A Quick Introduction to Plotting in PyLab

Lecturer: John Guttag

Text Can Be Useful

6.00x started on October 1, 2012

The instructors are Eric Grimson, Chris Terman, and John Guttag

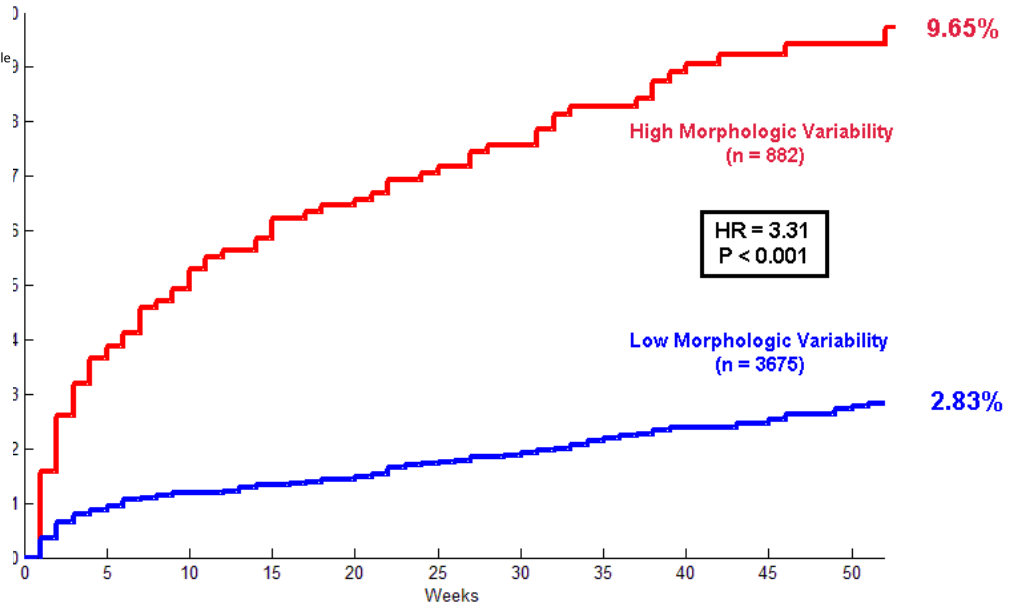
It's lots of work!

A Picture Is Worth 10,000 Words

related to cardiac health, can lead to a more accurate evaluation of the current state of a patient and the likely impact of various therapies. We refer to the information generated by such algorithms as computational biomarkers, which we define as a characteristic that is not directly measurable but that can be computationally derived from measurable data.

Our research leverages advanced analytical techniques from machine learning and data mining to extract potentially valuable information that is often overlooked in large volumes of cardiovascular data, but that may identify high-risk patients after ACS. We have developed three computational biomarkers [morphologic variability (MV), symbolic mismatch (SM), and heart rate motif (HRM)], using sophisticated machine learning and data mining techniques, that can be used to risk stratify post-ACS patients. Each metric is derived from long-term Holter ECG signals and its derivation is fully automated. Morphological variability (MV) assesses myocardial instability by quantifying low-amplitude probabilistic variability in the shape of the ECG waveform over long periods of time (17). SM quantifies the degree to which long-term ECG signals of individual patients are anomalous relative to those of other patients with a similar clinical history. It is based on detecting shifts in the morphology and dynamics of functional units of cardiac activity over long periods (18). HRM integrate the frequency with which high- or low-risk heart rate patterns reflecting autonomic function appear in a patient's ECG over long time periods (19). Each of these computational biomarkers uses time-series analytical techniques to extract new types of information that are presently unappreciated in large volumes of continuous cardiovascular data. These computational biomarkers measure subtle features of long-term ECG data that are outside the scope of human visualization but are consistently associated with future risk.

