Time Series Analysis
Recovering A Stochastic Signal

June 12, 2016

1 Time Series Analysis

1.1 Recovering A Stochastic Signal

- By Daniel Cuneo

This is a pretty basic example of how to filter and recover a random signal from a time series that that has a linear combination of confounding noise.

```
In [1]: #%install_ext https://raw.githubusercontent.com/rasbt/watermark/master/watermark.py
        %reload_ext watermark
        %watermark -p numpy,scipy,pandas,matplotlib
```

```
numpy 1.10.1
scipy 0.16.0
pandas 0.16.2
matplotlib 1.4.0
```

```
In [294]: import matplotlib.pyplot as plt
          import numpy as np
          import pandas as pd
          import scipy.signal as signal
          %matplotlib inline
```

```
In [295]: df = pd.read_csv("data.csv")
```

```
In [296]: df.head()
```

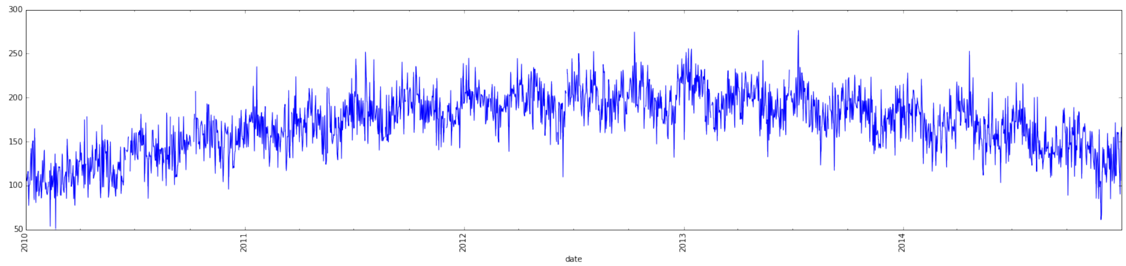
Out[296]:

	date	day.of.week	car.count	weather
0	2010-01-01	friday	94.5	-0.1
1	2010-01-02	saturday	108.4	-2.4
2	2010-01-03	sunday	105.5	-0.5
3	2010-01-04	monday	109.6	-2.1
4	2010-01-05	tuesday	116.1	1.9

```
In [297]: # I like using Pandas b/c of the datetime features, resample or groupby
          df['date'] = pd.to_datetime(df['date'])
          df.set_index(df['date'], inplace=True)
```

1.2 Initial Plot

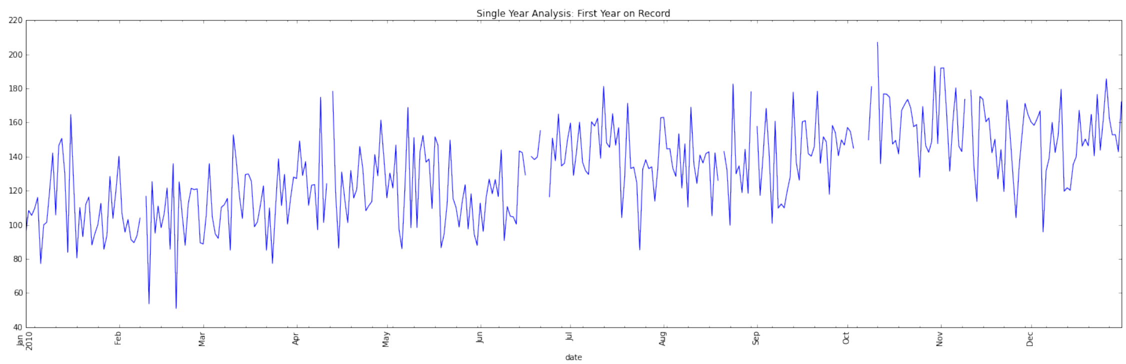
```
In [298]: df['car.count'].plot(rot=90, figsize=(25, 5));
```



I'd guess that we have a linear combination of a quadratic, sinusoid and random stochastic signal.

