CS170A — Mathematical Modeling & Methods for Computer Science — Winter 2016
HW#5 *optimization*
Due: 11:55 pm Sunday February 28, 2016

D. Stott Parker, Yun Lu, Yunzhong He, Luis Angel Larios-Cardenas

Using CCLE, please upload a PDF document with your work (including programs and their output).

1 Income Tax

You can get income tax returns from presidential candidates at
`http://www.taxhistory.org/www/website.nsf/Web/PresidentialTaxReturns?OpenDocument`
Some of these returns are impressive. Bernie Sanders' income is barely enough for his family to live in Los Angeles, and Hilary Clinton's is the highest of the 2016 candidates. Mitt Romney's 2011 tax return is 379 pages long!

In this assignment, take the file `HR_Clinton_2014_tax_return_numbers.txt` listing numbers in Hilary's tax return and determine (using the method in the course notes) whether the *unique* numbers entered in this form (please omit duplicates) violate Benford's Law.

2 Particle System

In this exercise you will create a particle system by using *Forward Euler Integration* and *Newton's Method*. As we saw in class, *Forward Euler Integration* is the simplest method to integrate forces acting on a body into acceleration, acceleration into velocity, and velocity into a new position. Your objective is to simulate particles in free fall, which will bounce off an elliptic boundary. In order to generate the bouncing effect, you have to prevent the particles from drifting beyond the ellipse by "pushing" them back to their *nearest location* on the curve.

Before delving into the simulation, you must first understand how we deal with collisions of particles against the ellipse. A key part of this process consists of particle relocation: When a particle goes beyond the boundary, you must figure out its corresponding *nearest point* on the curve and push the particle back to such location. As you may see in figure 1, the boundary can be represented as:

$$\mathbf{f}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} a\cos(t) \\ b\sin(t) \end{pmatrix}, \qquad 0 \leqslant t < 2\pi$$

where $\mathbf{f}(t)$ is the parametric form of an ellipse, centered at the origin, with horizontal semi-axis a, and vertical semi-axis b. Also, let $P_0 = (x_0,\ y_0)'$ be the current position of a particle that has gone through $\mathbf{f}(t)$, and now lies outside the enclosed area of the ellipse (i.e. a collision has been detected).

Furthermore, let $r(t) = ||\mathbf{f}(t) - P_0||_2^2$ be the *squared-distance* scalar function between a colliding particle and any point along the ellipse $\mathbf{f}(t)$. Since you are interested in "pushing" the particle back to its **nearest** location on the ellipse, we must find t for which $r(t)$ is minimal; that is, you must solve for t, such that

$$\frac{d}{dt}\left(r(t)\right) = \frac{d}{dt}||\mathbf{f}(t) - P_0||_2^2 = 0$$

Once you know that $r(t)$ is minimal for $t = t^*$, you can move P_0 to $\mathbf{f}(\mathbf{t}^*)$ and continue with the simulation as usual.

1. From the explanation above, given $\mathbf{f}(t) = (a\cos(t),\ b\sin(t))$, and $P_0 = (x_0,\ y_0)'$, provide a simplified expression for
$$\frac{d}{dt}||\mathbf{f}(t) - P_0||_2^2$$
You'll use this expression later to find t^*, the parameter for which the distance $||\mathbf{f}(t) - P_0||_2^2$ is minimal.

2. In the homework `*.zip` archive, you'll find a `particleSystemTemplate` script. This script is well-documented and guides you throught the process to complete your simulation. First, locate the **THREE** `TODO` sections enclosed in comment blocks. Then, read the file thoroughly and try to understand all the things it does to set up the environment and prepare the simulation variables. What is the timestep Δt? Which external forces are acting on the particles and what are their *vector representations*? How do you know if a particle is *outside* the ellipse?

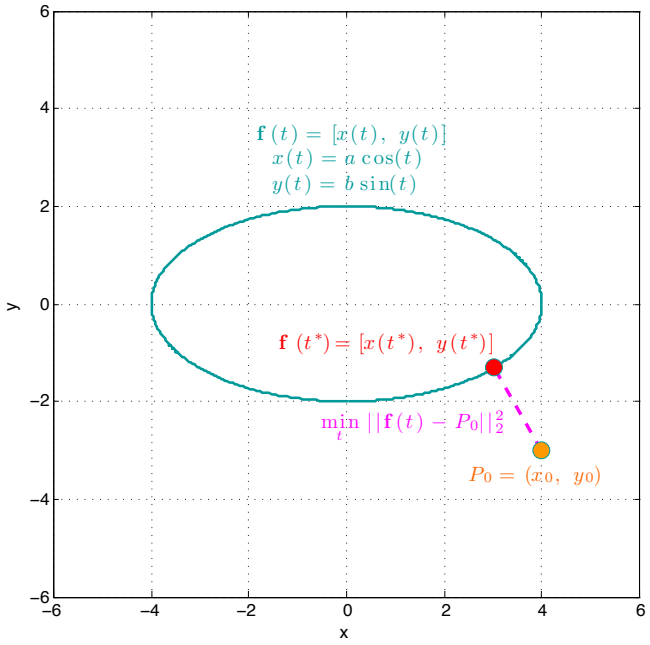

Figure 1: Collision detection

3. You have to simulate *10 particles*, whose properties are given in the `particleData.csv` file. Each row of this file contains information for one particle as follows:

```
Position: Column 1 (x) and column 2 (y).
Velocity: Column 3 (x) and column 4 (y).
Color: Column 5 (red), column 6 (green), and column 7 (blue).
Mass: Column 8.
```

   Go ahead and modify the first `TODO` block and write the statements necessary to load `particleData.csv` into your workspace. Then, do the necessary in order to assign to each particle object its corresponding initial position, initial velocity, color, and mass. Notice that we have located several `?` marks in places where you should precisely write the expressions that complete those statements.