Here, we assess the prognostic ability of these biomarkers in a blinded, prespecified, and fully automated study on more than 4500 patients in the MERLIN-TIMI36 (Metabolic Efficiency with Ranolazine for Less Ischemia in Non–ST-Elevation Acute Coronary Syndrome–Thrombolysis in Myocardial Infarction 36) trial. We evaluate MV, SM, and HRM in a population with detailed follow-up data and rich clinical metadata for all patients. This allows us to compare the ability of these computationally generated biomarkers to that of other metrics such as the TIMI risk score (TRS) (20), LVEF, and several other metrics that are based on long-term ECG time series (described in more detail in the Supplementary Material): heart rate variability (HRV), heart rate turbulence (HRT), deceleration capacity (DC), severe autonomic failure (SAF), and a fully automated version of modified moving average T-wave alternans (TWA). We also study the incremental information provided by computational biomarkers relative to existing metrics through orthogonal statistical approaches to assess their effect on discrimination and reclassification of CVD after ACS.



A Hierarchy of Open-source Python Libraries

NumPy adds vectors, matrices, and many high-level mathematical functions

Scipy adds mathematical classes and functions useful to scientists

MatPlotLib adds an object-oriented API for plotting

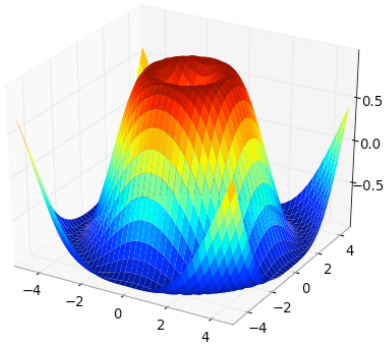
PyLab combines the other libraries to provide a MATLAB-like interface

Some Useful Web Pages

http://matplotlib.org/api/pyplot_summary.html

http://www.scipy.org/Plotting_Tutorial

http://matplotlib.sourceforge.net/users/customizing.html.



pylab.plot

The first two arguments to `pylab.plot` must be sequences of the same length.

First argument gives x-coordinates.

Second argument gives y-coordinates.

Points plotted in order. As each point is plotted, a line is drawn connecting it to the previous point.

Plotting Mortgages

Lecturer: John Guttag

```
class MortgagePlots(object):

    def plotPayments(self, style):
        pylab.plot(self.paid[1:],style,label=self.legend)

    def plotTotPd(self, style):
        totPd = [self.paid[0]]
        for i in range(1, len(self.paid)):
            totPd.append(totPd[-1] + self.paid[i])
        pylab.plot(totPd, style, label = self.legend)
```

```
def compareMortgages(amt, years, fixedRate, pts, ptsRate,
                     varRate1, varRate2, varMonths):
    totMonths = years*12
    fixed1 = Fixed(amt, fixedRate, totMonths)
    fixed2 = FixedWithPts(amt, ptsRate, totMonths, pts)
    twoRate = TwoRate(amt, varRate2, totMonths,
                      varRate1, varMonths)
    morts = [fixed1, fixed2, twoRate]
    for m in range(totMonths):
        for mort in morts:
            mort.makePayment()
    plotMortgages(morts, amt)
```

```
def plotMortgages(morts, amt):
    styles = ['b-', 'r-.', 'g:']
    payments = 0
    cost = 1
    pylab.figure(payments)
    pylab.title('Monthly Payments of Different $'\
                + str(amt) + ' Mortgages')
    pylab.xlabel('Months')
    pylab.ylabel('Monthly Payments')
    pylab.figure(cost)
    pylab.title('Cost of Different $' + str(amt)\
                + ' Mortgages')
    pylab.xlabel('Months')
    pylab.ylabel('Total Payments')
    .
    .
    .
```

```
for i in range(len(morts)):
        pylab.figure(payments)
        morts[i].plotPayments(styles[i])
        pylab.figure(cost)
        morts[i].plotTotPd(styles[i])
    pylab.figure(payments)
    pylab.legend(loc = 'upper center')
    pylab.figure(cost)
    pylab.legend(loc = 'best')
```

Random Walks and Simulation Models

Lecturer: John Guttag

Simulation Models

Simulation attempts to build an experimental device called a model

Kinds of Simulation Models

Deterministic simulations are completely defined by the model
    Rerunning the simulation will not change the result