1.2.1 Single Year Analysis: first year in the record

```
In [299]: # year = df['car.count'][0:365]  if you are in a rush
          year = df[df['date'] < pd.to_datetime('20110101')]['car.count']
          year.plot(rot=90, figsize=(25, 7), title="Single Year Analysis: First Year on Rec
```



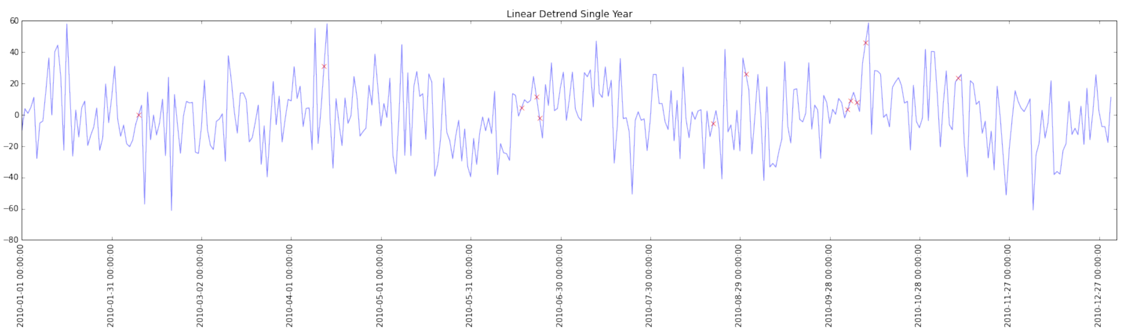
It's not easy to see, but there are missing values in the series. We need to treat those.

```
In [300]: #TODO: add to signal processing module

          def remove_nans(data, return_nan_index=False):
              nan_ind = np.nonzero(~np.isfinite(data))[0]
              good_data_ind = np.nonzero(np.isfinite(data))[0]
              good_data = data[good_data_ind]

              new_points = np.interp(nan_ind, good_data_ind, good_data)
              data[nan_ind] = new_points

              if return_nan_index:
                  return data, nan_ind
              else:
                  return data
```

```
In [301]: year, nan_ind = remove_nans(year.copy(), return_nan_index=True)
          year_linear_det = signal.detrend(year, axis=0, type='linear')
```

Single Year Linear Detrend Time Series

```
In [302]: plt.figure(figsize=(25, 5))
          plt.xticks(np.arange(year.shape[0])[0::30], year.index[0::30], rotation=90)
          plt.plot(year_linear_det, alpha=0.5)
          plt.title("Linear Detrend Single Year")
          plt.plot(nan_ind, year_linear_det[nan_ind], 'rx')
          plt.xlim(0, 366)

Out[302]: (0, 366)
```



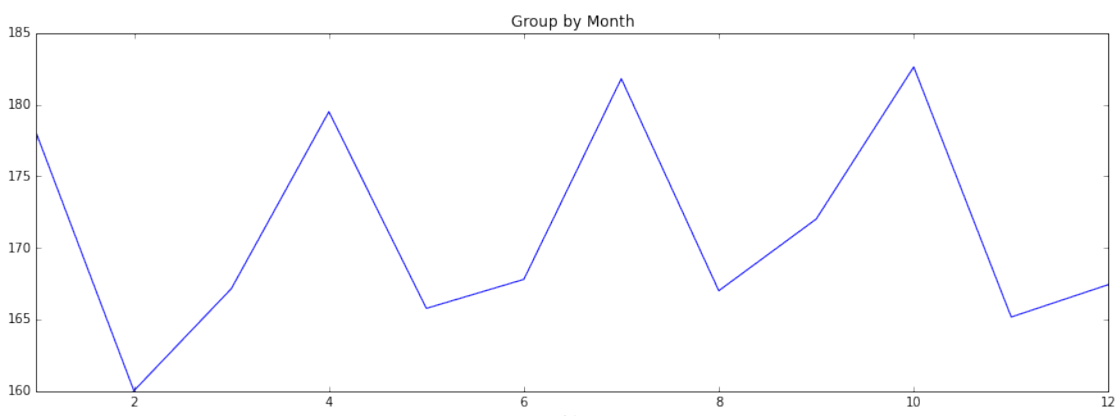
We see a ≈ 90 day period here.
Without more insight about the data we don't know if this is a nuisance or a feature we are looking for.
The NaN replacements look reasonable.

