# Modern scientific computing - problem set

***Release 0.1***

**Juan Carlos Cardona, Fernando Pérez, Diego Restrepo y Jorge Iván Zuluaga**

February 19, 2010

## Contents



We present below a list of selected problems intended to challenge the skills of Scientific Python Programmers. Problems are sorted according to their level of complexity and the increasing number of required skills needed to solve them.

Try to solve first the simpler problems and go next through the more complex ones.

## 1 Prime numbers and the sieve of Erathostenes

**Level**: Basic (language,basic libraries)

The sieve of Erathostenes is a simple and famous algorithm for factoring prime numbers. You will find a nice explanation of the algorithm in Wikipedia.

Try to implement it in Python, and think of different data structures you may use to do this.

Time the various implementations, do some analysis of the run times of each.

## 2 Monte Carlo integration

**Level**: Basic (language,basic libraries)

Compute $\pi$ via Monte Carlo integration. To do this, think of a function whose integral is related to $\pi$ (e.g. $\int_0^1 1/(1+x^2)dx$), and then compute this integral via Monte Carlo integration.

Try several different functions and compare.

## 3 Cellular automata

**Level**: Basic (language, basic libraries, plotting)

Compute the evolution of a 2D Cellular Automaton (CA) given a random initial state, a simple set of cell update rules and a given boundary condition.

As an example create one of the most famous CA, The Life Game:

1. Generate a 2D matrix of integer values between 0 and 3 (0: dead cell, 1: recently born, 2: alive, 3: recently dead.)
2. Update the state of each cell using the following conditions:

   1. If a cell is dead but there are exactly 3 neighbor cells alive, she will born (passes from state 0 to state 1)

   2. If a cell is alive and there are 2 or 3 neighbor cells alive she stays alive (passes from state 1 to 2 or stays in state 2), otherwise it dies (passes from state 2 to 3).
3. Use a *cold* boundary condition, i.e. all the cells beyond the boundary of the CA are dead.
4. Plot the state of the CA on each step.

**Bonus**: Create an animation of the evolution of the Automaton.

**Bonus: Change the boundary condition. Options are: *hot* boundary** condition (all the cells outside are alive), periodic boundary condition or a reflecting boundary condition.

References: http://en.wikipedia.org/wiki/Cellular_automaton

## 4 Simple fitting

**Level**: Medium (scientific libraries, plotting)

Test different methods of fitting using a signal with known source.

1. Generate data pairs $(x, y)$ in the range $x \in [0, 4]$ from the target function:

$$y(x) = a \exp(\alpha x) + k$$

   Using the following choice of constants: $a = 2, \alpha = -0.75, k = 0.1$.

2. Add zero-mean gaussian noise to $y$, with amplitude $0.1$ to create a noisy version of the signal.

3. Fit the noisy data using the following methods: least-square fit to the analytical form, splines (from `scipy.interpolate`), polynomial fit (from `numpy`) of orders 0, 1 and 2.

4. Plot the data and the results from the fitting procedures and Compare graphically the fitting methods.

## 5 Fractals

**Level**: Medium (plotting)

A simple ilustration of fractal curve is the von Koch curve K.

$$K = S_\infty \tag{1}$$

This curve is constructed by the following procedure mentioned below: In the iteration $S_0$ we take a line segment of longitud L and split it in three equal parts and the middle one is substracted. Then we replace the missing segment with the other two sides of an equlateral triangle.

The remaining iterations consist in repeat the procedure for each one or the segments in the previous approximation.

Write a python script to plot the six first iteration in the construction of the von Koch curve as ilustrated in the figure.

