High-level low-level programming integration

R. Pastor-Satorras

Departament de Física
UPC

Eines Informàtiques Avançades / EIA

High-level vs low-level programming controversy

You might have heard about the controversy between high-level vs low-level programming languages for scientific computation

- High-level (interpreted) languages, such as Python, are very easy to program, but very slow
- Low-level (compiled) languages, such as Fortran or C, are difficult to code, but very fast

The usual conclusion is that real computational science is to be performed in low- level languages, because they are the fastest

The trade-off between high- and low-level languages

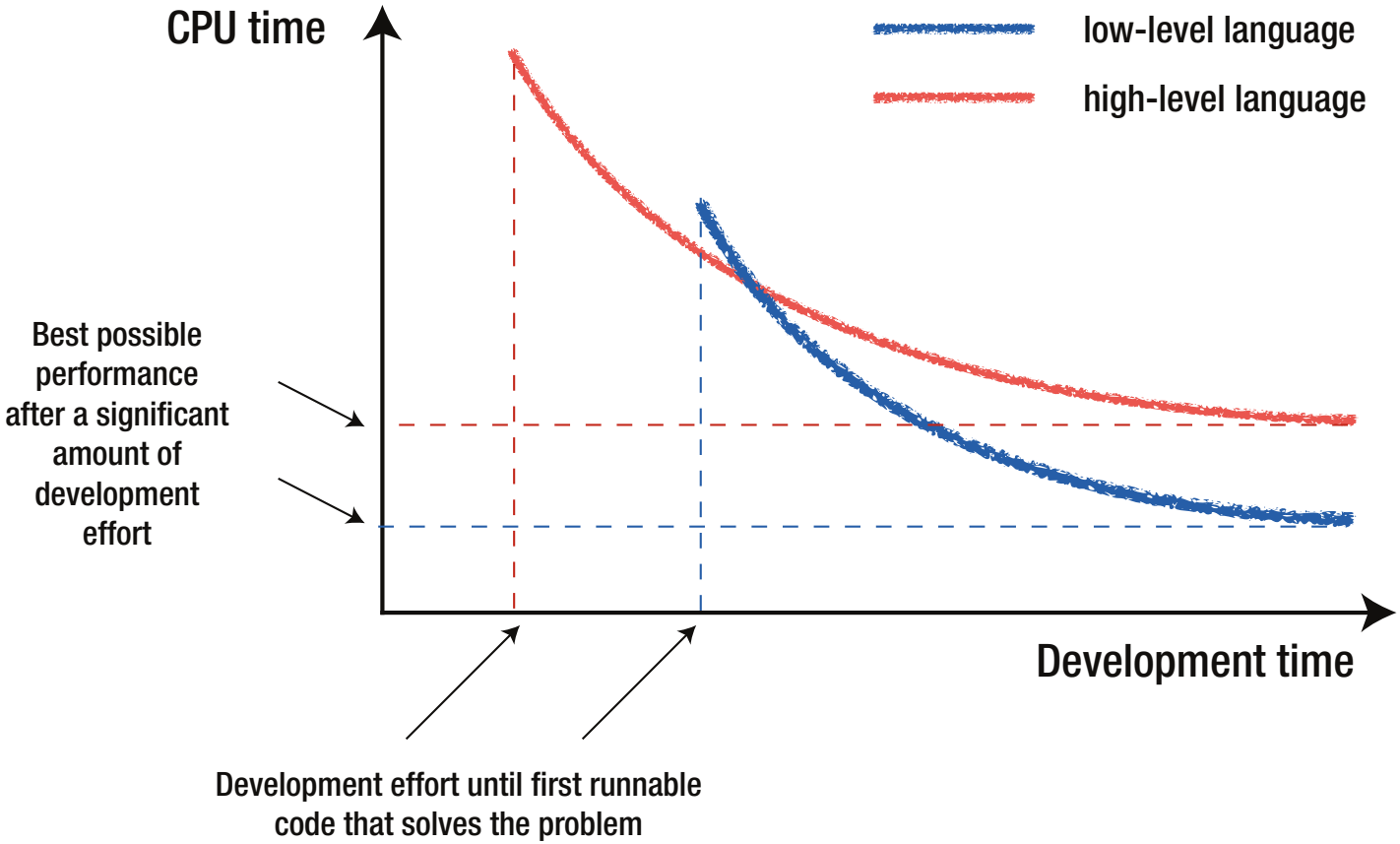
This simple conclusion however does not take into account a few hidden facts:

- Computer CPU is cheap, and becoming cheaper, and ever increasing in power
- Programmer time (your time) is expensive
- You don't always need the super-fastest code, but just a decent version, specially when you are experimenting with a new problem

The usual conclusion is that real computational science is to be performed in low- level languages, because they are the fastest

The trade-off between high- and low-level languages

A clarifying picture:



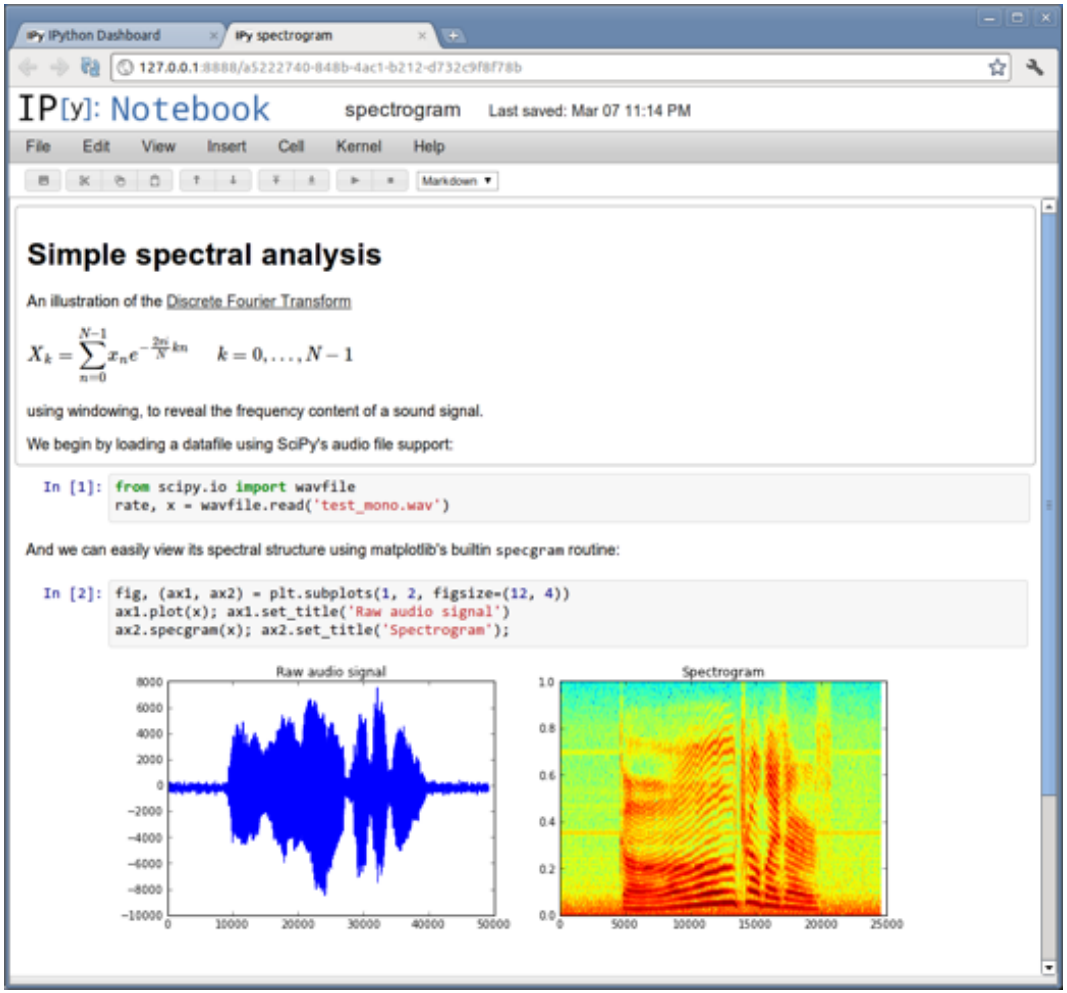
What high-level languages offers for scientific programming

In particular Python:

- Open software (free as in beer)
- Very fast development
- Programs are shorter, due to the higher level of the language (avoiding boilerplate code)
  - Less code = Less bugs
- Running is slower but writing can be much faster than a compiled language
- Easy to profile, to detect computational bottlenecks
- Huge ecosystem of user base and scientific libraries
- Many task, not strictly related to scientific computing or number crunching are easier (e.g string manipulation)

... and a huge plus

- A whole computing environment in which data analysis and graphics can be merged with calculations, i.e. **IPython**



Fortran vs Python

```python
import numpy as np

ITERATIONS = 100
DENSITY = 1000
x_min, x_max = -2.68, 1.32
y_min, y_max = -1.5, 1.5

x, y = np.meshgrid(np.linspace(x_min, x_max, DENSITY),
                   np.linspace(y_min, y_max, DENSITY))
c = x + 1j*y
z = c.copy()
fractal = np.zeros(z.shape, dtype=np.uint8) + 255

for n in range(ITERATIONS):
    print "Iteration %d" % n
    mask = abs(z) <= 10
    z[mask] = z[mask]**2 +  c[mask]
    fractal[(fractal == 255) & (-mask)] =
    254. * n / ITERATIONS

print "Saving..."
np.savetxt("fractal.dat", np.log(fractal))
np.savetxt("coord.dat", [x_min, x_max, y_min, y_max])
```

```fortran
program Mandelbrot
use types, only: dp
use constants, only: I
use utils, only: savetxt, linspace, meshgrid
implicit none

integer, parameter :: ITERATIONS = 100
integer, parameter :: DENSITY = 1000
real(dp) :: x_min, x_max, y_min, y_max
real(dp), dimension(DENSITY, DENSITY) :: x, y
complex(dp), dimension(DENSITY, DENSITY) :: c, z
integer, dimension(DENSITY, DENSITY) :: fractal
integer :: n
x_min = -2.68_dp
x_max = 1.32_dp
y_min = -1.5_dp
y_max = 1.5_dp

call meshgrid(linspace(x_min, x_max, DENSITY), &
    linspace(y_min, y_max, DENSITY), x, y)
c = x + I*y
z = c
fractal = 255

do n = 1, ITERATIONS
    print "('Iteration ', i0)", n
    where (abs(z) <= 10) z = z**2 + c
    where (fractal == 255 .and. abs(z) > 10) fractal = 2
end do

print *, "Saving..."
```

The things you can do with Python

Apart from writing general code for our numerical simulations, the many packages that are available in Python allow to do many other things

Let us see a few examples

Open a python session in your terminal and try to follow the examples

Algebra in Python with sympy

```
import sympy as sp

# symbolic variables must be  defined
sp.var("x y z  k, r1, r2")

# some variables are already defined
print "The pi number is", sp.pi

# let us define a simple function
f = x**3 + x**2 - x - 4
# solving f = 0
sols = sp.solve(f,x)
print "Solutions:", sols

# derivatives
print "Derivative:", sp.diff(f, x)

# integrals
print "An integral:", sp.integrate(sp.exp(-x)*x**(z-1), (x, 0, sp.oo))

# Laurent series
print "Laurent series:", sp.series(sp.cos(x)/sp.sin(x), x)
```

You can do all things available in `Mathematica` or `Maple` with open software. And mix them easily with numerical operations

Plotting in Python with matplotlib

```
import math
import matplotlib
import matplotlib.pyplot as plt

# first create some data, using pythonic list comprehensions
x = [i/10.0 for i in range(1, 100)]
y = [math.sin(float(i)) for i in x]
z = [math.sin(float(i)) for i in x]

# direct plot
plt.plot(x, y, "r-s")
plt.xlabel("x")  # write labels in the axis
plt.ylabel("sin(x)")
plt.show()

# some subplots
plt.subplot(2,2,1)
plt.plot(x, y, "r-")
plt.subplot(2,2,2)
plt.plot(x, y, "b-")
plt.subplot(2,2,3)
plt.plot(x, y, "g-")
plt.subplot(2,2,4)
plt.plot(x, y, "k-")

plt.show()
```

Many complex things, including 3D plots, animations, movies, etc. All inside Python, no need to go to gnuplot or other programs. Example ...