1.3 Group By for Basic Analysis

Grouping the data points into bins and taking the mean, is very simialr to a Fourier Transform.
Pandas makes this easy and there's no reason not to. Especially if the data is related to business trends.

1.3.1 Group by Month: Global monthly trend averaging over the 5 samples of each month

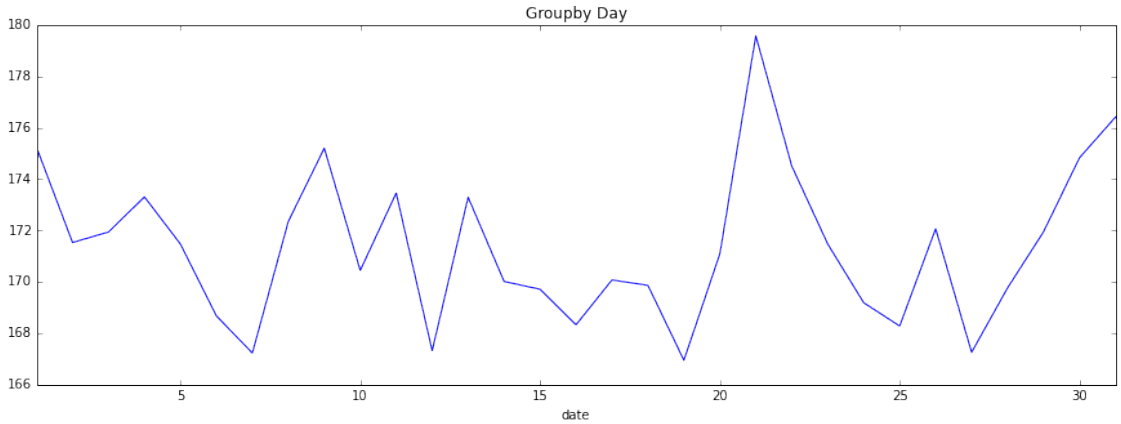
```
In [303]: grp = df.groupby(df.date.map(lambda x:x.month))
          grp.mean()['car.count'].plot(figsize=(15,5), title="Group by Month");
```



A group-by is sort of like a Fourier Transform where we choose just one frequency bin. There's the sinusodial period ≈ 90 days.

1.3.2 Group by Day: Global day trend averaging over the 5 samples of each day

In [304]:
```
grp = df.groupby(df.date.map(lambda x:x.day))
grp.mean()['car.count'].plot(figsize=(15,5), title="Groupby Day");
```



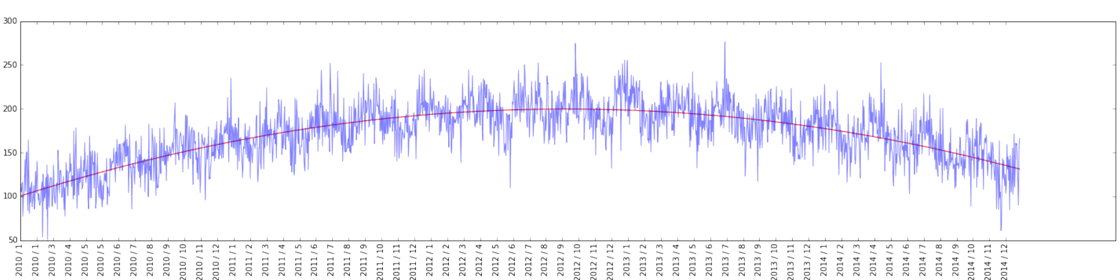
It would appear as though the 21st day of each month saw greater count.