## 6 The least action principle

**Level**: Medium (scientific libraries, algorithms)

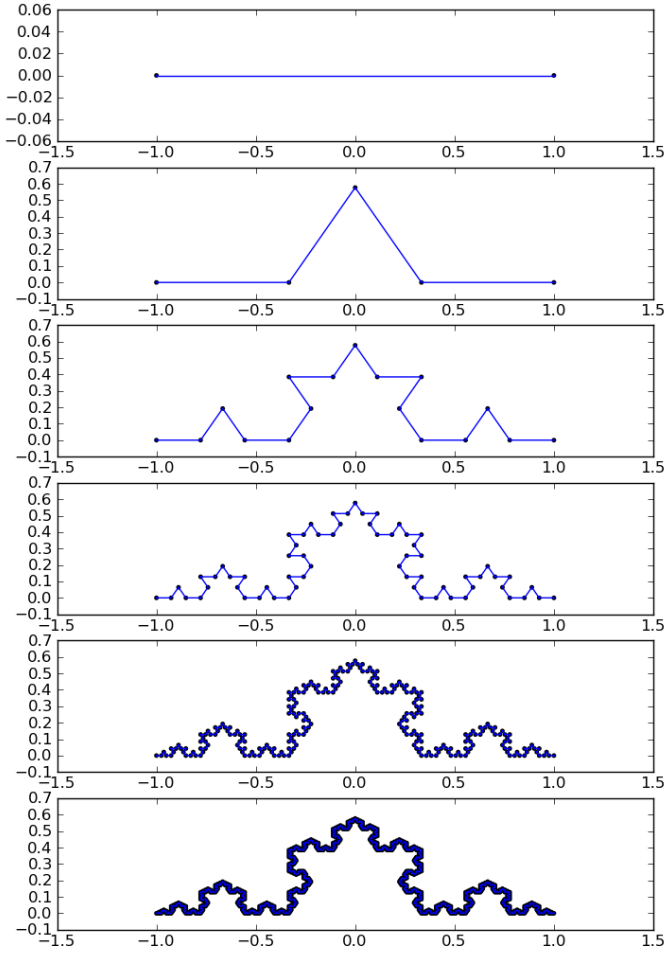
From: http://www.eftaylor.com/software/ActionApplets/LeastAction.html (With Java applets)

Java source code at: http://www.eftaylor.com/leastaction.html

Throw an apple vertically upward from the ground (zero height). We demand that 3 seconds later the apple return to our hand at the same height (zero) from which we launched it. What is the motion of this apple between the events of launch and catch? At what height can the apple be found at any given time? Or to express the question more technically: What is the worldline of the apple between launch and catch? We use the principle of least action to find answers to these questions.

$$S \equiv \int_{\text{entire worldline}} L\, dt = \int_{\text{entire worldline}} (T - V)\, dt$$

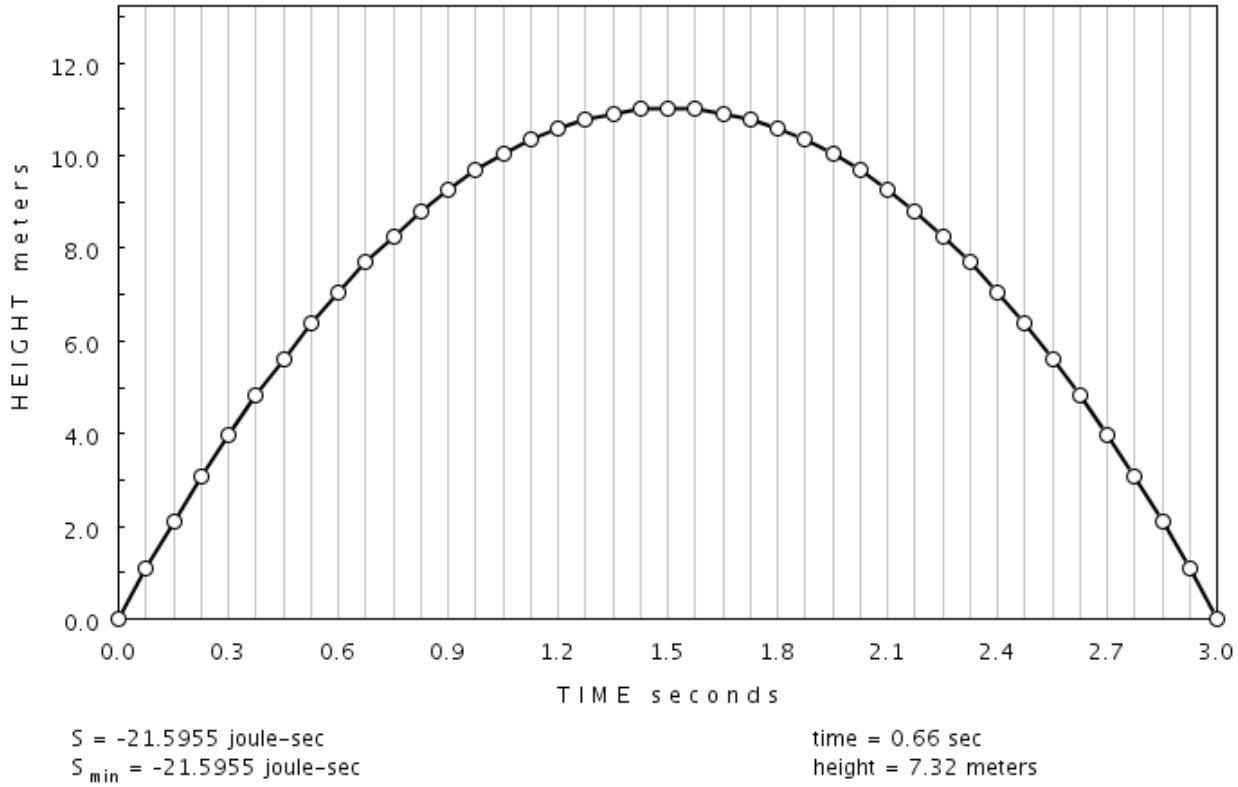
The principle of least action defines the action $S$ for motion along a worldline between two fixed events:


