



High Performance Python
EuroSciPy May 2014
License: CC ByAttribution
NonCommercial

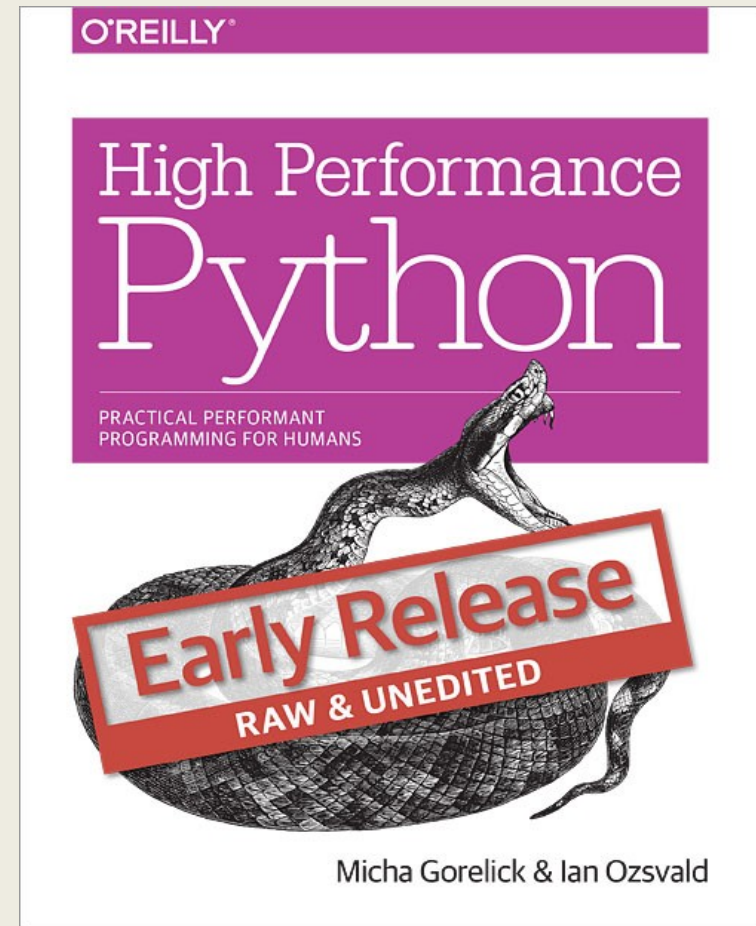
Ian Ozsvald @IanOzsvald MorConsulting.com

We'll cover

- Why we need to think about high performance
- Cython (pure Python and numpy)
- Numba
- Pythran
- PyPy

“High Performance Python”

- Published August
- Python 2.7 focused
- Lots of practical stuff



About Ian Ozsvald

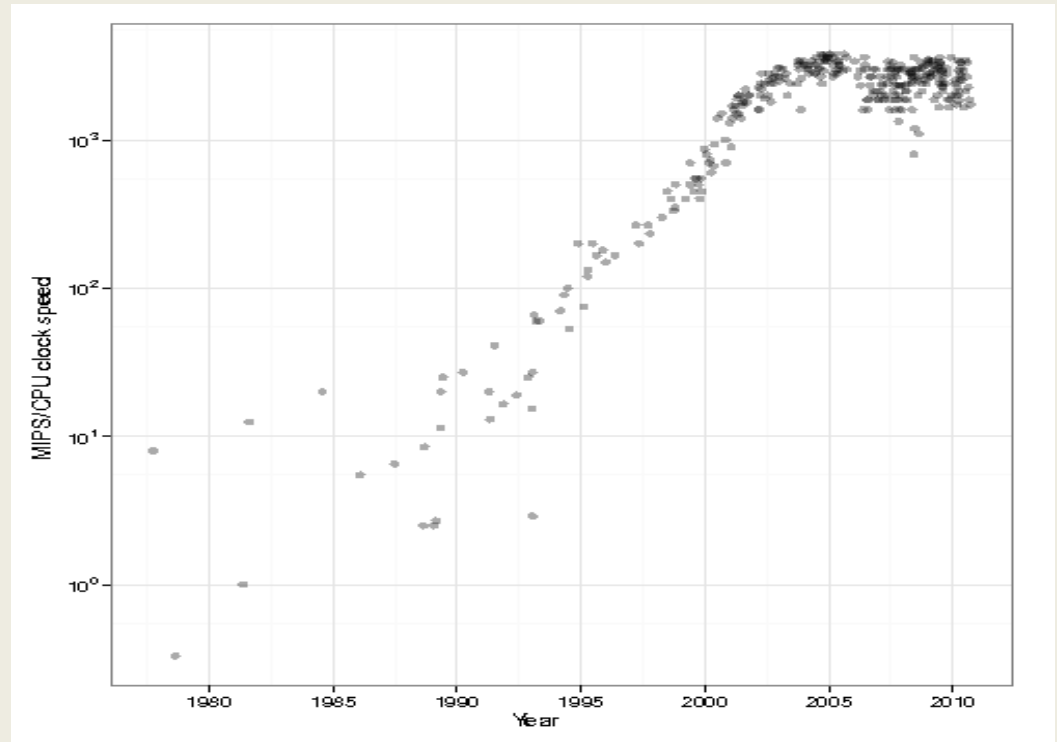
- “Exploiter of Data” in ModelInsight.io
- *I teach privately (modelinsight.io)!*
- Teacher: PyCon, EuroSciPy, EuroPython
- Various ML/Parallel/Data projects
- ShowMeDo.com
- IanOzsvald.com

Gordon Moore's Law

- Number of transistors on an IC doubles every 18-24 months
- Self fulfilling
- Clearly doesn't mean linear speed increases...