1.4 Removing Confounds

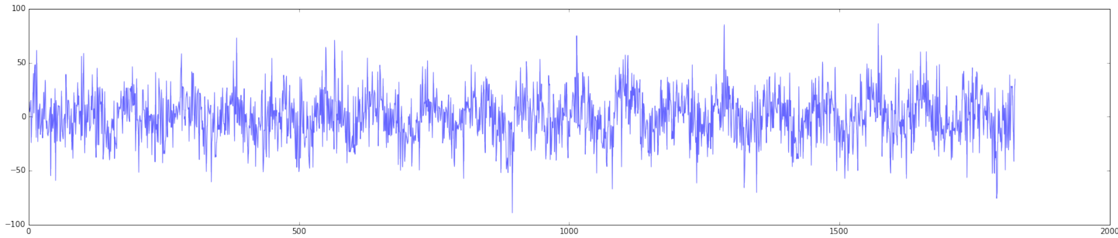
1.4.1 Quadratic Detrend Using PolyFit

In [305]:
```
poly =  np.polynomial.polynomial
counts = remove_nans(df['car.count'].copy(), return_nan_index=False)

t = np.arange(df.shape[0])
coefs = poly.polyfit(t, counts, deg=2, full=False)
fit_curve = poly.polyval(t, coefs)
```

In [306]:
```
plt.figure(figsize=(25, 5))

plt.plot(fit_curve, 'r')
plt.plot(t, counts, alpha=0.5)

labels = df['date']
date_str = map(lambda x: str(x.year) + " / "  + str(x.month), labels)
plt.xticks(t[0::30], date_str[0::30], rotation="vertical");
```



```
In [307]: det_curve = counts - fit_curve
          plt.figure(figsize=(25, 5))
          plt.plot(det_curve, alpha=0.6);
```



1.5 Further Confound Removal

Lets suppose that the quadratic is a measurement error and that the ≈ 90 day sinusodial is a well understood or nuisance, then we'll examine the remainder of the signal.

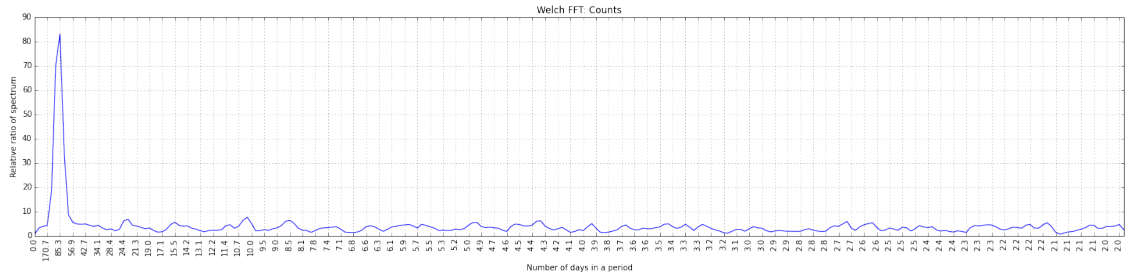
```
In [308]: import sys
          sys.path.append("/git/Python/Modules")
          import SignalProcessTools

          sigtools = SignalProcessTools.SignalProcessTools()
```

1.5.1 FFT

I keep this method handy and it should be in my Sigtools Module. It's just as well that you can see inside the Welch call.