4. Locate the second `TODO` section, which is within the *particles* evaluation loop. This loop traverses the particles array and updates their velocity and position at every time step. On this block, complete the statements:

```
%% In Matlab
particles(I).velocity = ?
particles(I).position = ?
```

```
## In Pylab
particle['velocity'] = ?
particle['position'] = ?
```

   Note, however, that when you compute a particle's new position, you must use the **old velocity**, NOT the new velocity you just approximated in the current time step.

5. The new particle position you computed in the previous step is not quite valid yet; you have to check if the particle has gone beyond the ellipse boundaries. If so, you should *push* the particle back to its **nearest** point on the ellipse. Likewise, the velocity vector must be flipped and scaled so that you have a "bouncing" effect. We have taken care of the latter – but you still have to solve for the *adjusted* position of a penetrating particle. To do so, locate the third `TODO` section, also within the particles evaluation inner loop.

   This `TODO` section will be executed when a particle is known to be outside the ellipse boundaries. Your goal here is to find the point on the ellipse, **nearest** to the current particle position, so that you can *move* the particle to that safe location (e.g. not penetrating). In order to do this, use the expression you found in the first exercise of this problem to create a *function object*:

```
%% In Matlab
df = @(t) ?
```

```
## In Pylab
df = lambda t: ?
```

Next, you'll rely on Matlab's `fzero` and/or SciPy's `newton` to obtain `tStar`. These methods take a function (e.g. `df`) and an initial value (e.g. `t0`), and return the t^* value (e.g. `tStar`) for which the input function becomes zero (see figure 1). Once you have obtained `tStar`, you may finally *relocate* the colliding particle, knowing that $\mathbf{f}(t^*) = [x(t^*),\ y(t^*)]$ lies exactly ON the ellipse.

***Catch**: Convergence on the* `fzero` *and* `newton` *methods greatly depends on the intial value,* `t0`*, that you supply. How can you pick a good* `t0` *such that these methods converge to the right answer in a short time?*

6. Finally, obtain a *screen shot* (i.e. save the figure as PNG/PDF) at the end of the simulation (i.e. *time* ≈ 10.00). If your simulation is correct, your screen shot should look like figure 2 (without the water mark, of course). Additionally, if everything goes smoothly, your particle system must behave like the movie we have included in the `*.zip` archive.

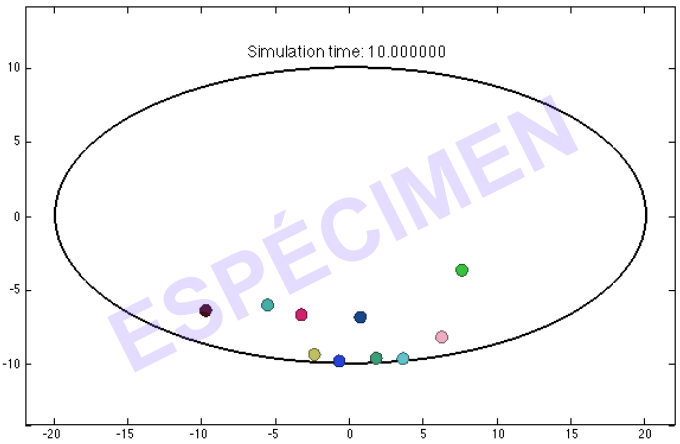

Figure 2: Particle system simulation at $t = 10.00$

3 Logistic Regression

In the previous homworks we learned how to project faces onto their first few principal components, so that they can be visualized as points in low dimensional space. And good faces and evil faces may remain separable in those spaces. For this exercie we want to build a classifier that finds a straight line to separate good and evil faces, such that all points beyond the line are considered good faces, and all points below the line are considered evil faces. Such a line is therefore called a "decision boundary".

For this exercise we assume all data points are already projected into two-dimensional space. And we use label $y = 1$ to indicate a good face, $y = 0$ to indicate an evil face. The decision boundary we want to find is defined as

$$ax_1 + bx_2 + c = 0$$

So finding the line is the same as finding parameters $\theta =< a, b, c >$ such that

$$\theta < x_1, x_2, 1 >^T = 0$$

Therefore, for $face_i = (x_1^{(i)}, x_2^{(i)})$, the classifier outputs "good" if $\theta < x_1^{(i)}, x_2^{(i)}, 1 >^T \geq 0$, and "evil" otherwise. In order to find the best θ, we define the following objective function $J(\theta)$ to minimize. So the θ that minimizes the function is what we want.

$$J(\theta) = \frac{1}{m}\sum_{i}^{m} -y^{(i)} \cdot log(h(x_1^{(i)}, x_2^{(i)}, \theta)) - (1 - y^{(i)}) \cdot log(1 - h(x_1^{(i)}, x_2^{(i)}, \theta))$$

where

$$h(x, y, \theta) = \frac{1}{1 + exp(-\theta < x_1, x_2, 1 >^T)}$$

1. In file cost.m/logisticRegression.py, implement the objective function above. Include your implementation in your report.

2. In file logisticRegression.m/logisticRegression.py, use fsearchmin/scikit.optimize.fmin, or any other optimization methods you like, to find the θ that minimize the objective funciton, and plot the decision boundary. Include a plot of the decision boundary you found in your report.