Python: The final battle

Today:

1. Solutions to example problems
2. Performance with Cython
3. Interactive plots

Basic coding exercise

Write a small Python program containing the following functions:

1. `factorial`: Given an integer, returns its factorial
2. `is_prime`: Given an integer, returns `True` if the number is prime, `False` otherwise. Hint: Python modulo operator: `x % 2`.
3. `deriv_fwd`: Given a function f, a position x and a step size h, returns the numerical derivative $(f(x+h) - f(x))/h$.

Basic coding exercise: factorial

```
def factorial(n):
    """ Computes the factorial of n. """
    if n > 0:
        # recursive
        return factorial(n-1)*n
    else:
        # stop recursion
        return 1

print "Factorial 5:", factorial(5)
print "Factorial 10:", factorial(10)
print "Factorial 0:", factorial(0)
```

Basic coding exercise: is_prime

```
from math import sqrt

def is_prime(n):
    """ Checks if n is prime"""
    # check up to sqrt(n), including sqrt(n)!
    for i in range(2, int(sqrt(n))+1):
        if n % i == 0:
            # found divisor, break
            break
    else:
        # no break occured -> prime
        return True
    return False

print "13 is prime:", is_prime(13)
print "49 is prime:", is_prime(49)
print "1301081 is prime:", is_prime(1301081)
```

Basic coding exercise: deriv_fwd

```
def deriv_fwd(f, x, h):
    """ Computes the numerical derivative of
    f at x using forward difference"""
    return (f(x+h)-f(x))/h

# import some functions for testing
from math import sqrt, sin

print "Derivative of sqrt at x=1.0:", \
    deriv_fwd(sqrt, 1.0, 1E-4)
print "Derivative of sin at x=0:", \
    deriv_fwd(sin, 0.0, 1E-4)
```

Array exercises

Note: Prefer array computations over `for` loops, as they are 100–300 times faster!

Array exercises

Note: Prefer array computations over `for` loops, as they are 100–300 times faster!

Given a function f and a discretized interval x:

1. Write a function that computes the numerical derivative $(f(x+h) - f(x))/h$ for all values in x.

2. Write a function that returns the positions of all local minima in x.

Here, a discretized interval means an array of equidistantly spaced values, e.g. obtained from `x = linspace(0.0, 10.0, 100)`.

Hint: Both functions can be implemented without loops.

Forward derivative

```
def deriv_fwd_array(f, x):
    """Computes the derivative of f
    for all points in x. """
    h = x[1]-x[0]  # get the step size
    dx = zeros_like(x)  # array for result
    # compute the first N-1 values at once
    dx[:-1] = (f(x[1:])-f(x[:-1]))/h
    # the last one separately
    dx[-1] = (f(x[-1]+h)-f(x[-1]))/h
    return dx

N = 1000  # number of discretization points
x = linspace(0.0, 10.0, N)
df = deriv_fwd(sin, x)
error = sum(abs(df-cos(x)))/len(x)
print "Numerical error of deriv:", error
```

Local minima

```
def get_minima(f, x):
    """ finds local minima of f """
    f_x = f(x[1:-1])     # values
    f_prev = f(x[:-2])   # values before
    f_next = f(x[2:])    # values after
    # local minima: value before and value
    # after is larger
    indices = logical_and(f_x < f_prev,
                          f_x < f_next)
    x = x[1:-1]  # the values used above
    return x[indices]

x = linspace(0.0, 15.0, N)
print "Local minima:", get_minima(sin, x)
print "Expected:", 1.5*pi, 3.5*pi
```

Final exercise: Precision of numerical derivative

Different numerical methods for approximate derivatives:

1. forward difference: $f'(x) = \frac{1}{h}\left(f(x+h) - f(x)\right) + \mathcal{O}(h)$
2. central difference: $f'(x) = \frac{1}{2h}\left(f(x+h) - f(x-h)\right) + \mathcal{O}(h^2)$
3. extrapolated difference:

$$f'(x) = \frac{1}{6h}\left[8\left(f(x+\frac{h}{2}) - f(x-\frac{h}{2})\right) - \left(f(x+h) - f(x-h)\right)\right] + \mathcal{O}(h^4)$$

.

- Compute the derivate of $\sin(x)$ at the point $x_0 = \frac{\pi}{6}$ using the three methods above for different values of $h = 10^{-14} \ldots 1$.
- Compute the *relative* error to the analytic result and visualize this error in a double-logarithmic plot.
- Additionally, plot the expected scaling behavior h^α in the same figure. Identify the optimal value h (order of magnitude).

Derivative Functions

```
def forw_diff(f, x, h):
    """ forward difference """
    return 1.0/h*(f(x+h) - f(x))

def cent_diff(f, x, h):
    """ central difference """
    return 0.5/h*(f(x+h) - f(x-h))

def extr_diff(f, x, h):
    """ extrapolated difference """
    d = (8*(f(x+h/2.0) - f(x-h/2.0)) -
         (f(x+h) - f(x-h))) / (6.0*h)
    return d
```

Derivative Functions

```
def forw_diff(f, x, h):
    """ forward difference """
    return 1.0/h*(f(x+h) - f(x))

def cent_diff(f, x, h):
    """ central difference """
    return 0.5/h*(f(x+h) - f(x-h))

def extr_diff(f, x, h):
    """ extrapolated difference """
    d = (8*(f(x+h/2.0) - f(x-h/2.0)) -
         (f(x+h) - f(x-h))) / (6.0*h)
    return d
```