Moore's Law - limitation

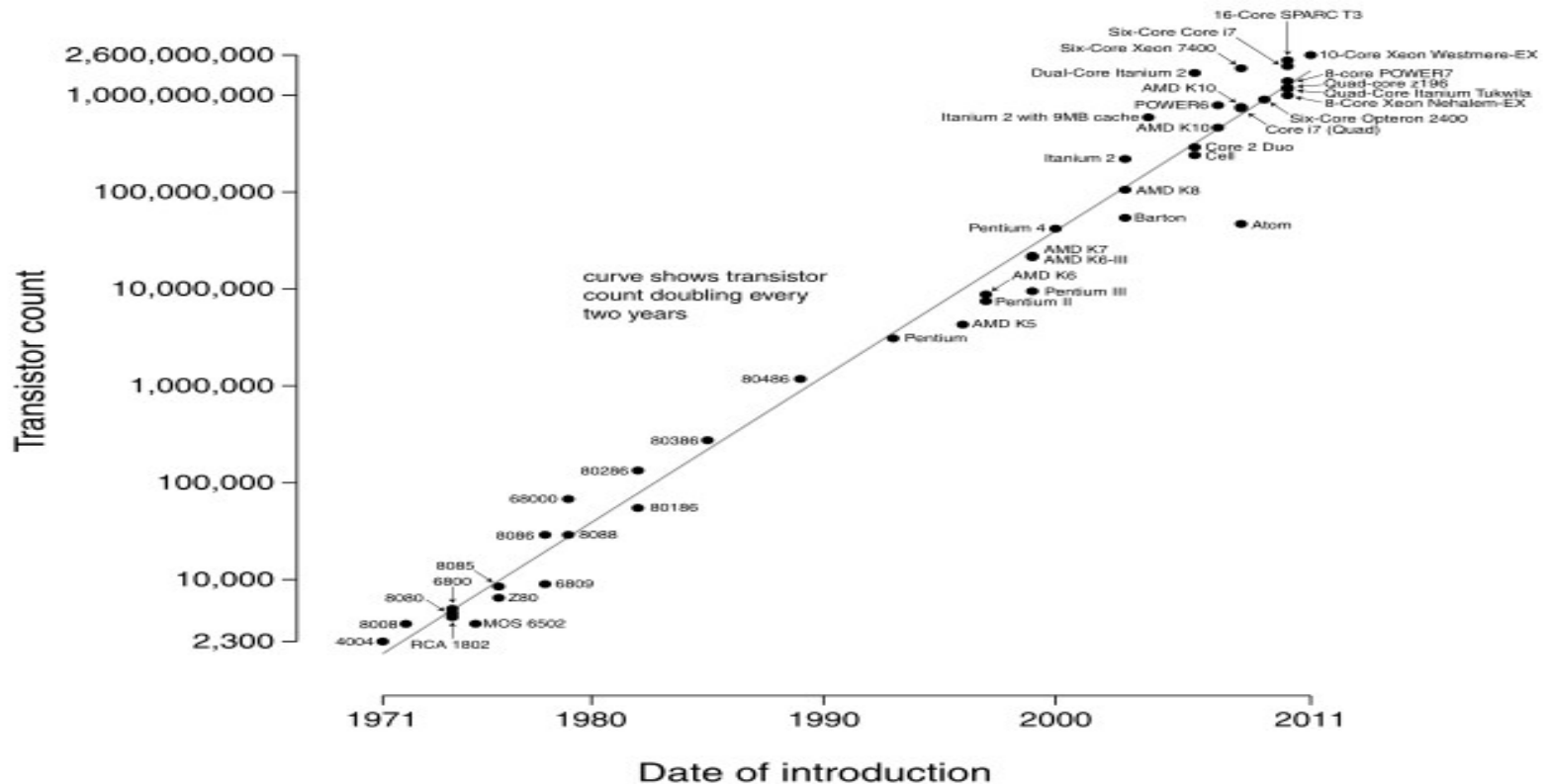
- 3.4GHz – why?



- <http://csgillespie.wordpress.com/2011/01/25/cpu-and-gpu-trends-over-time/>

Moore's Law

Microprocessor Transistor Counts 1971-2011 & Moore's Law



http://en.wikipedia.org/wiki/Moores_law

Proebsting's Law

- *“Proebsting’s Law asserts that improvements to compiler technology double the performance of typical programs every 18 years”*
- *“Pro. has suggested that ... communities should focus less on optimization and more on programmer productivity”*
- <http://www.cs.virginia.edu/~techrep/CS-2001-12.pdf>

Why use Python?

- Easy to use tooling
- Designed as beginner language
- Easy to keep in your head
- Large community (sci+eng)
- People are tackling *all the problems*
- Science, storage, visualisation, machine clustering, html, robustness, parsimonious coding

