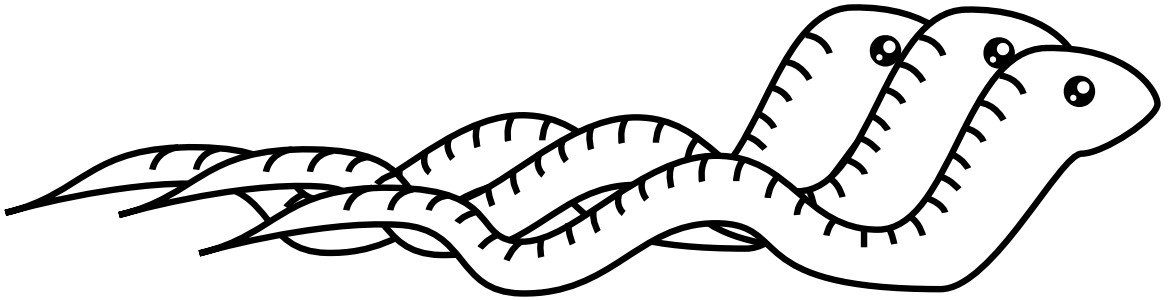
Python 3 for scientific computing

Lecture 7, 21.3.2018
Parallel computing; floating point; more libraries

Juha Jeronen
juha.jeronen@tut.fi



Spring 2018, TUT, Tampere
RAK-19006 Various Topics of Civil Engineering

TAMPERE
UNIVERSITY OF
TECHNOLOGY

Meta

Updated battle plan

- Starting the second (and final) part of the course: introduction to a variety of advanced topics.
  - **Lecture 7**: Parallel programming; behavior of floating point numbers
    - Also one slide on more libraries.
  - Lecture 8: High-performance computing in Python
    - At least Cython; possibly examples of Numexpr, Numba.
  - Lecture 9: Introduction to software engineering
  - Lecture 10: Gentle introduction to functional programming with Python
  - Lecture 11: Beyond Python
- Today, we will proceed as far as we have time; the rest, next week.
- The slide set on Cython will be shorter, completing next week's lecture.

Meta

About the final assignment

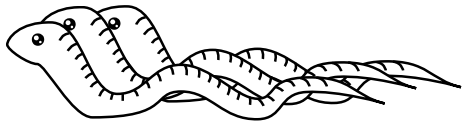
- Commented code is enough; no separate report.
  - At the beginning, include a short description of what your program does, and which algorithms it uses (if relevant).
  - If possible, provide web links (in comments or docstrings) regarding the algorithms, so that it's possible for a reader to quickly find out about them.

- The assignment can also be just a part for a larger project you're working on.
  - It doesn't have to be fully complete, as long as it runs, and does something sensible.
  - Just cut the project when some subset of functionality is done, or when the program is large enough to submit as a final assignment (≈100–1000 SLOC, depending on your background and the information density in those lines of code)

- Topic ideas, beside classical numerics:
  - Some complex data visualization?
  - Pick some interesting library you haven't yet used, figure out a use for it, and try it out.
  - Fractals – find a nice-looking one, and write a (parallel?) program to explore it.
  - Audio processor? Image processor? GUI app? Web app? Database app?
  - Something you eventually intend to deploy on your Raspberry Pi, if you have one?
  - Something specific that helps your studies and/or work?

- Really, the project can be just about anything, as long as it:
  - is implemented in Python (Cython is also ok),
  - uses something you have learned on this course,
  - is about the right size for a 5-credit course.

- Unsure about topic suitability or the size of your proposed project? Just ask.



Parallel computing

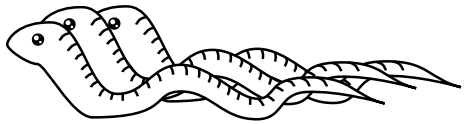
- Instruction-level parallelism in a single processing unit (core):
  - *Pipelining:* simultaneous execution of multiple instructions per clock cycle, in the *same* execution unit (such as ALU) on the processor, by dividing the unit to multiple simultaneously operating phases (steps). Often includes branch prediction.
  - *Out-of-order execution,* by the availability of input data and execution units rather than the original order in the program, taking into account possible dependencies.
  - *Superscalarity*: simultaneous dispatch of multiple instructions from one instruction stream (such as process or thread) per clock cycle, to *different* execution units.
  - *Simultaneous multithreading* (*SMT*; e.g. Intel's Hyper-threading): simultaneous dispatch of multiple instructions from *multiple* instruction streams per clock cycle.

- *Multi-core processor* ← Nowadays everywhere.

- *Symmetric multiprocessor* (*SMP*) ← A classical solution for servers, e.g. Intel Xeon.
  - System with multiple identical processors, shared memory, connected via a bus.

- *Distributed computing* (contrast *cloud computing*: comp. resources as a service)
  - Processing elements connected via a network.
  - *Cluster*: a group of computers that work together closely. Same task in each node.
  - *Massively parallel processor*: a single computer with many networked processors.
  - *Grid computing*: computers linked over the internet. Nodes may do different tasks.
  - *Volunteer computing*, e.g. Seti@Home, BOINC. Somewhat different from grid.