0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1

Here $L$ is called the Lagrangian. In simple cases the Lagrangian is equal to the difference between the kinetic energy $T$ and the potential energy $V$, that is, $L = T \breve{} V$. In this interactive document we will approximate a continuous worldline with a worldline made of straight connected segments. The computer then multiplies the value of $(T \breve{} V)$ on each segment by the time lapse $t$ for that segment and adds up the result for all segments, giving us an approximate value for the action $S$ along the entire worldline. Our task is then to move the connected segments of the worldline so that they result in the minimum total value of the action $S$.

In the following we assume a mass of 0.2 kilogram for the apple.

Try to obtain a plot similar to the obtained in "Display #2: Automatic hunting for worldline of least action", when the button `Hunt` is clicked:




## 7 Sequence alignment

**Level**: Advanced (algorithm, language)

Write a python Script to compute the global sequence alignment of two nucleotide sequences using a simple implementation of the Needleman-Wunsch algorithm.

1. Generate two random sequences of a given length using letters in the set $\{A, G, C, T\}$ (e.g. AGTGAC, TACGGA)

2. Construct a simple *similarity function* $S(A, B)$:

$$S(A,B) = \begin{cases} w_M & \text{if } A = B \\ w_U & \text{if } A \neq B \end{cases}$$

Where $A$ and $B$ are any letters in the symbols set and $w_M = 10$ $w_U = -5$ (gap scoring.) This function is used to score the matches between nucleotides in the sequences.

3. Construct the *F-matrix* of the sequences, i.e. the matrix which contains information about the matching of every single character in both sequences:

$$F_{ij} = \begin{cases} w_D & \text{if } i = 0 \text{ or } j = 0 \\ max(F_{i-1,j-1} + S(A_i, B_j), F_{i,j-1} + w_D, F_{i-1,j} + w_D) & \text{otherwise} \end{cases}$$

Where $A_i$ is the i-th letter in the sequence A and $w_D = -5$

4. The F-matrix has the instructions to build the final alignment. For the example sequences A=AGTGAC, B=TACGGA the F-matrix looks like:

$$F = \begin{pmatrix} -5 & -5 & -5 & -5 & -5 & -5 \\ -5 & -10 & -10 & 5 & 5 & 0 \\ -5 & -10 & -15 & 0 & 0 & 0 \\ -5 & -10 & -15 & -5 & 10 & 5 \\ -5 & 5 & 0 & -5 & 5 & 20 \\ -5 & 0 & 15 & 10 & 5 & 15 \end{pmatrix}$$

To build the aligned sequences $A', B'$, start in the lowermost, rightmost component of the matrix. Compare that element with their neighbors: diagonal $(i-1, j-1)$, left $(i, j-1)$ and up $(i-1, j)$. Find the larger value among them and *move* in that direction.

If your last movement was in the diagonal direction, add to aligned sequences the respective character in the original sequences.