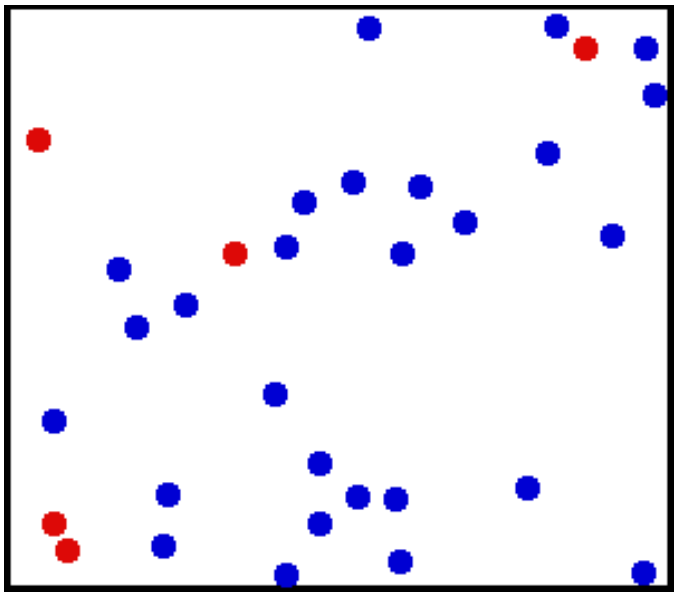
Stochastic simulations include randomness
    Different runs can generate different results

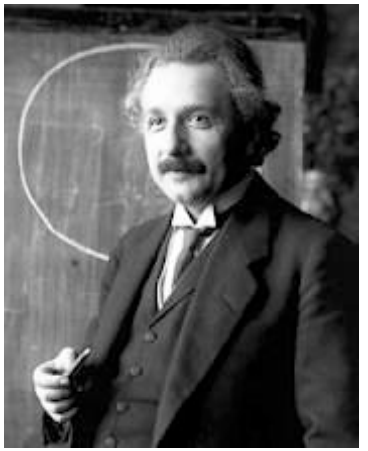
In a discrete model, values of variables are enumerable (e.g., integers). In a continuous model, they are not enumerable (e.g., real numbers).