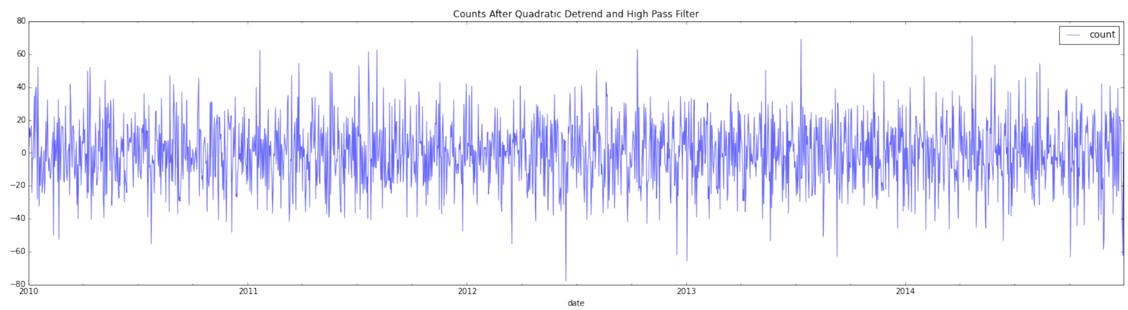
```
In [309]: def fft(data):
                '''Plot FFT using Welch's method, daily resolution '''
                f, y = signal.welch(data, fs=1.0, nperseg=365, noverlap=None, nfft=512, s

                interval = 3 # days
                periods = np.round(1/f[0::interval], 1)
                # clean up frequency of 0 Hz
                periods[0] = 0

                frqs = f[0::interval]
                plt.xticks(frqs, periods, rotation="vertical")

                plt.plot(f, y)

                plt.grid(True) # not working likely b/c of conflict with seaborn artist
                plt.title("Welch FFT: Counts")
                plt.ylabel("Relative ratio of spectrum")
                plt.xlabel("Number of days in a period")

                return f, y, frqs
```

```
In [310]: plt.figure(figsize=(25, 5))
          f, y, frqs = fft(counts)
```




```
In [311]: frq = 1 / 56.9 # from FFT output above
          out = sigtools.hi_pass_filter(det_curve, frq, 1.0, 3)

          dff = pd.DataFrame({'count':out}, index=df.index)
          dff.plot(title="Counts After Quadratic Detrend and High Pass Filter", figsize=(25
```

```
Out[311]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8673f5b9d0>
```