Numerical integration with scipy

$$y''(x) = \frac{(3x+2)y'(x)+(6x-8)y(x)}{3x-1}$$

```
import numpy as np
import scipy as sp
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def g(y, x):
    y0 = y[0]
    y1 = y[1]
    y2 = ((3*x+2)*y1 + (6*x-8)*y0)/(3*x-1)
    return y1, y2

# Initial conditions on y, y' at x=0
init = 2.0, 3.0
# First integrate from 0 to 2
x = np.linspace(0,2,100)
sol=odeint(g, init, x)
# Then integrate from 0 to -2
plt.plot(x, sol[:,0], color='b')
x = np.linspace(0,-2,100)
sol=odeint(g, init, x)
plt.plot(x, sol[:,0], color='b')

# The analytical answer in red dots
exact_x = np.linspace(-2,2,10)
exact_y = 2*np.exp(2*exact_x)-exact_x*np.exp(-exact_x)
plt.plot(exact_x,exact_y, 'o', color='r', label='exact')
plt.legend()

plt.show()
```

The downsize of coding in Python

Python is a very simple and easy programming language, but, when applied to scientific computing, it has a severe drawback:

- Due to the very fact that it is interpreted, it is much slower, sometimes an order of magnitude, than the corresponding algorithm coded in a compiled language

One of the main causes of the poor performance of Python is its lack of static typing

The fact that variables do not have a predefided type makes coding faster and easier, but it imposes an computation overload, since the type of each variable must be inferred at execution time

In compiled languages, the stric type of operations allows for a large number of optimizations, that are performed at compile time, and do not get reflected when excuting the code.

How to speed up Python code

Several approaches can be used to speed up Python code

- Access libraries coded in C/Fortran to use specific functions or methods
- Code specific parts of the algorithm directly in C/Fortran and link them into the Python source code
- Somehow, compile the Python code to make it run faster
  - This can be done appliying "Just-in-time" (JIT) technologies using a Low-level virtual machine (LLVM)
  - The first time the code (usually in the form of a function) is executed, it becomes compiled, and successive executions are performed on the compiled object, achieving thus better performance

When to speed up Python code

You should not start optimizing your code from the very beginning, and you should not try to optimize all of it

- Otherwise, start coding in Fortran and lose the advantages of Python

You should instead carefully profile and benchmark your code and identify the essential bottlenecks: Small pieces of code in which most of the CPU time is spent

You should then proceed to optimize only those bottlenecks, leaving the rest of the code in pure Python

- Do not struggle for days to achieve and improvement from $1\,s$ to $1\,ms$

Code modularization is essential here:

- Write your code as a sequence of functions, and optimize only those critically slow

Numpy

The main library to use in Python for scientific computing is `numpy`

The module `numpy` provides access to multidimensional arrays, stored in contiguous memory positions and having all of them the same type

`numpy` and the operations with them are hard coded in C, and therefore offer a much better performance that pure Python lists

`numpy` arrays behave as vectors, as in Matlab. Vectorialized code takes full advantage of the code written in C, and can accelerate enormously Python calculations

Additionally, there are many commodity functions in the `numpy` module, that are much faster on `numpy` arrays and provide a useful set of mathematical operations

Numpy: A simple example

Compute the scalar product of two random vectors

Pure Python version:

```
import random

n = 1000000
x = []
y = []
for i in xrange(n):
    x.append(random.random())
    y.append(random.random())

scalar_product = 0.0
for i in xrange(n):
    scalar_product += x[i]*y[i]
```

Numpy version:

```
import numpy as np

n = 1000000
x = np.random.rand(n)
y = np.random.rand(n)
np.sum(x*y)
```

Performance of the code:

- Pure Python: 733 *ms*
- Numpy: 22.8 *ms*

Scipy

As a added bonus to `numpy`, we have `scipy`, a campanion module containing a large number of subroutines useful for scientific computation. It is organized in a set of submodules, referred to different specializes areas:

- cluster: Clustering algorithms
- constants: Physical and mathematical constants
- fftpack: Fast Fourier Transform routines
- integrate: Integration and ordinary differential equation solvers
- interpolate: Interpolation and smoothing splines
- linalg: Linear algebra
- ndimage: N-dimensional image processing
- optimize: Optimization and root-finding routines
- signal: Signal processing
- sparse: Sparse matrices and associated routines
- special: Special functions
- stats: Statistical distributions and functions

The power of Scipy

A very interesting feature of scipy is that it provides bindings to 99% of all scientific functions present in GSL, with the benefit that they can be accesses in much simpler form

We have seen an example in integration (`odeint`), which is an specialized adaptive step size integration algorithm valid for most numerical integrations of differential equations

So, if your scientific computing problem involves essentially the integration of a differential equation, however complex, you can gain a huge time investment benefit in coding using only Python+scipy: The execution time in pure Python+scipy is essentially the same as Fortran/C equivalent code

But you have to work a lot less

Numba

When numpy/scipy are not an option, Numba is, by far, the simplest way to optimize Python code by jit-compiling it

One can achive this, in the simplest case, by just adding a **decorator** on top of the definition of a function

The performance of the jit-compiled code can be improved by adding additional information of the types of the parameters and output of the function, but in many cases it is not necessary

Caveat:

Numba not always provides a good improvement, it can even slow down execution
It is advisable specially when there are lots of nested loops in the code
You must perform tests of efficiency

Let us see numba in action

Example: The Mandelbrot set

A classical example of computer science and graphics is the Mandelbrot set, defined as the set of points in the complex plane that do not approach infinity under the iterative map

$$z_{n+1} = z_n^2 + c$$

That is, a complex number c is part of the Mandelbrot set if, starting form $z_0 = 0$, the application of the iteration leaves z_n bounded for any large value of n.

One way to depict it graphically is using the **escape time algorithm**:

- Fixing a number of iterations, for every point in the complex plane c one checks for which value of m the absolute value of z_m becomes larger than a given threshold
- Then the color associated to the complex number is given by m.

