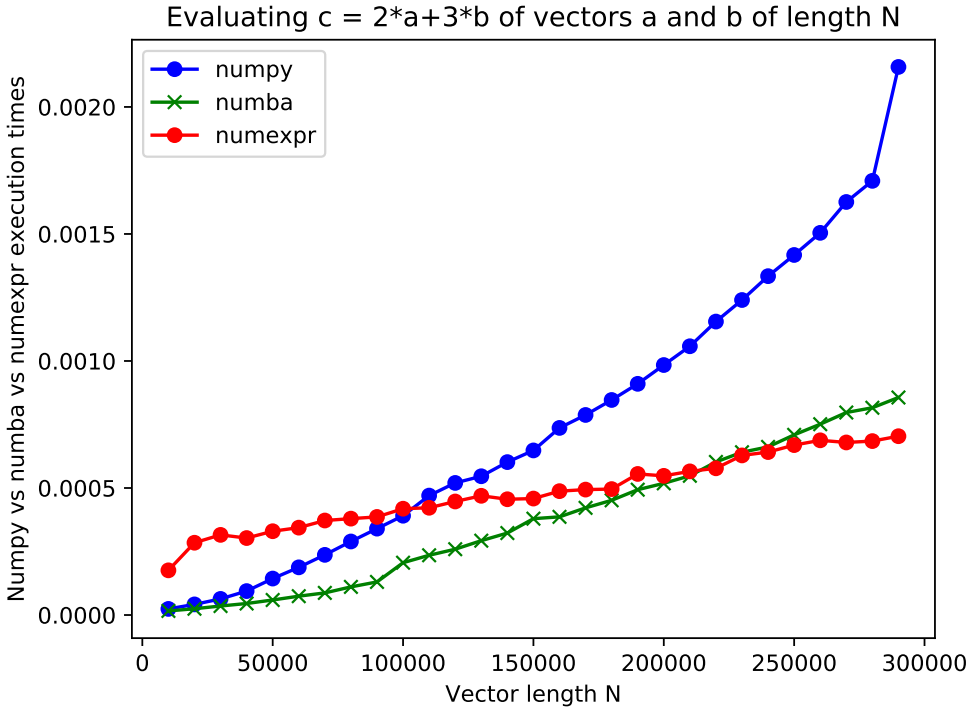
```
from numba import jit, int32, float32
@jit(["int32(int32)", "float32(float32)"])
def f(x):
```

8.2 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.

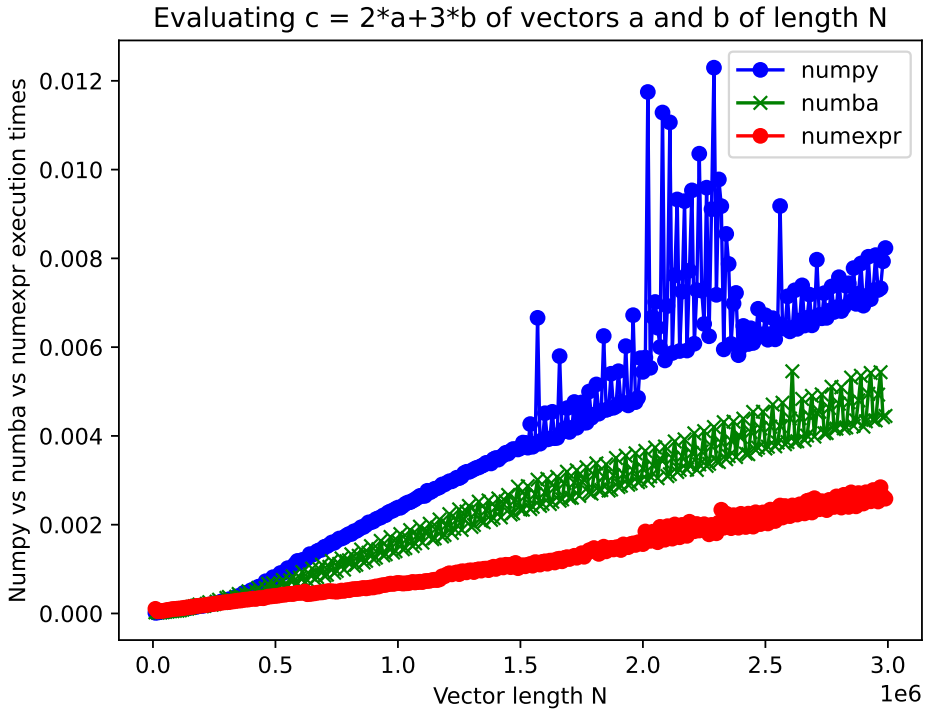
`Numba` and `NumExpr`, as well as `NumPy`, may autoparallelize execution, without you doing nothing.
Running an internally parallel code in a cluster will upset fellow researchers!

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. NumExpr can also use Intel MKL library routines.

Here's a longer test (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))
```

9 Machine learning (ML)

Python has become a very popular language in ML and data mining, which supposedly is behind the increasing popularity of Python. I'll introduce machine learning in this Python section, because I assume many of you are using Python in that context, but Julia is equally well suited. The bread and butter in ML is computation of gradients, Jacobians, and Hessians, and handling of tensors.

The Python ecosystem is broad and mature but fragmented, and has an overall heavy reliance on the NumPy/SciPy stack. The Julia ecosystem is more unified and composable, with an emphasis on scientific ML, such as differential equations.

9.1 Fully connected, dense neural network

A Neural Network (NN) does regression

```
y = neural_network(x) := NN(x)
```

where `x` is the input and `y` is the output. Obviously, we need some known `(x,y)` pairs to be able to ***teach*** the NN. Later we divide these known pairs to a `training set` and a `validation set`. The goal is to find a NN that can *generalize*, yield reasonable values `y` also for `x` not in the original set of known `(x,y)` pairs.

What fast operations can be applied to input data?
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)
```

It's also possible to do the matrix-vector operation the other way round,

```
y = np.dot(x, w) # w.shape is (n,m)
```

The `n*m` elements of the ***weight*** array are free 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 the 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. This problem is solved adding ***nonlinearity*** in the form of a ***nonlinear activation function***. Frequently used activations are $\tanh(x)$ and

$$\begin{aligned} \text{relu}(x) &= \max(0, x) \quad \text{ReLU (Rectifying Linear Unit)} \\ \text{sigmoid}(x) &= \frac{1}{1+\exp(-x)} \ . \end{aligned}$$

The ReLu activations just cuts off negative signals - that is enough for nonlinearity. Keras built-in activations can be listed and plotted using the code keras_activations.py.
Adding activation, we have a NN model

```
y = activation(np.dot(w,x) + b)
```

With ReLU activation and binary (1 or 0) output, this is called a ***perceptron***. A perceptron is not a good regression model for many functions $y = f(x)$, but adding more ***hidden neurons*** changes everything. Add one more layer of weights and biases, [15]

[15] A one-hidden layer NN can be written as a sum over hidden neurons, $y = \sum_i \alpha_i \sigma(w^i x + b^i)$.

```
y = np.dot(w2,activation(np.dot(w1,x) + b1) + b2) # One hidden layer universal approximator
```

In math, `activation()` is often written as $\sigma()$, so this means (I write the layer indices as superscripts to make way to later element notation such as w_{ij}^2)

$$\boldsymbol{y} = \boldsymbol{w}^2\sigma(\boldsymbol{w}^1\boldsymbol{x} + \boldsymbol{b}^1) + \boldsymbol{b}^2 \ .$$

We can split the operations and define a ***hidden layer*** between the input layer and the output layer,

```
x                       # input layer
z = np.dot(w1,x) + b1   # hidden layer input
h = activation(z)       # hidden layer output is output layer input
y = np.dot(w2,h) + b2   # output layer
```

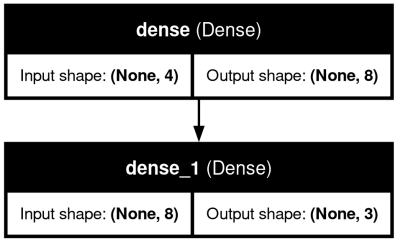
In Keras, the 4-8-3 NN model shown on the right can be defined like this:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.utils import plot_model

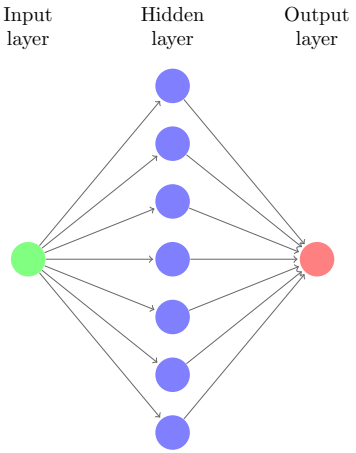
model = Sequential() # NN type
model.add(Input(shape=(4,)))
model.add(Dense(8, kernel_initializer='uniform', activation='relu'))
model.add(Dense(3, kernel_initializer='uniform'))

print(model.summary())
plot_model(model, to_file='model.png', show_shapes=True, expand_nested=True, show_layer_names=True)
```

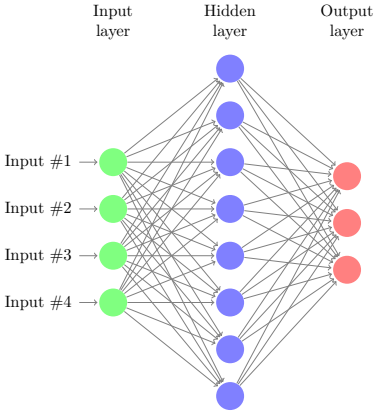
As usual, weights and biases are initialized to uniformly distributed random numbers. Summary gives the number of parameters, 40+27 = 67. [16]



[16] From weight (8,4) + bias (8,) = 8*4+8 = 40, and weight (8,3) + bias (3) = 8*3 +3= 27.



One input and one output



Many inputs and many outputs

Universal approximator theorem

The one-hidden-layer NN is a ***universal function approximator*** (more precisely given in Universal approximation theorem @Wikipedia)

> With sufficiently many hidden neurons and a suitable activation function, a one-hidden-layer NN can arbitrarily well approximate any continuous function $y = f(x)$, meaning there is a NN that satisfies
>
> $$|\mathrm{NN}(x) - f(x)| < \epsilon \ , \tag{9}$$
>
> for any x and any $\epsilon > 0$.

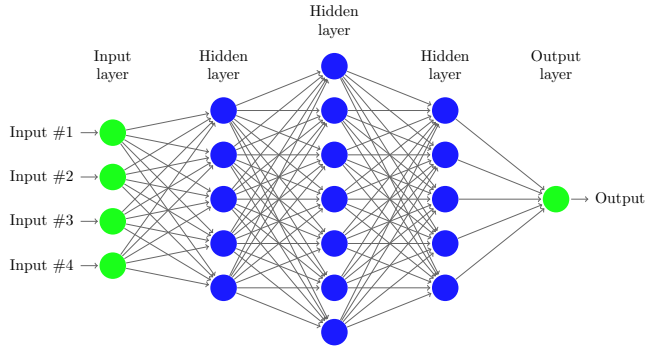
This is magnificient! In practise, one would need very many neurons in the hidden layer, making the NN shallow but very wide. A more efficient approach is to make the NN ***deep and narrow***.

Deep NNs

We can deepen the network by forwarding the hidden layer output to another hidden layer in a ***sequential model***. 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, Input
model = Sequential()
# define the topology of the model
model.add(Input(shape=(4,)))
model.add(Dense(5, 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. The `kernel_initializer` defines the random number distribution of initial weights, where we chose the uniform distribution.

So far we have only built a neural network model with ***hyperparameters***, just a few, which define the model structure and other parameters fixed 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 for weights and biases, we may as well guess them to get started:

Weights and biases are initialized to random values from a suitable distribution)

9.2 Training neural networks

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 do 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 piece of input data (a vector) 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***

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \,, \tag{10}$$

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 you that 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, send your input through the network - a feed forward - and evaluate the output error signal. 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 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 - 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 exactly how much, it only tells how "bad" the current setting is). You do small knob adjustments and the radio is better tuned and your song sounds 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.

9.2.1 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 it clear that there are two weights and two 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. The loss function $\mathcal{L}$ is the square deviation from the known training output $\boldsymbol{y}_{tr}$.

FORWARD PROPAGATION

$$
\begin{aligned}
& x_k && \boldsymbol{x} && \text{(layer 1 input)} \\
& z_i = \sum_k w^1_{ik} x_k + b^1_i && \boldsymbol{z} = \boldsymbol{w}^1 \boldsymbol{x} + \boldsymbol{b}^1 && \text{(layer 1 weighted input)} \\
& h_i = \sigma(z_i) && \boldsymbol{h} = \sigma(\boldsymbol{z}) && \text{(layer 1 output)} \\
& y_j = \sum_i w^2_{ji} h_i + b^2_j && \boldsymbol{y} = \boldsymbol{w}^2 \boldsymbol{h} + \boldsymbol{b}^2 && \text{(layer 2 output)} \\
& \mathcal{L} = \frac{1}{2} \sum_j (y_j - y_{tr,j})^2 && \mathcal{L} = \frac{1}{2} ||\boldsymbol{y} - \boldsymbol{y}_{tr}||^2 && \text{(a simple loss function) .}
\end{aligned}
\tag{11}
$$

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

$$
\bar{\boldsymbol{q}} := \frac{\partial \mathcal{L}}{\partial \boldsymbol{q}} \ . \tag{12}
$$

I emphasize that ***all adjoints are just numerical quantities we compute.***[17] The error signal backprobagation is initiated with the trivial

$$
\bar{\mathcal{L}} := \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1 \ . \tag{13}
$$

[17] 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.

The key is to apply ***the chain rule*** of differentiation,

$$\frac{\partial f(g(x,y))}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} \ , \tag{14}$$

The arrows show how the error signal backpropagates:

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{gradient used in updating } \boldsymbol{w}^2 \\
\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{gradient used in updating } \boldsymbol{b}^2 \\
\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{gradient used in updating } \boldsymbol{w}^1 \\
\frac{\partial \mathcal{L}}{\partial b^1_i} &:= \bar{b}^1_i = \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 b^1_i}}_{1} = \bar{z}_i && \bar{\boldsymbol{b}}^1 = \bar{\boldsymbol{z}} && \text{gradient used in updating } \boldsymbol{b}^1\ . \qquad (15)
\end{aligned}
$$

Notice that the quantities $\boldsymbol{y}$, $\boldsymbol{h}$, and $\boldsymbol{z}$ were computed and stored in forward propagation. The operator $\circ$ denotes the Hadamard (elementwise) product: $\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{a} \odot \boldsymbol{b} = a_i b_i$.

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).

9.2.2 Making use of the gradients: Gradient descent and fiends

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 ***learning rate*** η, a small, positive, adjustable hyperparameter:

WEIGHT AND BIAS UPDATE in Gradient descent

$$\begin{aligned}
\boldsymbol{w}^1 &\to \boldsymbol{w}^1 - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^1} = \boldsymbol{w}^1 - \eta \bar{\boldsymbol{w}}^1 \\
\boldsymbol{b}^1 &\to \boldsymbol{b}^1 - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^1} = \boldsymbol{b}^1 - \eta \bar{\boldsymbol{b}}^1 \\
\boldsymbol{w}^2 &\to \boldsymbol{w}^2 - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}^2} = \boldsymbol{w}^2 - \eta \bar{\boldsymbol{w}}^2 \\
\boldsymbol{b}^2 &\to \boldsymbol{b}^2 - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^2} = \boldsymbol{b}^2 - \eta \bar{\boldsymbol{b}}^2 \ .
\end{aligned} \tag{16}$$

Gradient descent is simple but not very robust. Adjusting weight and biases is just like any other optimization problem with known gradients, and there are many algorithms to choose from. The improved algorithms remember how succesfull the previous gradient updates were and keep revising their optimization strategy. Some popular algorithms are Adam, Adagrad, Momentum, AMSGrad, RMSProp, AdaMax and Nesterov, most of these are built-in in Keras/Tensorflow/PyTorch. See Sebastian Ruder: An overview of gradient descent optimization algorithms.

9.3 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) [18]

AD is a numerical method to compute gradients of arbitrary functions in machine precision accuracy.

The error backpropagation through a multilayer network relies on repeated application of the chain rule. In fact, *backpropagation is a special case of AD in reverse mode*. Reverse mode, because a forward pass is followed by a backward pass. Once you have defined a neural network, Tensorflow or some other clever software can backpropagate the error signal automatically. As you might anticipate, there's also a forward mode AD, to be discussed soon. [19]

Automatic differentiation is not numerical differentiation. The derivative of function $f(x)$ is

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \ , \tag{17}$$

but the ratio is numerically unstable in the limit $\Delta x \to 0$. Close to machine precision one has $x + \Delta x = x$ and $f(x + \Delta x) = f(x)$.

Automatic differentiation is not symbolic differentiation. All that's moving about in AD is numbers, not symbols. You start from the right with the gradient 1, and let it backpropagate. All adjoints are just numerical arrays.

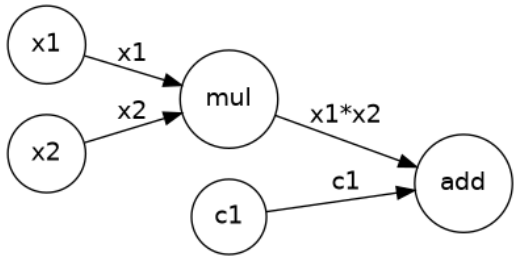
I recommend Jonathan Kernes' blog about automatic differentiation. Here's a short appetizer. Every function can be expressed as a combination of a few elementary functions, such as sin(), cos(), exp(), and all operators are also functions: $+$ is $add()$ and $*$ is $mul()$. In Python, $+$ means the magic method `__add__()` and $*$ means `__mul__()`:

[18] Robert Edwin Wengert, *A simple automatic derivative evaluation program*, Communications of the ACM 7(8):463–4, Aug 1964., AD @Wikipedia.

[19] One of persons to first 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.

```
>>> 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.

Forward mode AD and dual numbers Often good for few inputs and many outputs.

Forward mode AD uses ***dual numbers***, which are (value, derivative)-pairs. Define (mark with underscore dual numbers

and their operations)

$$\underline{a} = (a, a') \quad \text{a dual number, value and derivative} \tag{18}$$
$$\underline{b} = (b, b') \quad \text{a dual number, value and derivative} \tag{19}$$
$$\underline{x} = (x, 1) \quad \text{a dual variable } x \text{ and it's derivative}$$
$$\underline{c} = (c, 0) \quad \text{a dual constant } c \text{ and it's derivative}$$
$$\underline{a} \mathbin{\underline{+}} \underline{b} = (a, a') \mathbin{\underline{+}} (b, b') = (a + b, a' + b') \qquad \text{here } \underline{+} \text{ is dual plus, not plus-minus } \pm$$
$$\underline{a} \mathbin{\underline{*}} \underline{b} = (ab, a'b + b'a) \qquad \text{rule to derive } ab$$
$$\underline{a} \mathbin{\underline{/}} \underline{b} = (a/b, a'/b - ab'/b^2) \qquad \text{rule to derive } a/b$$
$$\underline{\sin}(\underline{a}) = (\sin(a), a' \cos(a)) \qquad \text{rule to derive } \sin(a)$$
$$\underline{\cos}(\underline{a}) = (\cos(a), -a' \sin(a)) \qquad \text{rule to derive } \cos(a) \ ,$$

and so on, write a derivation rule for every built-in operator (in NumPy, for example). Following these rules and keeping the order of operations, let's evaluate the derivative of $f(x) = x^2 \cos(2x)$ at $x = 3.0$. The dual argument is $\underline{x} := (3.0, 1)$, and the operation is, step by step,

$$\underline{f}(3.0, 1) = \underline{x} \mathbin{\underline{*}} \underline{x} \mathbin{\underline{*}} \underline{\cos}(\underline{2} \mathbin{\underline{*}} \underline{x}) \tag{20}$$
$$= (3.0, 1) \mathbin{\underline{*}} (3.0, 1) \mathbin{\underline{*}} \underline{\cos}(2.0 \underline{*} (3.0, 1)) \tag{21}$$
$$= (3.0 * 3.0, 1 * 3.0 + 3.0 * 1) \mathbin{\underline{*}} \underline{\cos}((2.0 * 3.0, 2.0 * 1)) = (9.0, 6.0) \mathbin{\underline{*}} \underline{\cos}((6.0, 2.0))$$
$$= (9.0, 6.0) \mathbin{\underline{*}} (\cos(6.0), -2.0 * \sin(6.0)) = (9.0 * \cos(6.0), 6.0 * \cos(6.0) - 2.0 * \sin(6.0) * 9.0)$$
$$= (8.641532579853294, 10.790500687482862) := (f(3.0), f'(3.0)) \ . \tag{22}$$

This looks a bit clumsy, but dual numbers and their operations are easily done by the computer. You can check the results,

$$f(3.0) = 8.641532579853294 \tag{23}$$
$$f'(x) = 2x \cos(2x) - 2x^2 \sin(2x) \Rightarrow f'(3.0) = 10.790500687482862 \ . \tag{24}$$

Works like a charm, with machine precision accuracy!

