Time Series Analysis: Recovering A Stochastic Signal

Daniel Cuneo

June 14, 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 [71]: ##install_ext https://raw.githubusercontent.com/rasbt/watermark/master/watermark.p
         %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 [72]: import matplotlib.pyplot as plt
         import numpy as np
         import pandas as pd
         import scipy.signal as signal
         %matplotlib inline
```

```
In [73]: df = pd.read_csv("/home/daniel/git/Python2.7/DataScience/notebooks/TimeSeries/data
```

```
In [74]: df.head()
```

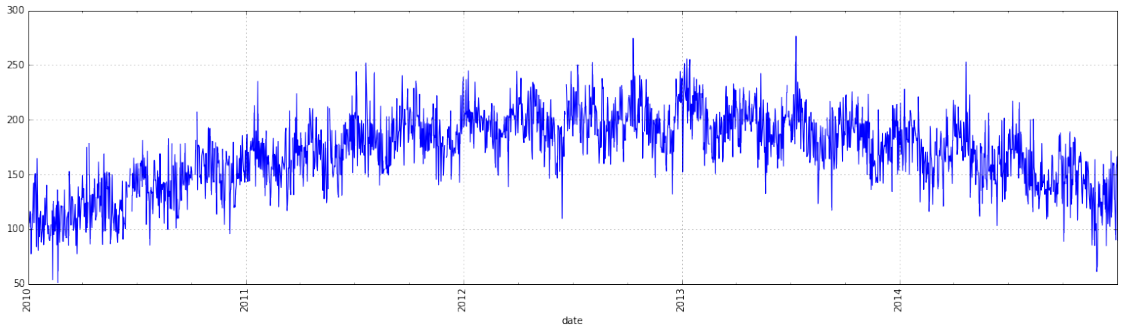
Out[74]:

	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 [75]: # 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

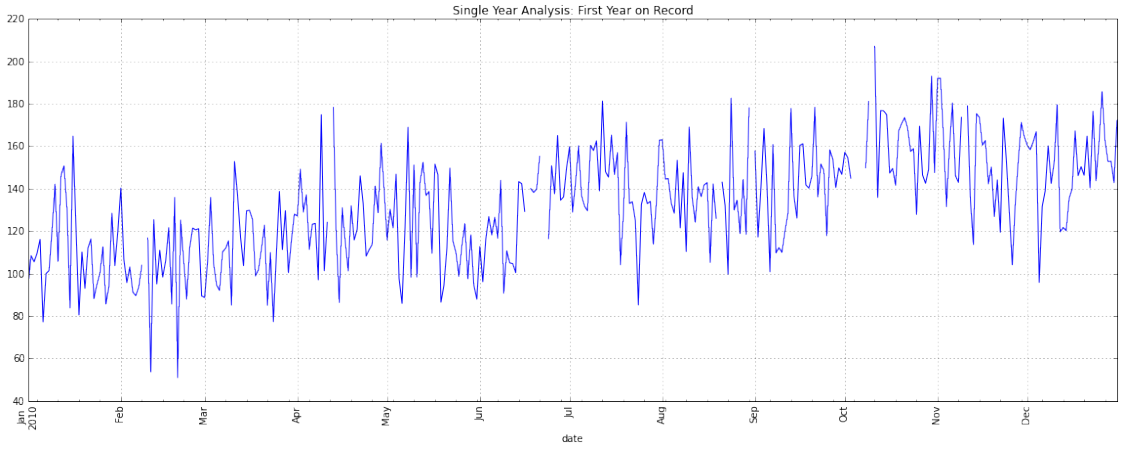
```
In [76]: df['car.count'].plot(rot=90, figsize=(20, 5), grid=True);
```



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 [77]: # 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=(20, 7), grid=True, title="Single Year Analysis: First
```



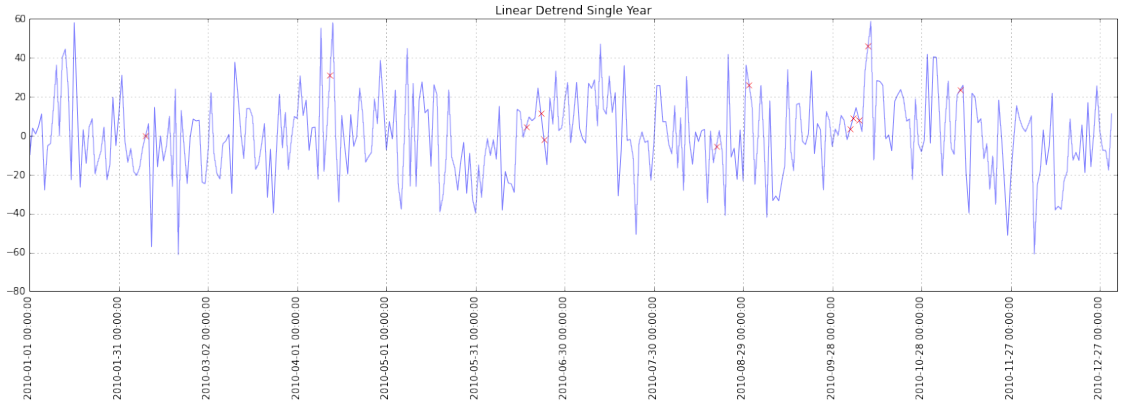
It's not easy to see, but there are missing values in the series. We need to treat those.

```
In [78]: #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 [79]: 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 [80]: plt.figure(figsize=(20, 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)
         plt.grid()
```