General go-fast rules

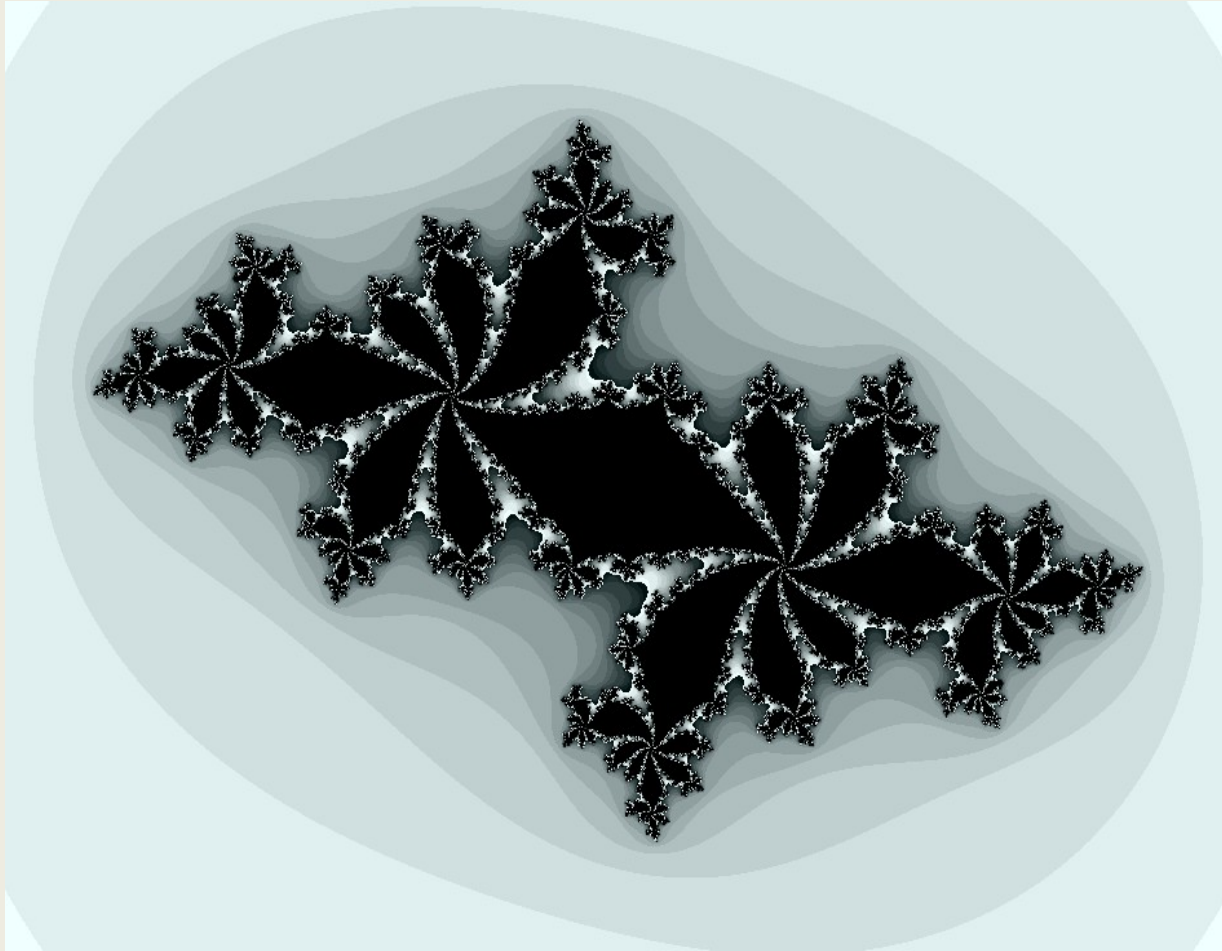
- Do as little work as possible

You won't beat grep:

<http://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html>

- Cache to avoid re-work
- Keep everything debuggable
- Keep everything documented

The Julia Set Fractal



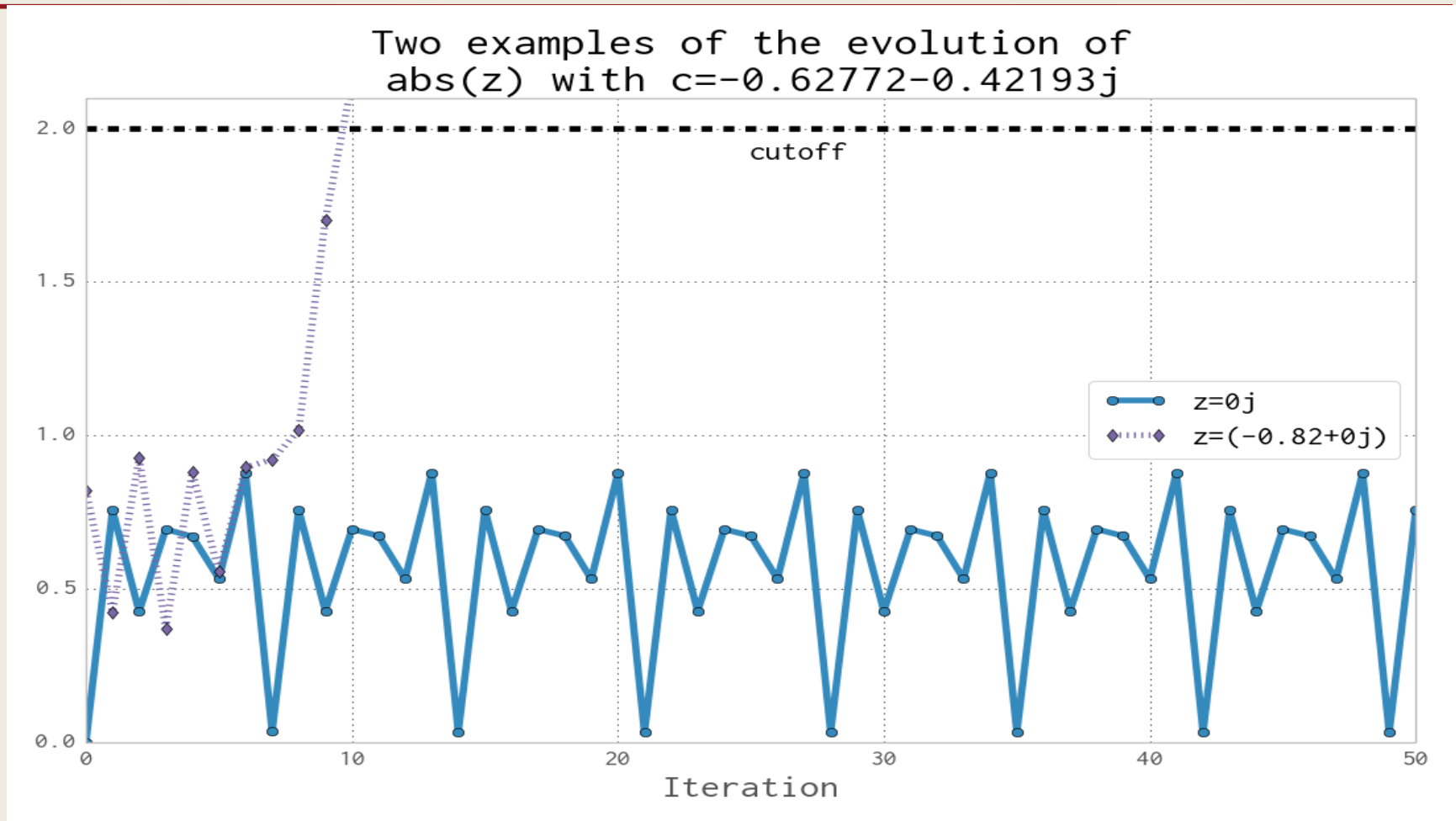
The Julia set

- Complex plane (just a co-ord set)
- Complex behaviour (what does this mean?)

$$f(z) = z^2 + c$$

- Embarrassingly parallel function
 - what does this mean?
- We're testing for bounded behaviour