If you move up, add to the aligned sequence $A'$ the respective character of $A$ and to $B'$ add a symbol '-' (insertion or deletion symbol).

Finally if you move to left, add to the aligned sequence $B'$ the respective character in $B$ and put a '-' symbol in $A'$.

Repeat this procedure until you reach the first line or column.

With the sequences provided above the final aligned sequences are:

–GTGAC

TAG-GA-

**Bonus**: Try to measure the aligning time of long sequences and to figure out the way as the time increases with the length of the sequences.

**Bonus**: Try changing the values of the gap-scoring ($w$ parameters).

A detailed but slightly different explanation of the algorithm could be found in the Wikipedia article: http://es.wikipedia.org/wiki/Algoritmo_Needleman-Wunsch.

## 8 Command line calculator

**Level**: Advanced (algorithm, language, scientific libraries)

Make a python program that works as a command line calculator, `pycalc` in the same spirit that the beloved UNIX program `bc`.

The program should work as in the following examples:

```
$ pycalc "2*pi^2*sin(0.7)/atan(60)"
8.1822869377000984
```

Note the use of `^` instead of `**`. The calculator should use by default the `math` library.

When dealing with complex numbers the python calculator should import the numpy library:

```
$ pycalc -c "2*pi^2*asin(2+3j)/atan(60)"
(7.247930188215947+25.191239093226159j)
```

where the `-c` options allows to work with complex numbers.

**Bonus 1**: Deal with rationals and integers:

```
$ pycalc 2/3+8/5
34/15
```

**Bonus 2**: Input from pipes:

```
$ echo 2./3+8/5. | pycalc
2.26666666667
```

## 9 Tree methods for sumations

**Level**: Advanced (algorithm, language, Object Orientated Programming)

One of the most common numerical tasks in problems involving the simulation of dynamical processes in N-body systems is the computation of large number of summations with the form:

$$f_i = \sum_{j=1}^{j=N} g_{ij}$$

One example is the computation of a long-range force exerted on a particle by the rest of particles in a given physical system.

To compute the evolution of that kind of systems $O(N^2)$ computations of the $g_{ij}$ terms are required and with a large value of $N$ the computational cost is huge.

One of the most clever solution of this problem is the so-called *tree-method for summation*. In that method the value of $g_{ij}$ corresponding to distant particles are summed up by constructing a hierarchical structure of nested cells (tree).

The tree is constructed using the following procedure:

1. Divide the initial *volume* in $2^D$ equal regions ($D = 1, 2, 3$ is the number of space dimensions of the problem)
2. Compute the physical properties of each cell: total mass or total charge, gravitational or electrical multipole momenta, etc.
3. Store the geometrical paramters of each cell: center of mass or charge, dimensions, position of its vertices.
4. Store a list of the particles contained on each cell.
5. Tag the cell with an identification number.
6. Store the tags of the father cell (0 for the original volume) and a list of the tags of the child cells if there is any (see next step).
7. Repeat 1-6 for every child cell that contains more than 1 particle.

The summation for each particle is performed in the following way:

1. Loop on the tree starting in the highest level (tag 1, 2, ... skip the 0 level)
2. For each cell compute the distance from the centroid to the particle (d).
3. If $s/d < 0.5$, where $s$ is the size of the cell compute the force exerted by the whole cell on the particle.
4. Else descend on the child cells and repeat from step 2.

At the end the computation of all the summations for the $N$ particles is performed in $O(NlogN)$ instead of the original $O(N^2)$.

Write a python script to compute the gravitational force on each particle of a system composed by N randomly distributed particles in a 1D line and using a tree-method to compute the summation.

The specific gravitational force on particle i exerted by another particle is given by:

$$f_i = -\frac{M}{R_i^3}\vec{R}_i$$

Where $R_i = \vec{r}_i - \vec{r}$, $M$ and $\vec{r}$ is the mass parameter and position of the other particle.

**Bonus: Compare the time required to perform the operation using** the direct summation and the tree mathod.

**Bonus**: Write the program to perform the summation in 2 and 3D.

In addition to the problems listed here, you should study the Scipy Cookbook (http://www.scipy.org/Cookbook), where a lot of very good problems are presented with full solutions.