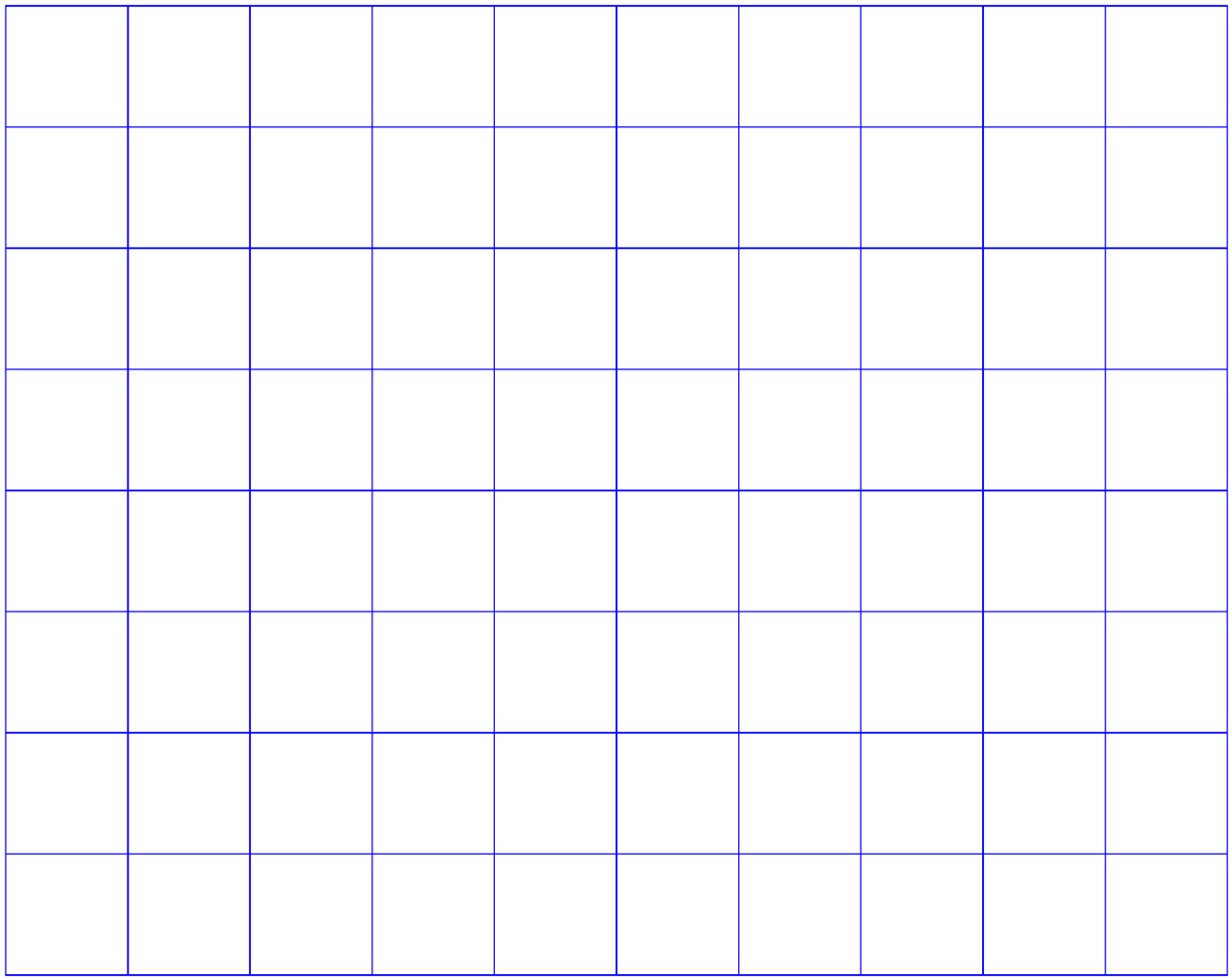
Random Walks and Simulation Models

Lecturer: John Guttag

Brownian Motion







Notable Aspects of Class Location

Notable Aspects of Class Field

```
class Drunk(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'This drunk is named ' + self.name

import random

class UsualDrunk(Drunk):
    def takeStep(self):
        stepChoices =\
            [(0.0,1.0),(0.0,-1.0),(1.0, 0.0),(-1.0, 0.0)]
        return random.choice(stepChoices)
```