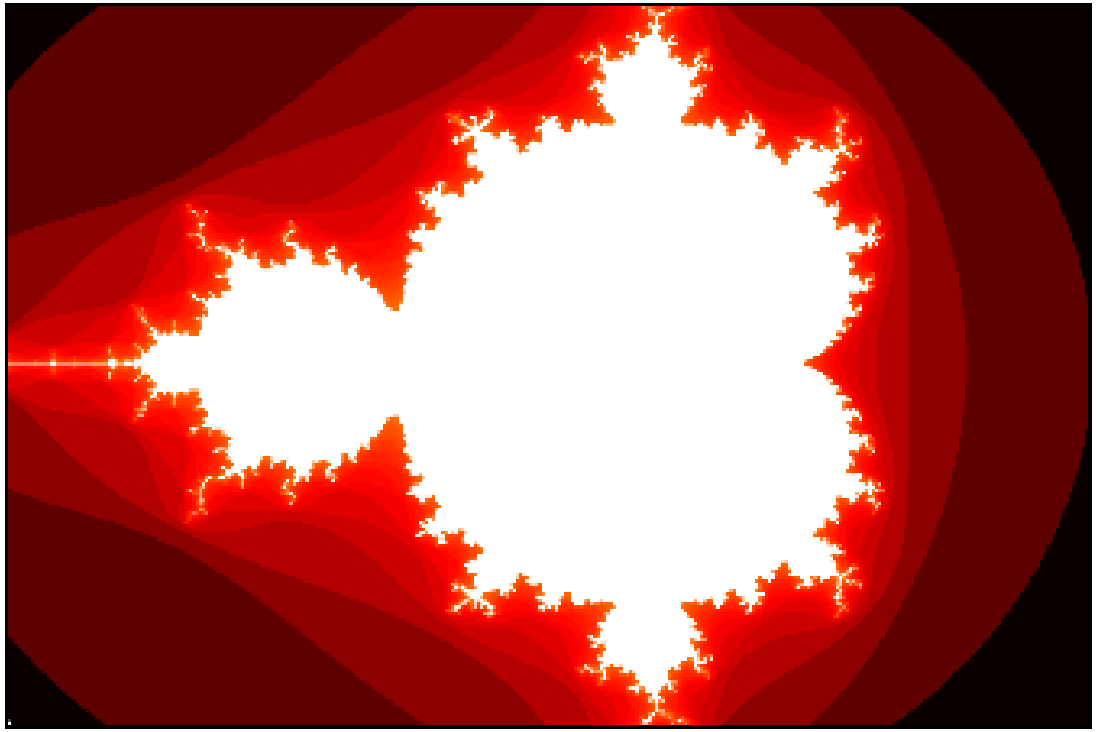
The Mandelbrot set: Python and Numpy

```
import numpy as np

def create_mandelbrot_python(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = color_mandelbrot_python(real, imag, iters)
            image[y, x] = color

    return image

def color_mandelbrot_python(x, y, max_iters):
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z**2 + c
        if abs(z)**2 >= 4:
            return i

    return 255

# create the image and plot it
image = np.zeros((500, 750), dtype=int)
# imshow shows an image encoded in colors for each pixel
m = create_mandelbrot_python(-1.75, 1.0, -1.0, 1.0, image, 20)
```

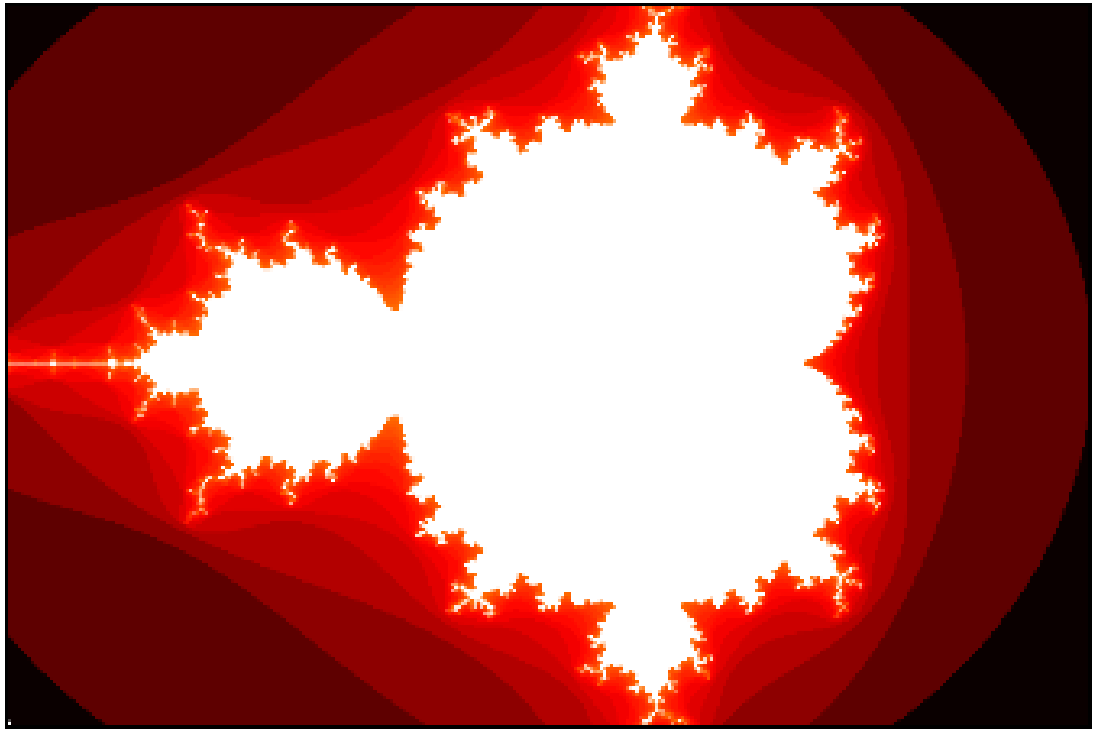
The Mandelbrot set: Numba

```
import numpy as np
from numba import jit

@jit
def create_mandelbrot_python(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = color_mandelbrot_python(real, imag, iters)
            image[y, x] = color

    return image
@jit
def color_mandelbrot_python(x, y, max_iters):
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z**2 + c
        if abs(z)**2 >= 4:
            return i

    return 255

image = np.zeros((500, 750), dtype=int)
m = create_mandelbrot_python(-1.75, 1.0, -1.0, 1.0, image, 20)
```