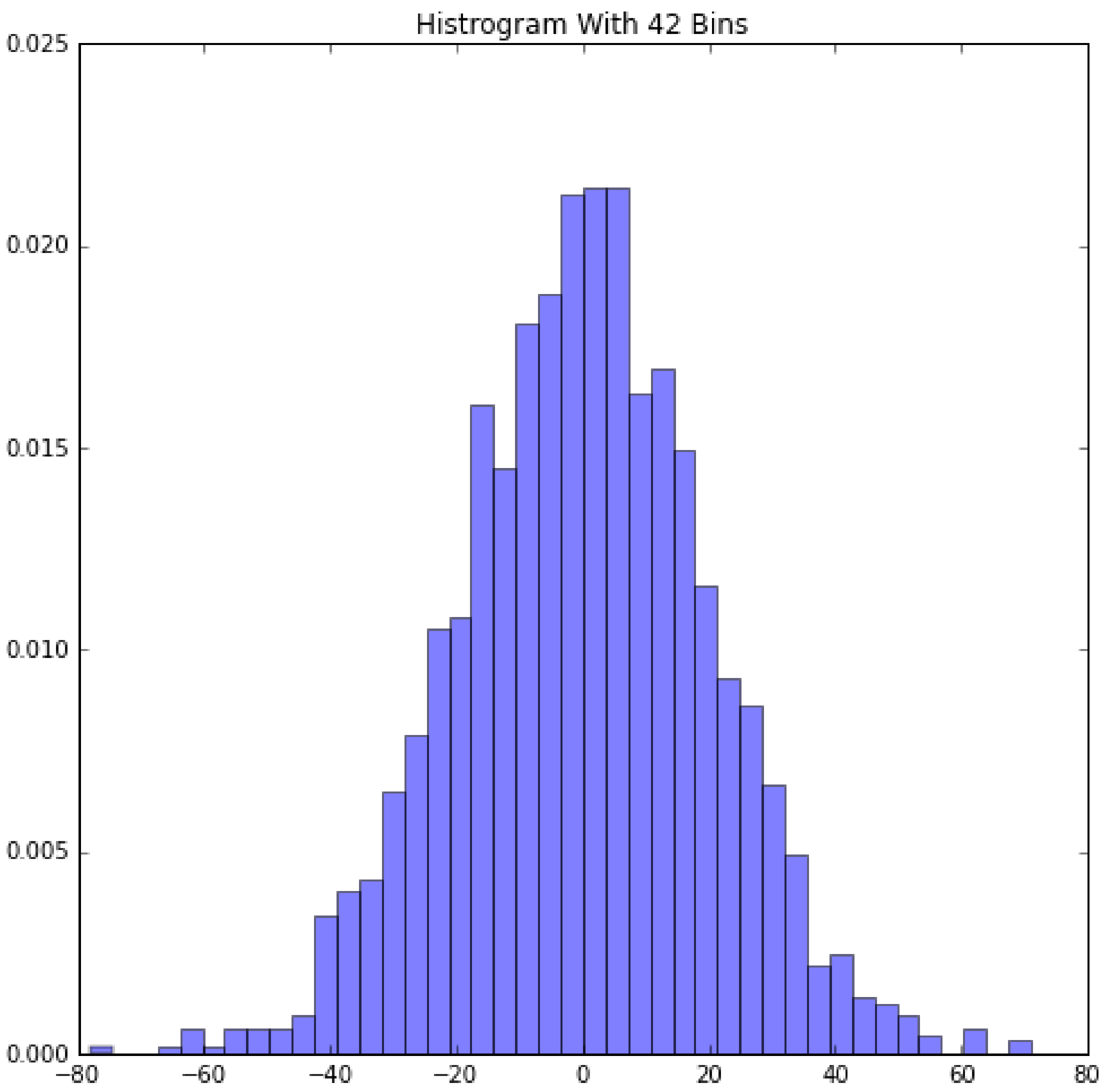


1.6 Fit A Distribution

The somewhat manual way and after with Seaborn

```
In [312]: plt.figure(figsize=(8, 8))

          nbins = np.floor(np.sqrt(out.shape[0]))
          y_counts, bin_x, patch = plt.hist(out, nbins, normed=True, alpha=0.5);
          string = "Histrogram With %s Bins" % str(int(nbins))
          plt.title(string)
```

```
Out[312]: <matplotlib.text.Text at 0x7f8673e2dfd0>
```



Perhaps a Gausian

```
In [313]: bin_x = bin_x[1:]# drop the first bin to match the array lengths
```

```
In [314]: from scipy.optimize import curve_fit
```

```
In [315]: def gauss(x, *p):
              A, mu, sig = p
              gau = A * np.exp(-(x-mu)**2 / (2 * sig)**2)
              return gau
```

```
In [316]: bin_width = np.round(bin_x[1] - bin_x[0], decimals=1)
          print bin_width
```