Random Walks and Simulation Models

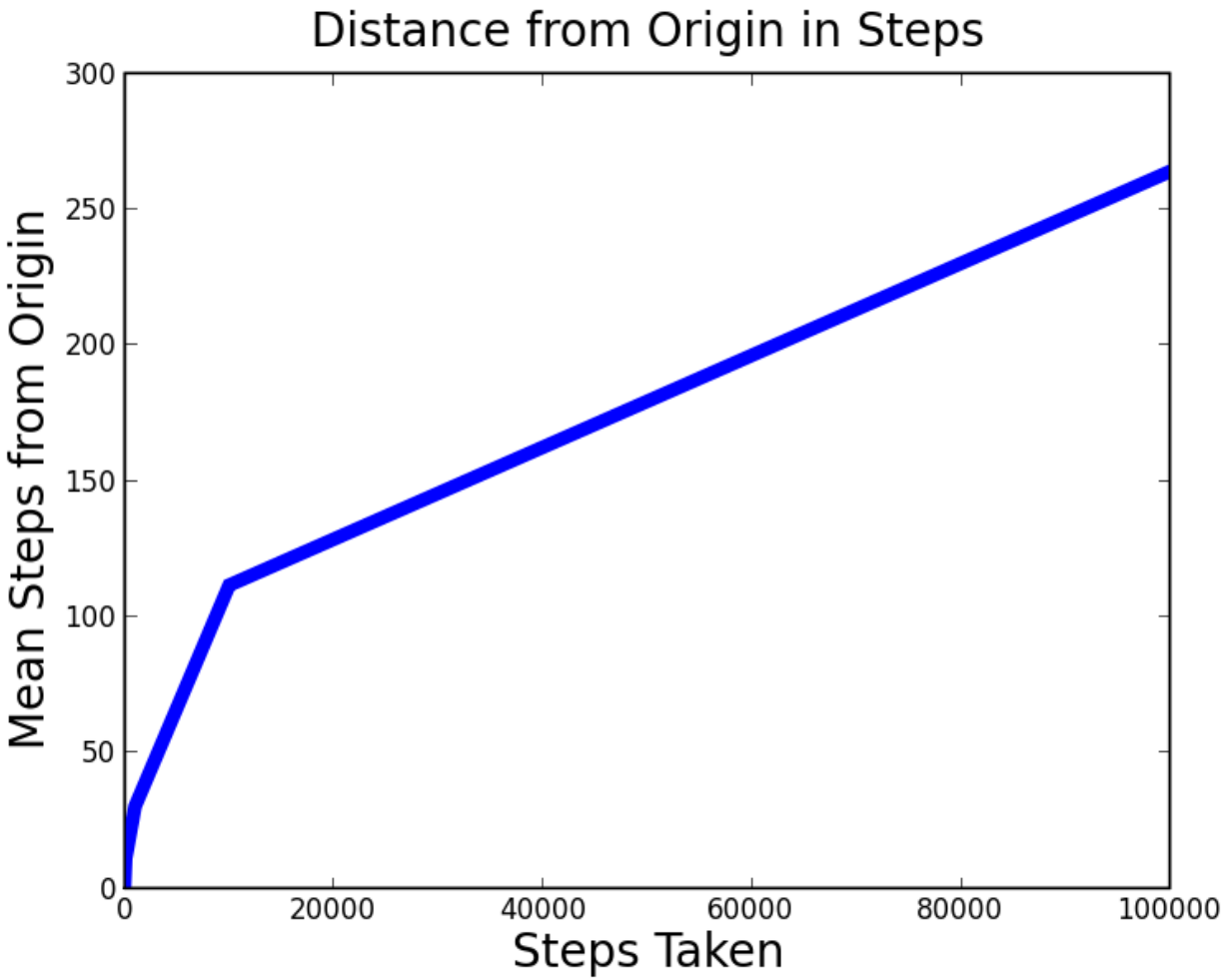
Lecturer: John Guttag

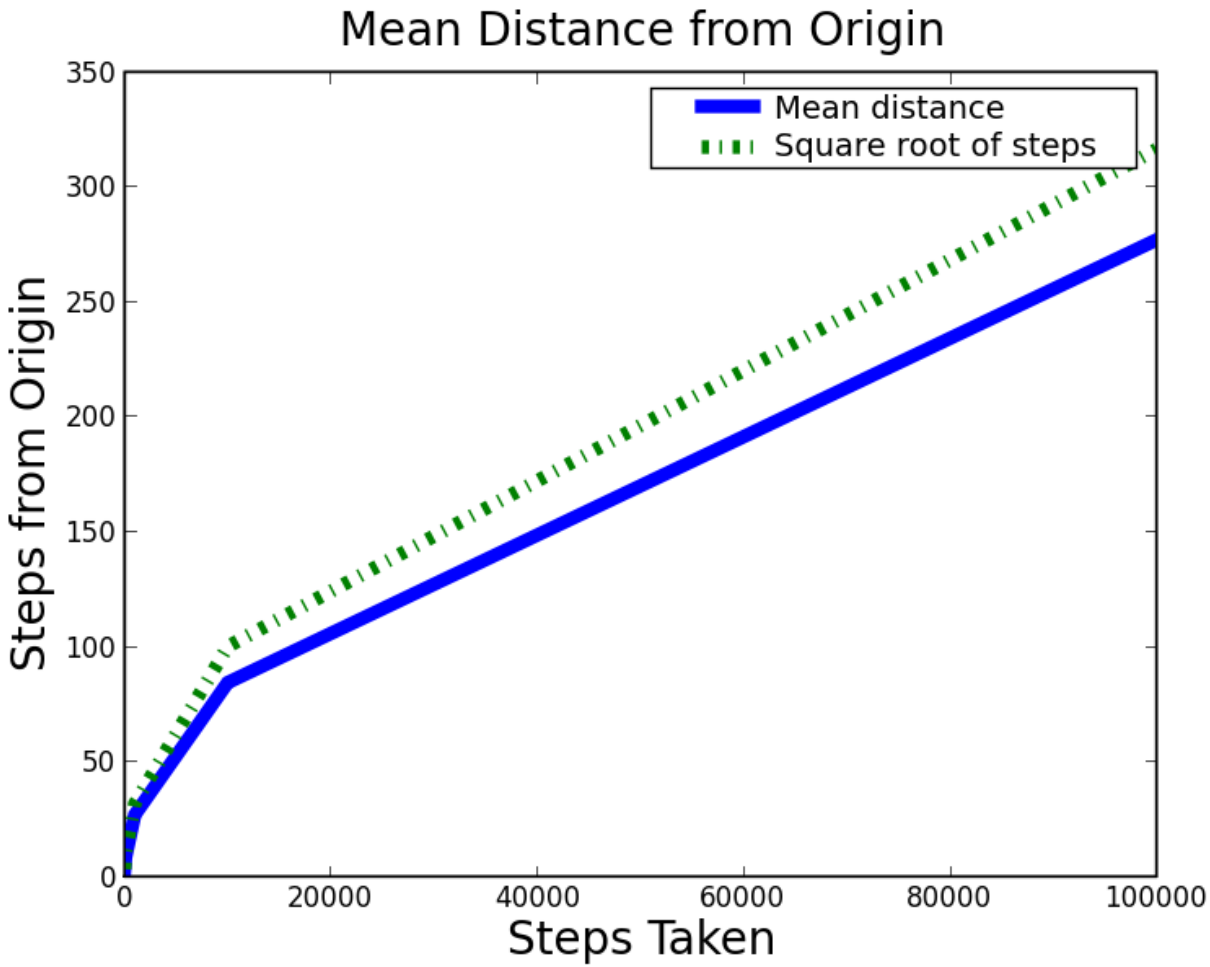
```
import random

def walk(f, d, numSteps):
    start = f.getLoc(d)
    for s in range(numSteps):
        f.moveDrunk(d)
    return(start.distFrom(f.getLoc(d)))
```

```
def simWalks(numSteps, numTrials):
    homer = Drunk('Homer')
    origin = Location(0, 0)
    distances = []
    for t in range(numTrials):
        f = Field()
        f.addDrunk(homer, origin)
        distances.append(walk(f, homer, numTrials))
    return distances
```