The Mandelbrot set: Output

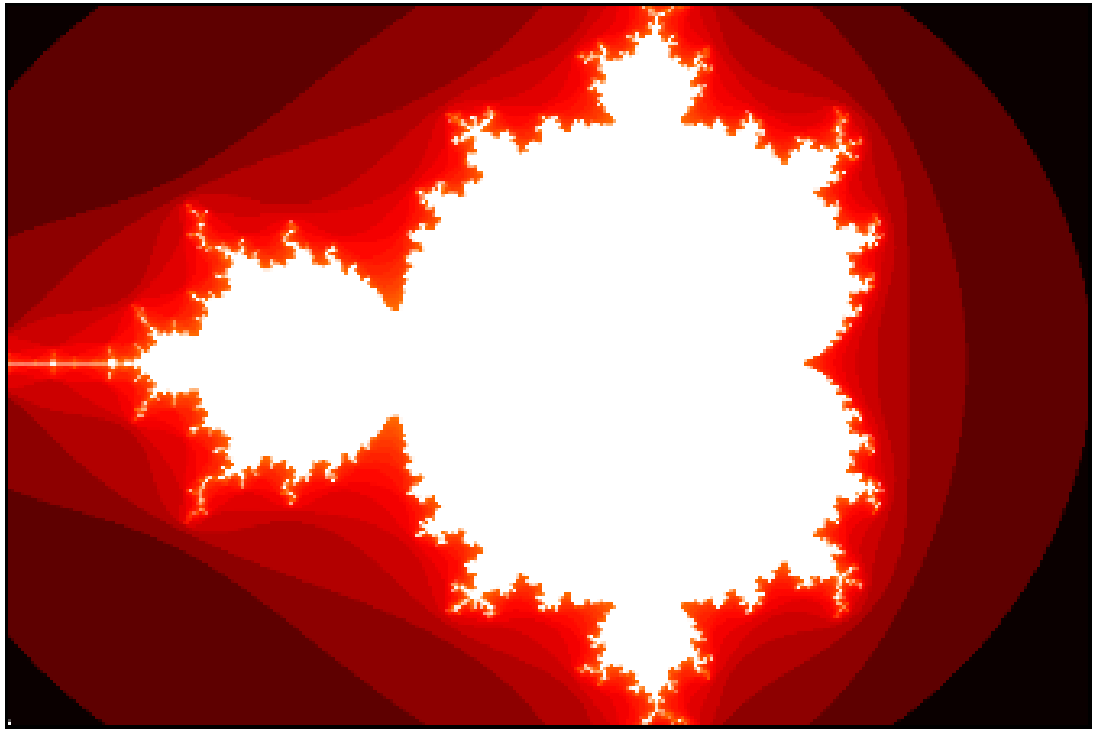


The Mandelbrot set: Output



- Execution time python + numpy: $4.46\,s$

The Mandelbrot set: Output



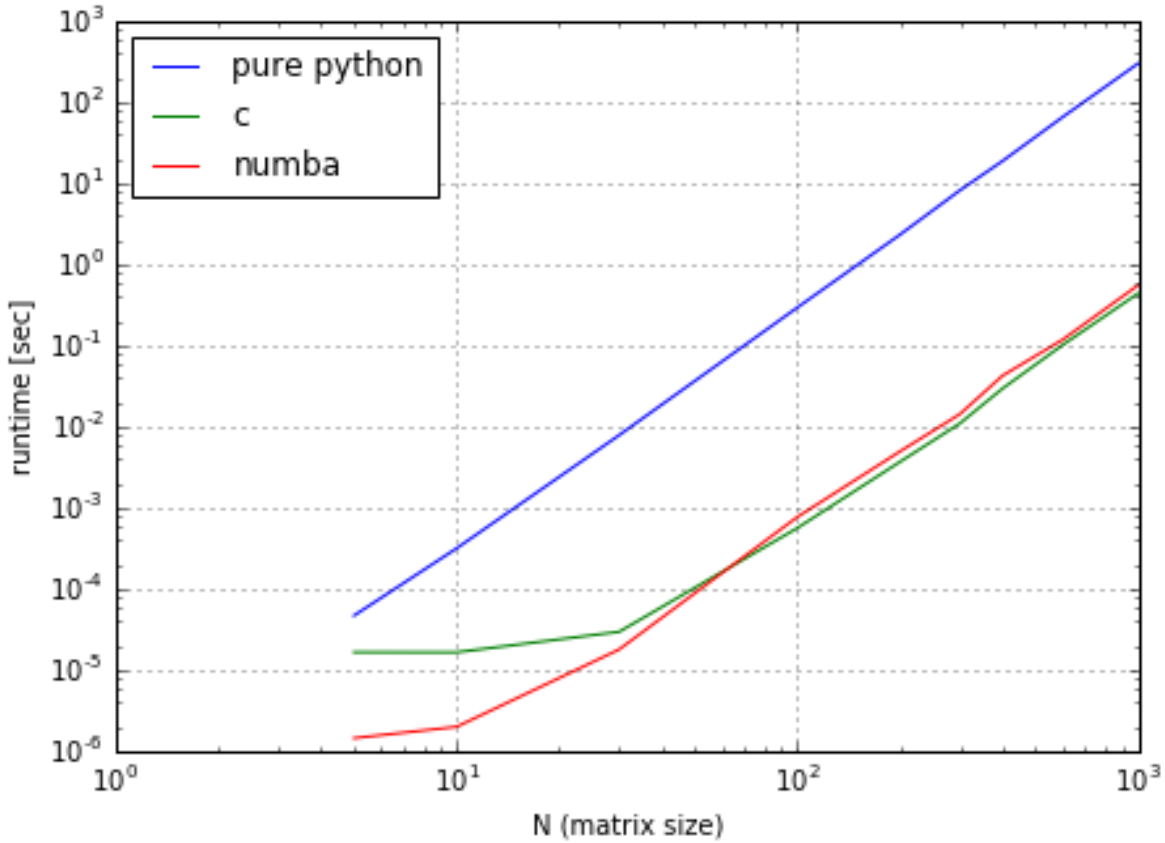
- Execution time python + numpy: $4.46\,s$
- Execution time Python + Numba: $0.13\,s$

Python vs C vs Numba comparison in a real problem

Chart comparing Python, Python with Numba and C for a the problem of LU factorization of random square matrices

Python vs C vs Numba comparison in a real problem

Chart comparing Python, Python with Numba and C for a the problem of LU factorization of random square matrices



f2py

`f2py` is a program that automatically wraps fortran code to use it into Python

We can compile fortran code and embed it into a module that we can then import into a Python program

`f2py` is part of numpy, so it can easily interact with it.