Reverse mode AD Often good for many inputs and few outputs.

As we saw, the reverse mode AD is frequently used in machine learning. Take the same example, $f(x) = x^2 \cos(2x)$ at $x = 3.0$. This example doesn't really show how the reverse mode should be coded, just how numerical data propagates.

- Forward pass, remember intermediate steps:

$$\begin{aligned} f(x) &= x^2 \cos(2x) \\ z_1 &:= x^2 = 3.0^2 = 9.0 \\ z_2 &:= 2x = 6.0 \\ z_3 &:= \cos(z_2) = \cos(6.0) = 0.960170286650366 \\ f(3.0) &= z_1 * z_3 = 9.0 * 0.960170286650366 = 8.641532579853294 \ . \end{aligned}$$

  Nothing spectatular, just *remember* z_1, z_2 and z_3.

- Reverse pass:

$$\begin{aligned} \frac{\partial f}{\partial z_1} &= z_3 = 0.960170286650366 && \text{computed earlier} \\ \frac{\partial f}{\partial z_3} &= z_1 = 9.0 && \text{computed earlier} \\ \frac{\partial z_3}{\partial z_2} &= -\sin(z_2) = -\sin(6.0) = 0.27941549819892586 \\ \frac{\partial f}{\partial z_2} &= \frac{\partial f}{\partial z_3}\frac{\partial z_3}{\partial z_2} = 9.0 * 0.27941549819892586 = 2.5147394837903327 && \text{chain rule} \end{aligned}$$

  Propagate back to input x, which appears in both $z_1 = x^2$ and $z_2 = 2x$. Path 1, through z_1, gives

$$\begin{aligned} \frac{\partial z_1}{\partial x} &= 2x = 6.0 \\ \frac{\partial f}{\partial x} &= \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x} = 0.960170286650366 * 6.0 = 5.761021719902196 && \text{chain rule} \end{aligned} \tag{25}$$

and path 2, through z_2, gives

$$\frac{\partial z_2}{\partial x} = 2$$
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z_2}\frac{\partial z_2}{\partial x} == 2.5147394837903327 * 2 = 5.0294789675806655 \qquad \text{chain rule ,}$$

and combine the two paths,

$$\frac{\partial f}{\partial x} = 5.761021719902196 + 5.0294789675806655 = 10.790500687482862 \; . \tag{26}$$

Computation of gradient and laplacian using AD AD can be very handy if you need, say, $\boldsymbol{\nabla}\phi$, $\nabla^2\phi$, or the jacobian or the hessian. On the JAX web page, a hessian is computed as

```
>>>import jax
>>>g = lambda x: x[0]**3 - 2*x[0]*x[1] - x[1]**6
>>>print(jax.hessian(g)(jax.numpy.array([1., 2.])))
[[   6.   -2.]
 [  -2. -480.]]
```

which is computes the hessian elements

$$h_{ij} = \frac{\partial^2 g}{\partial x_i \partial x_j} \tag{27}$$

for $g(x_0, x_1) = x_0^3 - 2x_0x_1 - x_1^6$ at $(x_0, x_1) = (1.0, 2.0)$. The code AD_grad_grad2.py shows how the Python module `jax` can be used to evaluate accurate gradients and laplacians, and compares the results with the numerical values of the analytical expressions and with the (inaccurate) finite difference numerical derivatives. The example uses JAX in higher precision float64 with `jax.config.update("jax_enable_x64", True)`. The default accuracy float32 is adequate for many purposes, and `jax` code can also be `jit`ted.

9.4 Terminology: 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.

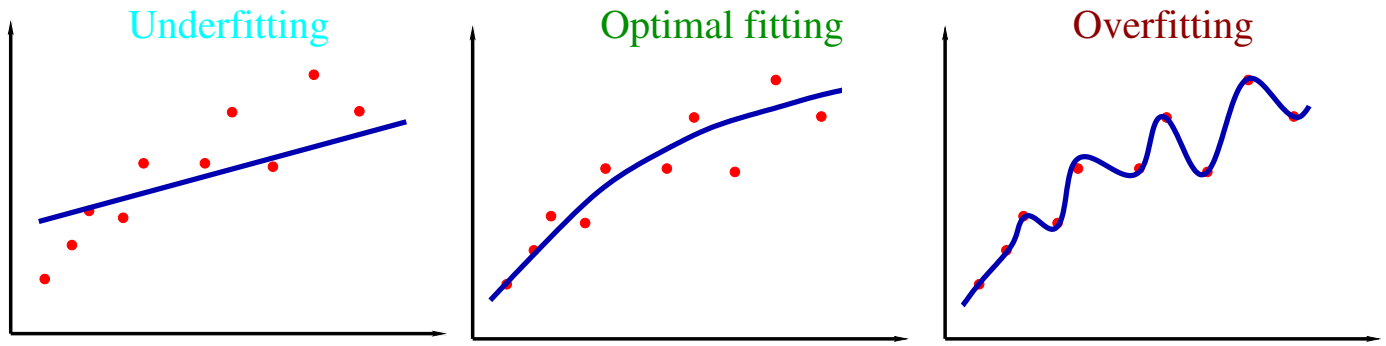
- ***A batch***: The training data set is divided into batches,

$$\begin{aligned} &\text{batch } 1: \quad \{x_{tr}\}_{1,\ldots,Nb} \to \{y_{tr}\}_{1,\ldots,Nb} \\ &\text{batch } 2: \quad \{x_{tr}\}_{Nb+1,\ldots,2Nb} \to \{y_{tr}\}_{Nb+1,\ldots,2Nb} \\ &\ldots \end{aligned} \tag{28}$$

  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.

9.5 Learning problems

In a deep NN, one may encounter problems that prevent NN from learning - "deep-learning problems". Changes in weights and biases are done based on gradients, computed from the loss function and backpropagated through the NN. A few things can go wrong:

- Bad initialization: signal dies or blows up as it propagates
- Vanishing gradient: Backpropagated to hidden layers, a layer may get very small or exactly zero gradients. The layer weights and biases are not changing ⇒ part of the network is not learning.[20]The first cure found to the problem is the ***long short-term memory (LSTM)***, a light-weight, computationally inexpensive network architecture. In many applications LSTM has been surpassed by ***transformers***, which are used in Large Language Models ***LLM***s, such as GPT-5. A key concept is ***self-attention***. It all started with the article (inspired by a Beatles song) "Attention is all you need", (link), and the model BERT (Bidirectional Encoder Representations from Transformers, 2018). **Also transformers are taught using backpropagation, so we are still using it.** If you are interested in how LLMs work, I recommend Youtube video about LLMs.
- Exploding gradient: Backpropagated to hidden layers, a layer may get huge weight and bias gradients, and the loss jumps.
- ***Degradation problem***: adding more layers increases training error, although one expects the NN to become more descriptive. Often the loss surface (see discussion later) is so anisotropic that gradient descend zigzags around.
- The so-called "dying-ReLU problem": If a layer has $xw + b < 0$, then ReLU$(xw + b)$=0. The network unit dies, and in many cases stays dead. Try "leaky-ReLU", where negative input does not give zero but a small "leak" output.
- Gradient noise: (stochastic) gradients may have so much noise that their direction is random. In such cases the problem is not overfitting, but stochastic noise.

[20]The problem was analyzed by Hochreiter in his diploma thesis in 1991. 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 that Uni, although in Physics.

What to do if NN is not learning

- A general advice is ***avoid sigmoid activation***, because it is prone to vanishing gradient. Use ReLU, GELU, SiLU ...
- ***Important!*** Preserve variance across layers by initializing weights using the ***Xavier/Glorot*** scheme. If a layer has n_{in} inputs and n_{out} outputs, choose weight with variance [21]

$$\sigma_w^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}} \ . \tag{29}$$

For a normal distribution initialization, use directly ($\sim$ means *sampled from*)

$$w_{ij} \sim \mathcal{N}(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}) \ . \tag{30}$$

For a uniform distribution initialization, use

$$w_{ij} \sim \mathcal{U}[-a, a] \tag{31}$$

so $\sigma_w^2 = a^2/3$, and solve

$$\sigma_w^2 = \frac{a^2}{3} = \frac{2}{n_{\text{in}} + n_{\text{out}}} \Leftrightarrow a = \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}} \ . \tag{32}$$

- One effective remedy to the degradation problem and also to the vanishing gradient problem is to use ***skip connections***, also called ***residual connections***. The principle is simple: Instead of learning

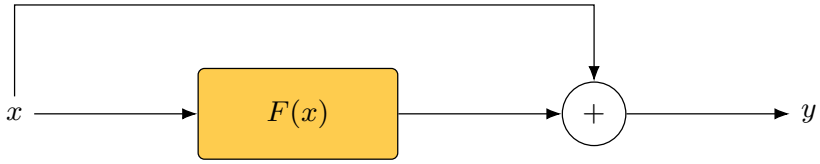
$$y = F(x) \tag{33}$$

that has the problem

$$\frac{\partial y}{\partial x} = \frac{\partial F}{\partial x} \leftarrow \text{very small gradient in problematic situations} \ . \tag{34}$$

An even worse problem emerges if $y = x$, then the deep NN would have to tune weight so that $F(x) = x$; a very fragile job. A better strategy is to ***learn a correction to identity***

$$y = x + F(x) \ . \tag{35}$$



Block of a neural network

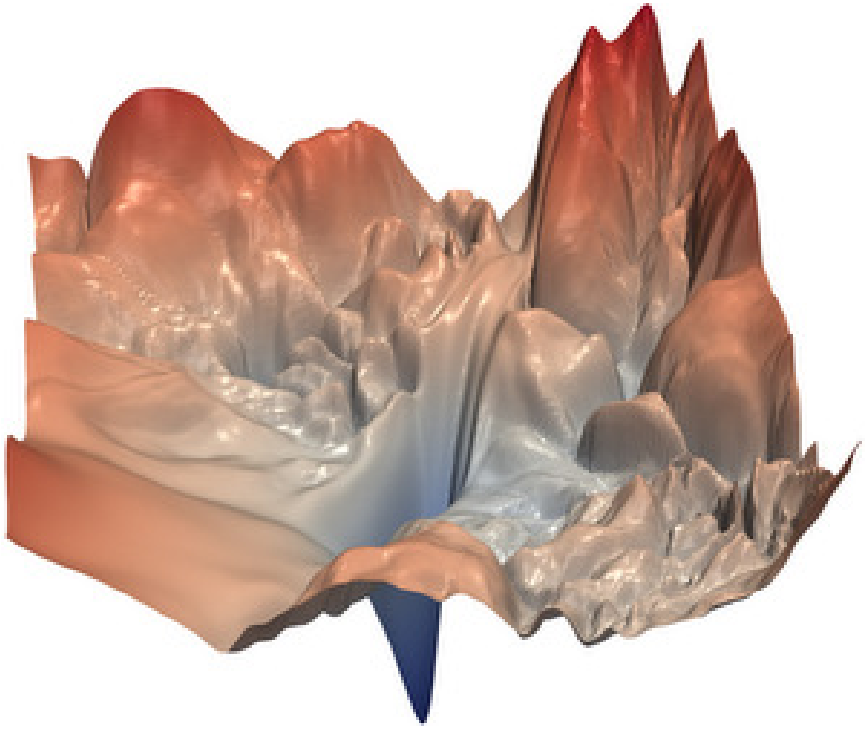
The gradient

$$\frac{\partial y}{\partial x} = I + \frac{\partial F}{\partial x} \ , \tag{36}$$

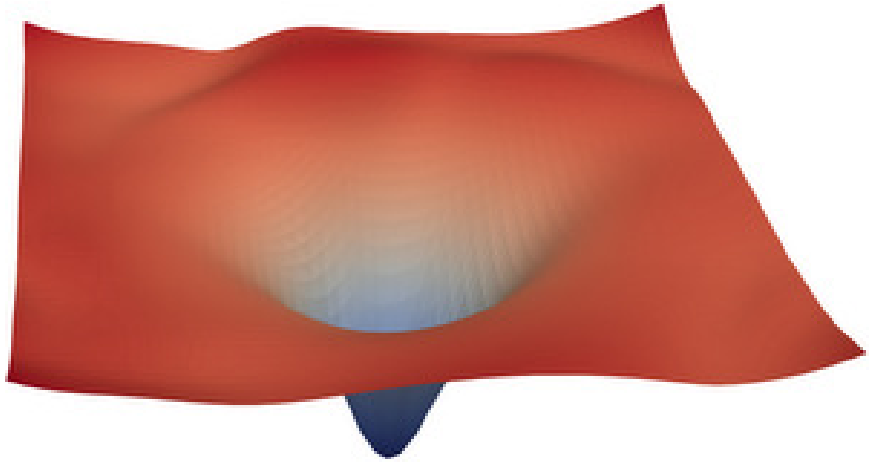
is not going to zero even if $\frac{\partial F}{\partial x} \approx 0$; in such a case the layer is effective "skipped", and the input is fed directly to the output. [22] The degradation problem can be attacked by adding residuals between many blocks in the NN, not just between input and output. In such a ResNet @wikipedia blocks can use the outputs of any of the previous blocks, skipping blocks in between them. This make the loss landcape a lot smoother, and effectively solves the degradation problem.

[22] If the input and output shapes differ, so you need to do $y = P(x) + F(x)$, where the projection $P(x)$ can be a linear layer, $P(x) = wx + b$, with its own (learning) weights and biases.

Here is a visualization of the loss landscape from github.com/tomgoldstein, published in arXiv:1712.09913:



Without residual connections



With residual connections

- Keep the signal within the effective range of activation function by using ***BatchNorm***, ***LayerNorm*** or ***RMSNorm***, which shift and normalize the signal $z \to (z - \mu)/\sigma$. Now $\sigma(z)$ and $\sigma'(z)$ are in a stable range.
- The ***learning rate schedule*** should use a high rate at early stages to explore the terrain, and a reduced rate later to allow for convergence. This is familiar from any optimization problem.

[22] The reason for using both n_{in} and n_{out} is that forward pass signal depends on the former and the backpropagating gradient signal depends on the latter.

9.6 Teaching `scikit-learn` datasets

The package `scikit-learn` contains datasets suitable for ML testing. After `pip install scikit-learn`, import it as `sklearn`.

```
>>> from sklearn import datasets
>>> help(datasets)
```

The code cancer_keras.py uses `tensorflow.keras`. If available, Tensorflow tries to use CUDA and Nvidia GPU, which you see during the calculation using

```
$ nvidia-smi
```

If your CUDA installation segfaults (possibly wrong library version), turn it off and live with a slower code:

```
$ export CUDA_VISIBLE_DEVICES=''
$ python ...
```

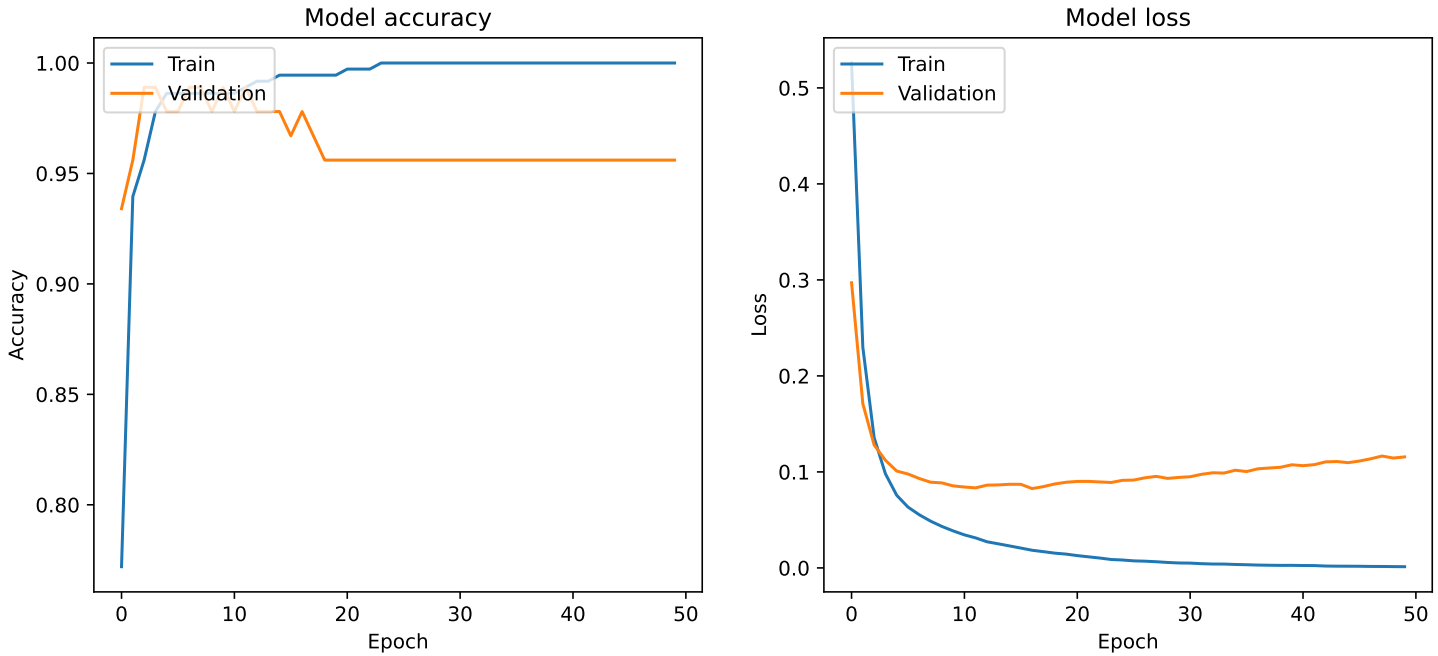
9.6.1 Classification

The Wisconsin breast cancer dataset can be classified using ML. The samples contains 212 malignant and 357 benign cases, and the task is to teach ML to classify them based on 30 "features" (see `DESCR` below):

> *Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.*

```
>>> from sklearn.datasets import load_breast_cancer
>>> cancer = load_breast_cancer()
>>> cancer.data.shape # type numpy.ndarray
(569, 30)
>>> dir(cancer)
['DESCR', 'data', 'data_module', 'feature_names', 'filename', 'frame', 'target', 'target_names']
>>> print(cancer.DESCR) # long description of the dataset, use print() to format output
```

After 50 epochs, the code cancer_keras.py gives the following plot:

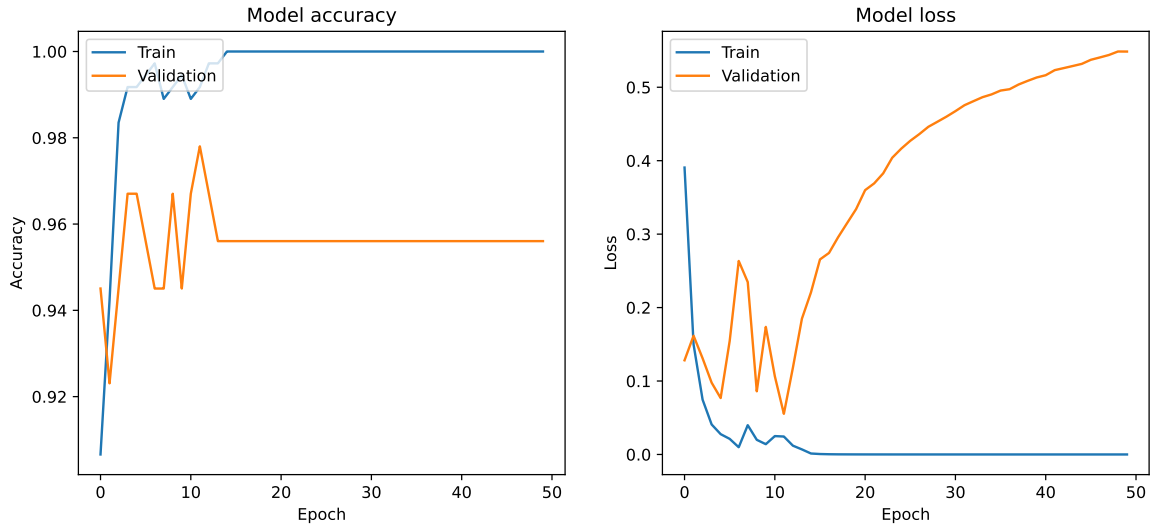


The training session tries to minimize a certain loss function, and the model seems to be quite good. The details of how accuracy and loss evolve with increasing number of epoch varies from run to run, but the final accuracy is about 96 percent. With more epochs, the predictive power of the model is no longer improving.