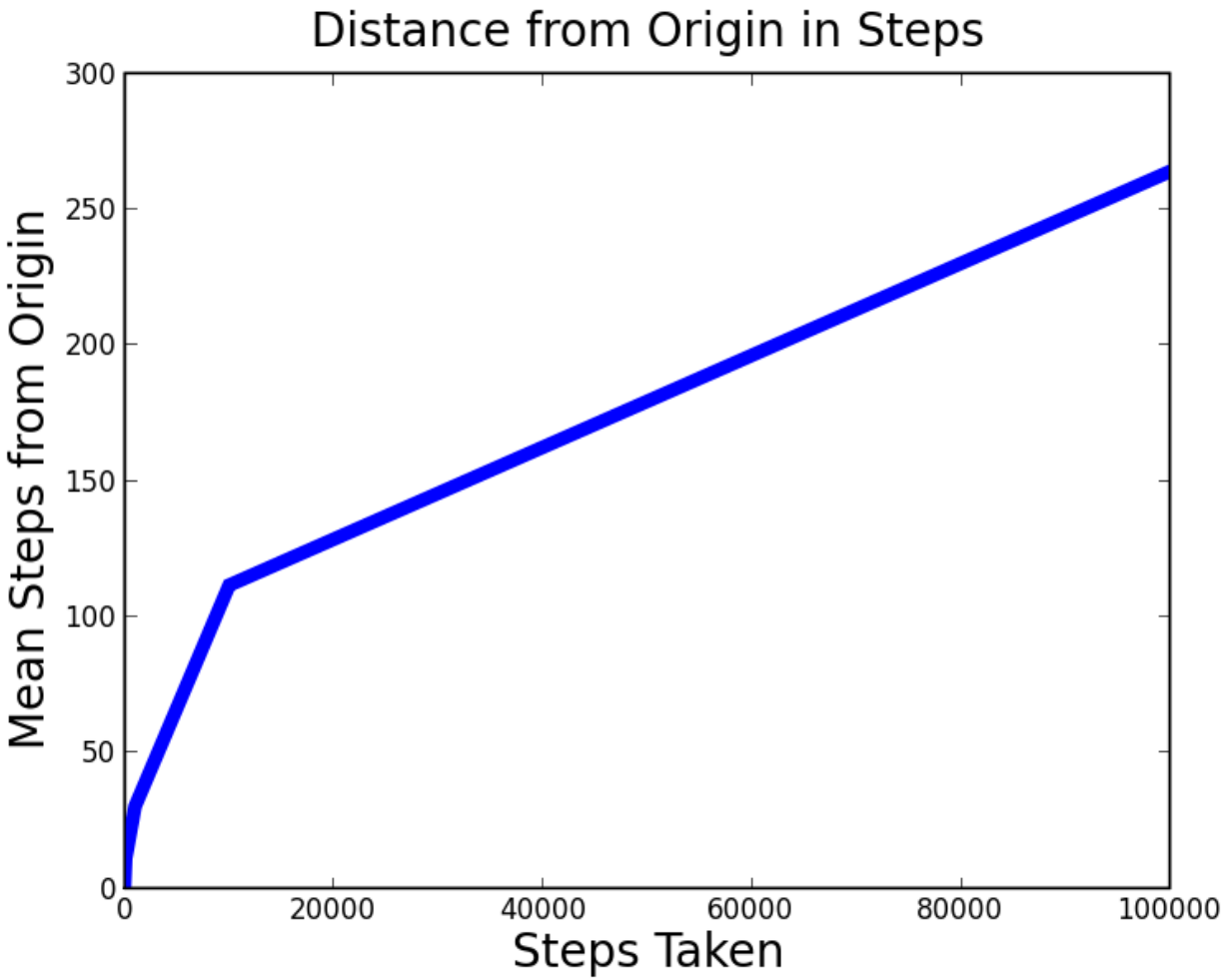
```
def drunkTest(numTrials):
    for numSteps in [10, 100, 1000, 10000, 100000]:
        distances = simWalks(numSteps, numTrials)
        print 'Random walk of ' + str(numSteps) + ' steps'
        print '  Mean =', sum(distances)/len(distances)
        print '  Max =', max(distances), 'Min =', min(distances)
```