Warning

Notice the `f2py` admits only old fortran 77, not the new versions (at least in my implementation of `f2py`)

A simple example

How to do it

Let us first create a Fortran module defining a function:

```
!!
!!  Fortran code in file prod.f
!!  It computes the scalar product of two
!!  real vectors of lenght N
!!

real function prod(x, y, n)
  real x(n)
  real y(n)
  integer, intent(in) :: n


  prod = 0.0
  do i=1,n
     prod = prod + x(i)*y(i)
  enddo

end function prod
```

This file must be compiled in order to create a module that can be access by Python, using the command:

```
:> f2py -c prod.f -m prodf
```

This command creates a module `prodf` that can be loaded into a Python program with the command

```
import prodf
```

The content of `prodf` (the function `prod`) can be accessed using the standard python notation for objects

```
prodf.prod(x, y, N)
```

Cython

`Cython` is both a language (a superset of Python) and a library

It allows to start with pure Python, and add annotations about the type of the variables

Cython translates the Python code into C and compiles it into a module that we can later use into any Python program. The compiled part can achieve C-like speed

It can be used in a two-fold way:

- Translate a pure Python code into C and then compile it
- Add pieces of code written in C into an otherwise pure Python code

Warning

`Cython` is extremely powerful but difficult to master. Beware

A simple example: The Fibonacci series

How to do it

Let us first create a some function in Python (notice file extension):

```
#
#  Python code stored in file fib.pyx
#

def fib(n):
    """Print the Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a + b
```

Now, create a `setup.py` file to compile this code:

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

setup(
    ext_modules=cythonize("fib.pyx"),
)
```

Now we have to build the `cython` expression. With the setup file, we can do it easily:

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

This command creates a module `fib` that can be loaded into a Python program with the command

```
import fib
```

The content of `fib` (the function `fib`) can be accessed using the standard Python notation for objects

```
fib.fib(10)
```

Improving Cython

The performance of `Cython` can be improved using numpy arrays, as well as adding annotations to the types of the objects involved, giving the compilers more hints about how to implement more aggressive optimizations

```
#
#  Cython function with type annotations
#

cimport numpy as np
def c_scalar_product_python_2(np.ndarray[np.float64_t, ndim=1] x, np.ndarray[np.float64_t, ndim=1] y):
    cdef float scalar = 0.0
    cdef int i
    for i in xrange(len(x)):
        scalar += x[i]*y[i]

        return scalar
```

The best of two worlds

The best of two worlds

A programming language to rule them all

The Julia programming language

Quoting the creators

We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

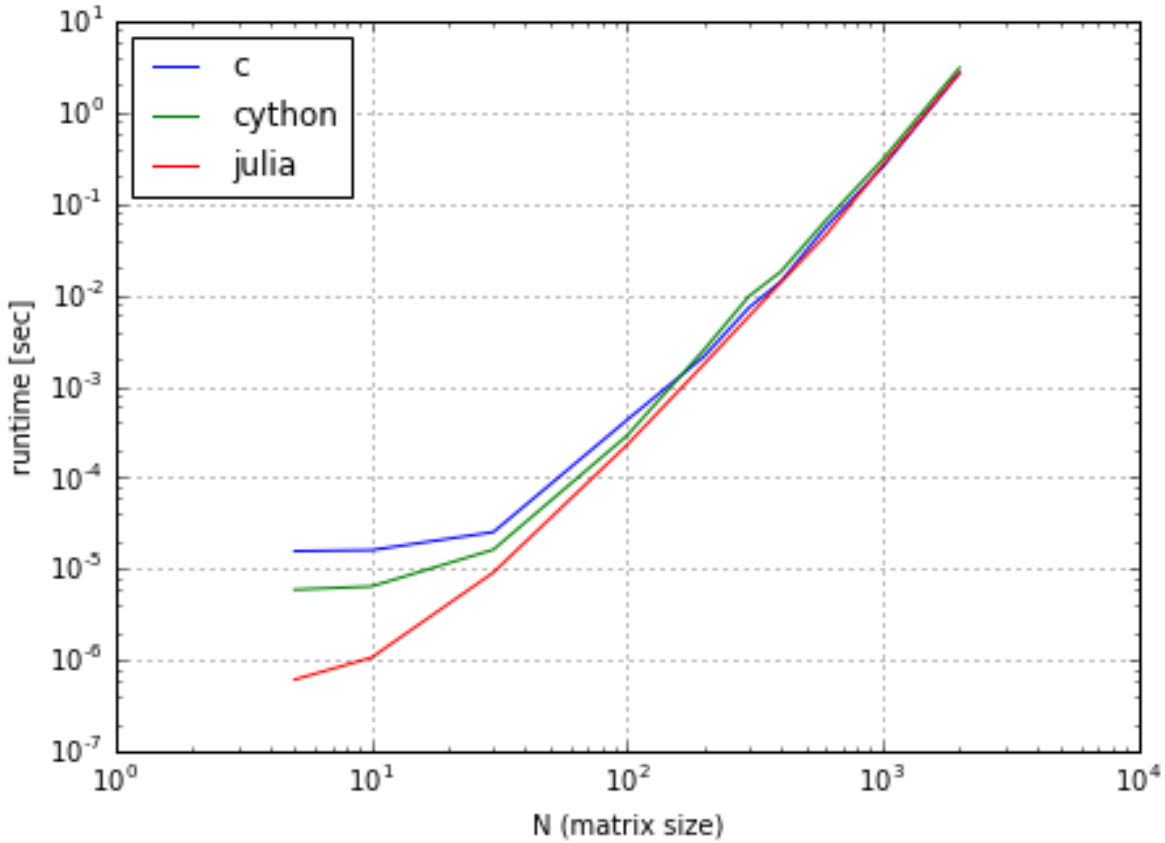
(Did we mention it should be as fast as C?)

The Julia programming language

- Julia is a high-level dynamic programming language designed to address the needs of high-performance numerical analysis and computational science while also being effective for general-purpose programming
- It is dynamic, untyped and interpreted; but it can be optionally typed for performance
- It is designed to be an easy-to-use and learn, elegant, clear and interactive. Its syntax is very similar to Matlab and fortran
- It has built-in and simple to use concurrent and parallel capabilities
- Julia has no static compilation step. It achieves its speed generating machine code just-in-time by an LLVM-based JIT compiler (in the same way as `numba`)
- The whole language is designed to achieve maximal performance by the JIT compiler

The speed of Julia

Chart comparing Julia, C and Python with Cython for the problem of LU factorization of random square matrices



The downsize of Julia

Julia has however a clear downsize: It is too young

At present, the version that can be downloaded is 0.6.2 (updated recently!)

It is young and in beta version, and therefore it is still evolving: Books written last year focusing on versions 0.3.5 and 0.4 are already obsolete, at least in the details

- Implementations are changing
- Functions and external packages are changing

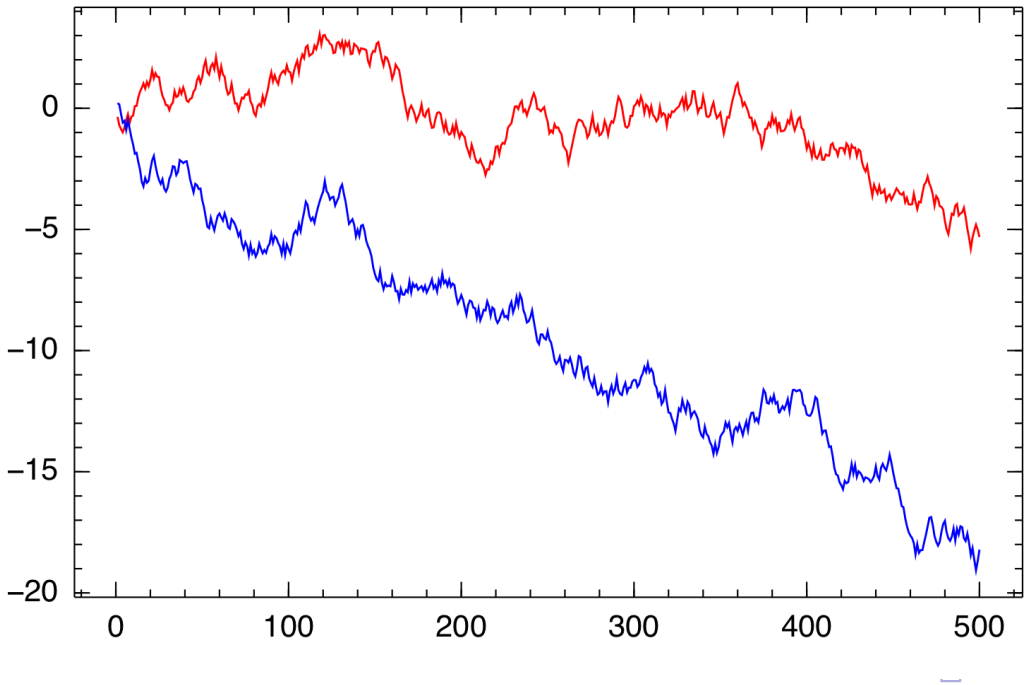
It is by all means possible to use Julia for production software (i.e. run real simulations), but be ready to relearn a few things in the course of the next years until the final stable release 1.0 is out

An example of Julia code