- GPGPU, heterogeneous computing (e.g. OpenCL), NUMA, …

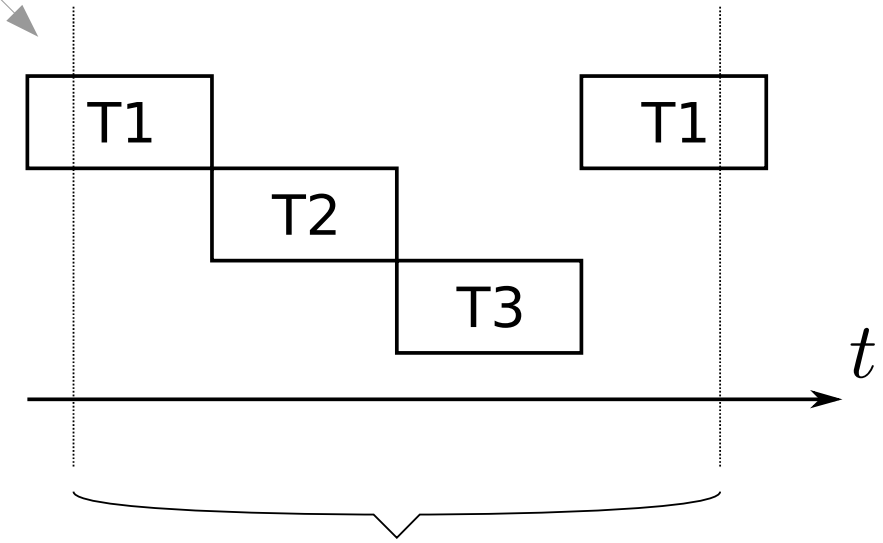


Parallel computing

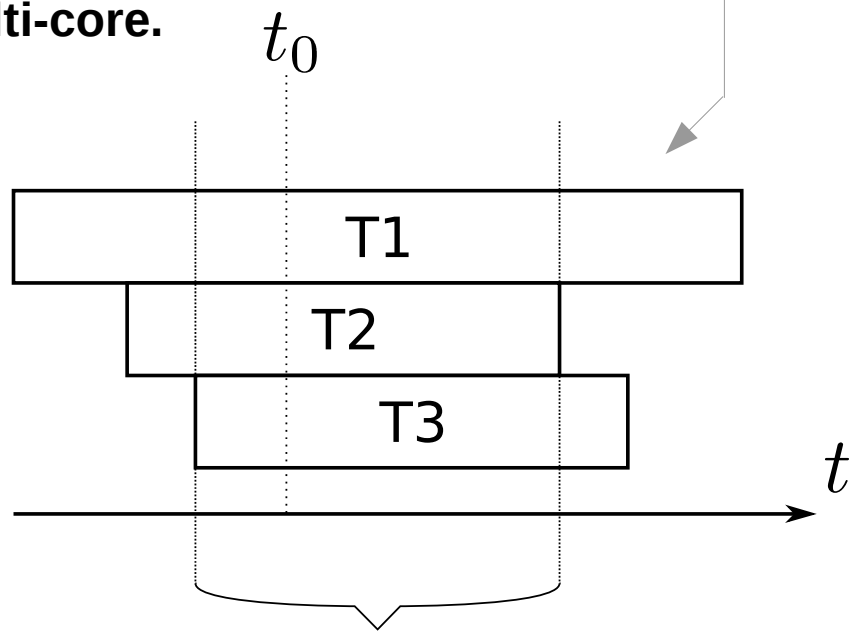
- *Concurrency vs. parallelism*:
  - ***Concurrent***: at least two tasks are making progress during a *period* of time.
  - ***Parallel***: at least two tasks are making progress at the same *point* in time.

- Concurrency useful for *I/O-bound* workloads, e.g. web servers.
  - Each task spends the majority of its time waiting for I/O, can be suspended.

- Scientific computing can be *CPU-bound*, *memory-bound* or *cache-bound*.
  - Each task spends the majority of its time either computing or transferring data.
  - **CPU-bound tasks need parallelism for multi-core.**



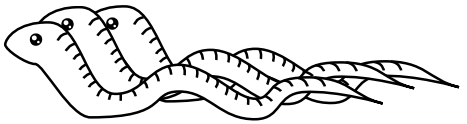
Time period considered



Parallel execution of T1, T2, T3
at all t_0 in this interval

This illustration uses Round-robin scheduling.

Parallel computing



Interaction strategies

- *Shared memory*
  - *Locking* ensures a consistent state when accessing shared data, i.e. *preserves the program's internal invariants* about the shared state.
    - This is done by serializing (synchronizing) access to the shared data.
    - Makes operations that mutate shared state *appear* atomic.
    - E.g. when a write into a shared data structure is in progress, other tasks must wait (before reading that data) until the write completes.
    - Often, *advisory* implementation; requires each thread to play by the rules.

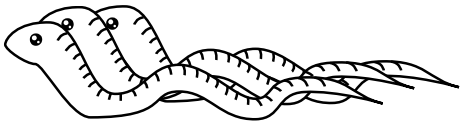
- *Transactional memory*
  - Reads/writes in shared memory *actually are* atomic (as seen by the user code).
  - No explicit locks in user code. Requires hardware or language support.
  - Related: non-blocking algorithms, lock-free data structures, compare-and-swap.
  - Bartosz Milewski (2010): Beyond locks and messages: The future of concurrent programming

- *Message passing*
  - General name for a set of approaches where all memory is private, and tasks communicate by sending messages.

- *Implicit parallelism*
  - Built into the language; e.g. Concurrent Haskell, Concurrent ML.

- https://en.wikipedia.org/wiki/Parallel_programming_model

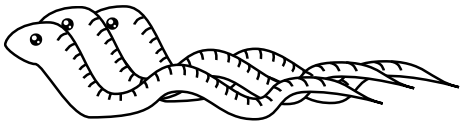
Parallel computing



Shared memory

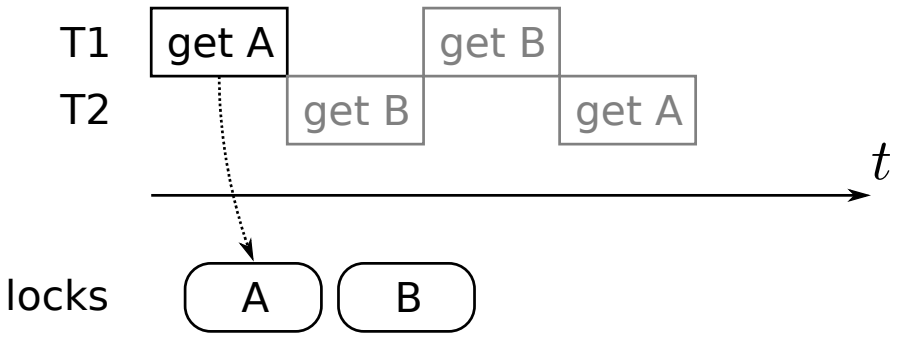
- Tasks usually modeled as threads in the same process.
- Does not scale when many tasks need access to the same resource; the mutually exclusive access becomes a bottleneck.
- Useful on a single computer, when the amount of shared data is large.
- Related term: *reentrant* code: code that can be safely interrupted and called again ("*re-entered*") before its previous invocations finish.
  - Documentation will typically warn if a routine is not reentrant (or *thread-safe*).
- Disadvantages:
  - *Deadlock*: two tasks T1, T2 that need two shared resources A, B may get stuck waiting for each other. (Example on next slide.)
  - *Livelock*: each task attempts to dodge the other's locking behavior, making no progress (like two people in a hallway trying to pass each other).
  - What happens if we don't lock?
    - *Race condition*: the result may depend on the ordering of concurrent events. (Example below.)

