Greg Czerniak's Website

Code | Games | Guides

Home

Kalman Filters for Undergrads Part I: Linear Kalman Filters

NOTE: This guide is currently a work in progress.

Table of contents [hide]

1. Introduction
2. When Can Kalman Filters Help?
3. Linear Kalman Filters
   1. Prerequisites
   2. Basics
   3. The Equations
4. Single-Variable Example
   1. Situation
   2. Simulation
   3. Preparation
   4. Execution and Python code
5. Multi-Variable Example
   1. Situation
   2. Simulation
   3. Preparation
   4. Execution and Python code
6. Conclusion
7. References
8. Special Thanks

Introduction

Kalman filters let you use mathematical models despite having error-filled real-time measurements. Programmers dealing with real-world data should know them. Publications explaining Kalman filters are hard for Computer Scientists/Engineers to understand since they expect you to know control theory. I wrote this guide for people who want to learn Kalman filters but never took a control theory course.

DISCLAIMER: This guide teaches amateur-level Kalman filtering for hobbyists. If lives depend on your Kalman filter (such as manned aviation, ICBMs, medical instruments, etc), do not rely on this guide! I skip a lot of details necessary for serious use!

This guide will cover:

- When Kalman filters can help.
- Examples of solving simple problems with Kalman filters.
- Examples of how to convert normal-looking equations into Kalman filter matrices.
- Example code implementing Kalman filters in Python.

This guide WON'T cover:

- Kalman filter history. Please consult the University of North Carolina at Chapel Hill's **great website** for information on this subject.
- When and why Kalman filters are optimal.
- How to tune Kalman filters for performance.
- In-depth details (such as exceptions to guidelines).

This guide aims to quickly get you 80% of the way toward understanding Kalman filters. I won't cover the remaining 20% since that's not the point of this guide. If you need to know Kalman filters to that depth, find a more detailed guide.

When Can Kalman Filters Help?

Kalman filters can help when four conditions are true:

1. You can get measurements of a situation at a constant rate.
2. The measurements have error that follows a bell curve.
3. You know the mathematics behind the situation.
4. You want an estimate of what's really happening.

There are exceptions, but I'll let other publications explain them.

Linear Kalman Filters

Prerequisites

To understand these linear Kalman filter examples, you will need a basic understanding of:

- High school physics
- Matrix algebra
- Python for the programming parts.

Basics

Here are the most important concepts you need to know:

- Kalman Filters are discrete. That is, they rely on measurement samples taken between repeated but constant periods of time. Although you can approximate it fairly well, you don't know what happens between the samples.
- Kalman Filters are recursive. This means its prediction of the future relies on the state of the present (position, velocity, acceleration, etc) as well as a guess about what any controllable parts tried to do to affect the situation (such as a rudder or steering differential).
- Kalman Filters work by making a prediction of the future, getting a measurement from reality, comparing the two, moderating this difference, and adjusting its estimate with this moderated value.
- The more you understand the mathematical model of your situation, the more accurate the Kalman filter's results will be.
- If your model is completely consistent with what's actually happening, the Kalman filter's estimate will eventually converge with what's actually happening.

When you start up a Kalman Filter, these are the things it expects:

- The mathematical model of the system, represented by matrices A, B, and H.
- An initial estimate about the complete state of the system, given as a vector x.
- An initial estimate about the error of the system, given as a matrix P.
- Estimates about the general process and measurement error of the system, represented by matrices Q and R.

During each time step, you are expected to give it the following information:

- A vector containing the most current control state (vector "u"). This is the system's guess as to what it did to affect the situation (such as steering commands).
- A vector containing the most current measurements that can be used to calculate the state (vector "z").

After the calculations, you get the following information:

- The most current estimate of the true state of the system.
- The most current estimate of the overall error of the system.

The Equations

NOTE: The equations are here for exposition and reference. You aren't expected to understand the equations on the first read.

The Kalman Filter is like a function in a programming language: it's a process of sequential equations with inputs, constants, and outputs. Here I've color-coded the filter equations to illustrate which parts are which. If you are using the Kalman Filter like a black box, you can ignore the gray intermediary variables.