```
# function to calculate the volume of a sphere
function sphere_vol(r)
    # julia allows Unicode names (in UTF-8 encoding)
    # so either "pi" or the greek letter can be used
    return 4/3*pi*r^3
end

# functions can also be defined more succinctly
quadratic(a, sqr_term, b) = (-b + sqr_term) / 2a

# calculates x for 0 = a*x^2+b*x+c, arguments types can be defined in function definitions
function quadratic2(a::Float64, b::Float64, c::Float64)
    # unlike other languages 2a is equivalent to 2*a
    # a^2 is used instead of a**2 or pow(a,2)
    sqr_term = sqrt(b^2-4a*c)
    r1 = quadratic(a, sqr_term, b)
    r2 = quadratic(a, -sqr_term, b)
    # multiple values can be returned from a function using tuples
    # if the return keyword is omitted, the last term is returned
    r1, r2
end

vol = sphere_vol(3)

@printf "volume = %0.3f\n" vol


quad1, quad2 = quadratic2(2.0, -2.0, -12.0)
println("result 1: ", quad1)

println("result 2: ", quad2)
```

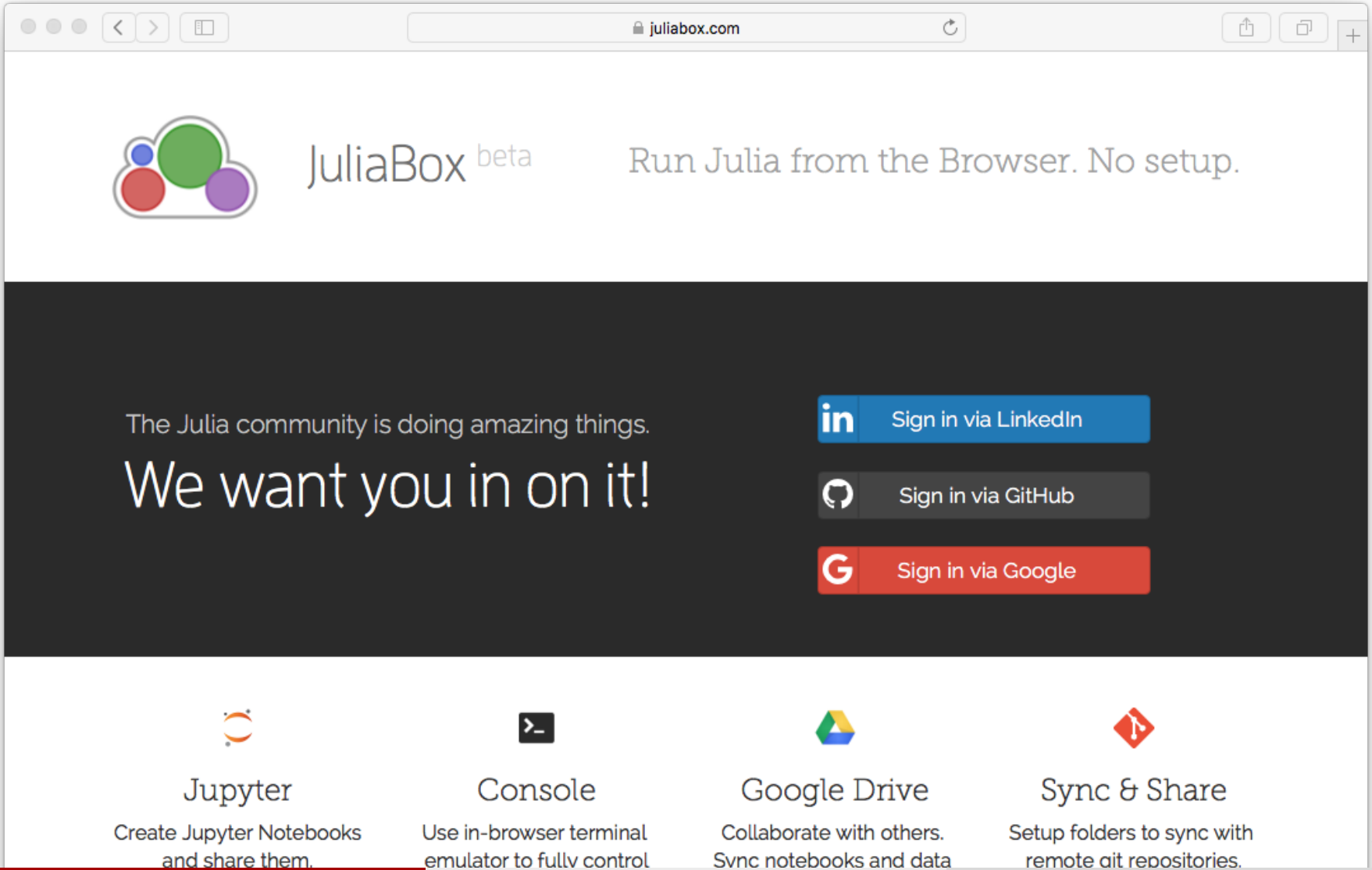
Plotting with Julia