The Julia set - evolution



The Julia set

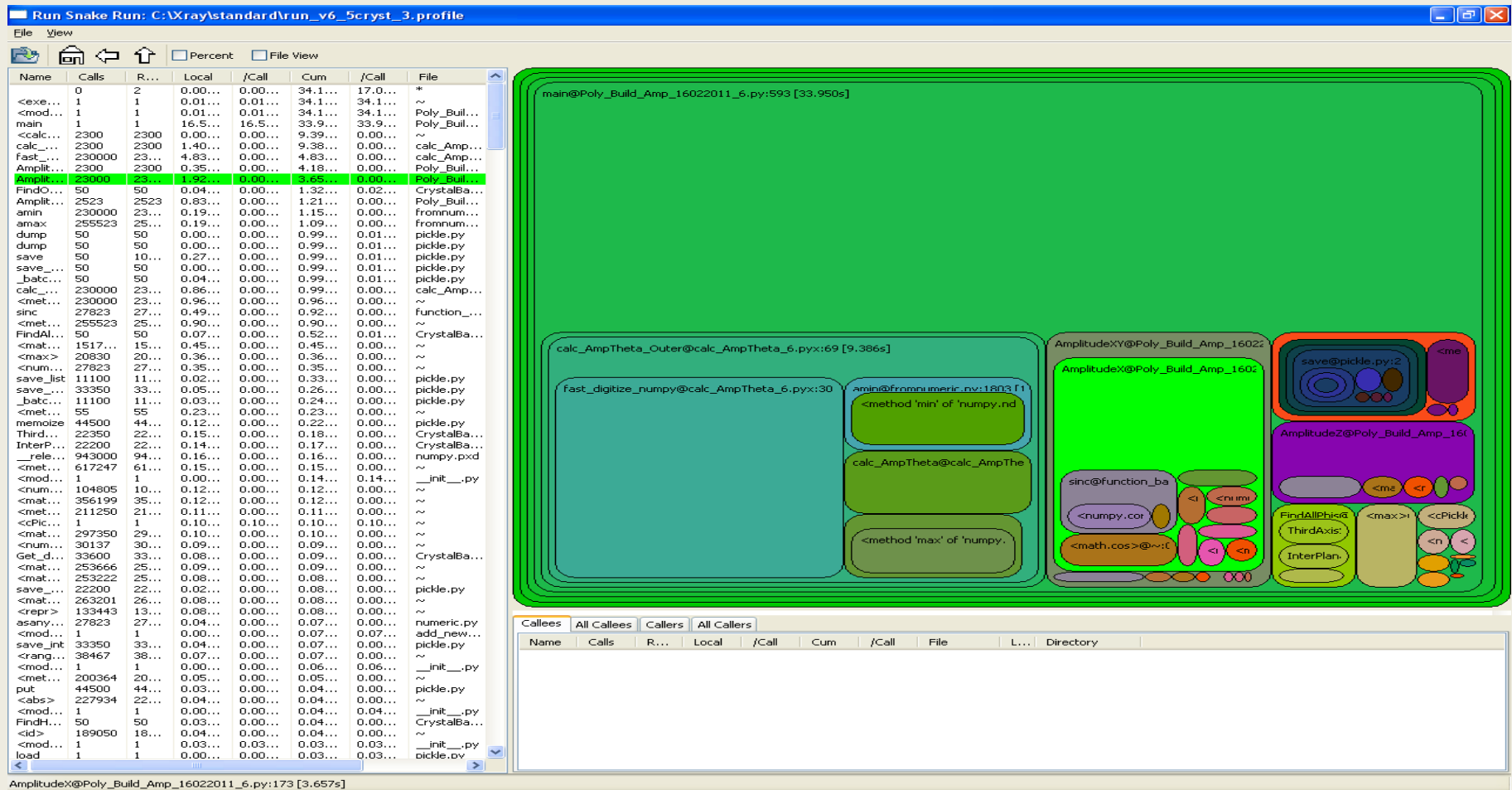
- Simple loop in code
- Let's review the code in julia.py (it is deliberately written suboptimally)
- We have a 1000 x 1000 array

```
while abs(z) < 2 and n < maxiter:  
    z = z * z + c  
    n += 1  
output[i] = n
```

Profiling the CPU

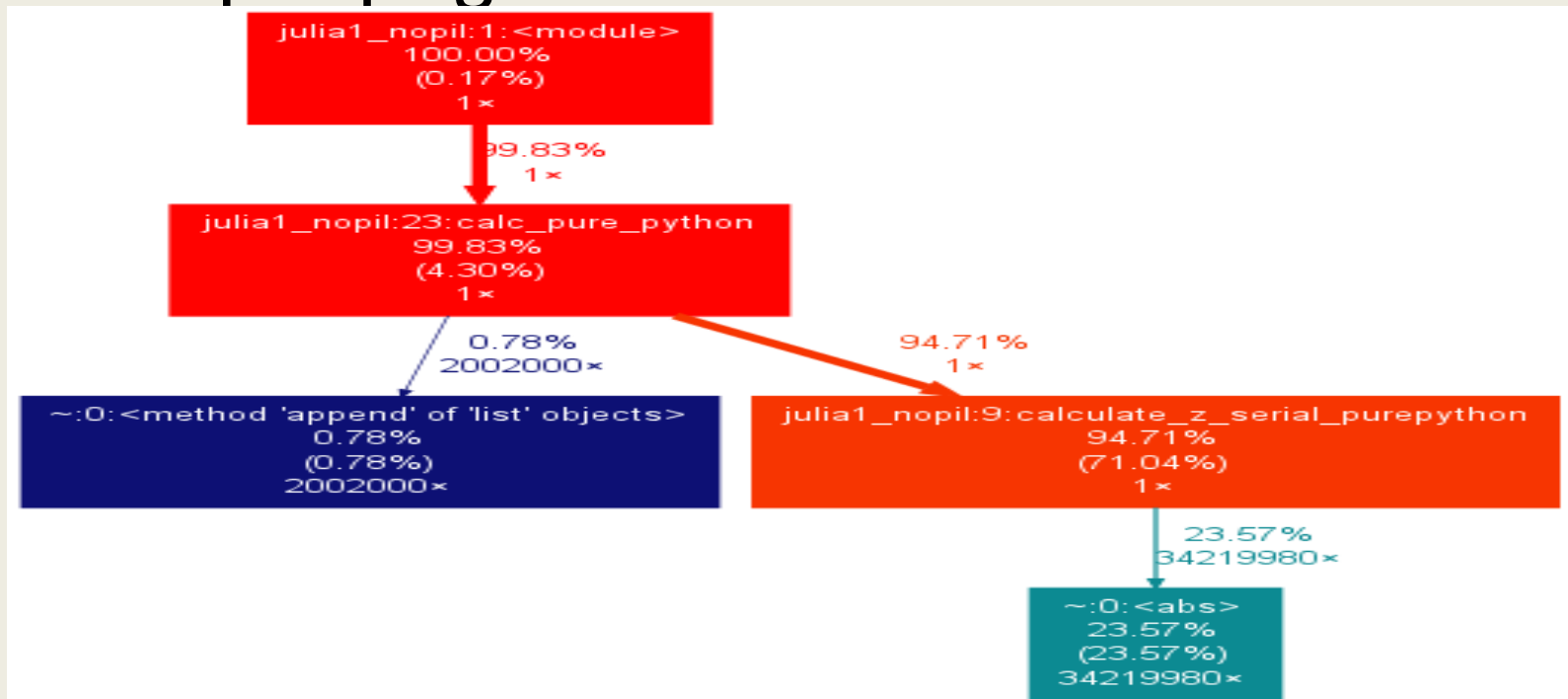
- “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil” - Donald Knuth
- Figure out what's slow, only optimize if it is worth it
- Optimizing takes time, costs mental cycles, introduces more complex code