Parallel computing

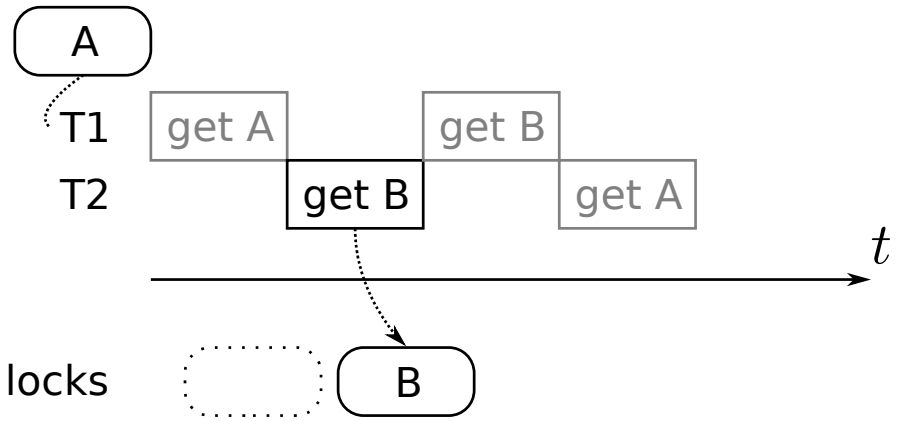


Deadlock example

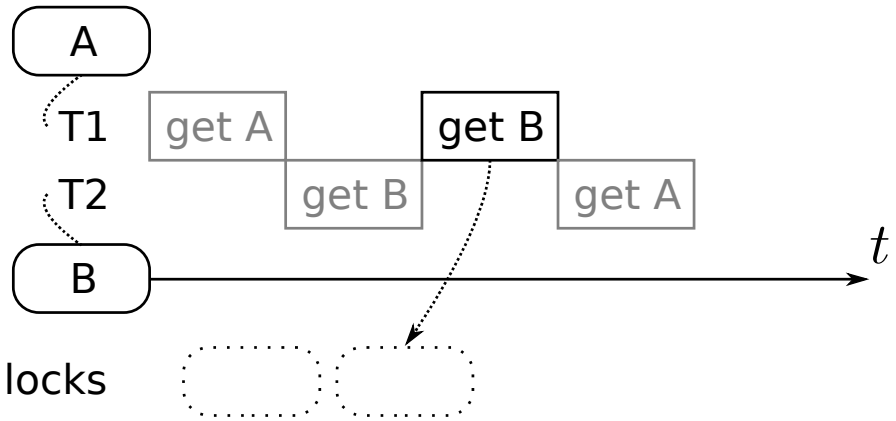
①



②

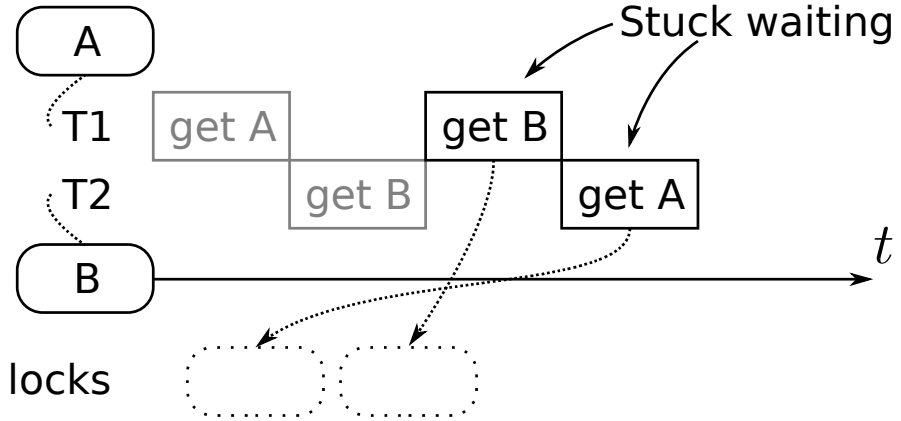


③



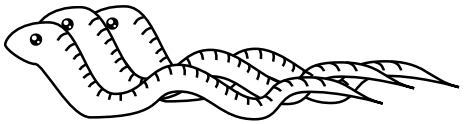
...but B is already taken.

④



Also A is already taken.

Parallel computing



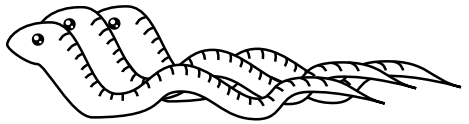
Race condition example

- What is the final value of *x*?

```
import time
import threading    # see standard library docs

def f(k):
    global x
    y = x                 # read shared state
    time.sleep(1e-6)      # wait 1 µs to simulate a “long” computation
    y += k                # compute (this step does not access x)
    x = y                 # write shared state

x = 1
T1 = threading.Thread(target=f, args=(2,))    # target is the thread's “main function”
T2 = threading.Thread(target=f, args=(3,))
T1.start()
T2.start()
T1.join()    # wait until the thread finishes
T2.join()
print(x)
```

- In *f()*, we could just as well replace the last three lines with *x* += *k*, but the above version makes the read-compute-update steps more explicit.

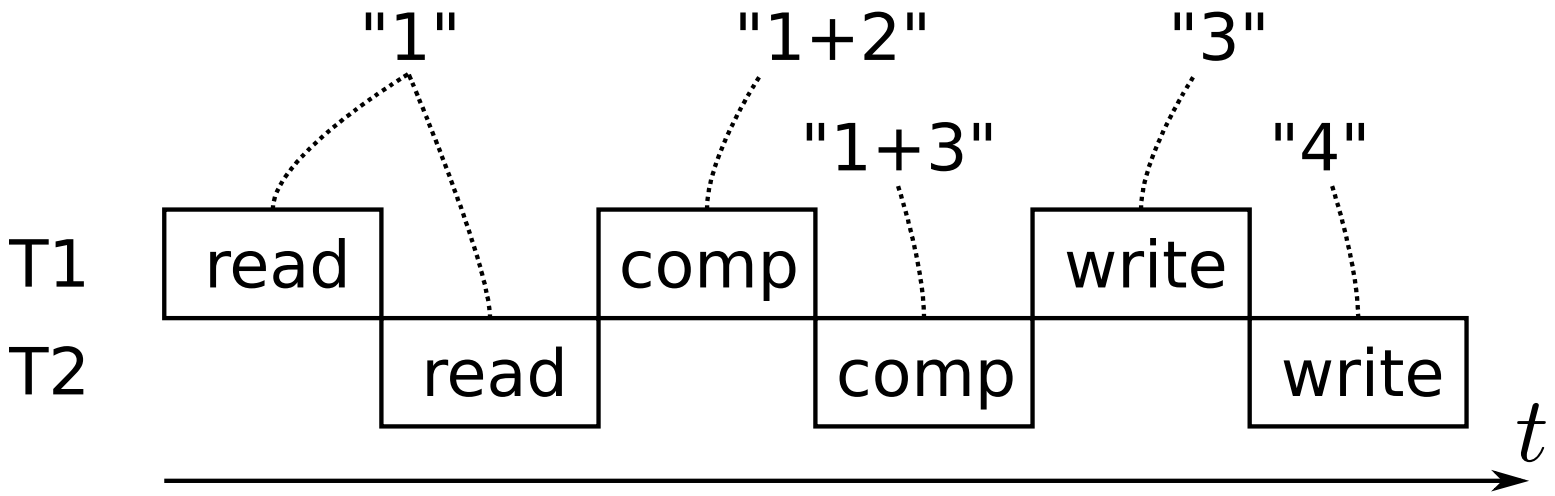
Parallel computing



Race condition example

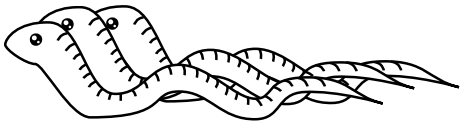
```
def f(k):
    global x
    y = x                  # read shared state
    time.sleep(1e-6)       # wait 1 µs to simulate a “long” computation
    y += k                 # compute (this step does not access x)
    x = y                  # write shared state

x = 1
# …T1, T2, …, start(), join(), …
```

- **Bug: *race condition*: the result depends on the order of execution**. For example:



- The fix is to add locking around the accesses to *x*. Lock, read, unlock, and lock, write, unlock; or lock, read-and-update, unlock.

Parallel computing



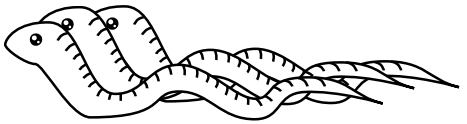
Race condition example

```
import time
import threading
import numpy as np

def f(k):
    global x
    y = x
    time.sleep(1e-6)
    y += k
    x = y

out = []
for i in range(1000):
    global x
    x = 1
    threads = [(lambda k: threading.Thread(target=f, args=(k,)))(k=j) for j in range(1, 21)]
    for T in threads:
        T.start()
    for T in threads:
        T.join()
    out.append(x)

hist,bin_edges = np.histogram(out)
bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2
print(hist, bin_centers)
print("Most common bin center: {v}".format(v=bin_centers[np.argmax(hist)]))
```

Exercises:

- Try this version for a real-world test.
- Protect accesses to *x* with Threading.Lock. Check that the modified program works.
- Read about other useful parts of *threading*, such as thread-local storage and events.
- How would you use events to signal worker threads to exit their main loops when the user presses *Ctrl+C* (which makes Python raise KeyboardInterrupt in the main thread)? *Hint / material*.