BLUE = inputs ORANGE = outputs BLACK = constants GRAY = intermediary variables

State Prediction (Predict where we're gonna be)	$\mathbf{x}_{predicted} = \mathbf{A}\mathbf{x}_{n-1} + \mathbf{B}\mathbf{u}_n$
Covariance Prediction (Predict how much error)	$\mathbf{P}_{predicted} = \mathbf{A}\mathbf{P}_{n-1}\mathbf{A}^T + \mathbf{Q}$
Innovation (Compare reality against prediction)	$\tilde{\mathbf{y}} = \mathbf{z}_n - \mathbf{H}\mathbf{x}_{predicted}$

Innovation Covariance (Compare real error against prediction)	$\mathbf{S} = \mathbf{H}\mathbf{P}_{predicted}\mathbf{H}^{\mathbf{T}} + \mathbf{R}$
Kalman Gain (Moderate the prediction)	$\mathbf{K} = \mathbf{P}_{predicted}\mathbf{H}^{\mathbf{T}}\mathbf{S}^{-1}$
State Update (New estimate of where we are)	$\mathbf{x}_n = \mathbf{x}_{predicted} + \mathbf{K}\tilde{\mathbf{y}}$
Covariance Update (New estimate of error)	$\mathbf{P}_n = (I - \mathbf{K}\mathbf{H})\mathbf{P}_{predicted}$

Inputs:

Un = Control vector. This indicates the magnitude of any control system's or user's control on the situation.

Zn = Measurement vector. This contains the real-world measurement we received in this time step.

Outputs:

Xn = Newest estimate of the current "true" state.

Pn = Newest estimate of the average error for each part of the state.

Constants:

A = State transition matrix. Basically, multiply state by this and add control factors, and you get a prediction of the state for the next time step.

B = Control matrix. This is used to define linear equations for any control factors.

H = Observation matrix. Multiply a state vector by H to translate it to a measurement vector.

Q = Estimated process error covariance. Finding precise values for Q and R are beyond the scope of this guide.

R = Estimated measurement error covariance. Finding precise values for Q and R are beyond the scope of this guide.

To program a Kalman Filter class, your constructor should have all the constant matrices, as well as an initial estimate of the state (x) and error (P). The step function should have the inputs (measurement and control vectors). In my version, you access outputs through "getter" functions.

Here is a working Kalman Filter object, written in Python:

```
# Implements a linear Kalman filter.
class KalmanFilterLinear:
  def __init__(self,_A, _B, _H, _x, _P, _Q, _R):
    self.A = _A                      # State transition matrix.
    self.B = _B                      # Control matrix.
    self.H = _H                      # Observation matrix.
    self.current_state_estimate = _x # Initial state estimate.
    self.current_prob_estimate = _P  # Initial covariance estimate.
    self.Q = _Q                      # Estimated error in process.
    self.R = _R                      # Estimated error in measurements.
  def GetCurrentState(self):
    return self.current_state_estimate
  def Step(self,control_vector,measurement_vector):
    #---------------------------Prediction step-----------------------------
    predicted_state_estimate = self.A * self.current_state_estimate + self.B * control_vector
    predicted_prob_estimate = (self.A * self.current_prob_estimate) * numpy.transpose(self.A) + self.Q
    #--------------------------Observation step-----------------------------
    innovation = measurement_vector - self.H*predicted_state_estimate
    innovation_covariance = self.H*predicted_prob_estimate*numpy.transpose(self.H) + self.R
    #-----------------------------Update step-------------------------------
    kalman_gain = predicted_prob_estimate * numpy.transpose(self.H) * numpy.linalg.inv(innovation_covariance)
    self.current_state_estimate = predicted_state_estimate + kalman_gain * innovation
    # We need the size of the matrix so we can make an identity matrix.
    size = self.current_prob_estimate.shape[0]
    # eye(n) = nxn identity matrix.
    self.current_prob_estimate = (numpy.eye(size)-kalman_gain*self.H)*predicted_prob_estimate
```

Single-Variable Example

Situation

This is a **classic example**. We will attempt to measure a constant DC voltage with a noisy voltmeter. We will use the Kalman filter to filter out the noise and converge toward the true value.

