Exercise05

February 21, 2015

```
In [1]: import os
        import sys
        import glob
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        %matplotlib inline
        %precision 4
        plt.style.use('ggplot')
```

```
In [2]: from sympy import Symbol, exp, I, pi, N, expand
        from sympy import init_printing
        init_printing()
```

0.1 Background for Exercises 1 and 2

The first 2 exercises are about using Newton's method to find the cube roots of unity - find z such that $z^3 = 1$. From the fundamental theorem of algebra, we know there must be exactly 3 complex roots since this is a degree 3 polynomial.

We start with Euler's fabulosu equation

$$e^{ix} = \cos x + i \sin x$$

Raising e^{ix} to the nth power where n is an integer, we get from Euler's formula with nx substituting for x

$$(e^{ix})^n = e^{i(nx)} = \cos nx + i \sin nx$$

Whenever nx is an integer multiple of 2π, we have

$$\cos nx + i \sin nx = 1$$

So

$$e^{2\pi i \frac{k}{n}}$$

is a root of 1 whenever $k/n = 0, 1, 2, \ldots$.

So the cube roots of unity are $1, e^{2\pi i/3}, e^{4\pi i/3}$.

While we can do this analytically, the idea is to use Newton's method to find these roots, and in the process, disover some rather perplexing behavior of Newton's method.

```
In [3]: expand(exp(2*pi*I/3), complex=True)
```

Out[3]:

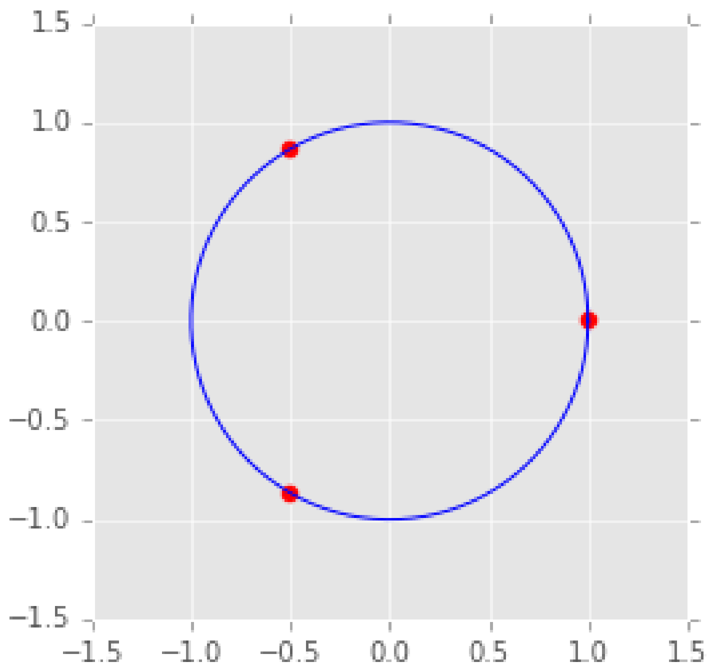
$$-\frac{1}{2} + \frac{\sqrt{3}i}{2}$$

In [4]:
```
expand(exp(4*pi*I/3), complex=True)
```

Out[4]:

$$-\frac{1}{2} - \frac{\sqrt{3}i}{2}$$

In [5]:
```
plt.figure(figsize=(4,4))
roots = np.array([[1,0], [-0.5, np.sqrt(3)/2], [-0.5, -np.sqrt(3)/2]])
plt.scatter(roots[:,0], roots[:,1], s=50, c='red')
xp = np.linspace(0, 2*np.pi, 100)
plt.plot(np.cos(xp), np.sin(xp), c='blue');
```



Exercise 1 (10 points). Netwon's method for functions of complex variables - stability and basins of attraction.

1. Write a function with the following function signature `newton(z, f, fprime, max_iter=100, tol=1e-6)` where
   - `z` is a starting value (a complex number e.g. `3 + 4j`)
   - `f` is a function of `z`
   - `fprime` is the derivative of `f` The function will run until either max_iter is reached or the absolute value of the Newton step is less than tol. In either case, the function should return the number of iterations taken and the final value of `z` as a tuple (`i`, `z`).
2. Define the function `f` and `fprime` that will result in Newton's method finding the cube roots of 1. Find 3 starting points that will give different roots, and print both the start and end points.

In [6]:
```
# your code here
```

Exercise 2 (10 points). Write the following two plotting functions to see some (pretty) aspects of Newton's algorithm in the complex plane.

1. The first function `plot_newton_iters(f, fprime, n=200, extent=[-1,1,-1,1], cmap='hsv')` calculates and stores the number of iterations taken for convergence (or max_iter) for each point in a 2D array. The 2D array limits are given by `extent` - for example, when `extent = [-1,1,-1,1]` the corners of the plot are `(-i, -i), (1, -i), (1, i), (-1, i)`. There are `n` grid points in both the real and imaginary axes. The argument `cmap` specifies the color map to use - the suggested defaults are fine. Finally plot the image using `plt.imshow` - make sure the axis ticks are correctly scaled. Make a plot for the cube roots of 1.

2. The second function `plot_newton_basins(f, fprime, n=200, extent=[-1,1,-1,1], cmap='jet')` has the same arguments, but this time the grid stores the identity of the root that the starting point converged to. Make a plot for the cube roots of 1 - since there are 3 roots, there should be only 3 colors in the plot.

```
In [7]: # your code here
```

Exercise 3 (10 points). Consider the following function on $\mathbb{R}^2$:

$$f(x_1, x_2) = -x_1 x_2 e^{-\frac{(x_1^2+x_2^2)}{2}}$$

1. Use `sympy` to compute its gradient.
2. Compute the Hessian matrix.
3. Find the critical points of f.
4. Characterize the critical points as max/min or neither. Find the minimum under the constraint

   $$g(x) = x_1^2 + x_2^2 \leq 10$$

   and

   $$h(x) = 2x_1 + 3x_2 = 5$$

   using `scipy.optimize.minimize`.
5. Plot the function using `matplotlib`.

```
In [8]: # your code here
```

Exercise 4 (20 points). Find the maximum likelihood function for a logistic regression. Load the file "train.csv" (located in this directory) into python. This file is data from the survival data from the Titanic (from https://www.kaggle.com/). You are to use a logistic regression to model survival as a function of gender, age and class (of travel). Find the maximum likelihood estimator of β by numerical optimization using stochastic gradient descent as follows:

1. Stochastic gradient descent. In this method, gradient descent is used essentially by fitting *one data point at a time.* Recall the usual gradient descent step:

$$\beta_{i+1} = \beta_i - \nabla \ell(\beta_i)$$

where

$$\nabla \ell(\beta_i) = \sum_{j=1}^{n} \nabla \ell(\beta_i, x_j, y_j)$$

and ℓ is the log-likelihood function. All of the data is used to make the next step toward the optimal β. In stochastic gradient descent, only one point at a time is used to determine the next β:

$$\beta_{i+1} = \beta_i - \alpha \nabla \ell(\beta_i, x_j, y_j)$$

where α is the step size. For simplicity, we'll take a constant $\alpha = 1$.
Implement the following stochastic gradient algorithm:

a. Shuffle data points (i.e. randomly permute the order of the (x_j, y_j))

b. Refine beta using the iterative formula above over each data point.

c. Repeat a and b until convergence is reached.

Apply the algorithm to the given data set to find the best-fit logistic regression coefficients. Do not forget that your optimization should include a tolerance and a limit on the number of iterations.

In [9]:
```
# your code here
```