Parallel computing



Message passing

- Tasks usually modeled as processes.
  - All memory is private to each task. ***No shared data, no need for locking***.
  - Tasks communicate by sending messages.

- Scales well, unless the amount of data to be sent is so large that messaging becomes a bottleneck.

- Good for distributed computing.

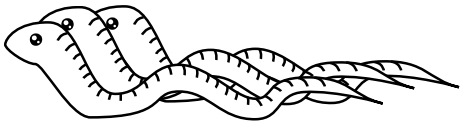
- Tasks fully independent. In principle, may run completely different code.
  - For implementing such a collection of parallel programs, see e.g. the ØMQ library.
  - However, *SPMD* (*single program, multiple data*) is a common design in scientific applications; e.g. the MPI (*Message Passing Interface*) framework is SPMD.

- Time spent communicating is time *not* spent performing useful work. Eventually scaling leads to parallel slowdown: added communication > added comp. resources.

- Excess computation may be needed (compared to the serial case), if the algorithm requiring the least amount of operations is inherently serial.

- *Synchronous* communication used often (but not always): when a task is sending, another one must be listening. In such cases deadlock can still occur.

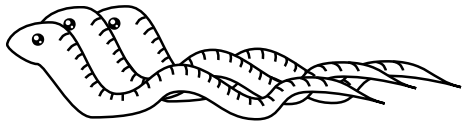
Parallel computing



SPMD (Single program, multiple data)

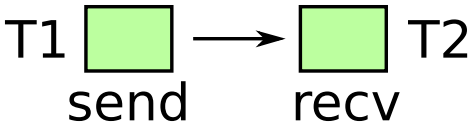
- ***All tasks execute (mostly) the same program, independently.***
  - This simplifies the user code: need to write only one program.
  - But beware: it's still a parallel program!
    - In general, more difficult to implement correctly than a serial program.
- At appropriate places, the program uses conditional statements (branches), which select the code to run depending on the task identity (number).
  - E.g. task 0 acts as a controller, whereas tasks ≥ 1 perform the actual computation.
- SPMD leads to some redundant computation, but not an issue:
  - Most of the computational workload is in the main solver algorithm; re-computing everything else does not cost much.
  - May actually be faster to redo “small” computations than compute once, then broadcast results.
    - Even in a PC, math is cheap, memory transfers expensive. In distributed computing, the messages must be sent over the network; even slower.

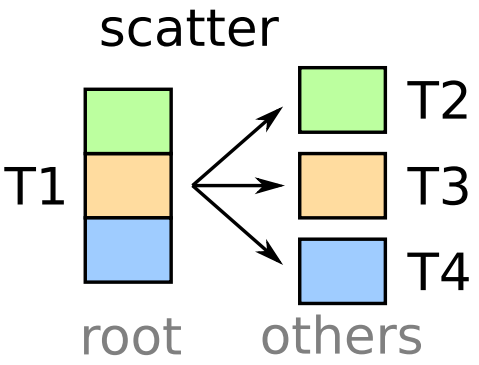
Parallel computing

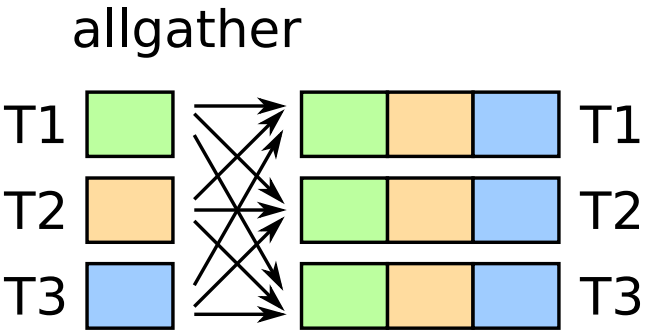


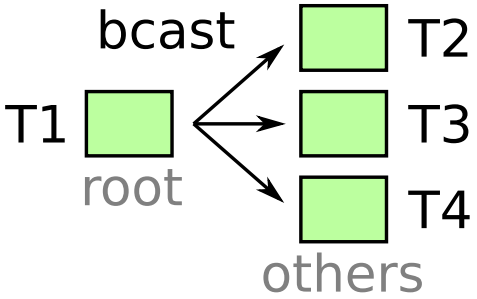
MPI (Message Passing Interface)

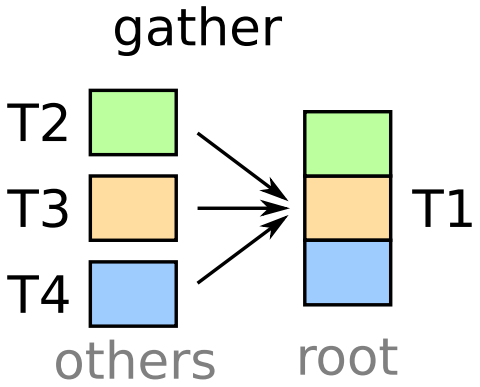
- SPMD – single program multiple data. MPI backend provides comm infra for clusters.
- ***MPI rank*** is the task identity (which is an integer); switch on that in your code.
- Some operations have a ***root process***, chosen by the user (often MPI rank 0).
- Tasks can be grouped by ***communicators***; default *MPI_COMM_WORLD* includes all
- **Operations**, in one picture. Finally, *barrier* waits until all tasks in the group reach it.





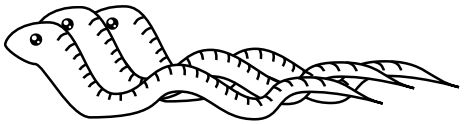






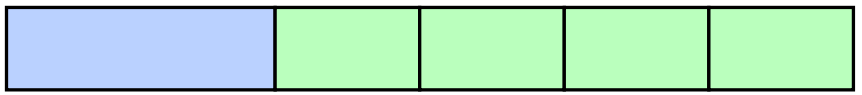
- For much more, see MPI Tutorials. Specifically for Python, see mpi4py tutorial.

Parallel computing

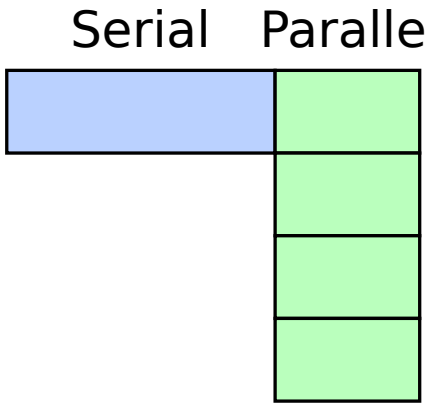


Scalability: Amdahl's law

- To perform *the same workload* in parallel (*strong scaling*), the maximum possible speedup (regarding overall latency) is limited by the fraction of code that cannot be parallelized.