The state transition equation:

$$V_n = V_{n-1} + w_n$$

Vn = The current voltage.

Vn-1 = The voltage last time.

Wn = Random noise (measurement error).

Since the voltage never changes, it is a very simple equation. The objective of the Kalman filter is to mitigate the influence of Wn in this equation.

Simulation

In Python, we simulate this situation with the following object:

```
class Voltmeter:
  def __init__(self,_truevoltage,_noiselevel):
    self.truevoltage = _truevoltage
    self.noiselevel = _noiselevel
  def GetVoltage(self):
    return self.truevoltage
  def GetVoltageWithNoise(self):
    return random.gauss(self.GetVoltage(),self.noiselevel)
```

Preparation

The next step is to prepare the Kalman filter inputs and constants. Since this is a single-variable example, all matrices are 1x1.

Matrix "A" is what you need to multiply to last time's state to get the newest state. Since this is a constant voltage, we just multiply by 1.

$$V_n = V_{n-1} \rightarrow A = \left[\begin{array}{c} 1 \end{array} \right]$$

A = 1

Matrix "H" is what you need to multiply the incoming measurement to convert it to a state. Since we get the voltage directly, we just multiply by 1.

H = 1

Matrix "B" is the control matrix. It's a constant voltage and there's no input in the model we can change to affect anything, so we'll set it to 0.
B = 0

Matrix "Q" is the process covariance. Since we know the exact situation, we'll use a very small covariance.

Q = 0.00001

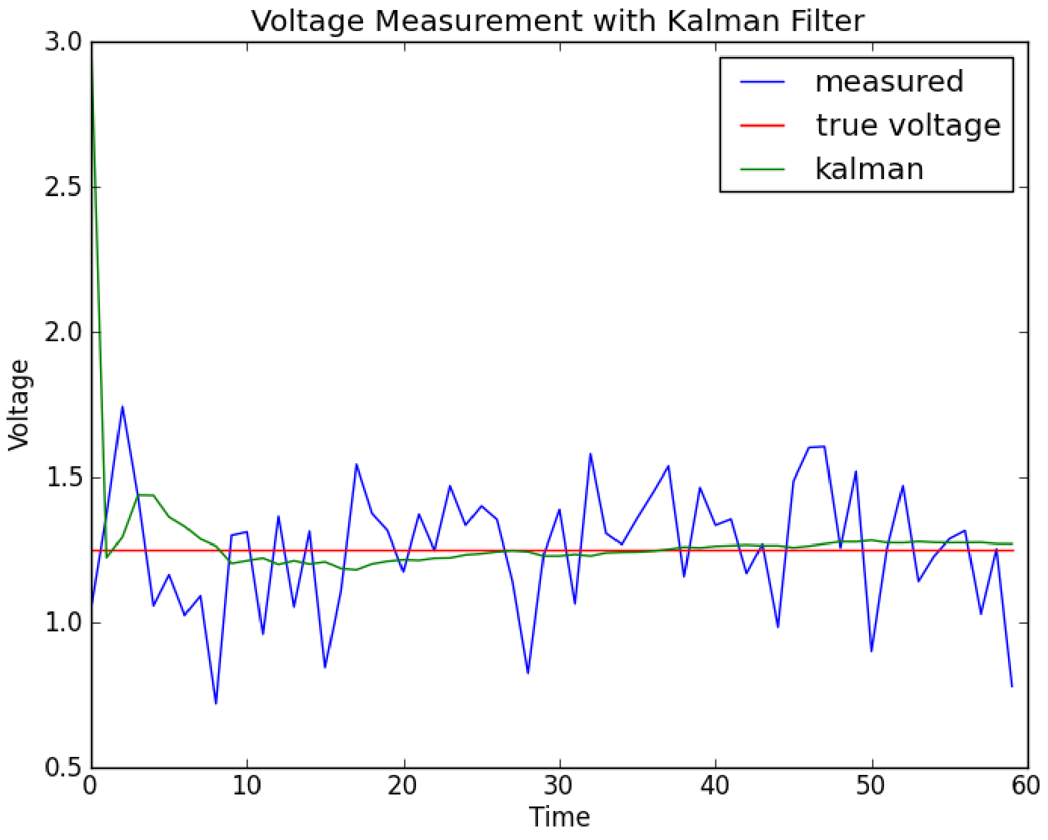
Matrix "R" is the measurement covariance. We'll use a conservative estimate of 0.1.
R = 0.1