cProfile & runsnakerun



cProfile and gprof2dot

```
gprof2dot -f pstats profile.stats | dot -Tpng  
-o output.png
```



line_profiler

- More informative, takes longer
- Line by line profiling
- Uses a C backend
- @profile – what is this?
- Make “julia_lineprofiler.py”, add @profile before calculate_z_serial_purepython
- !change max_iterations to 100 (from 300)
- !remove the assert

line_profiler

- `kernprof.py -l -v julia_lineprofiler.py`
- Run this first! It takes a while...
- *Can you explain the output to me?*
- *What is most costly?*
- We're using 100 `max_iterations` (not 300)
- More informative, takes longer
- Line by line profiling
- Uses a C backend

line_profiler (max 300 its.)

```
Line #      Hits          Time Per Hit   % Time  Line Contents
=====
     9                                @profile
    10                                def calculate_z_serial_purepython(maxiter, zs, cs):
    11                                """Calculate output list using Julia update rule"""
    12                1          6888    6888.0     0.0        output = [0] * len(zs)
    13       1000001       766409         0.8     0.8        for i in range(len(zs)):
    14       1000000       758497         0.8     0.8            n = 0
    15       1000000       817633         0.8     0.8            z = zs[i]
    16       1000000       757191         0.8     0.8            c = cs[i]
    17       34219980      36893641         1.1    36.7            while abs(z) < 2 and n < maxiter:
    18       33219980      31852838         1.0    31.7                z = z * z + c
    19       33219980      27775502         0.8    27.6                n += 1
    20       1000000        880494         0.9     0.9            output[i] = n
    21                1           5         5.0     0.0        return output
```

Profiling memory

- Samples system's memory report via psutil
- Can do line-by-line or graph
- What is using RAM in our Julia set? What do we expect to see? What is a surprise?

Profiling memory

- Make “julia_memoryprofiler.py”,
- add `@profile` before `calculate_z...` and `calc_pure_python`
- !Set `desired_width=100` (not 1000)
- !`max_iterations` can stay at 100
- `python -m memory_profiler julia_memoryprofiler.py # from line_pr...`

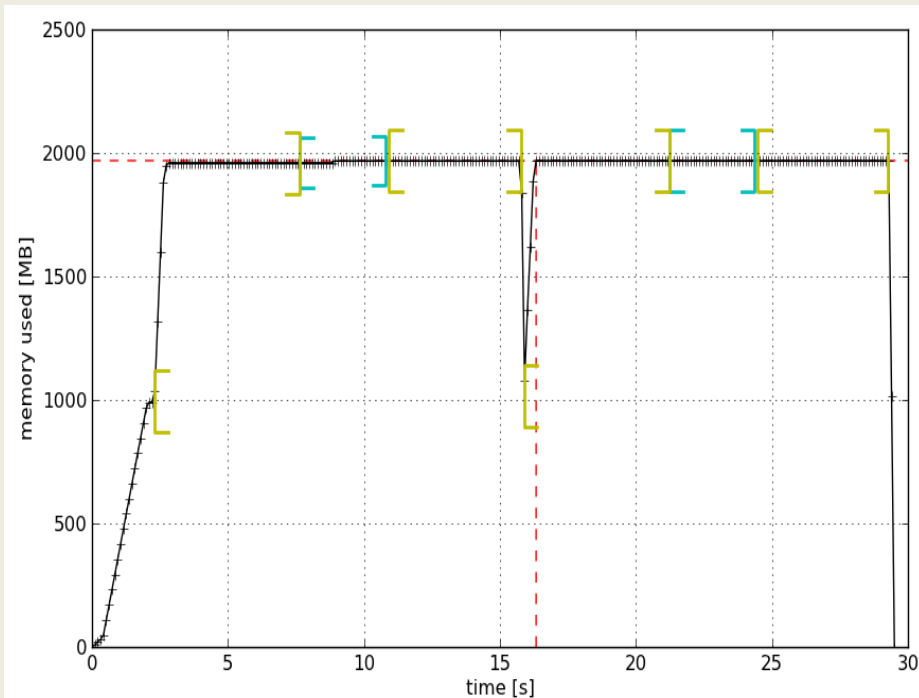
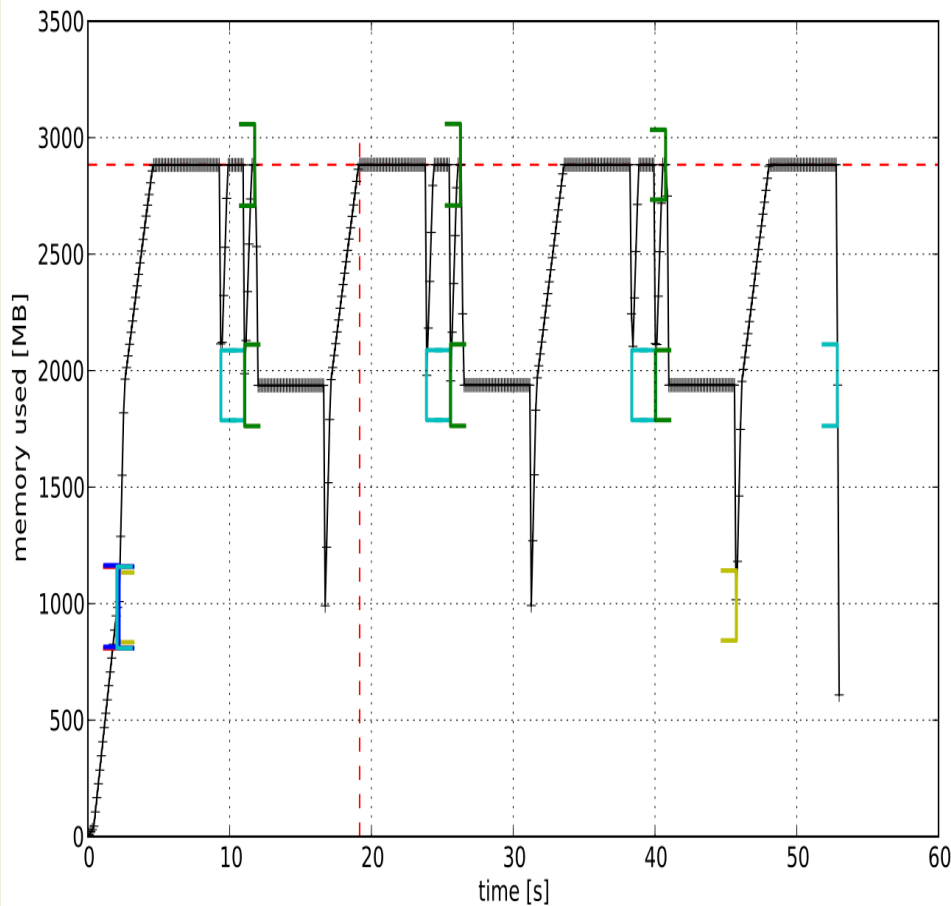
memory_profiler output

```
44 10.301 MiB 0.000 MiB zs = []
45 10.301 MiB 0.000 MiB cs = []
46 89.523 MiB 79.223 MiB for ycoord in y:
47 89.523 MiB 0.000 MiB     for xcoord in x:
48 89.523 MiB 0.000 MiB         zs.append(complex(xcoord, ycoord))
49 89.523 MiB 0.000 MiB         cs.append(complex(c_real, c_imag))
50
51 89.531 MiB 0.008 MiB     print "Length of x:", len(x)
52 89.531 MiB 0.000 MiB     print "Total elements:", len(zs)
53 89.531 MiB 0.000 MiB     start_time = time.time()
54                                     output = calculate_z_serial_purepython(max_iterations, zs, cs)
55 138.000 MiB 48.469 MiB     end_time = time.time()
56 138.000 MiB 0.000 MiB     secs = end_time - start_time
57 138.004 MiB 0.004 MiB     print calculate_z_serial_purepython.func_name + " took", secs,
```