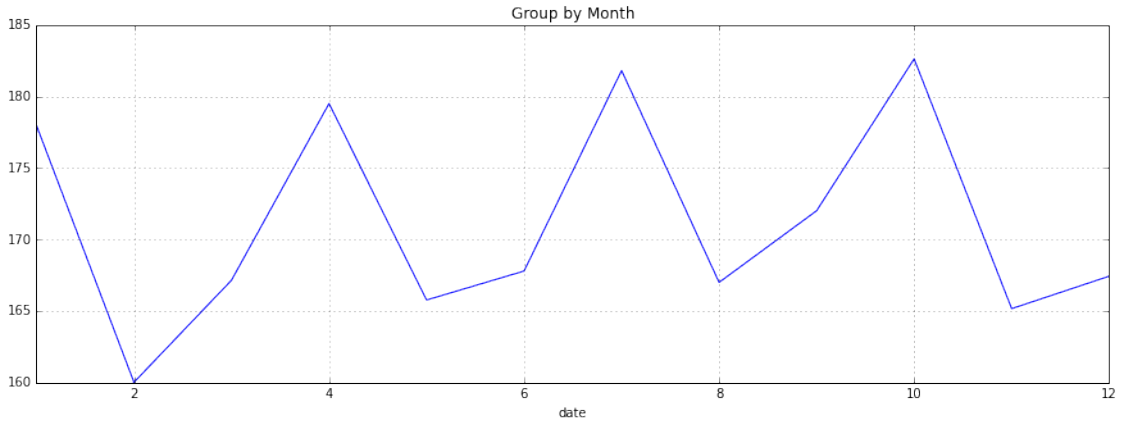
We see a ≈ 90 day period here.
Without more insight about the data we don't know if this is a nuisance or a feature we are looking for.
The NaN replacements look reasonable.

1.3 'Group By' for Basic Analysis

Grouping and aggregating is a simple and powerful way to gain insights into data sets. For large data, or even medium large, I use a SQL database. For small data like this, Pandas is perfect.

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

```
In [81]: grp = df.groupby(df.date.map(lambda x:x.month))
         grp.mean()['car.count'].plot(figsize=(15,5), grid=True, title="Group by Month");
```



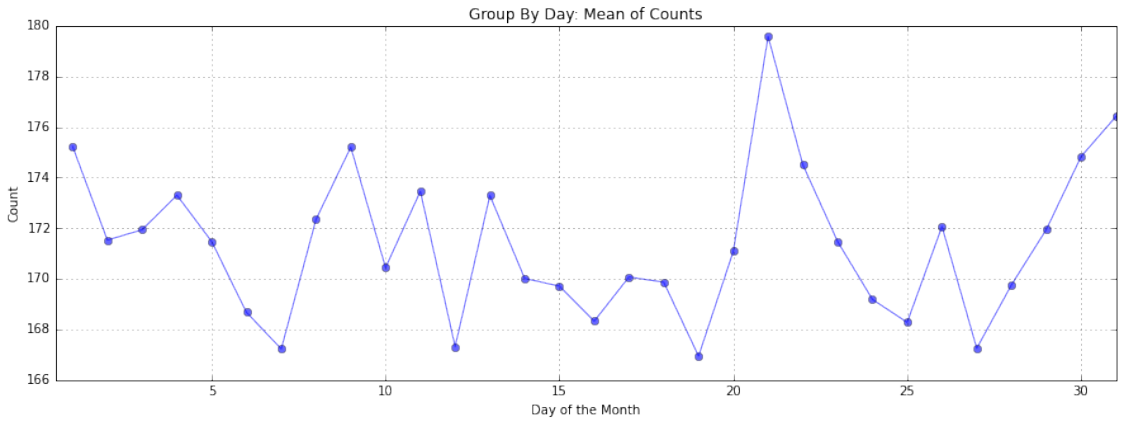
A group-by is sort of like a Fourier Transform where we choose just one frequency bin. There's the sinusodial period $\approx$ 90 days.