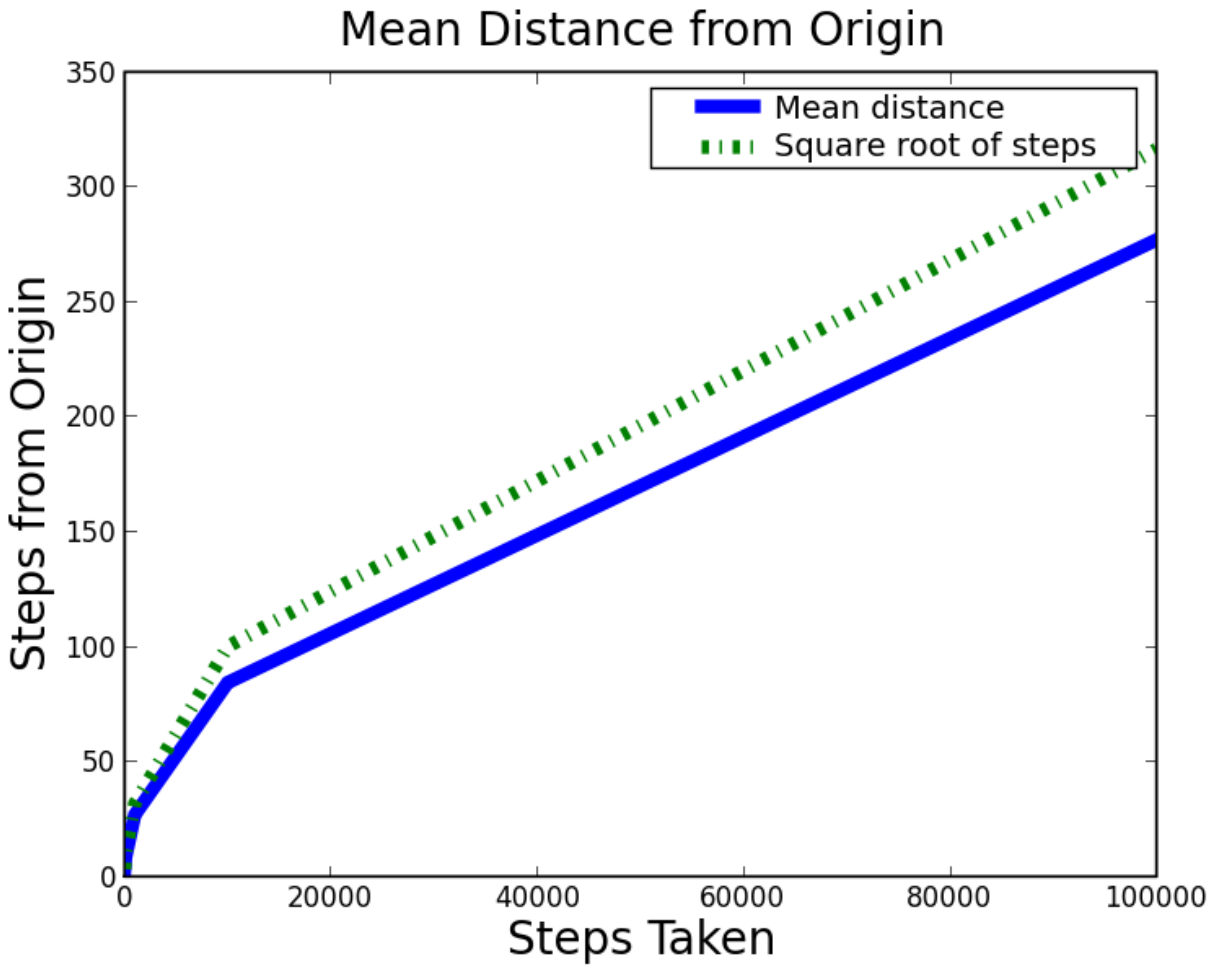

Distance from Origin in Steps
Mean Steps from Origin
Steps Taken
0
50
100
150
200
250
300
0
20000
40000
60000
80000
100000


Mean Distance from Origin
Mean distance
Square root of steps
Steps from Origin
Steps Taken
0
50
100
150
200
250
300
350
0
20000
40000
60000
80000
100000

Random Walks and Simulation Models

Lecturer: John Guttag


Distance from Origin in Steps
Mean Steps from Origin
Steps Taken
0
50
100
150
200
250
300
0
20000
40000
60000
80000
100000


Mean Distance from Origin
Mean distance
Square root of steps
Steps from Origin
Steps Taken
0
50
100
150
200
250
300
350
0
20000
40000
60000
80000
100000

```
class UsualDrunk(Drunk):
    def takeStep(self):
        stepChoices =\
            [(0.0,1.0),(0.0,-1.0),(1.0,0.0),(-1.0,0.0)]
        return random.choice(stepChoices)

class ColdDrunk(Drunk):
    def takeStep(self):
        stepChoices =\
            [(0.0,0.95),(0.0,-1.0),(1.0,0.0),(-1.0,0.0)]
        return random.choice(stepChoices)
```

```
class EDrunk(Drunk):
    def takeStep(self):
        deltaX = random.random()
        if random.random() < 0.5:
            deltaX = -deltaX
        deltaY = random.random()
        if random.random() < 0.5:
            deltaY = -deltaY
        return (deltaX, deltaY)
```

```
def simWalks(numSteps, numTrials):
    homer = UsualDrunk('Homer')
    origin = Location(0, 0)
    distances = []
    for t in range(numTrials):
        f = Field()
        f.addDrunk(homer, origin)
        distances.append(walk(f, homer, numSteps))
    return distances
```

```
def drunkTestP(numTrials = 100):
    stepsTaken = [10, 100, 1000, 10000]
    for dClass in (UsualDrunk, ColdDrunk, EDrunk):
        meanDistances = []
        for numSteps in stepsTaken:
            distances = simWalks(numSteps, numTrials, dClass)
            meanDistances.append(sum(distances)/len(distances))
        pylab.plot(stepsTaken, meanDistances,
                           label = dClass.__name__)
    . . .
```