Matrix "xhat" is your initial prediction of the voltage. We'll set it to 3 to show how resilient the filter is.

xhat = 3

Matrix "P" is your initial prediction of the covariance. We'll just pick an arbitrary value (1) because we don't know any better.
P = 1

Execution and Python code

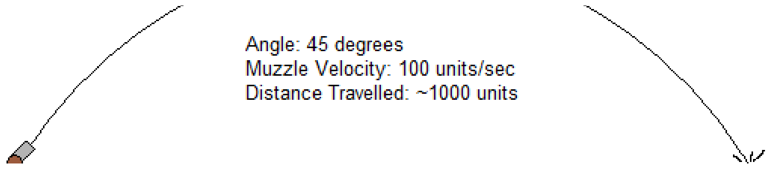


For brevity, I don't list all the code in this guide. The complete file is available: **kalman1.py** . I highly recommend you read through it and play with the starting values.

Multi-Variable Example

Situation

We will use a common physics problem with a twist. This example will involve firing a ball from a cannon at at 45-degree angle at a muzzle velocity of 100 units/sec. However, we will also have a camera that will record the ball's position from the side every second. The positions measured from the camera have significant measurement error. We also have precise detectors in the ball that give the X and Y velocity every second.


Angle: 45 degrees
Muzzle Velocity: 100 units/sec
Distance Travelled: ~1000 units

(not drawn to scale)

The kinematic equations for this situation, done to death in physics classes around the world, are:

$$\begin{aligned} x(t) &= x_0 + V_{0x}t \\ V_x(t) &= V_{0x} \\ y(t) &= y_0 + V_{0y}t - \tfrac{1}{2}gt^2 \\ V_y(t) &= V_{0y} - gt \end{aligned}$$

Our filter is discrete. Let's convert these equations to a recurrence relation, with delta-t being a fixed time step:

$$\begin{aligned} x_n &= x_{n-1} + V_{xn-1}\Delta t \\ V_{xn} &= V_{xn-1} \\ y_n &= y_{n-1} + V_{y_{n-1}}\Delta t - \tfrac{1}{2}g\Delta t^2 \\ V_{y_n} &= V_{y_{n-1}} - g\Delta t \end{aligned}$$

These equations can be represented in an alternate form shown here:

$$\begin{bmatrix} x_n \\ V_{xn} \\ y_n \\ V_{y_n} \end{bmatrix} = \begin{bmatrix} x_{n-1} & + & V_{xn-1}\Delta t & & & \\ & & V_{xn-1} & & & \\ & & & y_{n-1} & + & V_{y_{n-1}}\Delta t \\ & & & & & V_{y_{n-1}} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\tfrac{1}{2}g\Delta t^2 \\ -g\Delta t \end{bmatrix}$$

Note that rows are the summation of the influence of the variables in each column. This notation will be important to illustrate something in the Preparation step, so remember it.