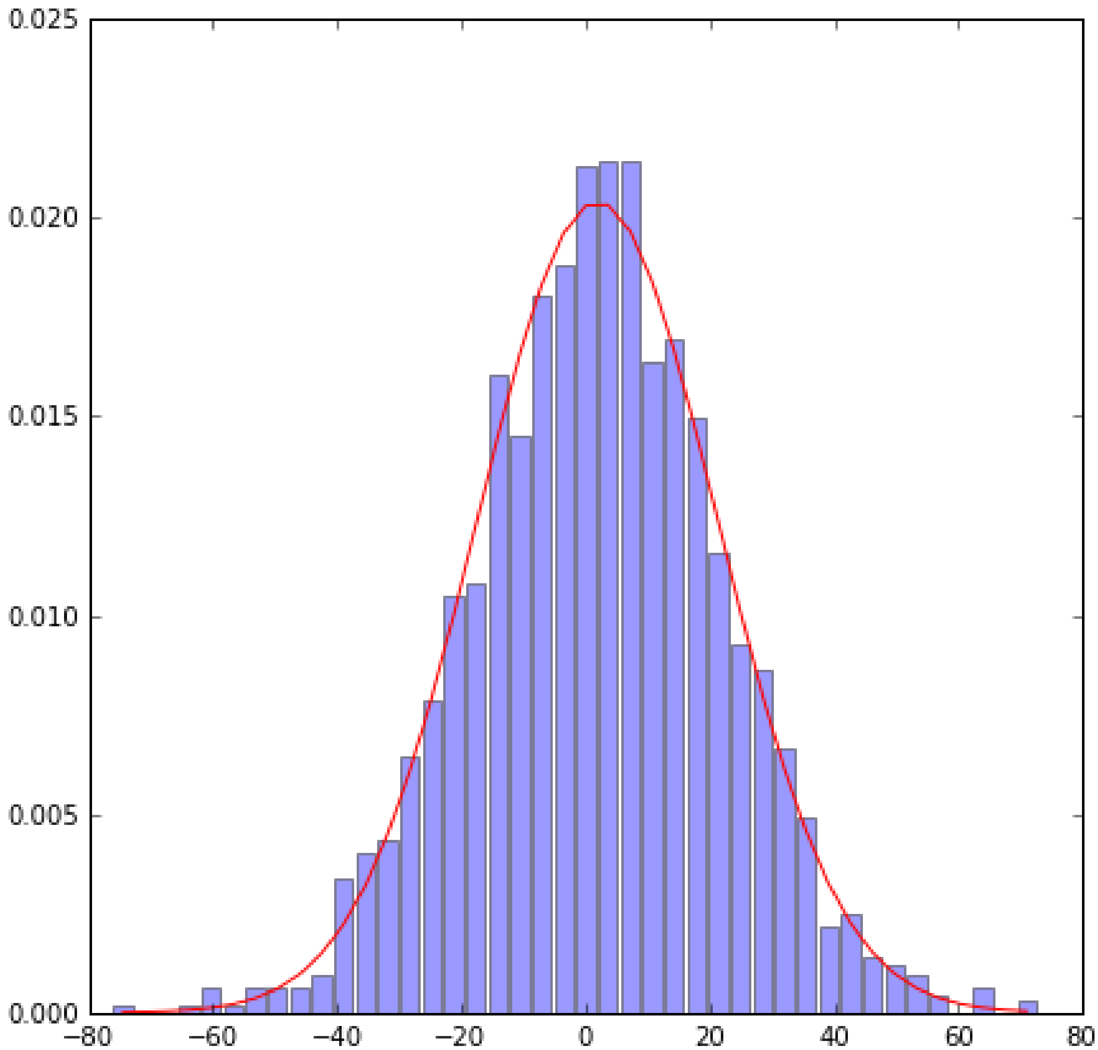
```
3.5
```

```
In [317]: coeff, var_matrix = curve_fit(gauss, bin_x, y_counts, p0=[0.1, 0.0, 1.0])
          print "Amplitude:%f  mean:%f   std:%f" %(coeff[0], coeff[1], coeff[2])
```

```
Amplitude:0.020397  mean:1.929398   std:13.724085
```

```
In [318]: fit_gau = gauss(bin_x, *coeff)
          plt.figure(figsize=(7, 7))

          plt.plot(bin_x, fit_gau, 'r')
          plt.bar(bin_x, y_counts, alpha=0.4, width=3, align='center')
```

```
Out[318]: <Container object of 42 artists>
```