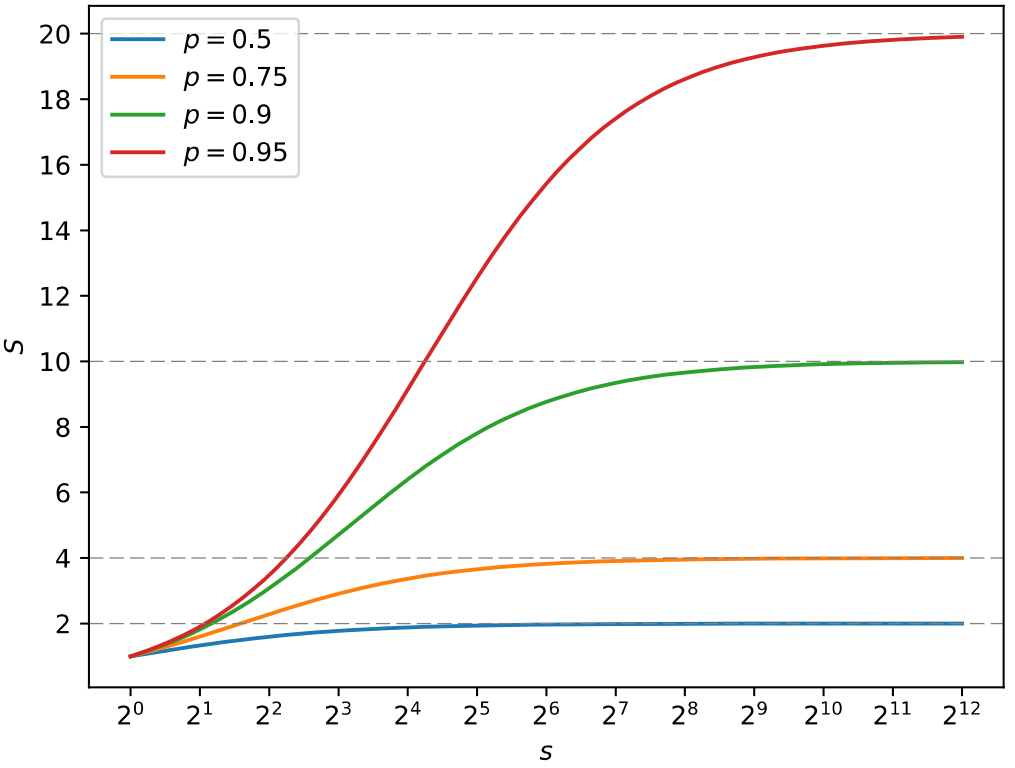


With the code parallelized...



Here $s = 4$.

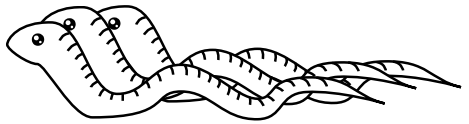
...we run it using s parallel tasks, to get it to finish as soon as possible.



$$S = \frac{1}{(1-p)+p/s}$$

S: theoretical speedup
p: fraction of code that can be parallelized
s: speedup of the parallelized part

Parallel computing



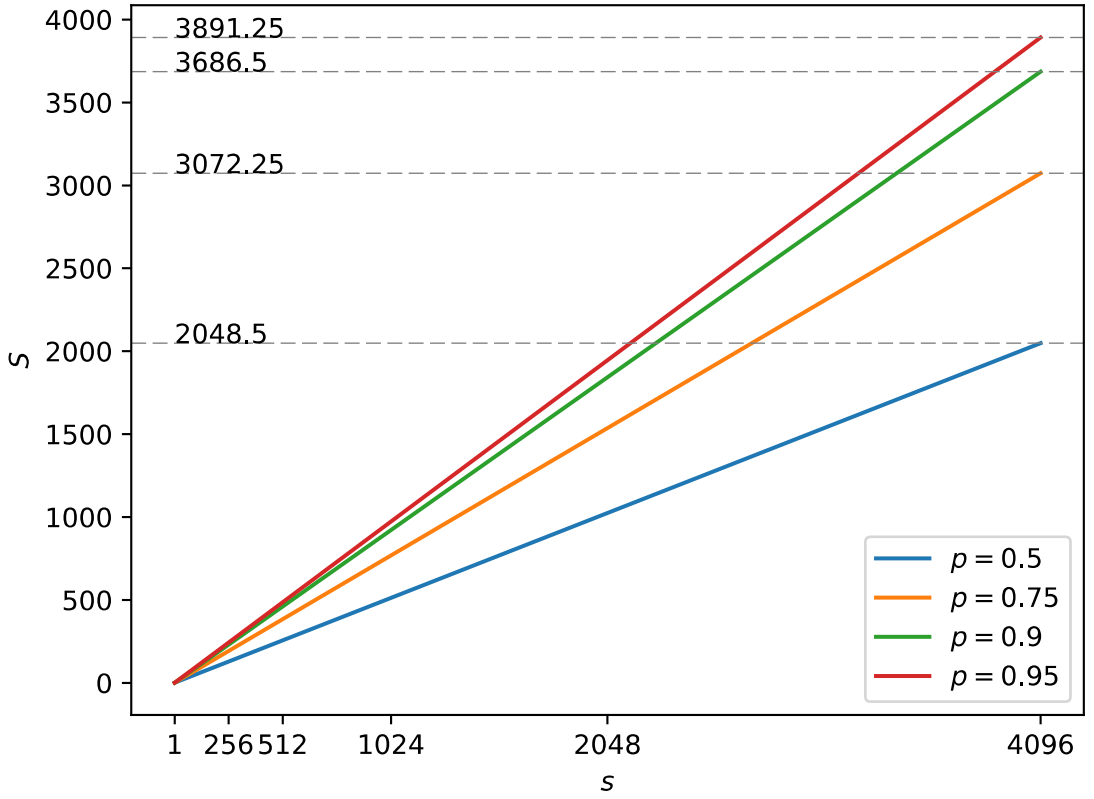
Scalability: Gustafson's law

- When resources increase, the workload is rarely kept the same; increased resources are often applied to a similarly larger workload (*weak scaling*).
- With the same *p* and *s* as before, but with *S* now denoting the theoretical weak-scaling speedup:

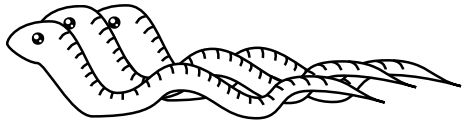
$$S = (1 - p) + sp$$

- Here we scale both the resources and the workload size by a factor of *s*, i.e. by as much as the parallelized code can take.
- Gustafson's law is much less pessimistic about the usefulness of parallelization than Amdahl's.

Slope = $p < 1$; a slope of 1 is ideal scaling (fully parallel code).



Parallel computing



Multicore Python

By core dev Nick Coghlan.
Recommended reading.

- ***Global interpreter lock (GIL)***: [1] [2] [3]
  - *The CPython interpreter is concurrent, not parallel*.

  - A Python thread can only execute with the GIL acquired.
    - Hence, Python threads compete for the GIL.

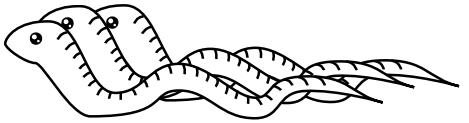
  - However, a Python thread will release the GIL, while:

    - Waiting for I/O – such as in server workloads.
      - Cannot proceed anyway until data available.

    - Executing a low-level function (C, Fortran) – such as those in NumPy and SciPy!
      - The low-level code cannot use Python objects. Can opt in to release the GIL.
      - If you code your own in Cython (next week), use Cython's **with** nogil: syntax.

  - Hence, whether the GIL presents a problem depends on how your program works:
    - Few, huge NumPy or SciPy operations – no problem; Python threads will run mostly in parallel.
    - Many small operations – threads not useful; consider a process-based approach.
      - multiprocessing, in the standard library. API similar to threading.
      - MPI with Python bindings, such as mpi4py. Requires an MPI backend such as OpenMPI or MPICH. Scales to clusters (if your algorithm does).