Let's make things worse. Add more and larger hidden layers (this uses another way to define layers in Keras),

```
Dense = layers.Dense
model = keras.Sequential()
model.add(Dense(64, activation="relu", input_shape=(X_train.shape[1],)))
model.add(Dense(64, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(1, activation='sigmoid')) # output is a binary number 0 or 1
```

According to `model.report()`, there are now 55085 parameters, so there is a high risk of overfitting.



As expected, the training set is almost perfectly reproduced, and the training loss is closing on zero. The model accuracy levels off, meaning the number of wrong predictions is not increasing. Meanwhile, the validation loss is increasing, so the probabilities the model is assigning to the two cathegories is getting worse. One says that the model is becoming ***overconfident***.

9.6.2 Continuous target

One of the continuous datasets is `load_diabetes`,

```
>>> from sklearn.datasets import load_diabetes
>>> diabetes = load_diabetes() # the type is ``Bunch''
>>> diabetes.data.shape # type numpy.ndarray
(442, 10)
>>> dir(diabetes)
['DESCR', 'data', 'data_filename', 'data_module', 'feature_names', 'frame', 'target', 'target_filename']
>>> diabetes.feature_names
['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
```

The set is 442 samples of 10 diabetes indicators ("dimensionality"). The target is a disease progression score, which is a ***continuous variable***, so a suitable measure of the prediction accuracy is the absolute deviation from the known correct value; For the whole dataset this is given by MAE (Mean Absolute Error). A Keras/Tensorflow solution is given in `diabetes_keras.py`

What about ChatGPT? Let's ask (one trillion is 10^{12}).

- GPT-2 (2019)
  - Parameters: 1.5 billion
  - Context window: 1,024 tokens
  - Training data: WebText (40 GB of text)
  - Impact: First widely known large LLM; OpenAI initially didn't release the full model due to "misuse concerns."
  - Limitation: Struggled with coherence in long passages.
- GPT-3 (2020)
  - Parameters: 175 billion
  - Context window: 2,048 tokens
  - Training data: Hundreds of billions of words (Common Crawl + books + Wikipedia, etc.)
  - Impact: Huge jump — capable of few-shot and zero-shot learning, first time LLMs became mainstream.
  - Limitation: Still hallucinated, lacked reasoning.
- GPT-3.5 (2022)
  - Parameters: Not officially disclosed (likely tens to hundreds of billions, maybe optimized architectures).
  - Context window: 4,096 tokens
  - Impact: Used in ChatGPT (Nov 2022) — the one that made LLMs go viral. Much more reliable than GPT-3 thanks to RLHF (reinforcement learning from human feedback).
  - GPT-4 (2023)
  - Parameters: Undisclosed (but estimated 1 trillion parameters with mixture-of-experts, meaning only a fraction active per token).
  - Context windows: 8k, 32k, and later even 128k tokens.
  - Impact: Major leap in reasoning, factuality, and multimodality (text + images).
  - Limitation: Still "hallucinates" but much less.
- GPT-5 (2025 – you're talking to it)
  - Parameters: Not disclosed.
  - Likely features:
    - Smarter efficiency (not just "bigger" but better training and mixture-of-experts).
    - Stronger reasoning (esp. step-by-step).
    - Better long-context understanding.
    - More multimodal capabilities.

Trend: GPT-2 → GPT-3 is mostly about scale (more parameters, bigger dataset). GPT-3 → GPT-4 → GPT-5 is about efficiency and architecture, not just raw size.

9.7 DeepONet: Operators as neural networks

Let's make more use of the fact that a simple feed-forward neural network can, in principle, approximate any function. Sometimes a common value-to-value function is insufficient, and one needs instead a ***mapping***. A mapping is an ***operator*** [23] $\mathcal{G}$ that does

$$\mathcal{G} : u(\cdot) \mapsto v(\cdot). \tag{37}$$

There are many notations for such an operator:

$$v(y) = G[u](y) \quad \text{notation in functional analysis} \tag{38}$$
$$v(y) = G(u)(y) \quad \text{notation in some ML papers} \tag{39}$$
$$v(y) = G(u; y) \quad \text{notation in some ML papers .} \tag{40}$$

For example, the derivative operator $\mathcal{D}$ maps a function to a function,

$$v(x) = \mathcal{D}[u](x) = \frac{d}{dx}u(x) = u'(x) \ , \tag{41}$$

and so does the cumulative integral

$$v(y) = \mathcal{G}[u](y) = \int_0^y u(x)dx \ . \tag{42}$$

Less well known universality theorem:

A single hidden layer can approximate any nonlinear continuous operator.

The idea is that we don't know what the operator is, but only that ***there is a unique operator that descibes*** $\mathcal{G} : u(\cdot) \mapsto v(\cdot)$ ***in our system***. Obviously we can't write down the operator (or we wouldn't need numerics), but, thanks to the

[23] An operator is not a functional, the latter is a mapping from a function to a number, for example $F[u] = \int_0^1 u(x)dx$.

universality theorem, we know there is a neural network that does the same job. But we have to know what the operator does for a wide variety of inputs u; that's our teaching material.

A ***DeepONet (Deep Operator Net)*** does that in practice. The question is, how do I build numerically another function

$$v(x) = \mathcal{G}[u](x) \tag{43}$$

from a known function $u(x)$? Starting from the end, the function $v(x)$ has an expansion in basis functions ϕ_k,

$$v(x) = \sum_k c_k \phi_k(x) \ . \tag{44}$$

Now we want to construct $v(x)$ from u, so the expansion coefficients c_k must depend on u. Since the coefficients c_k are just numbers, they are functionals of u, $c_k = c_k[u]$, and we have defined the operator

$$\mathcal{G}[u](x) = v(x) = \sum_k c_k[u] \phi_k(x) \ . \tag{45}$$

The basis functions ϕ_k are ***features of*** v. Let's assume we keep m important features,

$$\mathcal{G}[u](x) = v(x) \approx \sum_{k=1}^{m} c_k[u] \phi_k(x) \ . \tag{46}$$

You may now ask if we have achieved anything? We wrote an unknown operator $\mathcal{G}$ in terms of mysterious functionals $c_k[u]$, so how is that going to help us? Let me remind you that we are trying to write a very generic program that takes in any input function u we care to try it on, and lets us evaluate v at points x. Just like the cumulative integral operator we wrote earlier; it should work for any u. That's why we can't hard-code $u(x)$ in the program as, say, $\sin(x)$ or $exp(x^2 - 1)\cos(x)$. What we can do, is assume the code takes in ***samples*** of the function u evaluated at some points.

Since we can only work with a numerical representation of u, then for $c_k[u]$ we need something that takes the samples of u at a finite set of points, [24]

$$u \text{ is approximately represented by samples } \{u(x_1), u(x_2), u(x_3), ..., u(x_n)\} \ . \tag{48}$$

[24] A wave function can be written in the form

$$\psi(x) = \int dy \delta(x - y)\psi(y) \approx (\Delta x) \sum_{k=1}^{n} \delta(x - x_k)\psi(x_k) \ , \tag{47}$$

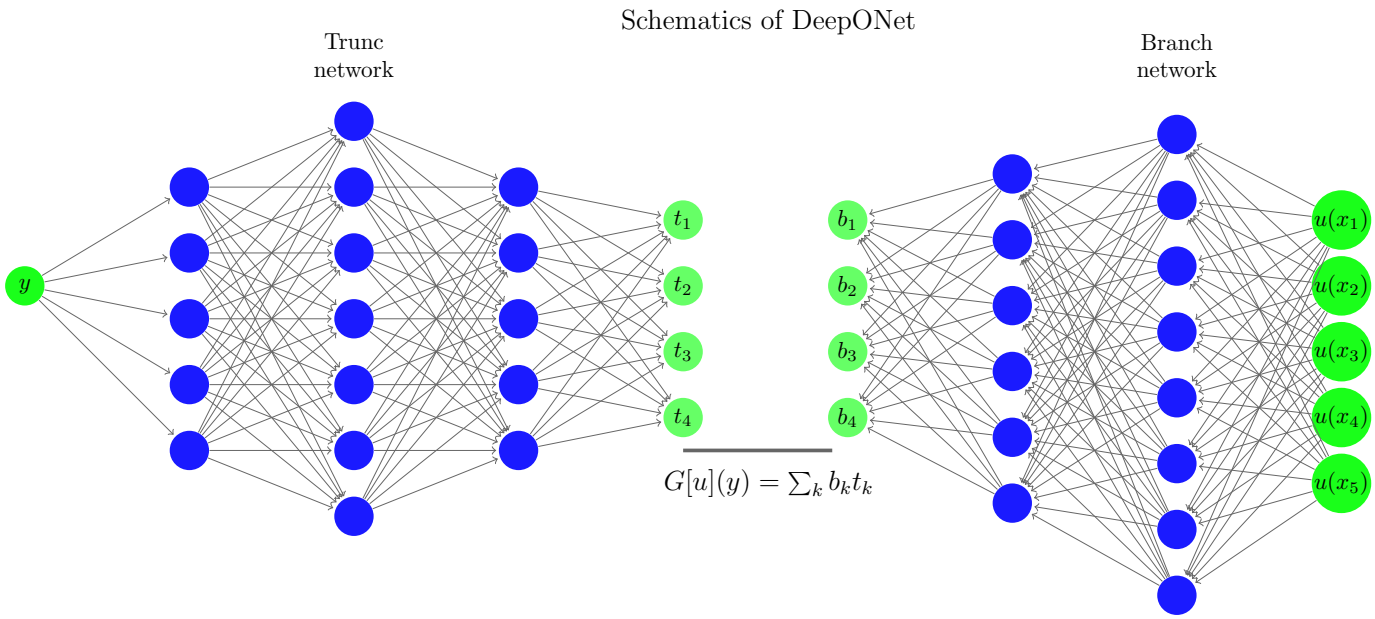
From this finite set of sampled points we construct $c_k[u]$,

$$c_k[u] \approx b_k(u(x_1), u(x_2), u(x_3), ..., u(x_n)) \ . \tag{49}$$

The functions b_k are **unknown**, but all we need is something that takes n inputs and gives $k = 1..m$ outputs – a neural network with n inputs and m outputs!. There you are! The rest is easy. The **unknown** basis functions ϕ_k take input y and gives m outputs: another neural network.

$$v(y) := \mathcal{G}[u](y) \approx \sum_{k=1}^{m} \underbrace{c_k[u]}_{\substack{\text{branch} \\ \text{network}}} \quad \underbrace{\phi_k(y)}_{\substack{\text{trunck} \\ \text{network}}} := \sum_{k=1}^{m} b_k t_k \ . \tag{51}$$

so $\psi(x)$ can be approximately represented by its values in n points $\psi(x_k)$. This is valuable in Diffusion Monte Carlo (variant of Quantum Monte Carlo), where the analytical form of the wave function is unknown. Nevertheless, we can draw samples from it.



Once you have for teaching (preferably many) known mappings $u \mapsto v$, you can teach the branch and trunc networks using AD and backpropagation in the hope that the networks learn the operator denoted as $\mapsto$. The codes `deeponet_simple.py` and `deeponet_simple_modified.py` show how to write a simple DeepONet using PyTorch.

For more information, see the articles DeepONet @ArXiv or DeepONet @Nature Machine iIntelligence, and the codes DeepONet (github) and DeepONet FNO (github).

9.8 Graph Neural Network (GNN)

Interactive intro: A gentle introduction to graph neural networks show how images, text, molecules and social networks can be encoded as graphs. A very enjoyable presentation!

Consider a many-body system of interacting particles. Each particle interacts with its neightbours, which interact with their neighbors. The interaction is typically a two-body interaction, such as the Coulomb interaction. Adding up all interactions is a fairly simple task, and you get the potential energy of a configuration. The kinetic energy, either classical or quantum, adds a new layer of complication to the problem, as you need to find out how the particle positions are *correlated*. Particles in an external potential (such as a harmonic trap) have a one-body correlation, and if the particles interact, you immediately observe a two-body correlation (particle pairs binding or repelling). That's the whole story only at a very low density, where having more than two particles close to each other is small. As the density increases, the more particles are felt present at once, and you see three-body or higher correlations. Such a hierarchy escalates out of hand very soon.

A related system is a social network. You contact a friend and ask his opinion on something. That friend may give his opinion, but maybe he contacts a friend and asks her opinion on the matter before telling you. What you hear is an opinion influenced by two opinions. In real life, opinions are affected by spouses, kids, school, work, and the internet.

One way to approximate correlations is to concentrate on passing *messages* between *nodes (vertices)*, and on how messages are stored in nodes. This keeps the communications on a simple one-to-one level. In the first example the nodes are particles and the edges can be their distances. In the second example, the nodes are persons, and the edges can be connections to influencial friends.

Let's assume first that each node i remembers it's own embedding (opinion or physics) h_i^0 on level 0. The embedding can be processed from some properties, Z_i, using an **unknown function** ψ_0,

$$h_i^0 = \psi_0(Z_i) \ . \tag{52}$$

Next, we add inter-node connections to this set of "non-interacting" embeddings. A node can receive *messages* m from other nodes. The node's embedding will be updated based on the current embedding and the incoming messages, each such update is done level by level. The level $\ell = 0$ embedding is h_i^0, the updated "interacting" embedding is h_i^1 and so on, up to some maximum level $\ell = L$. The importance of a message is quantified by the node-to-node *edge (link)* ϵ_{ij}.

The message is formed from the current h_i^ℓ, h_j^ℓ and ϵ_{ij}, processed by an **unknown functions** ϕ_ℓ,

$$m_{ij}^\ell = \phi_\ell(h_i^\ell, h_j^\ell, \epsilon_{ij}) \ . \tag{53}$$

The **unknown functions** ψ_ℓ interpret the current embedding and all the incoming messages, and encode how the these affect the embedding in node i,

$$h_i^{\ell+1} = \psi_\ell(h_i^\ell, h_j^\ell, \sum_{j(\neq i)} m_{ij}^\ell) \ . \tag{54}$$

The nodes and edges build up a ***graph***. In the particle example, a particle learns something about it's neighbours, which know something about their neighbours, and so on with increasing level ℓ.

By now you might have guessed what our strategy is:

Anywhere we have an unknown function we put a neural network

The promise of GNNs is that they allow scientists to retain structural symmetry, 3D geometry, and physical laws without flattening data into simple lists of numbers. Many physical problems have a natural graph structure. Think about, for example, molecules where atoms are nodes and bonds are edges.

Some recent applications of GNN in natural sciences:

- Weather forecasting, for example GraphCast by Google DeepMind
  Why weather forecasting? (DL=Deep Learning) [25]

  > Firstly, traditional DL-based methods, such as Convolutional Neural Networks (CNNs), are typically designed for structured data (images). However, meteorological data often forms irregular, spatially correlated graph structures (such as relationships between weather stations and interactions between atmospheric layers). As a result, conventional methods generally cannot handle such graph-structured data expediently. Secondly, the atmospheric environment includes complex regional interactions such as wind patterns, humidity, and temperature. CNNs excel at capturing local spatial information. However, they have limitations in modeling global and cross-regional dependencies. In contrast, GNNs are more effective in naturally capturing the complex spatial relationships and dependencies between data.

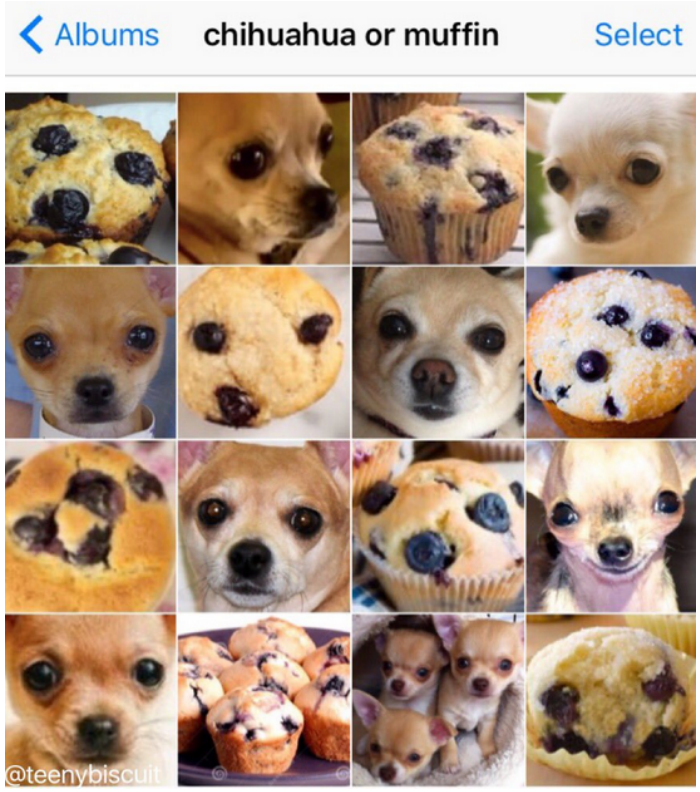
[25] Quote from Xiaoni *et al.*, *Utility of Graph Neural Networks in Short-to Medium-Range Weather Forecasting*, Computers, Materials and Continua, Volume 84, Issue 2, 2025, Pages 2121-2149.

- Accelerating Molecular Dynamics (SchNet, NequIP). Compared to traditional physics-based simulations of atom movements, geometric GNNs can simulate atomic forces exponentially faster while maintaining quantum accuracy.
- Reaction prediction. GNNs can predict how chemical reactions will occur.
- Particle physics. Reconstruct of particle tracks and classification of collision events from sensor network data.
- Fluid dynamics. GNNs can model interacting particles in liquids and gases. E(3) equivariant GNNs respect rotational, translational and reflection symmetries from the outset.

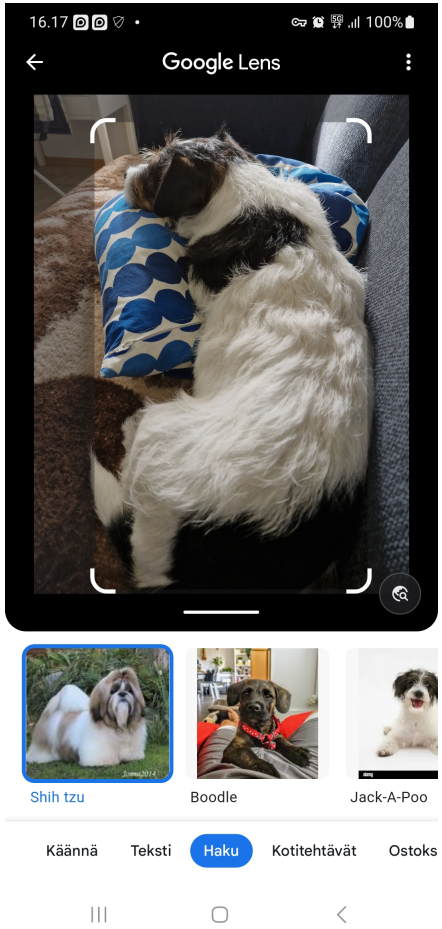
See Nature 2022; Graph neural networks for materials science and chemistry. I recommend warmly GNN for Scientific Applications by Maria Bankestad (Uppsala 2026).