mprof (memory_profiler)

<https://github.com/scikit-learn/scikit-learn/pull/2248>

Before & After an improvement

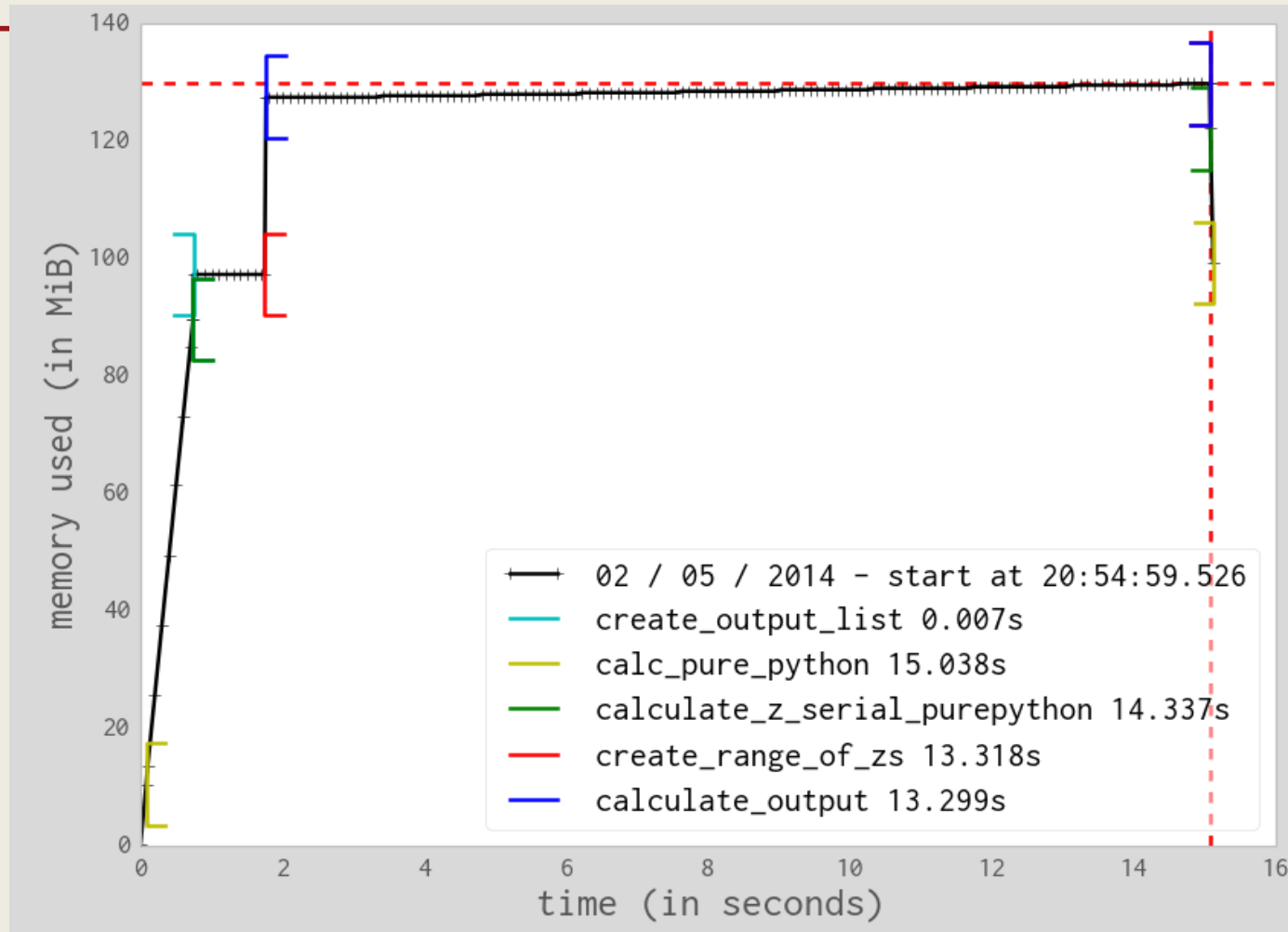


mprof – draw the mem. usage

- !desired_width=1000
- mprof run julia_memoryprofiler.py
- mprof plot # should show a graph

```
@profile
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    with profile.timestamp("create_output_list"):
        output = [0] * len(zs)
        time.sleep(1)
    with profile.timestamp("create_range_of_zs"):
        iterations = range(len(zs))
        with profile.timestamp("calculate_output"):
            for i in iterations:
```

mprof



mprof – final tweak

- What could we change the range call to?
- Make the change – how does mprof plot change?
- We could also add annotations beyond function names