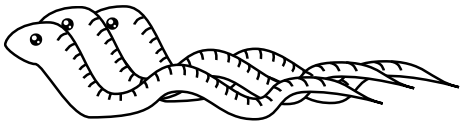
Parallel computing



Multicore Python

- **Further resources**:
  - *concurrent.futures.ProcessPoolExecutor* – a higher-level abstraction based on *multiprocessing*. Provides a context manager, futures, and a parallel *map()*.
  - Parallel programming with NumPy and SciPy (old, but still useful)
  - A Python introduction to parallel programming with MPI
  - David Beazley (2009): An introduction to Python concurrency
- Maybe of interest:
  - In a threaded Python program, watch out for import lock.
  - *Parallel design patterns*, such as Map and MapReduce.
  - PyPy interpreter with Software Transactional Memory (Python 2 only)

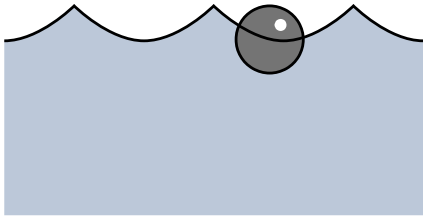
Parallel computing



Is it worth it?

- ***Do you really need to parallelize your code?***
  - If you intend to Monte Carlo, do sample-based uncertainty analysis, or run a parametric study (a.k.a. parameter sweep), then the complete workload is already embarrassingly parallel.
    - Simply run multiple serial simulations simultaneously.
  - If one large problem, then parallelize (if you really need more speed).
    - Take advantage of any easy opportunities for parallelism in your particular problem.
    - Parallelizing e.g. a linear equation system solver is difficult, but has been a topic of research, and software exists.
      - Parallel methods for linear equation systems (introductory material);
      - Dekker et al. (1994); Göddeke (2010) (research)
      - ScaLAPACK; PETSc; PLASMA; MAGMA; MUMPS (software).

Floating point



See also Wikipedia for a more detailed introduction.

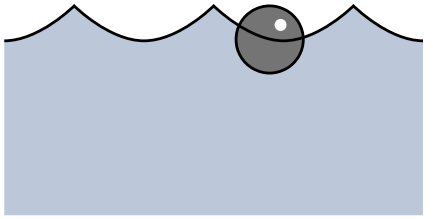
- Floating point arithmetic has an undue reputation as something that introduces random errors into numerical computations.
  - In reality, although subtle in many ways, floating point arithmetic is completely deterministic.

Related to the halting problem.

- **Main idea**:
  - Task: represent $\mathbb{R}$; but for arbitrary $x \in \mathbb{R}$, this is impossible.
    - Some real numbers are not even computable. Limits of Specker sequences, Chaitin's Ω.
    - For use with arbitrary numbers, a fixed-length, approximative storage scheme is desirable.
    - But such a scheme can only represent a finite subset of $\mathbb{Q}$.
      - Choosing this subset is a tradeoff of range vs. precision.
  - **Floating point**: *compute with a given number of significant digits*, regardless of the absolute magnitude. (The decimal point "floats".)

- Material:
  - *What every computer scientist should know about floating-point arithmetic*
    - David Goldberg's 1991 classic paper. Not only for CS experts; recommended!
    - TL;DR summary: *What every programmer should know about floating-point arithmetic*
  - *You're Going to Have to Think!*, a series of articles by Richard Harris. (Start here.)
  - *Bruce M. Bush (1996): The Perils of Floating Point*
  - Python tutorial: Floating point arithmetic – Issues and limitations
  - *IEEE Floats and Python*
  - *Comparing Floating Point Numbers, 2012 Edition*
  - *Stupid Float Tricks* – e.g. adjacent floats have an adjacent integer representation.
  - Computer mathematics – Floating-point representation (course material)

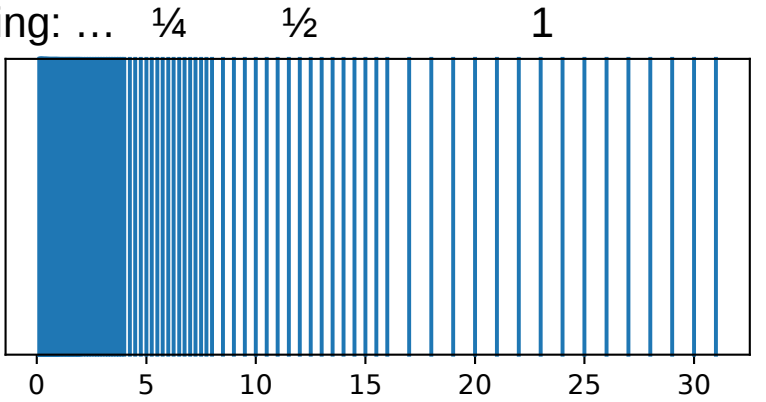
Floating point



- Roughly, the format is:

$$f = s \cdot m \cdot b^k$$

So, roughly, floats are logarithmically spaced.



where

s = sign, +1 or -1
m = mantissa; $m \in \mathbb{Q} \cap [1, b)$
b = base; integer ≥ 2. Usually $b = 2$.
k = exponent, integer

Representable numbers for “toy floats”: 4 bits for m, 3 bits for k, $b = 2$. Program.

Here k = -3, -2, …, 4. (3 bits ⇒ 8 values)

Note higher density near the origin, since m uses a fixed number of bits.

- Example with $b = 10$:

exponent

$$f = +1 \cdot 2.99792458 \cdot 10^8 = 299\ 792\ 458$$

sign mantissa base b = 10

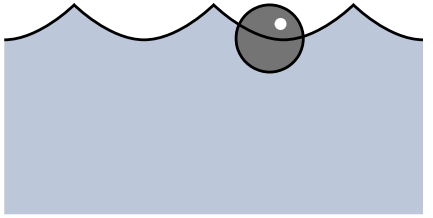
In Python, see *sys.float_info*.

The mantissa always has **the same number of significant digits**, specified by the storage format.

- Mantissa given as a ***binary fraction*** (like a decimal number, but in base-2).
- A *bias* is added to the exponent so that e.g. [0, 2047] means [-1022, +1023]

On this course, “float” means IEEE-754 double precision unless otherwise specified.

Floating point



- Commonly used floating point standard: IEEE-754. (Lecture note.)
  - ***Single precision***: 32 bits; ≈7.22 decimal digits
    - 23-bit mantissa, 8-bit exponent – **video games, graphics**; consumer GPUs use this format
    - Precision not sufficient for scientific use without special tricks
      - Especially, partial differential equation (PDE) solvers often need to handle both very small and very large numbers in the same expression.
  - ***Double precision***: 64 bits; ≈15.95 decimal digits
    - 53-bit mantissa, 11-bit exponent – **scientific applications**
    - Used in CPUs and specialist compute boards (AMD FirePro, NVIDIA Tesla, Intel Xeon Phi)

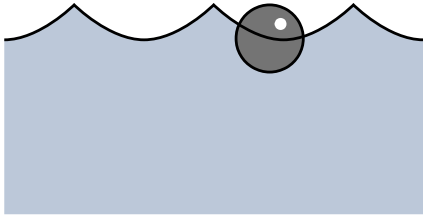
- **The exponent is normalized**. This brings the mantissa into the half-open interval [1, *b*).
  - E.g. in base-10, we can agree to represent 100 as $1 \cdot 10^2$ instead of e.g. $10 \cdot 10^1$ or $100 \cdot 10^0$.
  - In base-2 (only), the normalization fixes the leading bit of the mantissa as 1, so it does not need to be represented (*implicit bit convention*).