9.9 Odds and ins of ML: pattern recognition

AI is not infallible, see Karen Zack @teenybiscuit







Jackrusselterrier 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 recommend 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 Pandas, in a Jupyter notebook with inline Matplotlib plots.

I warmly recommend Florian Marquardt: Machine Learning for Physicists.

One trend in ML is obvious in the abstract from Smith *et al* (2019) [26]
(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 EMLM by Tommi Kärkkäinen, JYU. Compared to ELM, EMLM is ridge regression on distances to selected landmarks,

$$\text{ELM} : \sigma(w_i^T x + b_i) \rightarrow \text{EMLM} : d(r_i, x) = ||x - r_i|| = \sqrt{\sum_{l=1}^{n} (x_l - r_{il})^2} \ . \tag{55}$$

What makes EMLM attractive is that distances are always scalars in any dimensions.

[26] Smith *et al*, *Approaching coupled cluster accuracy with a general-purpose neural network potential through transfer learning* pdf (Nature Communications) .

One detail: EMLM has no (explicit) activation function, because

$$f(x) = \sum_i \beta_i \sqrt{\sum_{l=1}^{n} (x_l - r_{il})^2} \ , \tag{56}$$

so distance calculation provides a nonlinear function of x. Of course, without a nonlinear activation function ELM would collapse to a trivial linear function (more properly, an affine function; a linear function would have no bias),

$$f(x) = \sum_i \beta_i (w_i^T x + b_i) = a^T x + c \ . \tag{57}$$

[27] Using data gathered in lengthy DFT (density functional) computations to train EMLM, they demonstrated that one can reproduce the structural parameters and energetics at several temperatures with a reasonable accuracy. This approach is sometimes called ***ML model surrogate to DFT***. The speedup was several orders of magnitude. One thing learned in this work was that it's not enough to train on data of fairly good, low energy clusters because then EMLM easily goes astray and probes completely impossible structures.

In Summer 2021 two groups reported highly accurate predictions of ***protein structures***. The deep-learning models, AlphaFold2[28] and RoseTTAFold[29] 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.

For recent ML trends, see Machine Learning Trends 2026. More interesting, perhaps, is the list of old, outdated trends. It shows how ML trends melt like last Winter's snow.

[27] 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).

[28] J. Jumper *et al.*, Nature, Vol 596, 26 (2021).

[29] M. Baek *et al.*, Science 373, 871-876 (2021).

9.10 Probabilistic Programming and Bayesian Inference

Bayes' theorem [a]

Let A and B be events and let $P(A|B)$ be the conditional probability that A happens if B has happened.

$$\underbrace{P(A|B)}_{\text{what we want to know}} = \underbrace{\frac{P(B|A)\,P(A)}{P(B)}}_{\text{what we know}} \tag{59}$$

Likelihood $P(B|A)$ Prior $P(A)$

Posterior $P(A|B)$ Marginal likelihood or Evidence $P(B)$

[a] Bayes' theorem @Wikipedia

Bayes' theorem tells how to get $P(A|B)$ from known $P(B|A)$.

Bayes' theorem tells how to add new data to prior knowledge to get a more informed probability for the event.

Proof of Bayes' theorem. The definition of conditional probability that A happens given B has happened is

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \quad \text{if } P(B) > 0\ , \tag{60}$$

therefore the joint probability that both A and B happen is (multiplication or chain rule of probability) $P(A \cap B) = P(A|B)P(B)$ (alternative notation for $P(A \cap B)$ is $P(A, B)$). Then also

$$P(B|A) = \frac{P(B \cap A)}{P(A)} = \frac{P(A \cap B)}{P(A)}, \quad \text{if } P(A) > 0\ , \tag{61}$$

because a set intersection is commutative; eliminate $p(A \cap B)$ to get Bayes' theorem.

Bayesian inference in physics

In physics, we are often interested in predicting parameters (θ) of a ***model***, based on assumptions (the prior $p(\theta)$ and acquired data (y).[a]

In continuous models, the relevant quantities are not probabilities P but ***probability densities*** p. [b]

$$p(\theta|y) = \frac{p(y|\theta)\,p(\theta)}{p(y)} \tag{62}$$

θ is parameters (or hypothesis or proposition), and y is (new) data.

Likelihood Prior

Posterior Marginal likelihood or Evidence

The name "marginal" comes from the "marginalization", a marginal probability distribution has some variables have been integrated (summed) away. Now

$$p(y) = \int p(y|\theta)p(\theta)d\theta \ , \tag{63}$$

which shows that $p(y)$ is normalized $p(y|\theta)p(\theta)$, so the posterior is a normalized probability density function (PDF).

Where did the prior $p(\theta)$ came from? Edwin Thompson Jaynes (1922-1998) came to the conclusion that you can push things further back and say it comes from some unspecified source of information, past experiments or experience, that gave you the data y_0. We still have the Bayesian structure in the form

$$p(\theta|y, y_0) = \frac{p(y|\theta, y_0)\,p(\theta|y_0)}{p(y|y_0)} \tag{64}$$

You'll recover the earlier Bayes' theorem if you leave y_0 implicit and use the prior without bothering to write down where it came from. Writing y_0 explicitly emphasizes the fact that the prior is based on pre-acquired information y_0. Humorously, y_0 is a stubborn opinion that remains unchallenged. With overwhelming new data the effects of a prior diminish, and you may conclude that Earth is round even though dad once said it's flat. Still, Earth would probably be a bit flatter because the prior does pull the posterior probability distribution.

[a] Internet is full of "Bayesian" vs. "Frequentist" discussions. Frequentists view probability as a frequency of occurrence in very many repetitions, while Bayesians see it as a degree of belief that can be updated with new data.

[b] Toussaint, *Bayesian Inference in Physics*, Rev. Mod. Phys., Vol. 83, (2011).

9.10.1 Credibility Interval (Bayesian) vs. Confidence Interval (Frequentist)

I'm mixing the terms easily, but for Bayesian results the correct terminology is the credibility interval.
"function" is a parameter or hypothesis
"data" is observations

	Credibility Interval (Bayesian)	**Confidence Interval (Frequentist)**
Philosophy	**Bayesian:** The true function is a random variable (modeled as a distribution because of your uncertainty); the data is fixed (the only concrete reality you have)	**Frequentist:** The true function is fixed; the data is a random variable (subject to sampling variations).
Meaning	There is a **95% probability** that the true function value lies within this specific interval.	If you repeat the experiment infinitely, **95% of the computed intervals** will contain the true fixed value.
Interpretation	Measures the **uncertainty of the estimate** given the observed data.	Measures the **reliability of the estimation process** over repeated trials.

Frequentist probability is a long-run frequency, Bayesian probability is a degree of belief.

I have taken a pragmatic approach and use Bayesian inference if I need to update the probability distribution when new data arrives.

9.10.2 Ordinary Least Squares vs. Bayesian linear regression

The problem is the familiar "which line best describes the data". If the data is points (x_i, y_y) , $i = 1..N$, then Ordinary Least Squares (OLS) finds parameters (a, b) that give the data as

$$y_i \approx a x_i + b \ . \tag{65}$$

Define

$$\beta = \begin{pmatrix} a \\ b \end{pmatrix} \text{ parameter vector} \tag{66}$$

$$\tag{67}$$

and stack the N known ***feature vectors*** $(x_i, 1)$ and known y_i to

$$X = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \\ x_N & 1 \end{pmatrix} \text{ features} \quad , \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \ . \tag{68}$$

Now the task is to find parameters β so that

$$y \approx X\beta \ , \tag{69}$$

and as a least squares problem one solves

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \, ||y - X\beta||^2 \ . \tag{70}$$

The solution is

$$\hat{\beta} = (X^T X)^{-1} X^T y \quad \text{, if } X^T X \text{ is invertible .} \tag{71}$$

More generally, the solution is $\hat{\beta} = X^{+}y$ where X^{+} is the Moore-Penrose pseudoinverse. The pseudoinverse chooses the minimum-norm solution

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} ||\beta|| \quad \text{s.t. } X\beta = y \ . \tag{72}$$

This gives a meaningful OLS with the aid of a singular value decomposition (SVD) even if $(X^T X)^{-1}$ doesn't exist.

What are the credibility limits? For that you'd have to make an extra assumptions, such as Gaussian noise,

$$y_i = ax_i + b + \epsilon_i \ , \quad \text{where } \epsilon_i \sim \mathcal{N}(0, \sigma^2) \ , \tag{73}$$

where $\sim$ means "sampled from". In this situation OLS becomes the so-called *Maximum Likelihood Estimate* (MLE).

Notation: Mathematically, the normal distribution with mean μ and variance σ^2 is $\mathcal{N}(\mu, \sigma^2)$, but in Python packages the second argument is σ, the standard deviation.

NumPy has `rng.normal(loc=mu, scale=sigma)`

SciPy has `scipy.stats.norm(loc=mu, scale=sigma)`

PyMC has `pm.Normal("x", mu=mu, sigma=sigma)`

PyTorch has `torch.distributions.Normal(loc=mu, scale=sigma)`

Also Julia `Distributions` calls this `Normal(mu, sigma)`.

`rng.normal(0.1,2.0)` means mathematically $\mathcal{N}(0.1, 4)$.

We can tackle the problem using **Bayesian inferense** and get the mean values a, b *and* their credibility limits. Moreover, we get the *full probability densities* of the all the parameters (a, b, σ)·

Priors for (a, b, σ)

The priors are reasonable but weak assumptions about the parameters. If a and b are somewhere around zero, then we use broad normal (Gaussian) distributions $\mathcal{N}$,

$$a \sim \mathcal{N}(0, \sigma_a^2) \quad \text{choose } \sigma_a^2 \approx 5 - 10 \tag{74}$$

$$b \sim \mathcal{N}(0, \sigma_b^2) \quad \text{choose } \sigma_b^2 \approx 5 - 10 \ . \tag{75}$$

The prior $p(\sigma)$ is safe to set to a broad distribution with $p(\sigma < 0) = 0$ to ensure positivity of σ. A suitable choice is the truncated Cauchy distribution, which has a long tail that doesn't cut off possible large-σ values.

Bayes' theorem and the posterior we want

We want the posterior $p(a, b, \sigma|\boldsymbol{x}, \boldsymbol{y})$ with data $(\boldsymbol{x}, \boldsymbol{y}) = (x_1, y_1, x_2, y_2, ...x_N, y_N)$. According to Bayes' theorem,

$$p(a, b, \sigma|\boldsymbol{x}, \boldsymbol{y}) \propto p(\boldsymbol{x}, \boldsymbol{y}|a, b, \sigma)p(a)p(b)p(\sigma) \ . \tag{76}$$

The rhs is simple to find. The noise in y_i are independent, so we have a product of Gaussians,

$$p(\boldsymbol{x}, \boldsymbol{y}|a, b, \sigma) = \prod_{i=1}^{N} \mathcal{N}(y_i|\mu_i, \sigma^2) = \prod_{i=1}^{N} \mathcal{N}(y_i|ax_i + b, \sigma^2) \tag{77}$$

$$= \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-(y_i - \mu_i)/(2\sigma^2)) \tag{78}$$

$$= (2\pi\sigma^2)^{-N/2} \exp(-\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - ax_i + b)/(2\sigma^2)) \ . \tag{79}$$

The priors $p(a)$, $p(b)$, and $p(\sigma)$ were set earlier, so the rhs of Eq. (76) is known.

The 3D distribution $p(a, b, \sigma|\boldsymbol{x}, \boldsymbol{y})$ has no closed form solution, but we can, nevertheless, *sample from it* - even though we don't know the normalization factor either. This is a task for *Probabilistic Programming*. We need an algorithm that effectively samples through the possible values of (a, b, σ) according to the distribution $p(\boldsymbol{x}, \boldsymbol{y}|a, b, \sigma)p(a)p(b)p(\sigma)$, which, according to Eq.(76), is **proportinal** to the quantity we want, $p(a, b, \sigma|\boldsymbol{x}, \boldsymbol{y})$. One such algorithm is Hamiltonian Monte Carlo (HMC).

Hamiltonian Monte Carlo (HMC)

HMC is inspired by physics. We want to sample from $p(q, data)$, which in the linear regression case was $p(q, data) := p(a, b, \sigma|\boldsymbol{x}, \boldsymbol{y})$.

Imagine classical particles with positions q. The particles are affected by a position-dependent potential and they have mass M and momenta p_i. In classical mechanics, the Hamiltonian function of positions and momenta is the total energy and has the unit of energy. We now shift to probabilistic quantities and write a function that is *analogous* to the Hamiltonian function,

$$H(q,p) = \frac{1}{2M}\sum_i p_i^2 + U(q) = \frac{1}{2}p^T M^{-1} p + U(q) \qquad \leftarrow \text{dimensionless! .} \tag{80}$$

The joint probability density of values (q, p) is (two different p's here, probability density and momentum)

$$p(q,p) \propto e^{-H(q,p)} = e^{-\frac{1}{2}p^T M^{-1} p} e^{-U(q)} \ . \tag{81}$$

This looks like the finite-temperature Gibbs distribution, but with dimensionless H and no temperature (or $k_B T = 1$). If we choose

$$e^{-U(q)} = p(q|data) \ , \tag{82}$$

we have

$$p(q,p) \propto e^{-H(q,p)} = e^{-\frac{1}{2}p^T M^{-1} p} p(q|data) = p(p)p(q|data) \ . \tag{83}$$

The momentum factor - introduced just to make this look like a dynamical system - is a quadratic form, and we can immediately see that momenta have a Gaussian distribution, $p \sim \mathcal{N}(0, M)$. The point of this exercise is that *we know how to simulate classical motion*, just do it now in (q, p)-space. In analogy to the Hamiltons equations of motion in classical mechanics we have

$$\frac{dq_i}{dt} = \frac{\partial H(q,p)}{\partial p_i} \ , \ \frac{dp_i}{dt} = -\frac{\partial H(q,p)}{\partial q_i} \Leftrightarrow \dot{q} = M^{-1}p \ , \ \dot{p} = -\boldsymbol{\nabla}_q U(q) \ , \tag{84}$$

and algorithms, such as **leapfrog**, solves the trajectories $(q(t), p(t))$ which keep $H(q, p)$ close to a constant (small fluctuations can be corrected with a Metropolis accept/reject step). The gradients of H are computed using AD. The main benefit of HMC is that stochastically independent momenta are sampled directly from $\mathcal{N}(0, M)$, and the whole (q, p)-space is sampled very effectively. Schematically, HMC does repeatedly these steps:

- At the current q, draw a fresh random momentum $p \sim N(0, M)$.
- Follow Hamilton's equations for some time, the trajectory takes to a proposed new point (q', p').
- Accept/reject it with a Metropolis question (corrects small energy fluctuation).
- Throw away p', keep q', and repeat. $\leftarrow$ discarding p' makes a jump to a new trajectory

After the initial transient/adaptation and assuming the Markov chain explores the target adequately, the retained q-values are distributed according to $p(q|data)$.

A few points about HMC:

- HMC trajectories need $\dot{p} = -\boldsymbol{\nabla}_q U(q)$, but isn't $U(q) = -\ln(p(q|data))$ unknown? Yes, but it's unknown only up to an unknown factor. From Bayes, we know that

$$p(q|data) = \frac{p(data|q)p(q)}{p(data)} \ , \tag{85}$$

  where the normalization is the unknown marginal distribution $p(data) = \int p(data|q)p(q)dq$ By definition,

$$\dot{p} = -\boldsymbol{\nabla}_q U(q) := -\boldsymbol{\nabla}_q \left(-\ln(p(q|data)\right) = \boldsymbol{\nabla}_q \left(\ln(p(data|q)p(q)\right) - \ln(p(data)) \tag{86}$$

$$= \boldsymbol{\nabla}_q \ln(p(data|q)) + \boldsymbol{\nabla}_q \ln(p(q)) - \underbrace{\boldsymbol{\nabla}_q \ln(p(data))}_{=0} \ , \tag{87}$$

  so don't need to know $p(data)$. Notice, that here $p(q)$ is the *prior* probability of q, a known quantity we feed in.

- Why is keeping $H(q,p)$ constant so important? We are sampling an artificial joint distribution

$$\Pi(q,p) \propto e^{-H(q,p)} \ . \tag{88}$$

  The trajectories $(q(t), p(t))$ keep $H(q,p)$ constant, so we don't change the distribution and the density of points on a trajectory is constant: $\Pi(q(t), p(t)) = \Pi(q(0), p(0))$. This is, however, not enough.

- To get actual probabilities, the probability density $\Pi(q,p)$ is integrated over a phase-space volume in question, in other words, the probabilities are found integrating the **probability measure** $\Pi(q,p)dqdp$ over a phase-space volume. Hamiltonian dynamics satisfies also this requirement, as it preserves the phase-space volume (Liouville's theorem). In HMC, the targeted probability measure remains constant,

$$e^{-H(q',p')}dq'dp' = e^{-H(q,p)}dqdp \ . \tag{89}$$

- We can't stay on one constant energy, because the distribution $\Pi(q,p) \propto e^{-H(q,p)}$ is for **many different energies**. Therefore HMC jumps between trajectories by updating the momemtum to a new value.

HMC sampling uses multiple trajectories, where, put together, the points have the correct distribution. The idea is related to **Langevin dynamics**, which has friction and stochastic force.

Bayesian linear regression model in Julia:
(snippet from Bayesian_linear_regression.jl) The Julia package `Turing.jl` uses near-mathematical notation,

```
@model function bayes_linreg(x, y)
   # Priors (very liberal, use large sigma to allow deviations from 0)
   a ∼ Normal(0, 5)
   b ∼ Normal(0, 5)
   σ ∼ truncated(Cauchy(0, 2); lower=0) # Cauchy to allow also large positives
   # Likelihood
   for i in eachindex(x)
      μ = a * x[i] + b
      y[i] ∼ Normal(μ, σ)
   end
end
model = bayes_linreg(x, y)  # create model from macro @model
chain = sample(model, NUTS(0.65), 2000) # HMC sampling under the hood
```

This generates 2000 moves in (q, p) space, and we can read the approximate distribution of $q = (a, b, \sigma)$ from the data.

Bayesian linear regression model in Python using `PyMC`:
(snippet from Bayesian_linear_regression.py)

```
import pymc as pm
...
with pm.Model() as model:
      a = pm.Normal("a", mu=0.0, sigma=5.0)
      b = pm.Normal("b", mu=0.0, sigma=5.0)
      sigma = pm.HalfCauchy("sigma", beta=2.0)

      # Likelihood
      mu = a * x + b
      pm.Normal("y_obs", mu=mu, sigma=sigma, observed=y)

      # Sample posterior
      trace = pm.sample(
          2000,
          tune=1000,
          chains=3,
          cores=3,
          target_accept=0.9,
          random_seed=1234,
          return_inferencedata=True,
      )
a_samps = trace.posterior["a"].values.flatten()
b_samps = trace.posterior["b"].values.flatten()
sigma_samps = trace.posterior["sigma"].values.flatten()
```

9.10.3 Gaussian Process Regression (GPR)

Let's generalize the Bayesian linear regression to infinitely many smooth functions. This can be done for two reasons:

- Functions are represented by a finite number of samples

$$f(X) = (f(x_1), f(x_2), ..., f(x_n))$$

- All distributions are taken to be Gaussians. Central Limit Theorem: The distribution of a *sum of random variables* approaches a Gaussian as more variables are added; this is true for very many distributions of the random variables.
  $\Rightarrow$ Gaussians appear naturally everywhere.

Suitable functions are picked by saying that some functions are more probable than the other:

Extend from the idea
A Gaussian distribution is a distribution of numbers
to the idea
A Gaussian process is a distribution of functions.

And why would we want to do GPR?

- 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.
- GPR gives ***credibility limits***.

What is a smooth function?

Smoothness means, roughly speaking, that if two x-values are close, the corresponding function values should also be close, in order to avoid large derivatives in $f(x)$,

$$x \approx x' \Rightarrow f(x) \approx f(x') \ . \tag{90}$$

A more precise definition is that function values of nearby x's are strongly correlated, [30]

$$\mathrm{Cov}[f(x), f(x')] = k(x, x') \ , \tag{91}$$

where the ***kernel function*** $k(x, x')$ has

$$k(x, x') \approx 1 \ \text{ if } x \text{ and } x' \text{ are close}$$
$$k(x, x') \approx 0 \ \text{ if } x \text{ and } x' \text{ are distant} \ .$$

One possibility (there are many) is squared-exponential quadratic kernel (RBF),

$$k(x, x') = \sigma_f^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right) \ . \tag{92}$$

The length scale ℓ defines what "close" means, $|x - x'| \ll \ell \Rightarrow k(x, x') \approx 1$. The output variance σ_f^2 controls the vertical scale: It determines how far the random curves can swing above and below the mean function (usually 0) on the vertical axis.