Note: Those function also work element-wise for arrays `h`

Derivative Functions

```
h = logspace(-14, 0, 100)
x0 = pi/6
df = cos(x0)  # the analytic result

# compute the rel error for all h at once
err_fw = abs((forw_diff(sin, x0, h) - df)/df)
err_ce = abs((cent_diff(sin, x0, h) - df)/df)
err_ex = abs((extr_diff(sin, x0, h) - df)/df)

plt.loglog(h, err_fw, 'b',
    label='Forward Difference')
plt.loglog(h, err_ce, 'r',
    label='Central Difference')
plt.loglog(h, err_ex, 'g',
    label='Extrapolated Difference')
plt.show()
```

Cython

Cython

- Basic idea: translate Python code to C, compile and then import the binary back in Python programs.
- Usual approach: "cythonize" only the performance critical functions of your script.
- With Cython you can:
  - Do everything Python can do.
  - Add type information for more efficient C code (`cdef`).
  - Import C functions (`cimport`).
  - Efficiently access `numpy` arrays.
- Cython modules: `*.pyx` endings

A simple example: Standard Map

$$\bar{x} = x + \bar{p} \mod 2\pi, \quad \bar{p} = p + K \sin x \mod 2\pi.$$

A simple example: Standard Map

$$\bar{x} = x + \bar{p} \mod 2\pi, \quad \bar{p} = p + K \sin x \mod 2\pi.$$

Python implementation:

```
def std_map_python(x0, p0, K, N):
    """ Standard Map on the torus
        x = 0..2pi, p = -pi..pi """
    x = np.empty(N)
    p = np.empty(N)
    x[0] = x0
    p[0] = p0
    for n in xrange(1, N):
        p[n] = (p[n-1] + K*np.sin(x[n-1]) \
                + np.pi) % (2*np.pi) - np.pi
        x[n] = (x[n-1] + p[n]) % (2*np.pi)
    return x, p
```

Standard Map: Python performance

```
M, N, K = 25, 100000, 1.5
initial_x = 4.0*np.ones(M)
initial_y = np.linspace(-np.pi, np.pi, M,
                        endpoint=False)
for i in xrange(M):
    start = time.clock()
    x, p = std_map_python(initial_x[i],
                          initial_y[i], K, N)
    runtime = (time.clock() - start)*1000
    print "Trajectory %d: %.3f ms" % (i, runtime)
    plt.plot(x, p, ',')
```

Standard Map: Python performance

```
M, N, K = 25, 100000, 1.5
initial_x = 4.0*np.ones(M)
initial_y = np.linspace(-np.pi, np.pi, M,
                        endpoint=False)
for i in xrange(M):
    start = time.clock()
    x, p = std_map_python(initial_x[i],
                          initial_y[i], K, N)
    runtime = (time.clock() - start)*1000
    print "Trajectory %d: %.3f ms" % (i, runtime)
    plt.plot(x, p, ',')
```

Performance:

```
$ python std_map.py
Trajectory 0: 461.257 ms
Trajectory 1: 451.459 ms
Trajectory 2: 449.373 ms
...
```

Standard Map: Cython implementation

```
import numpy as np
from libc.math cimport sin, fmod

def std_map_cython(x0, p0, K, N):
    cdef double[:] x = np.empty(N)
    cdef double[:] p = np.empty(N)
    cdef int n
    cdef double pi = np.pi
    cdef double k_c = K

    x[0] = x0
    p[0] = p0
    for n in xrange(1, N):
        p[n] = fmod(p[n-1]+k_c*sin(x[n-1]) +
                    3*pi,  2*pi) - pi
        x[n] = fmod(x[n-1]+p[n] + 2*pi, 2*pi)
    return x, p
```

Standard Map: Cython performance

```
# generate and compile Cython code
import pyximport
pyximport.install(setup_args={'include_dirs':
                              [np.get_include()]})
from std_map_cython import std_map_cython
# ...
for i in xrange(M):
    x, p = std_map_cython(initial_x[i],
                          initial_y[i], K, N)
```

Standard Map: Cython performance

```
# generate and compile Cython code
import pyximport
pyximport.install(setup_args={'include_dirs':
                      [np.get_include()]})
from std_map_cython import std_map_cython
# ...
for i in xrange(M):
    x, p = std_map_cython(initial_x[i],
                          initial_y[i], K, N)
```

Performance:

```
$ python std_map.py
Trajectory 0: 16.149 ms
Trajectory 1: 6.897 ms
Trajectory 2: 7.205 ms
Trajectory 3: 7.081 ms
...
```

Standard Map: Python vs Cython

Python: 450 ms
Cython: 7 ms
Factor 65 speed-up

Standard Map: Python vs Cython

Python: 450 ms
Cython: 7 ms
Factor 65 speed-up

Check the generated C code:

```
$ cython -a std_map_cython.pyx
```

Generates `std_map_cython.html` with useful source code information:

Yellow stuff: not efficient!

Interactive plots

Interactive plots

- connect functions to user events (e.g. mouse clicks/moves, pressed keys)
- in `matplotlib`: `plt.connect` function
- fast and easy wayt to create interactive plots

Interactive plots

- connect functions to user events (e.g. mouse clicks/moves, pressed keys)
- in `matplotlib`: `plt.connect` function
- fast and easy wayt to create interactive plots
- BUT: not quite a real GUI, for GUIs: TkInter, WxPython, ...

Interactive Standard Map

```
def trajectory(event):
    """ Iterates and plots new trajectory """
    # check if left mouse click inside axis
    if (event.button==1 and event.inaxes):
        print event.xdata, event.ydata, K, N
        x, p = std_map(event.xdata,
                       event.ydata, K, N)
        plt.plot(x, p, ls='', marker='o',
                 mew=0, ms=2)
        plt.draw()  # update plot

# set up plot window ...

# call `trajectory` when mouse is clicked
plt.connect('button_press_event', trajectory)

plt.show()
```