Compiling with Cython

- 2007 project (forked from Pyrex *.pyx*)
- Converts annotated Python into C
- You have to do the conversion
- We'll convert the plain Python version into C (we'll do numpy version later)
- We'll import a compiled version of the function

Cython

- Make “cython” directory, copy `julia_nopil.py` in there
- Make `cythonfn.py` (it'll become `cythonfn.pyx` soon)
- Move `calculate_z` function
- “`from cythonfn import calculate_z`”

Cython

- Once we know it works, rename to cythonfn.pyx (after pyrex project)
- `cython -a cythonfn.pyx`
- open “firefox cythonfn.html”

Cython – annotated

```
1: def calculate_z(maxiter, zs, cs):
2:     """Calculate output list using Julia update rule"""
3:     output = [0] * len(zs)
4:     for i in range(len(zs)):
5:         n = 0
6:         z = zs[i]
7:         c = cs[i]
8:         while n < maxiter and abs(z) < 2:
9:             z = z * z + c
10:            n += 1

    __pyx_t_2 = PyNumber_InPlaceAdd(__pyx_v_n, __pyx_int_1); if (unl
    __Pyx_GOTREF(__pyx_t_2);
    __Pyx_DECREF_SET(__pyx_v_n, __pyx_t_2);
    __pyx_t_2 = 0;
}

11:     output[i] = n
12:     return output
```

Note InPlaceAdd and Reference Counting

Cython – make setup.py

```
from distutils.core import setup  
from Cython.Build import cythonize
```

```
setup(  
    ext_modules = cythonize("cythonfn.pyx")  
)
```


Cython

- MOVE cythonfn.py → cythonfn.pyx
- To compile:

```
python setup.py build_ext --inplace
```

note build<under>ext dashdashinplace
- We should have a .c and a .so
- python julia_nopil.py
- This won't be much faster (and why is that?)

Cython – add annotations

```
def calculate_z(int maxiter, zs, cs):  
    """Calculate output list using Julia update rule"""  
    cdef unsigned int i, n  
    cdef double complex z, c  
    output = [0] * len(zs)  
    for i in range(len(zs)):  
        n = 0  
        z = zs[i]  
        c = cs[i]
```

Cython

- How could we remove the abs operation?

Cython

- How could we remove the abs operation?
- `abs(z)` just `sqrt(real^2 + imag^2)`

Cython – expand the math

```
def calculate_z(int maxiter, zs, cs):  
    """Calculate output list using Julia update rule"""  
    cdef unsigned int i, n  
    cdef double complex z, c  
    output = [0] * len(zs)  
    for i in range(len(zs)):  
        n = 0  
        z = zs[i]  
        c = cs[i]  
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:  
            z = z * z + c  
            n += 1  
        output[i] = n  
    return output
```

Cython

- Why do we expand the math?
- *Avoid doing work we don't have to do!*
- What else is `abs(z)` doing? We're forcing more specialisation
- We can disable bounds checking (but it doesn't change much)

Cython – tradeoffs

- Probably the fastest and most reliable solution for compiling
- You have to know some C
- You have to be happy working with C
- Removes generic behaviour, specialises your code (so less flexible)
- Use unit tests!
- Can compile with debug libs, easy enough just to use print statements

numpy serial version (slow!)

- Let's replace the Python lists with numpy arrays
- Look in src/numpy_version
- Walk through the new zs code first
- np.array is fast, right?
- Try the new demo <ouch> (>2 mins!)
- What's going on?

Cython and numpy

- We let C see the block of memory inside numpy arrays
- `arr.data[0]` → first byte
- `__array_interface__.items()` for the internal guts
- No need to manage access to Python objects any more
- What else might a C compiler do without the GIL restriction?
- Let's convert the numpy version with Cython

Cython and numpy

- Start with `cythonfn.py` and `julia_nopil.py` as before
- Check they run
- Copy `setup.py` from before
- “`python setup.py build_ext --inplace`”
- It'll take >2mins to run due to dereferencing cost

Cython and numpy

```
import numpy as np
cimport numpy as np

# to compile
# python setup.py build_ext --inplace

def calculate_z(int maxiter, double complex[:] zs,
                double complex[:] cs, long[:] output):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, length
    cdef double complex z, c
    #for i in range(len(zs)):
    length = len(zs)
    for i in xrange(length):
        z = zs[i]
        c = cs[i]
        output[i] = 0
```

Cython and numpy

- Now we're back to 4 seconds
- Can you expand the math like we did before?
- Does it run faster again? (it should be slightly faster to what we had for the lists version)
- Adding early binding, type specialisation and going to the raw low level objects means C can compile it very efficiently
- Could a non-Cython colleague understand this code?

OpenMP

- What does OMP give us?
- *shared memory multiprocessing*
- Multi-platform, multi-OS, C/C++/Fortran
- We need to make the decisions
- parallel for, parallel reduce

Cython and OpenMP

- How we do annotate the loop?
- We have to tell the compiler to use OMP
- What is *static* and *dynamic* scheduling?

Cython and OpenMP

- Add “from cython.parallel import prange”
- Change the for loop:
with nogil:
 for i in prange(length,
 schedule="guided"):

Cython and OpenMP

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize
from Cython.Distutils import build_ext
ext_module = Extension("cythonfn",
    ["cythonfn.pyx"],
    extra_compile_args=['-fopenmp'],
    extra_link_args=['-fopenmp'])

setup(name = 'Cython fn',
    cmdclass = {'build_ext': build_ext},
    ext_modules = [ext_module])
```


Cython and OpenMP

- This is as fast as we can easily go!
- Fully exploits multiple cores
- Reductions are possible too

Pythran

- Somewhere between ShedSkin and Cython
- Has an annotation extension engine
- You supply the function annotation
- Works on Python and numpy variants
- Has interesting AST rebuilding and lightweight reimplemented modules
- Uses lightweight RefCounting (like CPython)
- CPython data must be copied into Pythran's memory space

Pythran

- Annotate: `#pythran export calculate_z(int, complex[], complex[], int[])`
- `pythran fn.py → fn.so`
- If you delete the `.so` then your original `.py` file will run unchanged – great for testing!

Pythran and OpenMP

- We can easily add OMP
- Add “#omp parallel for” before the for loop
- `pythran -fopenmp -march=corei7-avx cython_np.py`

Pythran specialisations

- Core library has been lightly reimplemented
- What if we take away a lot of the numpy machinery?
- It tries to auto-parallelise e.g. on a map

Pythran - tradeoffs

- Young project, very few users
- They're quick to respond
- Only some numpy modules supported
- Uses comments therefore does not disrupt code (unlike Cython)

Numba

- numpy-aware optimizing compiler
- Not a tracing JIT (unlike PyPy) but method-based (tracing is likely to be loop-based)
- Uses LLVM
- Requires a tiny bit of decoration
- GC handled by LLVM

Numba

- Add “from numba import jit”
- Add “@jit”, optionally add types