The kernel defines an infinite set of smooth curves, a *prior* $p(f)$ that is nonzero for smooth functions. The infinitely many functions defined by the chosen kernel are often marked

$$f \sim \mathcal{GP}(0, k) \ . \tag{93}$$

This is our prior. Before seeing the data, we regard $f(x)$ as a random function drawn from a Gaussian process whose mean function is zero and whose covariance is specified by the kernel k. This is just a mathematical notation and we need to express it using a finite set of random points $X = \{x_1, x_2, ..., x_n\}$. In this set, the kernel function defines the so-called ***covariance matrix*** K, with elements

$$K_{ij} = k(x_i, x_j) \ . \tag{94}$$

Translating the mathematical $f \sim \mathcal{GP}(0, k)$ to functions sampled from the finite set X, we have vectors

$$f(X) = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix} \ . \tag{95}$$

[30] Covariance $\mathrm{Cov}[A, B] = \mathbb{E}[AB] - \mathbb{E}[A]\mathbb{E}[B]$.

that are ***jointly Gaussian***,

$$f(X) \sim \mathcal{N}(0, K) \tag{96}$$

Jointly Gaussian means that all components are independently sampled from a Gaussian,

$$p(f(X)) = (2\pi)^{-n/2}|K|^{-1/2} \exp\left(-\frac{1}{2} f(X)^T K^{-1} f(X)\right) \ . \tag{97}$$

If a function is not smooth, then $f(X)^T K^{-1} f(X)$ is large, and the exponent kills the probability, $p(f(X)) \approx 0$. There you have it: the probability density $p(f(X))$ of smooth functions, later used as our prior in the Bayes' theorem.

If you want to see *some* of the smooth functions we use as prior you can follow this process:

1. Select the x-grid, e.g., 100 points spaced evenly from 0 to 10.
2. Build the covariance matrix K: Calculate the kernel function $k(x, x')$ (such as RBF) for every possible pair of those 100 points. This generates a matrix. Because of the kernel, points sitting right next to each other on the grid will have a correlation value near 1, while points on opposite ends will have a correlation near 0.
3. Draw multivariate normal samples: Feed the 100x100 covariance matrix and a vector of 100 zeros into a random number generator for a multivariate Gaussian distribution $y \sim \mathcal{N}(0, K)$.
4. Plot the result: The generator outputs a vector of 100 y-values. Because neighboring points are highly correlated, connecting these 100 points forms a continuous, smooth curve. Repeating this process generates a brand-new, completely different smooth curve.

In Python,

```
import numpy as np
import torch

def rbf_kernel(x1, x2, length_scale=0.1):
  """Squared exponential kernel"""
  x1 = x1[:, None]  # shape (m,1)
  x2 = x2[None, :]  # shape (1,m)
  return np.exp(-0.5 * ((x1 - x2)/length_scale)**2) # shape (m,m)

m = 100  # number of x points
N = 1000 # number of smooth functions
x = np.linspace(0, 1, m)
K = rbf_kernel(x, x, length_scale=0.5)
Y = np.random.multivariate_normal(mean=np.zeros(m), cov=K, size=N)
for y in Y:
   ... # y is a set of m y-values on a smooth curve (x, y)
```

Technical remark: Componentwise, $p(f(X))$ gives the expectation value and covariance

$$\mathbb{E}[f(x_i)] = 0 \quad , \ \mathrm{Cov}[f(x_i), f(x_j)] = k(x_i, x_j) = K_{ij} \ . \tag{98}$$

Numerically, the probability distribution $p(f(X))$ gives under- or overflows, therefore it is always evaluated as a log-probability,

$$\ln p(f(X) = -\frac{n}{2} \ln(2\pi) - \frac{1}{2} \ln |K| - \frac{1}{2} f(X)^T K^{-1} f(X) \ . \tag{99}$$

One makes a Cholesky decomposition, with a lower triangular matrix L,

$$K = LL^T \ , \tag{100}$$

and now $\ln |K|$ becomes a log-trace,

$$\ln |K| = 2 \sum_i \ln L_{ii} \ , \tag{101}$$

and

$$f(X)^T K^{-1} f(X) = f(X)^T ((L^T)^{-1} L^{-1}) f(X) \ . \tag{102}$$

Define

$$\alpha = L^{-1} f(X) \Leftrightarrow L\alpha = f(X) \ , \tag{103}$$

so we find α by solving a triangular set of equations, and now

$$f(X)^T K^{-1} f(X) = \alpha^T \alpha = ||\alpha||^2 \ . \tag{104}$$

The Cholesky decomposition and memory requirements limit the size of the set of points X.

How to compute the probability of a function in the presence of data?

Let's assume we have a set of known data points $D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, and we want to pick all smooth curves that are consistent with the data. We assume that the data points are given by

$$y_i = f(x_i) + \epsilon_i \quad , \ \epsilon_i \in \mathcal{N}(0, \sigma^2) \ , \tag{105}$$

The data are a noisy observations of the function.

The variance σ^2 is a parameter we give to the data, it tells how trustworthy the data is:

$$\begin{aligned} \sigma^2 \to 0 \quad & \text{data is almost exact, functions interpolate the points} \\ \sigma^2 \gg 1 \quad & \text{data is noisy, functions smooth heavily .} \end{aligned}$$

Now comes a clever insight, which we already used in Bayesian linear regression:

We don't need to know how reliable the data is, we treat σ^2 as a parameter. In GPR, we only set a suitable prior $p(\sigma)$.

The observations are independent Gaussians,

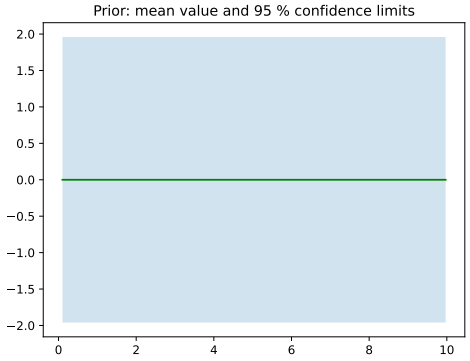
$$p(D|f) = \prod_i \mathcal{N}(y_i|f(x_i), \sigma^2) \ . \tag{106}$$

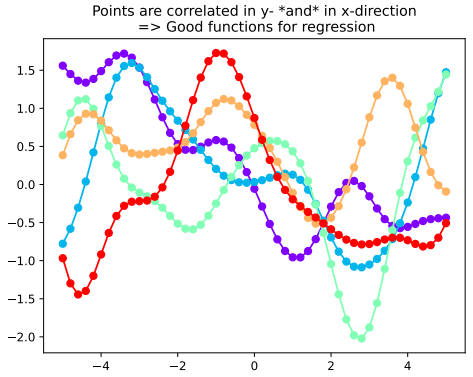
At this point Bayes' theorem comes to aid, remember it shows how to get $P(A|B)$ from known $P(B|A)$. Hence, we get from the probability density of data given f to probability density of f given data,

$$p(f|D) \propto p(D|f)p(f) \quad \text{probability density of function } f \text{ in the presence of data } D \ . \tag{108}$$

Gaussian processes are implemented in the Tensorflow-Keras framework as GPflow, and in Pytorch as gpytorch. See gaussian_process_intro.py.

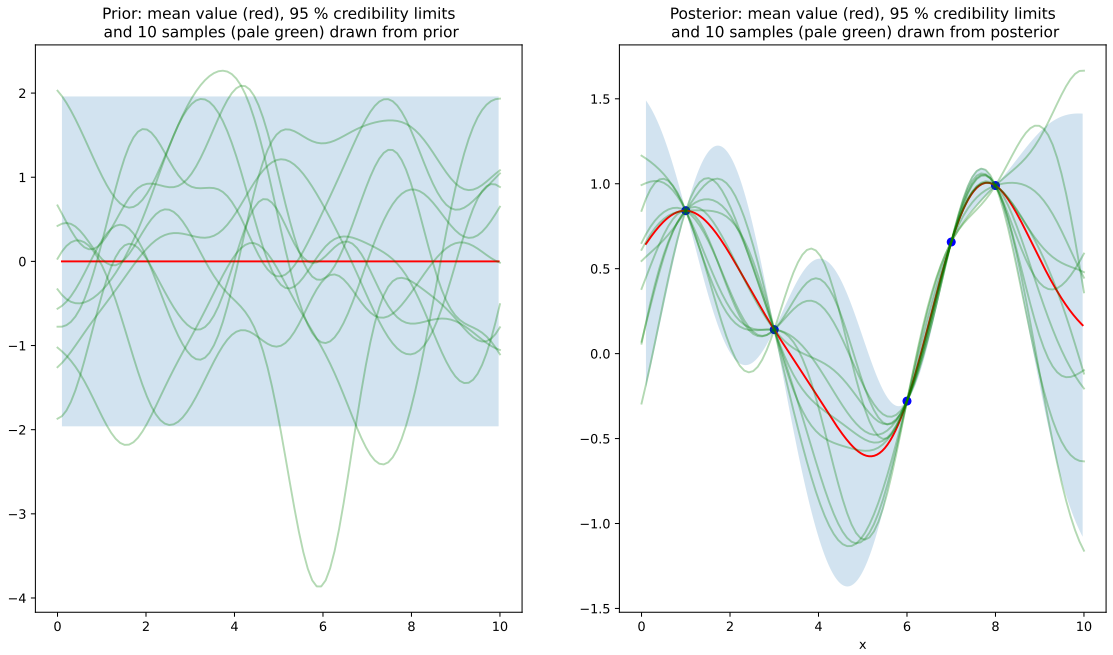
Left: Prior knowledge of the result. Right: Five of the infinitely many smooth Gaussian process functions $f \sim \mathcal{GP}(0, k)$ used as prior.





We don't need to actually generate the infinitely many smooth Gaussian process functions. All probabilities are Gaussians, and one can do the math and get the result as if we used infinitely many smooth Gaussian process functions.

I won't go through the details here.



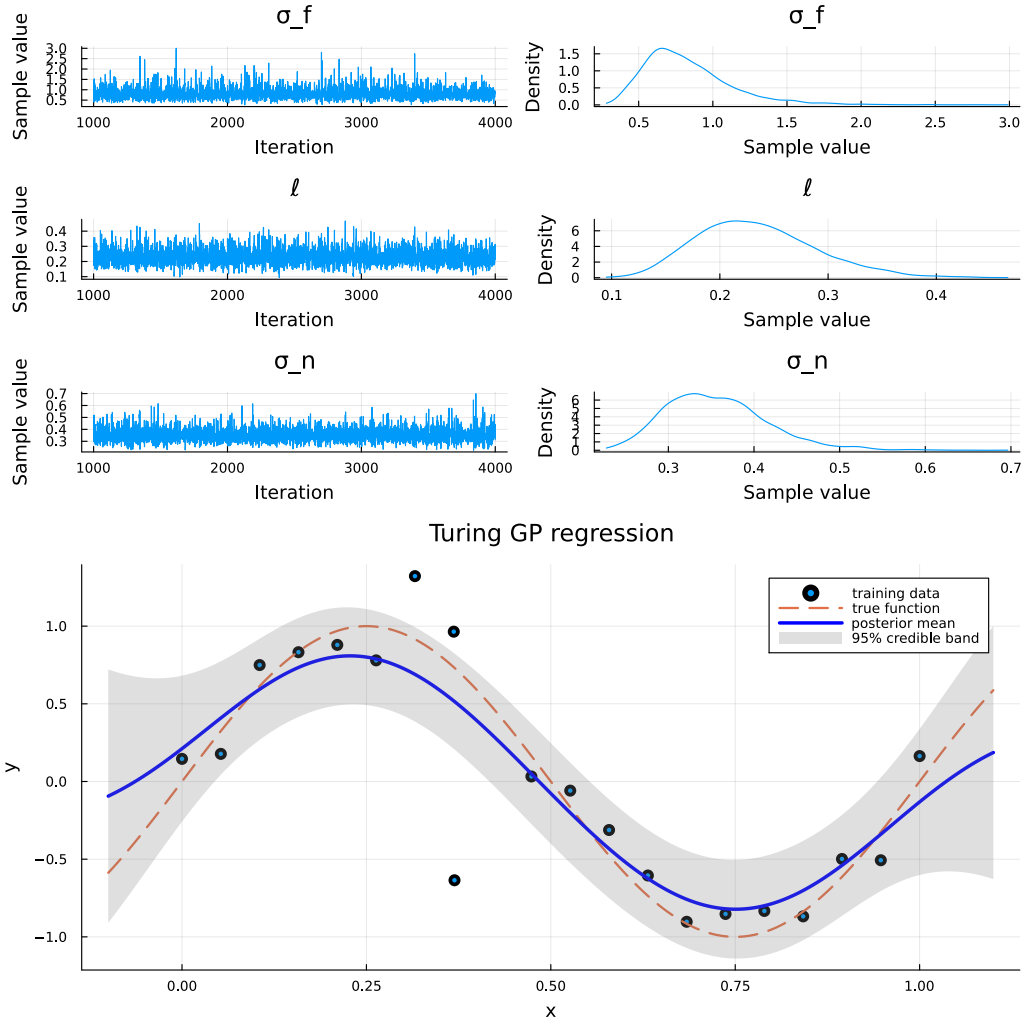
Prior is a rather liberal guess that most GPR functions lie within one-sigma limits with $\sigma = 1.0$ from the baseline $y = 0$. The 95% credibility limits of prior are $\pm 1.96 \times \sigma$.

Posterior is *accurate* observations $D = (x_1, y_1)...(x_5, y_5)$ added to the prior.
The posterior GPR functions are pinched to go through the data points, but fluctuate between them. Far away from data points the GRP functions are pulled toward the prior mean, in this case toward the $y = 0$ axis shown in the left panel.

"If there's no data nearby, use prior".

The code sklearn_GPR.py uses the Scikit-learn package `sklearn.gaussian_process`.

The `Turing.jl` package used to get a GPR for noisy data (with outliers), using the code GPR_with_Turing.jl. Notice that we don't know how realiable the data are, so we sample the variance σ^2 from a suitable prior.



9.11 JAX

It was difficult to deside where in these lexture notes 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.

As said, JAX can do automatic differentiation and jit compilation, and it can offload computations to GPU. JAX competes directly with NumPy and Numba, and a neural network software can be written based on JAX, because JAX can do both forward-mode and reverse-mode automatic differentiation, and backpropagation is a special case of the latter.

JAX can take advantage of fused multiply–add (FMA) instructions. Both Numba and JAX need a first call to trigger compilation. Technically, Numba and JAX are different in that Numba specializes on Python and NumPy types, while JAX specializes on shapes and dtypes and builds an XLA (Accelerated Linear Algebra by Google) computation graph.

jax_jit_test.py

```
import jax.numpy as jnp
from jax import jit
import numpy as np
from numba import njit
import timeit

def f(x):
  return x*x + 5*x

jax_f = jit(f) # compilation outside timing
numba_f = njit(f)
numba_f(np.ones((50, 50))) # compilation outside timing

jax_x = jnp.ones((5000, 5000))
numpy_x = np.ones((5000, 5000))

nrep = 100
print(f"timeit averages over {nrep} reps")

t_basic = timeit.timeit("f(numpy_x)", globals=globals(), number=nrep)
t_jax = timeit.timeit("jax_f(jax_x)", globals=globals(), number=nrep)
t_numba = timeit.timeit("numba_f(numpy_x)", globals=globals(), number=nrep)
print(f"  Basic Python with NumPy array {t_basic/nrep:.5f} s")
print(f"              JAX with JAX array {t_jax/nrep:.5f} s")
print(f"         Numba with NumPy array {t_numba/nrep:.5f} s")
```

Sample result:

```
Basic Python with NumPy array 0.09416 s
            JAX with JAX array 0.01350 s
       Numba with NumPy array 0.07152 s
```

10 Parallel Python

There are almost too many avenues to take, see wiki about Python parallel processing. I'm reasoning that you have a multi-core computer, so what you'd want is ***shared-memory parallel programming***, which leads us to Python `multiprocessing` or `concurrect.futures`.

If memory is not shared, like between nodes in a computer cluster, you need ***message passing***. For that we have ***MPI***, the message passing interface, which is between Python and the hardware. Popular ones are OpenMPI, MPICH, Microsoft MPI (MS MPI), and Intel MPI. Python uses MPI in modules such as `mpi4py`. Apart from MPI, there are other parallel paradigms, too, such as the Hadoop framework for processing large volumes of data.

Let's first get those cores work in parallel.

10.1 Threads, Multiprocessing, and Free-Threaded Python

The module `threading` starts threads that live in the same process and use the same address space.

```
import threading

t = threading.Thread(target=work, args=(arg,))
t.start()
t.join()
```

A longer `threading` code with timing.

threads.py

```
import threading as th
from time import perf_counter as T

def count(n):
    while n>0:
        n -= 1

N = 100000000
# serial
tic = T()
count(N)
count(N)
toc = T()-tic
print(f"  serial: {toc:.3f} 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(f"threaded: {toc:.3f} seconds")
```

An "ordinary" CPython interpreter has the GIL (**G**lobal **I**nterpreter **L**ock) enabled, and only one thread executes Python bytecode at a time. With the GIL, CPU-bound tasks that spend most of their time computing gain nothing from `[threading`, although it remain a valid approach for IO-bound tasks that spend most of their time waiting for data.

Free-threaded CPython (no GIL) has existed since version 3.13 and is officially supported in 3.14, although it remains an optional build and extensions can re-enable the GIL. I remind that many C/Fortran numerical kernels, including many NumPy modules, release the GIL while working. NumPy threading was already capable of real parallel execution long before free-threaded Python.

Installation of free-threaded Python:

```
  conda install conda-forge::python-freethreading
```

Testing:

```
import sys
import sysconfig

print(sys.version)
print(sysconfig.get_config_var("Py_GIL_DISABLED"))
print(sys._is_gil_enabled())
```

Sample timings of `threads.py` using a Python 3.14 interpreter with the GIL enabled:

```
  serial: 4.084 seconds
threaded: 3.934 seconds
```

Sample timings using a free-threaded Python 3.14 interpreter:

```
  serial: 3.394 seconds
threaded: 1.966 seconds
```

These are not serious benchmarks, just short tests to get a first impression.