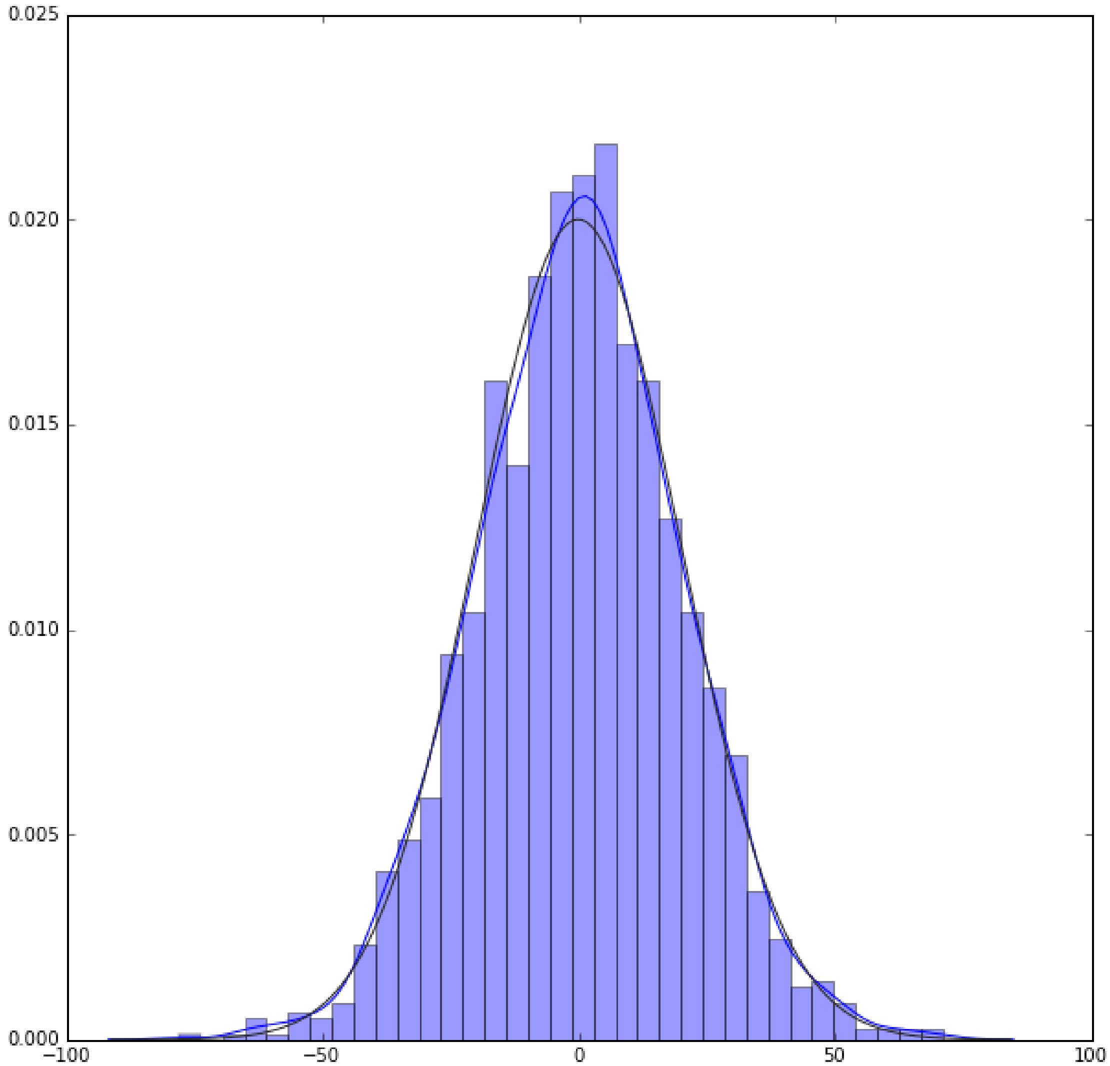


1.6.1 Try That Again With Seaborn

Here we see the Kernel Density Estimate and a Gamma fit, which is a prior for a Gaussian.

```
In [324]: import seaborn as sns
          from scipy import stats

          plt.figure(figsize=(10, 10))
          sns.distplot(out, kde=True, fit=stats.gamma)
```

Out[324]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8674078650>



1.7 Bootstrap Variance Estimation

```
In [320]: def bootstrap_mean(n, df, key):
              mu = 0
              for i in range(1, n+1):
                  update = df[key].sample(n=100, replace=True).std()
                  mu += (update - mu) / float(i)

              return mu
```

```
In [321]: dff = pd.DataFrame({"count":out})
          n = 2000

          psi_bar = bootstrap_mean(n, dff, 'count')
```

In [322]:
```
print psi_bar
```

```
19.8688617857
```

In [326]:
```
#sns.reset_orig()
plt.figure(figsize=(10, 10))

plt.plot(bin_x, fit_gau, 'r')
plt.bar(bin_x, y_counts, alpha=0.4, width=3, align='center')
plt.vlines(psi_bar, 0, 0.025, 'g')
plt.vlines(coeff[2], 0, 0.025, 'k')

plt.legend(('Gassian Fit', 'Bootstrap Std', 'Gaussian Fit Std'))
```

Out[326]: <matplotlib.legend.Legend at 0x7f86743e5b10>