```
# install one package (e.g. Winston) and all its dependencies:

Pkg.add("PyPlot")
using PyPlot

figure(width=600, height=400)
# plot some data
pl = plot(cumsum(rand(500) .- 0.5), "r", cumsum(rand(500) .- 0.5), "b")

# save the current figure
savefig("randomwalks.png")
# .eps, .pdf, & .svg are also supported
```



Using Julia

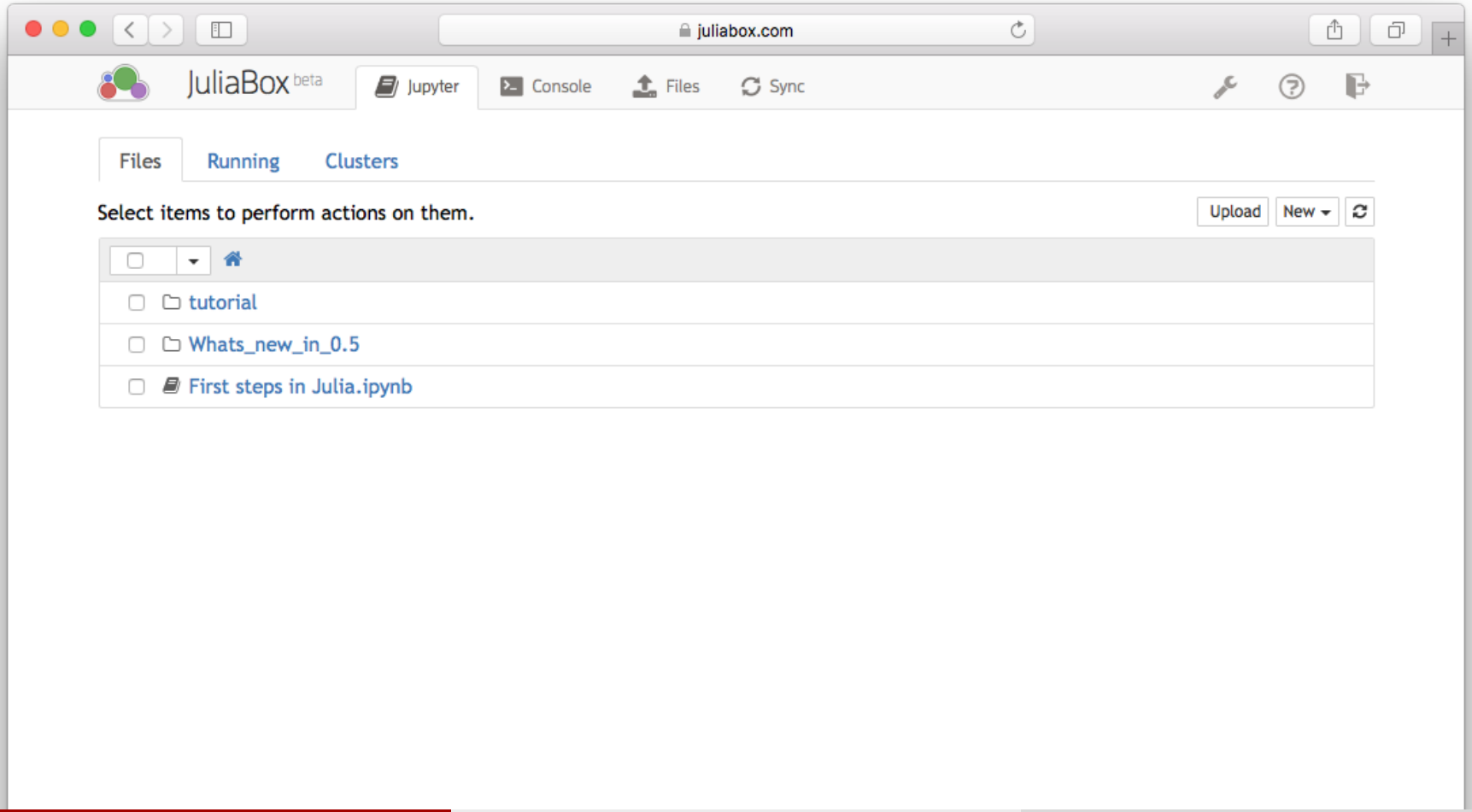
You can install Julia locally or play with it online at www.juliabox.com



JuliaBox

In JuliaBox, you access a Julia Jupyter notebook

in a nutshell, open a `New` notebook, type code in the cells and execute with `Shift+Intro`



Good places to learn Julia

`Hands on Julia`: https://github.com/dpsanders/hands_on_julia

A set of Julia notebooks, which you can study, download on a local installation, or simply cut and paste on Juliabox

Notice however that it is a bit outdated

For a more updated intro, you can watch the official **Intro to Julia** video

https://www.youtube.com/watch?v=zJ_d3oWuogw