GIL or no GIL, one can start multiple interpreters to work on multiple tasks. The module `multiprocessing` starts separate processes, which all have their own interpreter and address space.

```
from multiprocessing import Process

p = Process(target=work, args=(arg,))
p.start()
p.join()
```

Using a `Pool` of workers,

```
from multiprocessing import Pool

with Pool(4) as pool:
    result = pool.map(work, data)
```

multiprocesses.py

```
import multiprocessing as mp
from time import perf_counter as T

def count(n):
    while n>0:
        n -= 1

N = 100000000
# serial
tic = T()
count(N)
count(N)
toc = T()-tic
print(f"         serial: {toc:.3f} 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(f"multiprocessing: {toc:.3f} seconds")
```

Python 3.14 gets double the speed with two processes. Now we are getting somewhere!
However, in *free-threaded Python* the code produces a lengthy error message:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
    import sys; from multiprocessing.forkserver import main; main(3, 5, ['__main__'],
 ...
```

This brings us to the topic of ***start methods***.

10.1.1 Start Methods

The ***global start method*** defines how a new process is created and initialized.

- `fork`: Linux start method before Python 3.14
  - inherited address space and do copy-on-write memory management
    All processes are sharing the same set of memory pages and each one gets its own private copy when it wants to modify a page.
  - Very fast and convenient for some numerical workloads but discouraged as a general default.
- `forkserver`: Python $\geq$ 3.14 Linux start method default
  - Safer multithreading
- `spawn`: Start a fresh Python interpreter.
  - Default on Windows and macOS
  - Highest startup and serialization cost

  An example of how fork can be unsafe tells a story about an office with a kitchen and a coffee machine. The parent process is the original office, and thread A goes to the kitchen to make coffee and locks the door. While thread A is making coffee, thread B replicates the office using fork. Everything in the office is replicated to thread B, the whole memory state of the original office. But forking only replicates the parent, not thread A. Therefore, thread B sees a locked kitchen door, and if it wants coffee it goes to the door and waits for it to open. Only thread A could unlock the door, but in thread B universe there is no thread A to unlock the door. No coffee for thread B. Ever.

Test:

```
>>> import multiprocessing as mp

>>> mp.get_start_method()
```

If you want, you can set the start method to fork. The previous example works with fork but fails with forkserver.

The error is raised by the fact that spawn and forkserver require the main module to be safely importable. When the execution reaches the top-level process creation lines

```
t1 = mp.Process(target=count,args=(N,))
t1.start()
t2 = mp.Process(target=count,args=(N,))
t2.start()
```

the running interpreter can be forked to a new child and everything works. However, forkserver starts from the onset and whenever a new process is needed, the parent process connects to the server and requests a new, pristine process. Notice how the forkserver hands out processes, not the running process. This is what fails in our example: the forkserver process is itself initializing when it is asked to create more processes. This causes a bootstrapping error we saw. The guard

```
def main():
    ...
if __name__ == '__main__':
    main()
```

is no longer optional, because forkserver uses it to isolate the main module. A fixed version of code is

multiprocesses_fixed_main.py

```python
# guard of main module added for forkserver and spawn start methods

import multiprocessing as mp
from time import perf_counter as T

def count(n):
    while n>0:
        n -= 1

def main():
    N = 100000000
    # serial
    tic = T()
    count(N)
    count(N)
    toc = T()-tic
    print(f"         serial: {toc:.3f} 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(f"multiprocessing: {toc:.3f} seconds")

if __name__=='__main__':
    main()
```

10.1.2 `concurrent.futures`

`concurrent.futures` provides one high-level interface for thread pools, process pools, and – using free-threaded Python – isolated interpreter pools.

Instead of manually spawning, joining, and queuing workers, you interact with asynchronous tasks via a concept called ***Futures***.

- `ThreadPoolExecutor` uses a pool of *threads* to execute calls asynchronously - remember GIL.
- `ProcessPoolExecutor` uses a pool of *processes* to execute calls asynchronously.

`concurrent.futures` with threads:

```
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max_workers=4) as ex:
    result = list(ex.map(work, data))
```

`concurrent.futures` with processes:

```
from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor(max_workers=4) as ex:
    result = list(ex.map(work, data))
```

The `concurrent.futures` abstraction lets you switch between threads and processes by editing one single line of code.

An example of `InterpreterPoolExecutor` is `interpreter_pool_executer.py`.

In an old Python interpreter the code will throw `ImportError: cannot import name 'InterpreterPoolExecutor' from 'concurrent.futures'`.

Summary:

- ThreadPoolExecutor
  - shared memory
  - ordinary threads
- InterpreterPoolExecutor

  - same process
  - isolated interpreters
  - separate GILs
- ProcessPoolExecutor
  - separate processes
  - separate address spaces

10.1.3 Parallelization and what you should *not* parallelize

I don't want to play down the importance of parallel execution, but you should realize that not all tasks parallelize well.

> Each worker must get a sufficient amount of computing in order to pay the parallelization overhead.

If the time it takes to coordinate workers is greater than the time saved by dividing the work, parallelization actually makes the program slower.

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
```

- Serial execution: 0.004 seconds. Parallel execution: 1.95 seconds, almost 500 times ***slower***.
- NumPy is very fast, it's compiled C-code for vectorized calculations.
- Starting a process — in this case a full CPython interpreter — has a large overhead (handing work to a running process has an overhead, too).

The parallel speed in this example is so miserable that there must be something else than just overheads. First, ***prosesses don't share memory space***, so `multiprocessing.pool.map` copies the whole, large list `arg` to every process. 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*.

One divide the workload to bigger chunks, something like

```
nproc = 10
chunks = np.array_split(arg, nproc) # Split array into multiple sub-arrays of equal size.
with mp.Pool(nproc) as pool:
    parts = pool.map(np.sqrt, chunks)
root = np.concatenate(parts)
```

This is a lot better, but `np.sqrt()` is a tiny operation, and a simulation taking seconds may better amortize the parallelization overhead.

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 reason to use it in numerics.

10.1.4 Examples of `concurrent.futures`

A rather good task to parallelize is finding text in files. The `glob` module offers "Unix style pathname pattern expansion",

concurrent_findtxt.py

```
# test if files *py contain "numpy"
import glob
from time import sleep
from concurrent.futures import ProcessPoolExecutor
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 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,it`
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.

Instead of map, you can submit tasks and get their results as_completed.

concurrent_submit.py

```
from concurrent.futures import as_completed, ProcessPoolExecutor
import numpy as np
from time import sleep
from time import perf_counter as T
import os

print(f'Your machine has {os.cpu_count()} logical CPUs visible to the system.')

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 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 ProcessPoolExecutor() as executor:
        for arg in xsl:
            future = executor.submit(cube, arg)
            futures[future] = arg
        for f in 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 asynchronously, 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` it corresponds to.

Another practicality: After the job is done, the context manager automatically calls `Executor.shutdown` that does automatic process cleanup.

10.2 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.

10.2.1 Safe locking with a context

Workers, threads or processes, may share resources but shouldn't access them simultaneously: ***locking*** limits access to shared resources. 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.

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('good_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')
```

Race conditions

Locking is one way to avoid the so-called ***Producer-Consumer problem*** (Wikipedia) caused by concurrency. The Producer manufactures items and the Consumer consumes. 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.*

Another race condition is mentioned in `numba_cuda_ising.py`, where the 2D Ising model spin update is done in parallel. A spin flip probability depends on the direction of neigboring spins, so flipping any of the neighboring spins simultaneously changes the conditions while evaluating them.

10.3 Subprocess: embarrasingly parallel jobs

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 bash shell command `ls -la` [31],

```
import subprocess
subprocess.run(["ls", "-la"])
```

A more general usage is also simple (you should provide `command`)

```
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

[31] `subprocess` module is overtaking modules `os` and `spawn`.

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. [32]

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.

[32] This code uses `time.time` to better reflect the result.

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.

10.4 MPI Parallelism with `mpi4py` (*MPI for Python*)

MPI is the standard parallelization method in clusters with many computing nodes.

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).[33]

Before communication takes place, Python objects are serialized using Python's native `pickle` protocol (see section 14.15.8), and afterwards data is deserialized. The small overhead of pickling is often tolerable.
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
```

[33] In 2009, `mpi4py` was rewritten in Cython from scratch.

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`.

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
```

In a SLURM batch queue system your resource reservation is automatically passed on to `mpirun`:

```
#!/bin/bash
#SBATCH -J mytest
#SBATCH -o out
#SBATCH -e errorout
#SBATCH -n 4

mpirun python mpi4py_1.py  # starts as many processes (4) as the SLURM script reserves
```

Here *#SBATCH* is not a commented line, a comment line has multiple #s. If you execute

```
mpirun python mpi4py_1.py
```

in a computer with no batch queue system it'll use all visible cores.

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 value using `np.mean()` and the standard deviation using `np.std()`. For a single value `np.std()` is 0, therefore I block tasks to `nb` batches. The ***statistical error*** of data X is

$$SE(X) = \frac{S_X}{\sqrt{\text{nb}}} , \quad S_X = \text{np.std}(X, \text{ddof} = 1) . \tag{109}$$

`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`. [34]

Collective calls

Logically, `scatter()` and `bcast()`, `allgather`, and `alltoall` are ***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 of values from workers
```

Here `MPI.SUM` tells to reduce the sum, product would be reduced with `MPI.PROD`.

10.4.1 send/recv vs. 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`.

[34] The type of `None` is `<class 'NoneType'>`. Clever Python.

NumPy array MPI communication is an example of a ***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(...define shape here...)  # 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

```python
# run with
# mpirun -np 2 python mpi4py_Send_Recv.py
from mpi4py import MPI
import numpy as np
from  time import perf_counter as T

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

N=1000  # number of repeats
shape=(100,5) # shape of NumPy array to communicate

if rank== 0: tic = T()
for _ in range(N):
    if rank== 0:
        data= np.random.random(shape)
        comm.send(data, dest=1, tag=4)
    elif rank== 1:
        data= comm.recv(source=0, tag=4)
if rank== 0:
    toc = T()
    timing1 = toc-tic

if rank== 0: tic = T()
for _ in range(N):
    if rank== 0:
        data= np.random.random(shape)
        comm.Send(data, dest=1, tag=5)
    elif rank== 1:
        data= np.empty(shape,dtype=float)
        comm.Recv(data, source=0, tag=5)

if rank==0:
    toc = T()
    timing2 = toc-tic
    print(f'{N} times MPI communicate NumPy array {data.shape}' )
    print(f'send/recv took {timing1} seconds')
    print(f'Send/Recv took {timing2} seconds')

#1000 times MPI communicate NumPy array (100, 5)
#send/recv took 0.013868566999999998 seconds
#Send/Recv took 0.005406212999999993 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}')
```

10.4.2 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,[35]

[35] 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.

10.4.3 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()
```

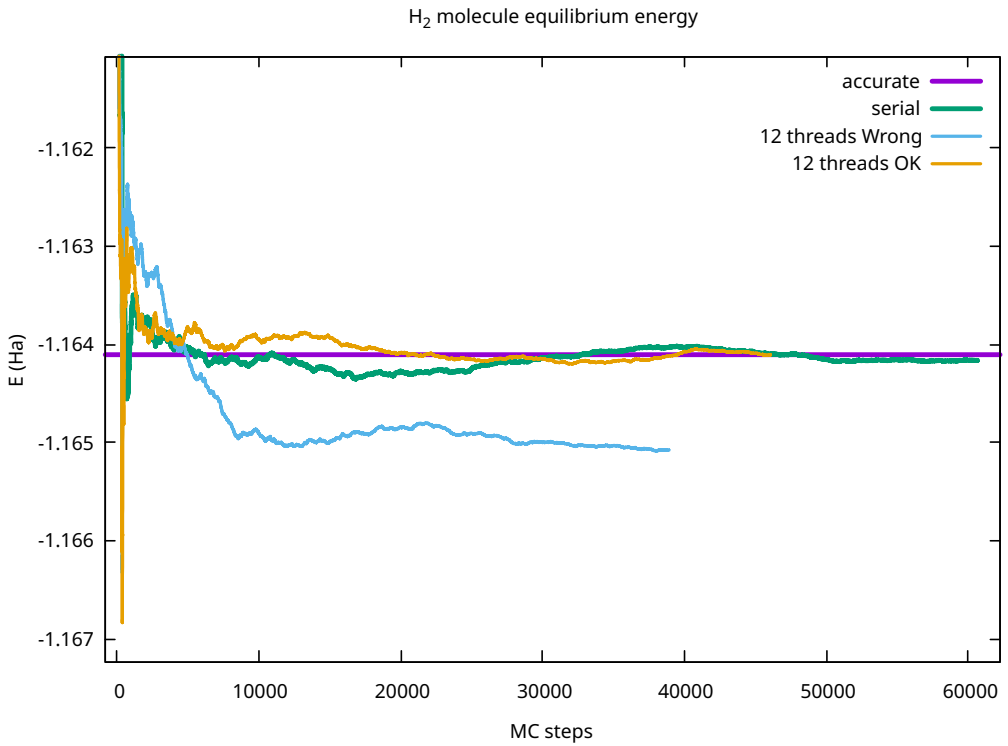
The `comm.isend` and `comm.irecv` methods return ***request instances*** (hence I called the variable `req`). These request instances must be completed with a blocking call to either `req.test` or `req.wait`. If you just want to inquire the status of the request `req` without stopping, then `req.test` will return `True` if the operation has completed.

10.5 When parallel code gives the wrong result

In a multithreaded code, consider these points (valid in many languages):

- Do you compute random numbers in a function that is accessed by multiple threads? The random number generator in many programming languages is a global generator, meaning it has a one single internal state. That internal state remembers that you've generated a random number using the algorithm and the next time it should generate a new one - if there were no such memory of a state, it would give you the same number every time you call the function. The problem is that the state may be updated by multiple threads asking for a number and it can cause a race condition.
- Did you inline the code that computes random numbers? That may cause trouble.

- Do you use a pre-allocated array that is used by a function? That array is shared by all threads, so it's elements may be used by a random thread. Happened to me while computing the energy of an H_2 molecule using Diffusion Monte Carlo. I had an array `F`, which I cleverly allocated before the loop to save time, and the loop was executed using 12 threads: Wrong result![36]



[36] The value $-1.1640\,\mathrm{HA}$ is a finite-τ result, the correct ground state energy is about $-1.1745\,\mathrm{Ha}$.

11 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++? [37]

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`.

11.1 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.

[37] SWIG (Simplified Wrapper and Interface Generator) has been largely superceded by pybind11, Cython, and cffi, so I leave SWIG at rest.

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.

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.

11.2 pybind11 and nanobind

pybind11 is a C++11 and Python intercommunication library:

> Think of this library as a tiny self-contained version of `Boost.Python` with everything stripped away that isn't relevant for binding generation.

One of the design goals was to keep it lightweight compared to Boost, which has bloated in attempt to support almost any C++ compiler, including the buggy versions. Stefan Welt has a blog (link) about which tool to choose from pybind11, Cython, and cffi (which I haven't mentioned yet; cffi knows only C, not C++, but if you use PyPy you want cffi). pybind11 has built-in support for NumPy, and you can map C++ pointers directly onto NumPy arrays without copying data.

nanobind (also by Jacob Wenzel) is an even more compact successor of pybind11 with less legacy C++ support, created to improve the compilation time and binary size bloating of pybind11. Multi-dimensional arrays are very efficient: nanobind passes data seamlessly between NumPy, PyTorch, JAX, and native C++ code. You need a C++17+ compiler.
Workflow:

1. Write C++ code `nanobind_example.cpp` : Use standard C++ classes and functions.
2. Define the bindings in the C++ code: State which classes and functions should be accessible to Python using nanobind macros (`NB_MODULE`, `m.def`).
3. Compile the C++ code to a shared library: The binary extension is .so on Linux/macOS or .pyd on Windows.
4. Import the shared library in Python: Import the shared library using `import nanobind_example` just like a module or a regular Python library.

See nanobind example @github.

11.3 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 recent (2024) account of Cython, see speeding-up-python-with-cython.

A canonical example is finding prime numbers (picked from cython_tutorial),

primes.pyx

```
#!/usr/bin/env python

def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    klim = 1000
    result = []
    try:
        if kmax > klim: raise ValueError
        k = 0
        n = 2
        while k < kmax:
            i = 0
            while i < k and n % p[i] != 0:
                i += 1
            if i == k:
                p[k] = n
                k += 1
                result.append(n)
            n += 1
        return result
    except:
        print(f"too many primes, >{klim}")
        return -1
```

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]
```

11.3.1 Creating a standalone executable with Cython

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 += 1
            if i == k:
                p[k] = n
                k += 1
                result.append(n)
            n += 1
        return result
    except:
        print(f"too many primes >{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,[38]

```
$  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` and link the Python library. This was a conda env, hence the nasty paths:

```
$  gcc  -I$HOME/miniconda3/envs/python_freethreads/include/python3.14t -o primesmain \
primesmain.c  -L$HOME/miniconda3/envs/python_freethreads/lib/ -lpython3.14t
```

Testing (free-threaded interpreter needs convincing that we don't need GIL),

```
$  LD_LIBRARY_PATH=$HOME/miniconda3/envs/python_freethreads/lib/ PYTHON_GIL=0 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

[38] 2023: 8375 lines. 2026: 10636 lines

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`.

12 GPU computations

There are many Python packages for GPU computations. If you are not interested in ML, then ***CuPy*** is for you. CuPy is an almost drop-in replacement of NumPy and Scipy, so moving computations to GPU is simple. Installation of CuPy relies on the underlying CUDA (Nvidia) or ROCm (AMD) graphics card software. These are proprietary software, and one option is to use OpenCL.

12.1 CuPy and Numba cuda

My machine used to have Python 3.12 and CUDA 12, so CuPy installation could be `pip install cupy-cuda12x`. As usual, this failed. In autumn 2025, the system gcc is version 15, while Nvidia CUDA compiler `nvcc` support only compiler versions up to 11. A typical version mismatch. Gcc 11 is not in the repo of my linux distro, so I had to install gcc 11 from source. Installing gcc-11 compiler using gcc-15 begs for problems. I had to hack `simple-object-mach-o.c` and modify `set_32`, made an environment module gcc/11 and tried `pip install cupy --ignore-installed`. This downloads the CuPy source package (doesn't use precompiled wheel package) and compiles it using gcc-11 and nvcc. More problems: `math_functions.h` and `math_functions.hpp` under `/usr/local/cuda/` are not compatible with those in the system files (missing `noexcept(true)` in functions `sinpi()` and `cospi()` - more hacking! The package installs to `$HOME/.local/lib/python3.13/site-packages/cupy`. Fingers crossed! `Successfully installed cupy-13.6.0`

Let's play with CuPy and Numba, first CuPy. Start ***ipython*** REPL or use a Jupyter notebook,

```
>>> import cupy as cp
>>> import numpy as np
>>> cpu_array = np.random.randint(0, 255, size=(4000, 4000)) # CPU random array: numpy.ndarray
>>> gpu_array = cp.random.randint(0, 255, size=(4000, 4000)) # GPU random array: cupy.ndarray
```

You can move arrays between ***host*** (CPU) to ***device*** (GPU on you graphics card). For more examples, see cupy docs.

Also Numba has cuda capabilities! Numba is a large package, and in 2024 the cuda part was separated to a standalone package, `numba_cuda`, which lives in nvidia.github.io/numba-cuda/. Both cuda versions, the one bundled with Numba and the one in `numba_cuda`, are imported as `from numba import cuda`, but beware that only `numba_cuda` will be maintained in the future. To see which cuda you are using, try `python -c "from numba import cuda; print(cuda.__file__)"`; for

`numba_cuda` in Linux it could show something like
`.local/lib/python3.13/site-packages/numba_cuda/numba/cuda/__init__.py`

```
>>> from numba import cuda
>>> cuda.detect() # should show your GPU
>>> numba_cuda_array = cuda.to_device(cpu_array) # from host to device using Numba cuda
>>> type(numba_cuda_array) # numba.cuda.cudadrv.devicearray.DeviceNDArray
>>> numba_cuda_array.copy_to_host() # back to cpu
>>> cp.asarray(numba_cuda_array) # from host to device using cupy
```

See cupy_scipy_FFT.speed.py

cupy_scipy_FFT.speed.py

```
import numpy as np
import cupy as cp
from numba import cuda
from time import time
from scipy import fft
import cupyx.scipy.fft as cufft
import matplotlib.pyplot as plt

# warm up cupy compilation: try without these to see the difference
gpu_arr = cp.random.randint(0, 255, size=(10,10))
cufft.fftn(gpu_arr)

res = []
# max N in cupy is limited by GPU memory,
# too large N gives a cupy.cuda.memory.OutOfMemoryError

for N in range(1000,9000, 1000):
    cpu_arr = np.random.randint(0, 255, size=(N, N))
    gpu_arr = cp.asarray(cpu_arr)

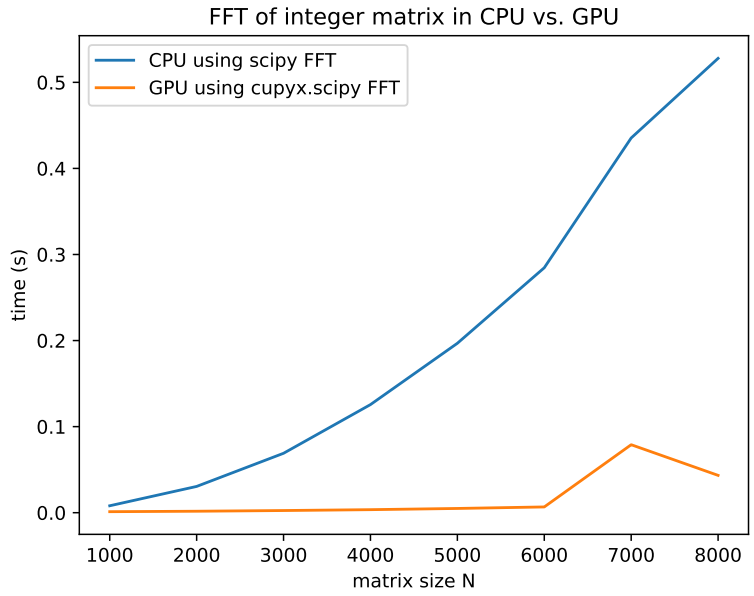
    t_cpu_start = time()
    fft_arr = fft.fftn(cpu_arr)
    t_cpu_end = time()

    t_gpu_start = time()
    fft_arr = cufft.fftn(gpu_arr)
    t_gpu_end = time()


    res.append([N, t_cpu_end-t_cpu_start, t_gpu_end-t_gpu_start])
    print(f'{res[-1]}, timing ratio = {res[-1][1]/res[-1][2]:8.4f}')


x, y1, y2 = zip(*res)
plt.plot(x, y1, label="CPU using scipy FFT")
plt.plot(x, y2, label="GPU using cupyx.scipy FFT")
plt.title('FFT of integer matrix in CPU vs. GPU')
plt.xlabel('matrix size N')
plt.ylabel('time (s)')
plt.legend()
plt.show()
```

```
In [2]: cuda.detect()
Found 1 CUDA devices
id 0    b'NVIDIA GeForce GTX 1070'
             Compute Capability: 6.1
                  PCI Device ID: 0
                     PCI Bus ID: 8
                           UUID: GPU-653bd2a2-9eef-fc1c-2202
                      Watchdog: Enabled
          FP32/FP64 Performance Ratio: 32
Summary:
 1/1 devices are supported
```



13 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 1.13.1 was released 25 September 2026.

How to get Julia:

- Homepage https://julialang.org
- Downloads https://julialang.org/downloads
- Source code https://github.com/JuliaLang/julia
- Packages https://juliapackages.com/

Contains:

- 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

For latest features, see Youtube videos from the yearly JuliaCon. The page A Deep Introduction to Julia for Data Science and Scientific Computing
contains a summary of syntax in MATLAB, Python and Julia,
MATLAB-Python-Julia cheatsheet.

13.1 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` arguments. In comparison, C/C++ can do optimization only at compile time, so do Numba and JAX. The strength of Julia is JIT, which compiles *and* optimizes code in run time.
- Easy to install - C++ needs the whole toolchain installed.
- Multiple dispatch (more about this later)
- Metaprogramming support (from Lisp, "code that writes code")
- LaTeX-style Greek letters: enhances readability; math code looks almost like math formulas.
- Built-in distributed computing: `using Distributed`.
- Extremely fast Static arrays: `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
```

13.2 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>`. Just like `ipython` strips away `>>>`.