- However, for large negative exponents, we can slightly extend the range – at the cost of some precision – if (for those numbers only!) we skip normalization, and do not use an implicit 1.
  - These are known as ***subnormal numbers*** (a.k.a. *denormal numbers*).
  - They linearly span the gap at the origin left by the normal floats.
  - Computing with subnormal numbers can be very very slow, due to lack of hardware support.

- Can also represent the limit-like quantities +∞, -∞, and NaN (*not a number*, such as 0 / 0).

- **Zero has a sign**; +0.0 and -0.0 are different floats.
  - Think limits; e.g. “1 / +∞ = +0.0” ⇔ $\lim_{x \to +\infty} 1 / x = 0$, the result reaching 0 from the + side.
  - The programming language may abstract this detail away; `1 / -float("inf") == 0  # True`

Floating point



- Error sources?
  - **Representation**, **Round-off**, **Cancellation**

- ***Representation***. Let's consider an example:
  - With base-2 floats, 2^k is exactly representable, also for $k < 0$. But humans use base-10.
    - Decimal floating point has not (yet?) caught on in hardware, although in 2008, the IEEE-754 standard added decimal64.
    - If you really need this capability (e.g. counting money), Python has a software-based (i.e. too slow for scientific computing!) *decimal* module.
    - For software-based (slow!) *arbitrary precision* base-2 floating-point, the *mpmath* library.

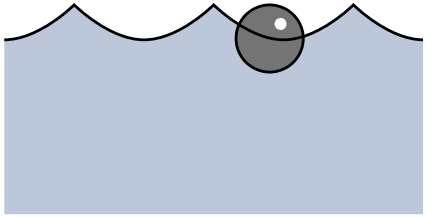
  - In base-2, *only fractions where the denominator is a power of 2 have terminating expansions*.
    - Cf. base-10; similarly, expansion terminates if the denominator has powers of 2 and 5 only.
    - For example, 0.1 decimal is $0.0\overline{0011}$ binary, where the overline denotes repeat.
      - *An exact 1 / 10 does not exist in binary*, no matter how many bits are used!

  - Unrepresentable numbers rounded to a representable number, ties usually rounded to even.

  - For example, at 28 decimals, float(19 / 20) is actually 0.9499999999999999555910790150
    - But in Python, the display handler performs magic to print 0.95. To work around that:

```
from decimal import Decimal, getcontext
print(getcontext())                          # show how many decimal places in use
r = (19 / 20).as_integer_ratio()             # create float “0.95”, get exact fraction
d = Decimal(r[0]) / Decimal(r[1])            # result rounded to getcontext().prec places

'{:0.18f}'.format(19 / 20)                   # or just use string formatting
```

Floating point



- ***Round-off*** of results.
  - 0.1 + 0.1 + 0.1 == 0.3 # False
    - Actually means float(0.1) + float(0.1) + float(0.1) != float(0.3), which is correct.
      - **The "+" is not exact**: floating-point summation often must round the result.
    - A combination of representation and roundoff errors. (As seen with *mpmath*.)
      - Both the above sum and 3 * 0.1 yield the result 0.30000000000000004.
      - Multiplying at higher precision, 3 * float(0.1) is 0.300000000000000016665.
      - float(0.3) is 0.29999999999999989 (representation error only).

  - sum(0.1 **for** k **in** range(100)) # 9.99999999999998
    - If you need an accurate sum (3 or more terms), consider, at the cost of more arithmetic:
      - *Kahan summation algorithm* (a.k.a. *compensated summation*), reduces round-off error.
        - Keeps a total of round-off, and adds it back to the sum when enough accumulated.
      - *Hettinger (2005): Binary floating point summation accurate to full precision*
        - In Python, *math.fsum()*. Completely eliminates both round-off and cancellation errors.
    - Useful for e.g. simulation of undamped vibrations, where errors accumulate.

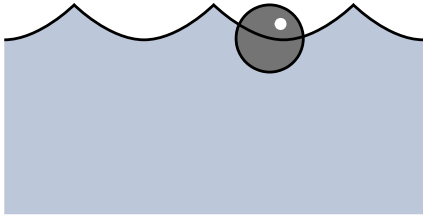
  - Useful: FMA (*fused multiply-add*); compute $x * y + z$ without rounding the intermediate result.

  - **Wilkinson's polynomial**. An example of an ill-conditioned problem.

    $w(x) = \prod_{k=1}^{20} (x - k) = (x - 1)\,(x - 2)\,\ldots\,(x - 20)$

    - Task: find the roots numerically. Obviously, $w(x)$ has 20 roots, at $x = 1, 2, 3, \ldots, 20$.
    - Highly sensitive to small perturbations in the coefficients. Changing the coefficient of x^{19} from -210 to -210.0000001192 (a shift of -2^{-23}), the value $w(20)$ decreases from 0 to $-2^{-23} \cdot 20^{19} \approx -6.25 \cdot 10^{17}$; the root at $x = 20$ changes to $x \approx 20.8$.

Floating point



- ***Cancellation***. Be careful when subtracting floats!
  - Especially *catastrophic cancellation*; a special case of ***loss of significance***.
    - For example, in decimal, consider this calculation, with 20 significant digits:

      0.12345678912345678 90 – 0.12345678900000000 00 = 0.00000000012345678 90
    - However, if we use only 10 significant digits:

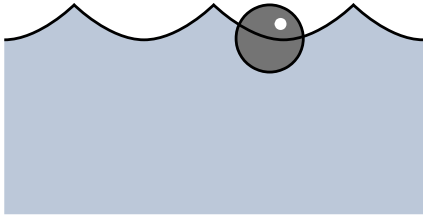
      0.1234567891 – 0.1234567890 = 0.0000000001

      As a (normalized) decimal floating point number with 10 significant digits, this answer is

      $1.000000000 \cdot 10^{-10}$

      Here most of the "significant" digits are doing nothing useful!
    - Informally, the problem is that the input numbers are very close to each other. Most of the digits of the representation are wasted on digits that are common to both of them, then canceling out in the subtraction.
    - *Serious problem for finite differences*. E.g. for $\partial v / \partial x \mid x=x_0 \approx (v(x_0 + h) - v(x_0 - h)) / (2\,h)$, there is an optimum h, below which cancellation destroys the accuracy.
      - (Exercise: try it! Take e.g. $v(x) = \sin(x)$. Then analytically, $\partial v / \partial x = \cos(x)$. At some x, plot the difference between the FD formula and the exact derivative as a function of h.)

Floating point



- Solving a quadratic equation:

  $a\,x^2 + b\,x + c = 0$.

  The schoolbook answer is [if you want to derive it yourself, *depress* the polynomial]:

  $x = (1 / 2\,a)\,(-b \pm (b^2 - 4\,a\,c)^{1/2})$

  - However, if $|4\,a\,c| \ll b^2$, then the magnitudes of b and $(b^2 - 4\,a\,c)^{1/2}$ are close to each other.
  - The combination with different signs (which one it is, depends on the sign of b) leads to **catastrophic cancellation**.
  - Solution: use a better algorithm, avoiding the subtraction of almost equal quantities:

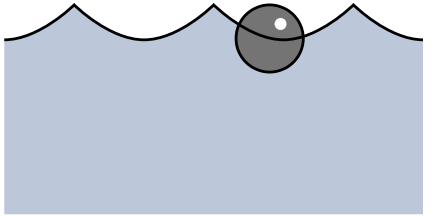
    $x_1 = (1 / 2\,a)\,(-b - \mathrm{sgn}(b)\,(b^2 - 4\,a\,c)^{1/2})$
    $x_2 = (2\,c)\,(-b - \mathrm{sgn}(b)\,(b^2 - 4\,a\,c)^{1/2})^{-1} = c / (a\,x_1)$

    copysign(1, b) from IEEE-754 is here better; works also in edge cases.