```
from numba import jit

@jit()
def calculate_z(maxiter, zs, cs, output):
```

- With the current version we have to pass in “output” from outside of the compiled function (but this hasn't always been the case)

Numba - tradeoffs

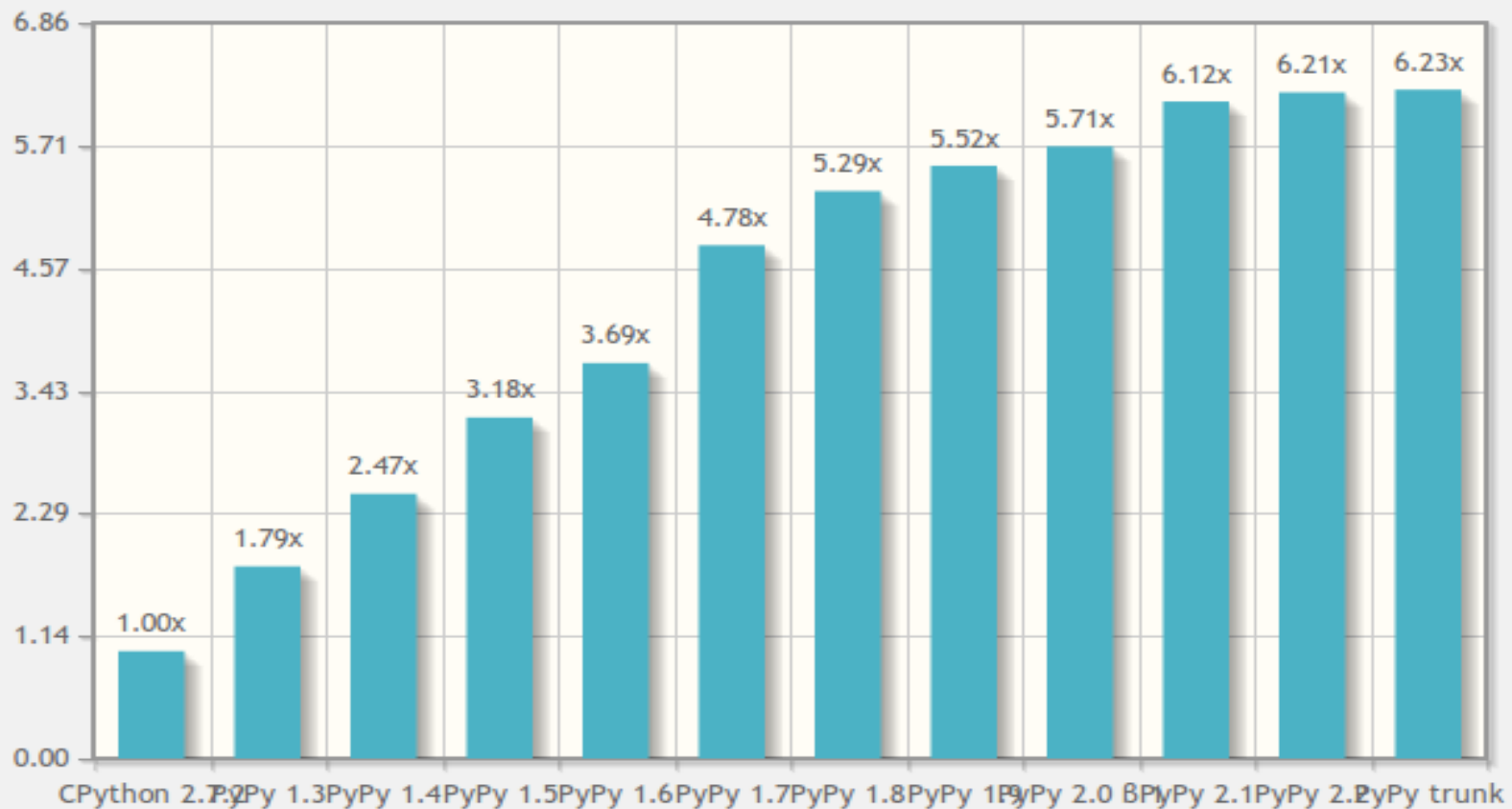
- Be aware that the API changes with each release
- Really needs Anaconda
- Note run 1 has compile cost, run 2 no additional cost
- Does nothing useful for non-numpy code (but does work)
- Somewhat mixed real-world reports
- Probably has best long-term future as 'drop in replacement' for numpy speed-ups

PyPy

- “Like CPython but 6.3* faster (ish)”
- <http://speed.pypy.org/>
- Different implementation of Python including different GC
- Tracing JIT – considers loops and frequent code paths rather than whole functions, then compiles the hot loops
- No annotation is required
- Does have a GIL
- Python 2.7 and Python 3 (beta)
- Written in RPython (restricted Python enabling easy inference of variable's type), not written in C
- Built out of Armin's psyco (32 bit JIT)

PyPy

How has PyPy performance evolved over time?



PyPy

- Run it, run with `julia_nopil.py`
- How much faster?
- Not bad for *no work at all*

Sidenote – ref counting GC in Python

- RefCounting to keep track of live objects
- When 0 references left – delete object
- This is a CPython implementation choice
- This is *not the only GC strategy*
- PyPy doesn't use RefCounting, it has a modified mark-and-sweep with nursery

PyPy

- Software Transactional Memory
- Replaceable Garbage Collectors
- Has had Java backend
- PyPy.js – RPython->C->Emscripten (C to JS via LLVM))->JS – faster than CPy but slower than PyPy
- JS & LLVM receiving lots of attention in the compiler community
- If you want to write your own efficient interpreter:
<http://www.wilfred.me.uk/blog/2014/05/24/r-python-for-fun-and-profit/>

PyPy tradeoffs

- There *is* numpy support (sort of)
- CPyExt sort-of provides access to C compiled extensions (and do we really need them?) e.g. cPickle in PyPy is not written in C any more
- CFFI is the right solution for C modules with Python + PyPy compatibility

“High Performance Python”

- I think I'm signing...
- Training courses in October in London
- pyvideo.org
- PyDataLondon meetup