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.[39]

13.3 Julia IDEs

13.3.1 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
```

[39] Interested in digging assembly codes produced by fortran, C or Julia? Take a look at the blog post by Hendrik Ranocha Optimizing Trixi.

13.3.2 Adding Julia to Jupyter notebook or Jupyter lab

Once you have installed Julia, install the `IJulia` package. From Julia console,

```
julia> using Pkg
julia> Pkg.add("IJulia")
```

and start the Jupyter Notebook on the command line,

```
$ jupyter notebook
```

13.3.3 Pluto

Pluto with the user interface `PlutoUI.jl` is an interactive Julia notebook running in a browser. You can bind a variable to a slider and see immediately how changes in that variable change all else that depends on it. The Live Docs (click a button to expand) show documentation of Julia objects under the cursor.

Install and start Pluto:

```
julia> using Pkg
julia> Pkg.add("Pluto")
julia> using Pluto; Pluto.run()
```

and move on to work on a Pluto browser tab. Most of the time you'd want to start with a line

```
using PlutoUI
```

see examples of how to create a slider, a button, and more: link to plutoui.jl. A Pluto notebook is divided into cells, much like a Jupyter notebook, and a cell is executed by clicking the cells play button or just by typing shift-enter. If the cell has a line `using Plots`, Pluto will, if needed, install and precompile the package `Plots`. If you want to see the progress (some packages are large and can take up to several minutes to install), click on the `Status` button to see an expanded view.

More importantly, the built-in Pluto package manager stores version information to the current **environment** so that the code is reproducible. Mind you, the Pluto environment you just filled with package `Plots` is not as such visible to Julia REPL,[40] you have to **activate** it to use it. The environment is the file `Project.toml`, and the path can be found using

```
Base.active_project()
```

In linux, the default environments are under `~/.julia/environments/v.11.1` (for Julia version 1.11.1), but Pluto env's are by default under the non-persistent directory `/tmp`. In Pluto, activate a persistent environment,

```
import Pkg; Pkg.activate("~/.julia/environments/my_pluto_notebook_env")
```

The Pluto environment focuses on managing package dependencies, the Julia interpreter itself is not stored.

Pluto notebook has autosave and the notebook can also be exported to pdf or html.

What I like most is that a Pluto notebook acts like a versatile spreadsheet, where changes propagate immediately in the notebook and recalculations are done if needed. I also like the fact that deleted cells behave as if they've never existed. Old tries and definitions don't haunt you. If you set `a=1`, execute the cell, and later delete the cell, the variable `a` is gone in that notebook, and an error message

```
UndefVarError: `a` not defined in `Main.var"workspace#155"`
```

appears wherever `a` is used, no matter if it was done before or after the deleted `a=1` cell.

[40] REPL stands for Read-Eval-Print Loop. It's the thing with `julia>` prompt, in Python the prompt is `>>>`.

13.4 Calling Julia from Python

The project pyjulia @github provides Julia calls from Python,

```
$ python -m pip install julia
```

or

```
$ conda 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
```

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.

13.5 Calling Python from Julia

Julia has the module `PyCall`, `?PyCall` shows usage examples,

```
using PyCall
so = pyimport(``scipy.optimize'')
so.newton(x -> cos(x) - x, 1)
# or
sc = pyimport(``scipy'')
sc.optimize.newton(x -> cos(x) - x, 1)
#
np = pyimport("numpy")
a = np.random.random((3,4))
#
# A macro exists for mimicking Python's "with statement". For example:
#
@pywith pybuiltin("open")("file.txt","w") as f begin
   f.write("hello")
end
```

The call `a = np.random.random((3,4))` creates first a NumPy row-major array (C-ordered), then converts is using PyCall to Julia column-major array (F-ordered); this involves copying the data.

For convenience, there is a module `SciPy` that does `PyCalls`,

```
using SciPy
SciPy.fft.fft([1, 2, 3, 4])
SciPy.integrate.quad(x -> x^2, 0, 1)
```

13.6 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}
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`, [41]

[41] Some type names have changed, the now 5-element VectorInt64 used to be called 5-element ArrayInt64,1.

```
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
```

13.7 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
julia> c=broadcast(*,a,b)    # 4x5x5 Array, same as broadcast(*,b,a)
julia> c=a.*b                # 4x5x5 Array, same as broadcast(*,a,b)
```

Notice 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.

13.8 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
```

13.9 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. If $\phi(x)$ is the many-body wave function, and you need the gradient $\boldsymbol{\nabla}\phi(x)$ at some x, you can compute it with AD,

```
function grad_phi(x ::MMatrix{dim,N,Float64}, phi ::Function) where {dim, N}
   grad = ForwardDiff.gradient(phi, x)
end
```

The code looks better in the editor, because you can actually use ϕ instead of `phi`. Of course, AD is a bit slower than computing and coding the gradient manually, but this works no matter what ϕ is!. I used mutable static matrix `MMatrix` to store the coordinates (see StaticArrays in 13.12). The return value is `grad`, because it's the last evaluated expression, so `return grad` is unnecessary.

If you don't want to return anything, do

```
function f(x)
    open(``output'','w') do file
        write(f,x)
    end
    nothing # nothing is returned
end
```

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,".") # add current directory to module search path
using 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
# 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()
```

13.10 Julia Differential Equations

Let's solve a set of differential equations, the Lorentz equations,

diffeq.jl

```
#http://docs.juliadiffeq.org/latest/tutorials/ode_example.html
#Example-2:-Solving-Systems-of-Equations-1

# activate local project
using Pkg
Pkg.activate(".")

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 = (0.0,100.0)
println("solving...")
prob = ODEProblem(lorenz,u0,tspan)
sol = solve(prob)

println("plotting...")
using Plots
#pyplot() # pyplot backend, looks nicer
s=plot(sol,idxs=(1,2,3),linewidth = 0.1)
display(s)
readline()
```

13.11 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.

13.12 Julia StaticArrays

See StaticArrays.jl. `StaticArrays` are stored in ***stack***, memory that is faster than ***heap***, where usual `Array`s are stored. `StaticArrays` will speed up small array calculations, up to about 100 elements long, where the benefit can be quite noticeable. I'm using StaticArrays a lot, mostly their modifiable versions `MMVector` and `MMatrix`.

```
using StaticArrays

v = @MVector zeros(10)   # macro creates a static vector that can be modified (M)
S = @MMatrix rand(10,10)  # same for a static matrix that can be modified (M)
```

Attempt to create too large static arrays will give `ERROR: LoadError: syntax: expression too large`. Run these in REPL or in Jupyter notebook:

```
using StaticArrays, BenchmarkTools

# Create a small 3x3 StaticMatrix and regular Array
A_static = @SMatrix rand(3, 3)
B_static = @SMatrix rand(3, 3)
A_regular = rand(3, 3)
B_regular = rand(3, 3)

# Benchmark StaticArrays multiplication
@benchmark $A_static * $B_static

# Benchmark regular Array multiplication
@benchmark $A_regular * $B_regular

# Create a small 3x3 SMatrix (immutable) and MMatrix (mutable)
```

```
A_smatrix = @SMatrix rand(3, 3)
B_smatrix = @SMatrix rand(3, 3)

A_mmatrix = @MMatrix rand(3, 3)
B_mmatrix = @MMatrix rand(3, 3)

# Benchmark SMatrix multiplication
println("SMatrix multiplication:")
@benchmark $A_smatrix * $B_smatrix

# Benchmark MMatrix multiplication
println("MMatrix multiplication:")
@benchmark $A_mmatrix * $B_mmatrix
```

One run showed these medians:

regular	25 ns
`MMatrix`	9.4 ns
`SMatrix`	3.7 ns

Notice the macro `@benchmark` and the dollar signs in `$A_mmatrix`. Macros such as `@benchmark` evaluate the entire expression, and using the $ operator tells the macro to use the value directly rather than treating it as a variable name. Otherwise the macro might not evaluate the variable properly, leading to unexpected results. The $ operator is meant for ***interpolation*** within expressions, it tells Julia to replace the variable with its value in contexts where expressions are being constructed.

13.13 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.

13.14 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 160 methods (depends on installed packages), all specialized to adding different kinds of data types. That many, because addition makes sense for many types of data. For example, adding matrices could be

```
+(H::LinearAlgebra.UpperHessenberg, x::LinearAlgebra.Bidiagonal)
   @ LinearAlgebra /usr/share/julia/stdlib/v1.11/LinearAlgebra/src/hessenberg.jl:120
```

Julia does JIT compilation in run-time, and, as Stefan Karpinsky points out, a code like this is possible:

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(d::A) = f(d)  # Julia: d is of type A *or* any subtype of A.
# d can be any of type A, B or C, because it's decided in runtime

println(g(B())) # decide in runtime that the argument has type B
println(g(C())) # decide in runtime that the argument has type C

# output:
# it's a B
# it's a C

# In static function overloading g(d) would always give "it's an A",
# because the decision is made at compile time,
# and you'd need to define what g(d::B) and g(d::C) do.
# => less generic code
```

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

C++ has ***compile-time dispatch based on function overload resolution***[a]. For example,

compile_time_dispatch.cpp

```cpp
// C++ compile-time dispatch
#include <iostream>

struct Circle { double r; };
struct Square { double a; };

void intersect_area(const Circle& c1, const Circle& c2) { std::cout << "Circle-Circle intersection\n"; }
void intersect_area(const Circle& c1, const Square& s1) { std::cout << "Circle-Square intersection\n"; }
void intersect_area(const Square& s1, const Circle& c1) { std::cout << "Square-Circle intersection\n"; }
void intersect_area(const Square& s1, const Square& s2) { std::cout << "Square-Square intersection\n"; }

int main() {
    Circle c{1.0};
    Square s{2.0};

    intersect_area(c, c);
    intersect_area(c, s);
    intersect_area(s, c);
    intersect_area(s, s);
}
```

The crucial point here is the `const` in the arguments.

[a]For run-time decisions of what method to use you need `std::virtual` or `std::variant` and `std::visit`.

In Julia, the same task is done by

multiple_dispatch_example.jl

```
struct Circle; r::Float64; end
struct Square; a::Float64; end

function intersect_area(s1::Circle, s2::Circle); println("Circle-Circle intersection"); end
function intersect_area(s1::Circle, s2::Square); println("Circle-Square intersection"); end
function intersect_area(s1::Square, s2::Circle); println("Square-Circle intersection"); end
function intersect_area(s1::Square, s2::Square); println("Square-Square intersection"); end

c = Circle(1.0)
s = Square(2.0)

intersect_area(c, c)  # Circle-Circle
intersect_area(c, s)  # Circle-Square
intersect_area(s, c)  # Square-Circle
intersect_area(s, s)  # Square-Square
```

Use concrete types, avoid `Any` and avoid boxing arguments

Julia compiler can (usually) optimize code if a variable has a ***concrete type***. If Julia can't establish the argument type it resorts to `Any` type, and the type has to be inferred in runtime, which is slow. The macro `@code_warntype` highlights all types `Any` in red, and for a good reason.

Boxing arguments hides the concrete types and leads to slow code. See Julia performance tips:

capture_and_boxing.jl

```
# Poor performance using closure
# https://docs.julialang.org/en/v1/manual/performance-tips/#man-performance-captured
#
using InteractiveUtils

function abmult(r::Int64)
    if r < 0
        r = -r
    end
    f = x -> x * r   # closure, captures r
    return f
end

@code_warntype abmult(1)
```

The code emits `r@_7::Union{Int64, Core.Box}` in red, because the compiler can't be sure that `r` is always an `Int64` (it thinks type may change in the closure). As a result, `r` is `Core.Box`ed.

One can also deliberately obscure the type using `Any`,

runtime_vs_static_dispatch_simple.jl

```
using InteractiveUtils

println("-"^5*"static dispatch, one specialization"*"-"^5)
f(x) = x + 1
@code_warntype f(5) # no Any
println(@code_typed f(5)) # short output
@show f(5)

println("-"^5*"dynamic dispatch"*"-"^5)
f(x) = x + 1

function g(y)
    v = Any[y]           # Boxing y to Any erases type information
    return f(v[1])       # compiler doesn't know the concrete type
end
@code_warntype g(5) # many red Anys
println(@code_typed g(5)) # very long output
@show g(5)
```

`Any[y]` prevents storing `y` simply as an integer to a CPU register, and forces it to be allocated as a small object on the heap. the garbage collector works hard to clean up these heap objects. With `Any`, dynamic dispatch is forced to do a runtime lookup of the type inside the box.

Take-home message:

If `@warn_type` shows anything in red, you may loose performance.

FFT

In C, methods are chosen based on their name, hence GSL (Gnu Scientific Library) has FFT methods

```
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 need distinct *function names* for all different FFTs. In C++, you could, for example, overload `fft_forward()` and `fft_backward()`. You can try out Julia FFT in REPL,

```
julia> using FFTW # Julia uses FFTW backend for FFT
julia> [1.0, 2.0, 3.0, 4.0] |> fft              # real FFT
julia> ComplexF64[1.0, 2.0, 3.0, 4.0] |> fft    # complex FFT
julia> B = plan_fft([1.0, 2.0, 3.0, 4.0])  # create plan B
julia> x = [1.0, 2.0, 3.0, 4.0]
julia> B*x # use plan B for faster FFT
julia> x = [5.0, 6.0, 7.0, 8.0]
julia> B*x # use plan B for faster FFT
```

The pipe operator expression `a |> f` means `f(a)`. You can pipe multiple function calls, For example

```
julia> sum(abs(fft([1.0, 2.0, 3.0, 4.0])))
```

can be written as

```
julia> [1.0, 2.0, 3.0, 4.0] |> fft |> abs |> sum
```

The method `ifft()` computes the inverse FFT,

```
julia> using FFTW
julia> Fa = [1.0, 2.0, 3.0, 4.0] |> fft  # forward FFT
julia> Fa |> ifft                        # inverse FFT
```

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++, 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')
```

After a fashion, we solved the expression problem. However, 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()`! This process is affectionately called *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. [42] Visitor pattern is double dispatch, the method to execute depents on what object defines the

[42] In Python, you can inject a method to an existing class.

visitor and on the object that calls the visitor. Another option is to make a generic external function using templates, without attaching it to a specific class.

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)
```

What can Julia multiple dispatch do in science?
For example, measured data has error bars,

$$x = 1.45 \pm 0.01 \quad \text{new data type ,}$$

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.

Is multiple dispatch = vectorization + function overloading ? No, you get a lot more:

- Fine-grained control over method selection based on the types of all arguments.
- Dynamic behavior that adapts to input types.
- Easier maintenance and extensibility of code.
- Optimized performance through specialized implementations

Julia shines in code reusability, there's less need for boilerplate code.

14 Appendix: Python basics

Python was invented by Guido Van Rossum (1956-) in December 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.

14.1 Python package managers and IDEs

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

Wheel packages

The package standard is called ***wheel***; wheels have the suffix `.whl`. Wheels have been compiled for common user cases, indicated in the name: language and version tag (e.g. 'py27', 'py2', 'py3') abi tag (e.g. 'cp33m', 'abi3', 'none'), and platform tag (e.g. 'linux_x86_64', 'any'). Usually wheel installation is automatic, fast, and reliable, without any need for a working C/C++ compiler. If a wheel doesn't exist, installation takes the slower `setup.py` route and needs a compiler. Some wheels are compiled with GPU support (NVidia Cuda support is indicates as `cuda` in the wheel name), which makes a huge difference in machine learning.

14.2 Python Ecosystem Overview

- Python ecosystem
  - Package managers: pip, conda
  - Distributions: Anaconda, Miniconda

  - Environments: venv, virtualenv, conda env
  - Interface: Jupyter (uses kernels tied to environments)

pip installs Python packages from PyPI, works also inside virtual environments

conda manages environments and packages (including non-Python libs)

Anaconda a full distribution (Python + conda + many scientific packages)

virtualenv / venv lightweight Python-only environments

Jupyter interface; runs code via kernels tied to environments

14.2.1 `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,

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

For example,

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

or

```
$ pip install numpy scipy matplotlib
```

Avoid installing Python packages as root/admin.

Having multiple installations of packages causes version mismatches.

14.2.2 Where does Python search for modules?

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

You either haven't installed the module `nnn`, or it's not in the Python search path. The search path can be printed with the command

```
$ python -m site
```

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 often leads to confusion.

How to exclude user-installed packages
One way is to define an environment variable,

```
$ export PYTHONNOUSERSITE=1
```

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

14.2.3 Python virtual environments

A Python virtual environment keeps a project detached from outside world and makes sure the code uses the correct module versions.

Basic Python virtual environment
The venv Python environment is create with the command

```
$ python -m venv <path>
```

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 3.14.3 ...
$ deactivate
```

The virtual environment can't see the packages installed outside the sandbox.

> Warning: The virtual environment turns unusable after Python version change! Use `pip freeze` before upgrade!

`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.

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.
>>>
```

Removing an environment is done with

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

14.2.4 VS Code, Spyder, iPython, Jupyter Notebook, and Jupyter Lab

VS Code (Visual Studio Code) is a free cross-platform IDE with syntax highlighting, git support, IntelliSense automatic filling, AI assistance, and Jupyter notebook support. **Spyder** is an open-source cross-platform IDE, with editing, shell, graphics output, and online help on the same platform. **iPython** is the "official" interactive command line Python interpreter, described at length in discussion @Stackoverflow .

Conda virtual environment and Jupyter notebooks inside VSCode is a very effective environment. You can get detailed installation instructions (pdf, written for another course) here.