    Python: *math.copysign*. Also in C99.

- For much more on pitfalls and how to avoid them, see *Numerical Recipes* by Press et al.
  - This book is highly useful in that it does not assume exact arithmetic; finite precision floating point arithmetic is accounted for.
  - But if you code open source, avoid looking at the code examples; they are all rights reserved.

Floating point



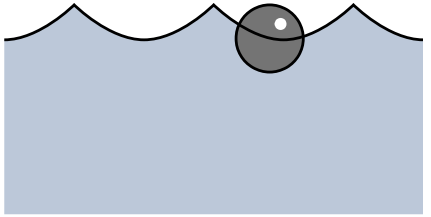
- Some final things:
  - *ε* (*machine epsilon*): upper bound on the *relative* error due to rounding in floating point arithmetic.
    - ***ε = spacing between +1.0 and the next larger float***
      - IEEE-754 double precision: *ε* = 2.220446049250313e-16
      - *Despite the name,* ***not*** *the smallest representable float.*
        - Smallest positive *normal* float: *sys.float_info.*min = 2.2250738585072014e-308
        - Largest positive finite float: *sys.float_info.*max = 1.7976931348623157e+308

```
import sys
ε = sys.float_info.epsilon
print(1 + ε)      # 1.0000000000000002
print(1 - ε)      # 0.9999999999999998
print(1 + ε/2)    # "1.0" – exact result not representable, since ε is the spacing at 1.0
print(1 - ε/2)    # 0.9999999999999999 – next smaller exponent; hence spacing is ε/2
```

  - *ULP* (*Unit in the Last Place*)
    - ULP = the value the least significant digit represents if it is 1.
    - Spacing toward the next larger float (in absolute value) at arbitrary *x*. For normal floats:

```
import sys, math
def ulp(x):
    ε = sys.float_info.epsilon
    m_min = 2**math.floor(math.log2(abs(x))) # abs. value represented by a mantissa of 1.0
    return m_min * ε
```

Floating point



- In IEEE-754, +, –, *, /, and square root are guaranteed to be accurate to the ULP.
  - But for “library functions” such as *cos* and *log*, **this is not mandated**!

- Often heard in the programming community: “*never compare floats for equality*.”
  - “*Never*” is overly harsh; should be “*almost never*”.
    - For example, it may make sense to compare for equality when checking whether some iterative algorithm has converged all the way to the ULP.
    - Mainly useful in low-level building blocks. In the real world, aside from floating point, there are often much larger error sources (modeling, truncation, measurements).

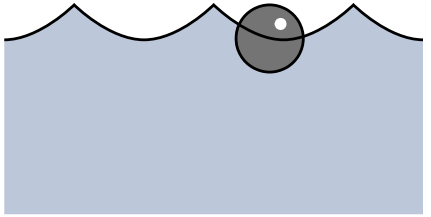
  - In many cases, a tolerance helps. Paraphrasing the floating point guide:

    - An absolute tolerance is almost always wrong: **don't** abs($x - y$) < *tol*.
      - The main idea of floating point is to be flexible for use in a wide variety of applications. Even if the exponents of x and y match, it is impossible to know *a priori*, in the general case, what the value of that exponent will be.

    - A relative tolerance, abs(($x - y$) / y) < *tol*, can also fail:
      - If $x = y = 0$. Either returns NaN, or raises an exception (Python: ZeroDivisionError).
      - If $x \neq 0$, $y = 0$. Either ∞ or an exception. If $x > 0$, then +∞, which ≥ *tol*, even if $x <$ *tol*.
      - If x and y are very small but on opposite sides of zero.
        - *Even when they are the smallest possible non-zero numbers.*
      - *Not commutative*: the result depends on which number is x and which is y.

    - See the guide for one solution; also see Comparing Floating Point Numbers, 2012 Edition.

Floating point



- *Try higher precision to be sure?*
- **Rump's example**: a reasonable increase in precision does not always help!

```
f = lambda x, y: ((333.75 - x**2) * y**6
                  + x**2 * (11 * x**2 * y**2 - 121 * y**4 - 2)
                  + 5.5 * y**8 + x/(2*y))

a = f(77617, 33096)    # approx. 1.1726039400531787 – with 32-, 64- and 128-bit floats

import mpmath    # How about unreasonable precision (still with base-2 floats)?
mpmath.mp.dps = 1000    # 1 000 decimal places (≈3 325 bits)
b = f(mpmath.mpf(77617), mpmath.mpf(33096))    # -0.827396059946821 3… – correct value
```

- **Totally wrong!** Sign, order of magnitude, digits. No indication of problem even at 128 bits!
  - To be fair, it's not just numerics; mathematics is full of surprises: Φ_{105}, Weierstrass function, Cantor function, fractals, long line, linear PDE with no solution, right-angled regular pentagon.

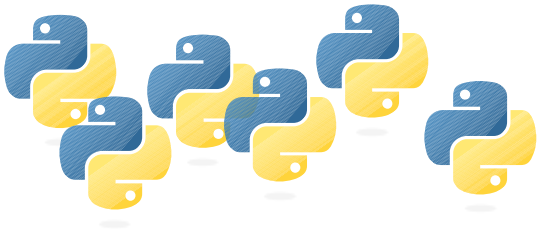
- Original source: Rump (1988): Algorithm for verified inclusions – theory and practice. *Reliability in computing*, Academic Press, pp. 109–126.

- Loh & Walster (2002): Rump's Example Revisited. *Reliable computing*, **8**(3), 245–248.
  - Updated version that fails also under IEEE-754 arithmetic.

- Taschini (2008): Interval Arithmetic: Python Implementation and Applications. *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, pp. 16–22.
  - Includes essentially the above Python code snippet to evaluate the updated example.

More libraries

As collected from lecture material, section 2. Links added.



- Accelerators:
  - Easy parallelization of NumPy expressions: NumExpr
  - JIT compiler for a subset of Python: Numba
  - GPGPU: Theano
- Data analysis for relational and labeled data: Pandas
- Machine learning: scikit-learn
- Image processing: scikit-image
- Splines:
  - B-splines for low-level work: bspline
  - For simple least-squares fitting, scipy.interpolate.LSQUnivariateSpline is fine
- Conformal mapping: cmtoolkit
- PDE (boundary value problem or initial-boundary value problem) solvers:
  - SfePy: Simple Finite Elements for Python – solver construction kit
  - The FEniCS computing platform – features automatic discretization of weak forms
  - Importing meshes: GmshTranslator (for Gmsh .msh files, good for academic work)
- Toolkits for graphical user interfaces (GUI):
  - PyQt: based on Qt (developed since 1995), which is widely used. Spyder uses this.
  - PyGTK: based on GTK (developed since 1998), which is also widely used.
  - Kivy (developed since 2011): main target smartphones, but can run unmodified also on PCs.
  - Plotting in a GUI: PyQtGraph, or Kivy's garden.matplotlib
- Frameworks for web applications: Flask, Django
- Databases:
  - SQL: sqlite3 (in standard library), python-sql (for queries), MySQL Connector/Python [docs].
  - NoSQL: TinyDB, redis-py [for redis] [docs] [tutorial], python-memcached [for memcached]
  - Probably others; I'm no database specialist!

Meta

Next time

- High-performance computing in Python.
  - At least Cython; possibly examples of Numexpr, Numba.

- See you next week!