Simulation

We will simulate the cannonball's trajectory mid-flight with the following Python object:

```
# Simulates the classic physics problem of a cannon shooting a ball in a
# parabolic arc.  In addition to giving "true" values back, you can also ask
# for noisy values back to test Kalman filters.
class Cannon:
  #--------------------------------VARIABLES----------------------------------
  angle = 45 # The angle from the ground to point the cannon.
  muzzle_velocity = 100 # Muzzle velocity of the cannon.
  gravity = [0,-9.81] # A vector containing gravitational acceleration.
  # The initial velocity of the cannonball
  velocity = [muzzle_velocity*math.cos(angle*math.pi/180), muzzle_velocity*math.sin(angle*math.pi/180)]
  loc = [0,0] # The initial location of the cannonball.
  acceleration = [0,0] # The initial acceleration of the cannonball.
  #---------------------------------METHODS-----------------------------------
  def __init__(self,_timeslice,_noiselevel):
    self.timeslice = _timeslice
    self.noiselevel = _noiselevel
  def add(self,x,y):
    return x + y
  def mult(self,x,y):
    return x * y
  def GetX(self):
    return self.loc[0]
  def GetY(self):
    return self.loc[1]
  def GetXWithNoise(self):
    return random.gauss(self.GetX(),self.noiselevel)
  def GetYWithNoise(self):
    return random.gauss(self.GetY(),self.noiselevel)
  def GetXVelocity(self):
    return self.velocity[0]
  def GetYVelocity(self):
    return self.velocity[1]
  # Increment through the next timeslice of the simulation.
  def Step(self):
    # We're gonna use this vector to timeslice everything.
    timeslicevec = [self.timeslice,self.timeslice]
    # Break gravitational force into a smaller time slice.
    sliced_gravity = map(self.mult,self.gravity,timeslicevec)
    # The only force on the cannonball is gravity.
    sliced_acceleration = sliced_gravity
    # Apply the acceleration to velocity.
    self.velocity = map(self.add, self.velocity, sliced_acceleration)
    sliced_velocity = map(self.mult, self.velocity, timeslicevec )
    # Apply the velocity to location.
    self.loc = map(self.add, self.loc, sliced_velocity)
    # Cannonballs shouldn't go into the ground.
    if self.loc[1] < 0:
      self.loc[1] = 0
```

Preparation

Remember this alternate form of the kinematic equations I showed you?

$$\begin{bmatrix} x_n \\ V_{xn} \\ y_n \\ V_{y_n} \end{bmatrix} = \begin{bmatrix} x_{n-1} + V_{xn-1}\Delta t & & \\ V_{xn-1} & & \\ & & y_{n-1} + V_{y_{n-1}}\Delta t \\ & & V_{y_{n-1}} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{2}g\Delta t^2 \\ -g\Delta t \end{bmatrix}$$

The first part is A in disguise (with the variables removed), and the second part is the control vector, provided you use the B matrix shown here:

Compare:

$$\mathbf{x}_{predicted} = \mathbf{A}\mathbf{x}_{n-1} + \mathbf{B}\mathbf{u}_n$$

to:

$$x_n = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} x_{n-1} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -\frac{1}{2} g \Delta t^2 \\ -g \Delta t \end{bmatrix}$$

When a state is multiplied by H, it is converted to measurement notation. Since our measurements map directly to the state, we just have to multiply them all by 1 to convert our measurements to state:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X0 is the initial guess for the ball's state. Notice that this is where we break the muzzle velocity into its X/Y components. We're also going to set y way off on purpose to illustrate how fast the filter can deal with this:

$$x_0 = \begin{bmatrix} x \\ V_x \\ y \\ V_y \end{bmatrix} = \begin{bmatrix} 0 \\ 100cos(\pi/4) \\ 500 \\ 100sin(\pi/4) \end{bmatrix}$$

P is our initial guess for the covariance of our state. We're just going to set them all to 1 since we don't know any better (and it works anyway). If you want to know better, consult a more detailed guide.

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Q is our estimate of the process error. Since we created the process (the simulation) and mapped our equations directly from it, we can safely assume there is no process error:

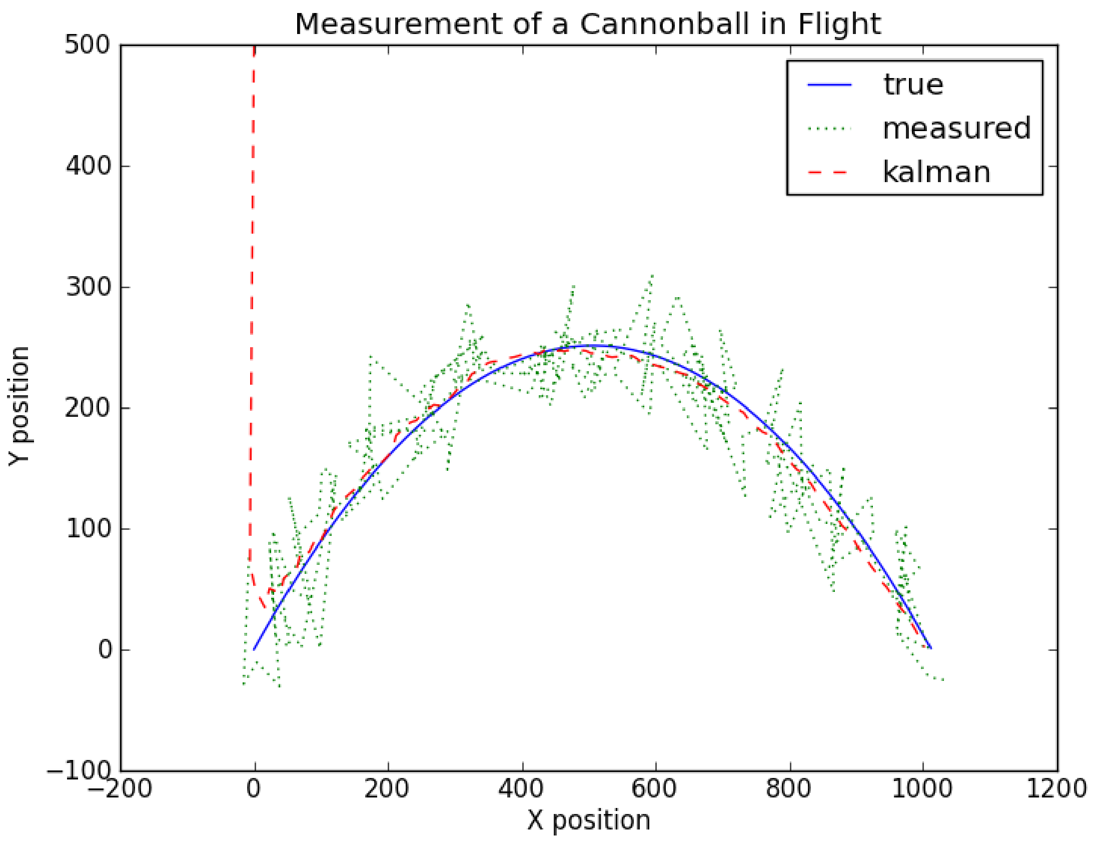
$$\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

R is our estimate of the measurement error covariance. We'll just set them all arbitrarily to 0.2. Try playing with this value in the real code.

$$R = \begin{bmatrix} 0.2 & 0 & 0 & 0 \\ 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 0.2 \end{bmatrix}$$

Execution and Python code



That's a pretty good estimate, given how much error our measurements have!

The complete code for this example is available at: **kalman2.py**

Conclusion

I hope that this guide was useful to you. This is a living document. If you have any suggestions, comments, etc, please send me a message at greg {at] czerniak (dot} info so I can make this guide even better. ESPECIALLY send me a message if you got confused somewhere -- you're probably not the first or last person that will get hung up at that point.

References

Kalman Filters

- **The Kalman Filter**: A great launching point for information about the Kalman filter.
- **Kalman Filter**: Wikipedia page. I modified the equations from this page for my version.
- **Simultaneous Localization and Mapping (SLAM)** : Seeing the matrices in the PowerPoint slides inspired me to write this guide. Most of my examples are variations of examples in these slides.
- **A 3D State Space Formulation of a Navigation Kalman Filter for Autonomous Vehicles** : This text goes in-depth about using Kalman filters in robotics, and it has great introductory material.
- **Optimal Filtering** : This book goes into detail on using Kalman filters.

Tools

- **Latex2png, latex2gif, latex2eps, latex2jpg** : This site generates image files from LaTeX code.
- **PBworks** : This is how I organize my thoughts.

Special Thanks

- Special thanks to Stefan Wollny from Korea for detecting and reporting a variance/covariance typo.
- Special thanks to Dave Bloss of Topsfield, Massachusetts and Robby Nevels for detecting and reporting a mistake with the cannonball physics equations (1/2gt^2 instead of gt^2).
- Special thanks to Robby Nevels for detecting and reporting a mistake with the Kalman filter code (innovation was being calculated wrong).

Attachment	Size
kalman1.py.txt	3.15 KB
kalman2.py.txt	7.71 KB

Guides