Many use iPython with the **Jupyter notebook**, installation is

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

Basic iPython has syntax highlighting and suggests line completion. Jupyter notebook starts on your web browser,[43]

```
$ jupyter notebook
```

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,

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

```
$ 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 (file suffix `.ipynb`) can be converted to Python scripts using

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

One reason you might convert a notebook to a standalone script is that you're done with playing with ideas and want to embed your code as a module to a larger codebase. The output `your_notebook.py` can be run using `ipython`. A notebook can be converted to html,

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

14.3 Updating Python packages using `pip` package manager

A single package update as non-root user:

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

or

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

How about updating *all* packages? One possibility is (you may need to add `--user`)

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

Another way: 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
```

14.4 Arbitrary precision calculations

14.4.1 Large integers

Python can handle arbitrarily large integers.

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)
```

14.4.2 Long floats with `mpmath`

An arbitrary precision floating point arithmetics is provided by mpmath .

Example: Riemann hypothesis

All nontrivial zeros of $\zeta(s)$ are on the critical line $s = \frac{1}{2} + i\,t$.

The trivial zeros of the Riemann zeta function are real, $\zeta(s) = 0$ for $s = -2, -4, -6, \ldots$.

`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) # 1.233700550136169749038117515738122165203094482421875
# more precision:
#mp.prec = 120 # set number of bits
mp.dps=100    # set number of decimal points
print(mp)
# calculate with better precision:
res=mp.quad(f, [0, 1], [0, 1])
mp.nprint(res,100)
#1.233700550136169827354311374984518891914212425
#   905098828301668672027505602802400655375278389267013232
mp.nprint(mp.pi**2/8,100)
#1.233700550136169827354311374984518891914212425
#   905098828301668672027505602802400655375221675464819029
```

The default double-precision accuracy has `mp.dps = 15` and `mp.prec = 53`.

I used `mpmath` in evaluating the partition function of three fermions in a harmonic trap,

$$Z_3(\beta) = \frac{1}{3}\Big((e^{\frac{1}{2}\hbar\omega\beta} - e^{-\frac{1}{2}\hbar\omega\beta})^{-9} - 3(e^{\frac{1}{2}\hbar\omega(2\beta)} - e^{-\frac{1}{2}\hbar\omega(2\beta)})^{-3}(e^{\frac{1}{2}\hbar\omega\beta} - e^{-\frac{1}{2}\hbar\omega\beta})^{-3} + 2(e^{\frac{1}{2}\hbar\omega(3\beta)} - e^{-\frac{1}{2}\hbar\omega(3\beta)})^{-3}\Big)\ , \tag{110}$$

where $\beta = 1/(k_B T)$ is the inverse temperature. The terms have huge cancellation! Using $\hbar\omega = 1$ and $\beta = 1/0.001$,

$$Z_3 \approx 1.576\,559 \times 10^{-1955} - 4.729\,676\,6 \times 10^{-1955} + 3.153\,118 \times 10^{-1955} \approx 7.311\,709 \times 10^{-2823}\ . \tag{111}$$

The smallness of the numbers is not a problem because one needs only $\ln(Z_3(\beta))$, the problem is the small signal-to-noise ratio. Notice that $1.576559 - 4.7296766 + 3.153118 = 0.0000004$, so the few digits shown above are inadequate to get any meaningfull result. To get the result $7.311\,709 \times 10^{-2823}$ I had to use `mp.dps`=900 (`mp.prec` = 2993), because more than 800 decimals cancel exactly.

14.5 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, because in Python `for` doesn't start a new scope. Meanwhile, list comprehension has a local scope,

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

You may 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 Python lambda function (an unnamed function, defined on the spot),

```
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(); filter() is <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
```

14.6 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
s = ''.join(random.choices(string.ascii_lowercase, k=N))
print(s)
```

14.7 Advanced unpacking

14.7.1 Unpacking of 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):
   print(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):
   print(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. Keyword arguments must follow the positional ones: `f(1,z=4,3)` is wrong.

- ***default arguments*** are set in function defition,

```
def f(x,y,z=4):
   print(x,y,z)
f(1,2)   # calls f with x=1, y=2, and z=4 (default z)
f(1,2,3) # calls f with x=1, y=2, and z=3 (uses given z)
```

14.7.2 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.

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 `fib1()` 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
# question: test = 2, so why does test if 2 succeed?
```

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]
```

14.8 Decorators

Decorators are wrappers that can be used to add extra functionality to a function, without modifying the body of the function. This is what makes decorators so good: the API is not changed. You don't want to write an address on a gift, instead you wrap the gift and write the address on the wrapper. The following relies on the fact that function arguments are either positional arguments and/or keyword arguments or none at all, so all possible arguments can be passed without knowing them. *Any* function can be wrapped like this:

```
def universal_decorator(any_function):
    def wrapper(*args, **kwargs):
       #... add extra functionality here ...
       return any_function(*args, **kwargs)
    return wrapper
```

This needs editing to actually do more than just 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

```
from time import process_time as T

if __debug__:
    def mydecorator(any_f):
        def wrap(*args, **kwargs):
            tic = T()
            res = any_f(*args, **kwargs)
            toc = T()
            print(f'timing: {toc-tic:<.6e}')
            return res
        return wrap
else:
    def mydecorator(any_f):
        return any_f


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


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

Can a decorator have arguments? Absolutely! We'll return to them. Nothing prevents re-decorating a decorated function.

14.8.1 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 to turn on `trackcalls`. 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 some time to interpret, and that a decorated function can't be undecorated.

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

I haven't seen this approach suggested elsewhere, but someone must be using this approach. 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.

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, the next example will be executed in the order `deco1` calling `deco2` calling `func`.

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

14.8.3 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` ... .
The outmost function `requires_permission(sPermission)` is a ***decorator factory***, because it returns a different decorator based on the argument `sPermission`. A working example is in file `decorator_with_arguments.py`.

14.9 Iterables, generators and `yield`

If you can write a loop `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 ***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. The `for` loop asks the generator to generate elements, in this case all of them.

A generator `yield`s 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))  # return a generator expression


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.

The generator functions body won't execute until you ask it to.

The generator `yield`s after receiving a request `next()`, which is also how `list` asks a generator to `yield` values. Also `for`-loops use `next()` to iterate.

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 that can be passed around.

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***. Each `yield` has only one number to yield, so successive `next()`s move down the `yield`s until there are none left.

Example: 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` is a pure Python implementation.

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.

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

```
$ watch filename
```

will show the contents, updated once in a second.

14.9.1 Generator pipelines

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

Generator pipelines are similar to `bash` shell commands, such as

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

where | pipes the output to the next command for further manipulation.

Example: Read data from file and process it

The data file has on each column an integer followed by two floating point numbers. The task is to find all even index y-values, add them together and print the sum. We write each sub-task as a generator function and pipeline them:[44]

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")

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

[44] Warning: this uses the legacy `np.random.random()`, see section 5.9.

- 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.
- 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.
- In debugging generators, be aware that ***asking a generator to `yield` intermediate values breaks the generator pipeline*** - a leaking pipeline. For testing, you may 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.

14.10 Underscores

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 values,

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

  Does the same as `first,last = a[0],a[-1]` .

- In the REPL (Read-Eval-Print Loop) 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.

14.10.1 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 method  _bar1() \n")
    def __bar2():
        print("this is class Foo method __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.

14.10.2 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(f'Initializing to x = {value}')
        self.x = value # This x is an instance variable
    def do_things(self, y):
        print(f'doing things with x = {self.x} and y = {y}')



if __name__=='__main__':
    # instantiate an instance of class MyClass
    my = MyClass(10)      # this calls __init__ , Initializing to x = 10
    my.do_things(20)      # doing things with x = 10 and y = 20
    another_my = MyClass(1)    # this calls __init__ and sets another_my.x = 1
    another_my.__init__(2)     # this calls __init__ directly, and sets another_my.x = 2
    another_my.do_things(20)   # another_my.x = 2 and y = 20
    my.do_things(20)           # my.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 a class may be, the bread-and-butter operations on class instances should be simple and plausible. Take a look at addition and comparisonm of two instances of `Vector2D`:

vector_class_prelim.py

```python
# 2D vector with addition, subtraction and length comparison
# A BAD implementation
from dataclasses import dataclass

@dataclass
class Vector2D:
    x : float
    y : float
    def add(self, other):
        if not isinstance(other, Vector2D):
            return NotImplemented
        else:
            return Vector2D(self.x + other.x, self.y + other.y)
    def sub(self, other):
        if not isinstance(other, Vector2D):
            return NotImplemented
        else:
            return Vector2D(self.x - other.x, self.y - other.y)
    def gt(self, other):
        # comparison basd on length
        return self.x**2 + self.y**2 > other.x**2 + other.y**2
    def repr(self):
        print('called method repr()') # just to see when the method is called
        return f"Vector2D({self.x}, {self.y})"


if __name__=='__main__':
    v1 = Vector2D(1, 2)
    v2 = Vector2D(3, 4)
    v3 = Vector2D(5, 6)
    v4 = Vector2D(7, 8)
    print(v1.repr())
    print(v2.repr())
    print(v3.repr())
    print(v4.repr())

    print(f'{v1 = }\n{v2 = }\n{v3 = }\n{v4 = }')
    print(f'v1 + v2 + v3 + v4 = {v1.add(v2).add(v3).add(v4)}')
    print(f'v1 - v2 + v3 - v4 = {v1.sub(v2).add(v3).sub(v4)}')
    print(f'v1 longer than v2 ? {v1.gt(v2)}')
    print(f'v3 shorter than v4 ? {not v3.gt(v4)}')
```

I'm not complaining about `Vector2D` *per se*, since I won't often look at it. But computations such as

```
v1.add(v2).add(v3).add(v4)
```

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

```
v1 + v2 + v3 + v4
```

and this is where magic methods come into play.

vector_class.py

```python
# 2D vector with addition, subtraction and length comparison
from dataclasses import dataclass

@dataclass
class Vector2D:
    x : float
    y : float
    def __add__(self, other):
        try:
            return Vector2D(self.x + other.x, self.y + other.y)
        except:
            return NotImplemented
    def __sub__(self, other):
        try:
            return Vector2D(self.x - other.x, self.y - other.y)
        except:
            return NotImplemented
    def __gt__(self, other):
        # comparison basd on length
        return self.x**2 + self.y**2 > other.x**2 + other.y**2
    def __repr__(self):
        return f"Vector2D({self.x}, {self.y})"


if __name__=='__main__':

    v1 = Vector2D(1, 2)
    v2 = Vector2D(3, 4)
    v3 = Vector2D(5, 6)
    v4 = Vector2D(7, 8)
    print(f'{v1 = }\n{v2 = }\n{v3 = }\n{v4 = }')
    print(f'{v1 + v2 + v3 + v4 = }')
    print(f'{v1 - v2 + v3 - v4 = }')
    print(f'v1 longer than v2 ? {v1>v2}')
    print(f'v3 shorter than v4 ? {v3<v4}')

    # make a deliberate error:
    vx = (1,3)
    print(f'{vx = }')
    print(f'{v1 + vx =}')
```

The magic method `__add__` tells how the + operator acts on `Vector2D`s. 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***; the same idea is implemented in C++ (section **??**) and can be done in most languages.

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.[45] `__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 may have no idea how an instance should be printed, and that's where `__str__` or `__repr__` can lead the way. 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 [46]. The `Vector2D` class and most builtin classes pass the test. The canonical example is `datetime`,

```
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
```

[46] Without a `__eq__` method 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.

[46] See Discussion @Stackoverflow.

Context managers

A context manager typically makes sure that you properly finish what you've started. 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***,

```
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 [47]

[47] 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 = }')
            print(f'{exception_value = }')
            print(f'{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')

if __name__=='__main__':
    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
```

Notice how `__init__` and `__enter__` are called immediately when the context starts. The method `__exit__` can handle exceptions, such as running `th.fooling_around()`: [48]

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.

14.11 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 **??**), but they can switch between tasks really fast.

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

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

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 -> find_first_sequence -> find_second_sequence

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

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

14.12 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. For example,

```
def generator_A:
    yield from generator_B
```

Whatever comes from `generator_B` is transparently copied to `yield`. If you consider that a generator may return anything (results, exceptions, StopIteration, *etc.*), you realize that there are a great number of possibilities to cover: miraculously, `yield from` can handle anything! Obviously, `yield from` saves lot's of coding, and makes splitting generators to subgenerators an easy task, similar to splitting functions to subfunctions.

Consider the following codes (the output is 0 1 2 3 4):

yield_from_longway.py

```
def generator_B():
    for x in [1, 2, 3]:
        yield x

def generator_A():
    yield 0
    for value in generator_B():
        yield value
    yield 4

if __name__=='__main__':
    for val in generator_A():
        print(val)
```

yield_from_shortway.py

```
def generator_B():
    yield from [1, 2, 3]

def generator_A():
    yield 0
    yield from generator_B()
    yield 4

if __name__=='__main__':
    for val in generator_A():
        print(val)
```

Moreover, `yield from` is a two-way door, you may receive data but also send data; in this respect the word "from" in `yield from` is a bit odd. Consider accumulating a sum like this:

yield_from_both_ways.py

```
# yield from both sending and receiving data
def compute_sum():
    """Computes sum of numbers sent to it."""
    total = 0
    while True:
        value = yield total   # yield current average and wait for new value
        if value is None:
            break
        total += value

def accumulate_sum():
    res = yield from compute_sum()
    if res==None:
        pass
    else:
        print(f'accumulate_sum output {res = }')

if __name__ == '__main__':
    gen_sum = accumulate_sum()
    print(next(gen_sum)) # start generator
    for x in [1.0, 2.0, 10.0]:
        print(f'send {x} sum is {gen_sum.send(x)}')

    try:
        gen_sum.send(None)
    except StopIteration:
        pass
```

14.13 Pattern matching with `match-case` - 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,[49]

```
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.

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

Everything was fine, until someone noticed that `match-case` may sometimes rebind 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 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 like in the example, the cat turns to a zombie, both dead and alive. The problem is that the line `case CAT_DEAD:` treats `CAT_DEAD` as a non-existing variable, so it binds it to `CAT_ALIVE`. For many programmers this may come as a surprise. Preventing an unwanted rebind of `CAT_DEAD` can be done by replacing the line `case CAT_DEAD:` with `case _ if _==CAT_DEAD:`.

14.14 Python `class`

14.15 `@classmethod` and `@staticmethod` decorators

I just give an example:

class_decorators.py

```python
class DecoDemo:
    creation_date = '10.9.2025'
    def __init__(self, arr="None"):
        self._arr = arr

    @property
    def arr(self):
        return self._arr

    def instance_method(self):
        return ("instance_method", self)

    @classmethod
    def class_method(cls):
        return ("class_method", cls)

    @staticmethod
    def static_method():
        return ("static_method", None)

    @classmethod
    def demo_factory(cls):
        return cls([1,2,3,4])

    @classmethod
    def classmethod_creation_date(cls):
        return cls.creation_date

    @staticmethod
    def staticmethod_creation_date():
        return DecoDemo.creation_date


if __name__=='__main__':
    print('testing instance methods:')
    ins = DecoDemo()
    print(f'{ins.instance_method() = }')
    print(f'{ins.class_method() = }')
    print(f'{ins.static_method() = }')

    ins2 = DecoDemo([0,2,4,6])
    print(f'{ins2.arr = }')
    #ins2.arr = [1,3,5,7] # AttributeError: property 'arr' of 'DecoDemo' object has no setter
    print(f'{ins2.arr = }')

    print('testing class methods:')
    ins = None
    ins2 = None
    try:
        print(DecoDemo.instance_method())
    except:
        print("failed to call DecoDemo.instance_method()")

    print(f'{DecoDemo.class_method() = }')
    print(f'{DecoDemo.static_method() = }')
```

14.15.1 @dataclass decorator

The following class has no __init__():

pointclass.py

```
from dataclasses import dataclass

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

point1 = Point(1.1, 2.2, True)
point2 = Point(1, 4)
print(point1)
print(point2)
#Point(x=1.1, y=2.2, outlier=True)
#Point(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.

14.15.2 `@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__`

Example: Suppose a square side is L, but once set, you are not allowed to change L of an existing square. Also the area is not allowed to be larger than 10.0. Notice how the area is accessed as if it was an attribute `.area`, although it's actually computed.

square_property.py

```
class Square:
    def __init__(self, L):
        self._L = L # note the underscore
        A = self.area
        if A > 10.0:
            raise ValueError(f'Area {A} above limit 10.0')

    # getter, defines self.L
    @property
    def L(self):
        return self._L

    @property
    def area(self):
        return self.L**2

if __name__=='__main__':
    small_sq = Square(0.5)
    print(f'small square: L = { small_sq.L} area = {small_sq.area}')

    # you are not allowed to change L after init
    small_sq.L = 10.0 # raises AttributeError
    # you are not allowed to define squares with area > 10.0
    big_sq = Square(50.0) # raises ValueError
    print(f'  big square: L = { big_sq.L} area = {big_sq.area}')
```

The underscore in `self._L` hints that the attribute is not supposed to be accessed directly. Comment out the line `small_sq.L=10.0` to test the area limit.

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 modified using `object._pbc = ...`.
In `__init__`, it's initialized to the default value `array([False, False, False])`. To change the value you need to use a `setter()` method, and to read the value you need to use a `getter()` method.

The getter and the setter are defined as (NOTE: are my comments)

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

```
#NOTE: define a setter for pbc, without this pbc would be read-only property
@pbc.setter
def pbc(self, pbc):
    self._pbc[:] = pbc  # NOTE: in_place update

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`.

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("The 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.

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 tell them apart. The above code actually means

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

14.15.3 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` has a method whose implementation doesn't meet your needs.[50]

> Do not edit the library code, because your edits will be overwritten in a future 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:

```
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.

14.15.4 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

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

```
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)`.

14.15.5 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)
```

14.15.6 Simulation and Measurements in Python

Suppose you 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):
          raise ValueError(f'{measurement.name} has no callable method _measure')

      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 returns 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, so `super().__init__()` calls `MeasurementBase.__init__()`, and sets up the list `measurements` and the counter `steps`.

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 iter in range(5):
       print(u'\u2500'*80)  # thick horizontal line
       print(f'iter {iter})
       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 example is in measurement_example.py. It "measures" random numbers, and computes their average and lower and upper limits.

14.15.7 Saving and reading HDF5 data

HDF5 is a commonly used binary format of simulation data.

```
$ pip install h5py
```

hdf5_write_example.py

```
import h5py
import numpy as np

# Create some simulation results
ave = np.random.random((5,5))

# Write data in HDF5 binary format
with h5py.File("output.h5", "w") as f:
    grp = f.create_group("simulation1")
    grp.create_dataset("average", data=ave)
```

hdf5_read_example.py

```
import h5py
import numpy as np

# Read data from HDF5 file
with h5py.File("output.h5", "r") as f:
    try:
        data = f["simulation1/average"][:]   # NumPy array
        print("data is ")
        print(data)
    except:
        print("file has no dataset simulation1/average")
```

In linux, you can view the contents of `output.h5` using

```
$ h5dump output.h5
```

The dumped contents of the HDF5 file `output.h5` is

```
HDF5 "output.h5" {
GROUP "/" {
   GROUP "simulation1" {
      DATASET "average" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 5, 5 ) / ( 5, 5 ) }
         DATA {
```

```
        (0,0): 0.864017, 0.973946, 0.470324, 0.0210522, 0.508029,
        (1,0): 0.335475, 0.0320508, 0.800733, 0.873321, 0.136637,
        (2,0): 0.984769, 0.26255, 0.14579, 0.731064, 0.413553,
        (3,0): 0.0803955, 0.103341, 0.0810394, 0.322087, 0.00538296,
        (4,0): 0.390216, 0.232611, 0.695463, 0.645668, 0.0259468
        }
      }
   }
}
}
```

A HDF5 file usually contains metadata, such as simulation parameters and date.

14.15.8 Python Serialization

Serialization means converting an object to a byte stream, a form which can be stored or transfered. Serialization can be used to store data between calculations. 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 called ***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()
```

15 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. If you need both, try polars.

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 .

16 Polars

`polars` has superseced `pandas` in HPC: `polars` can be as much as 30x-50x faster. Critical parts are written in Rust to ensure fast parallel execution in multithreaded environment. The syntax is slightly different to `pandas`, but familiar dataframes are still there. See examples in getting started..