1.4 Tangent into statistical time series analysis

1.4.1 Group by Day: Global day trend averaging over the 5 years of 12 months

In the plot below, it would appear that the 21st day of every month, averaged over the 5 years, has had the highest count of cars. The 60 days which comprise the sample is not such a high number of samples. I'll take a moment to try out the bootstrap method of resampling.

```
In [82]: grp = df.groupby(df.date.map(lambda x:x.day))
         X = grp.mean()['car.count']

         # decided to not use Pandas plot method b/c I wanted the -o style of lines
         plt.figure(figsize=(15, 5))
         plt.plot(X.index, X, '-o', alpha=0.6);

         plt.title("Group By Day: Mean of Counts")
         plt.xlabel("Day of the Month")
         plt.ylabel("Count")

         plt.xlim(0.5, 31)
         plt.grid()
```



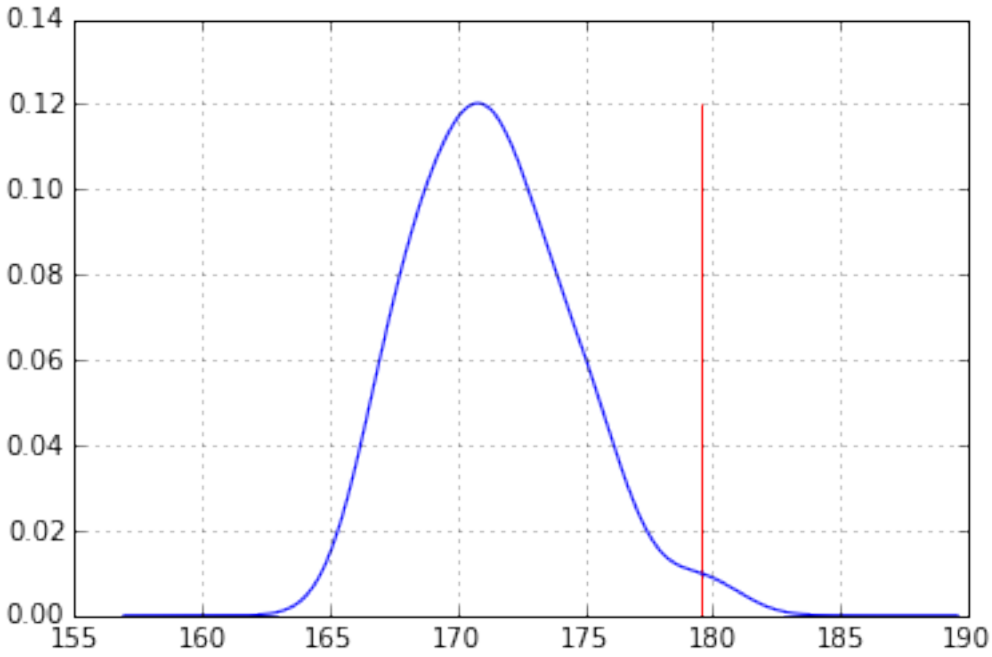
1.4.2 Significance

The density estimate below suggests that the max count on the 21st day is significant.

```
In [83]: from scipy.stats import gaussian_kde

         X = grp.mean()['car.count']
         N = X.shape[0]
         grid = np.linspace(X.min()-10, X.max()+10, 1000)

         kde = gaussian_kde(X, bw_method=None)
```

```
        out = kde.evaluate(grid)

        plt.plot(grid, out)
        plt.grid()
        max_cnt_ind = X.argmax()
        cnt_max = X[max_cnt_ind]

        ht = out.max()
        plt.vlines(cnt_max, 0, ht, 'r')

        max_ind = out.argmax()
        max_ = grid[max_ind]

        print "Max count in plot:    %.2f" % cnt_max
        print "Mean count over days: %.2f" % grid[max_ind]
```

```
Max count in plot:    179.57
Mean count over days: 170.79
```

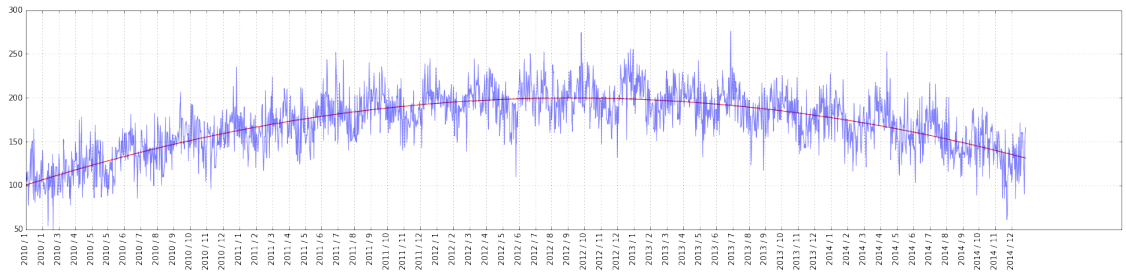


1.5 Removing Confounds

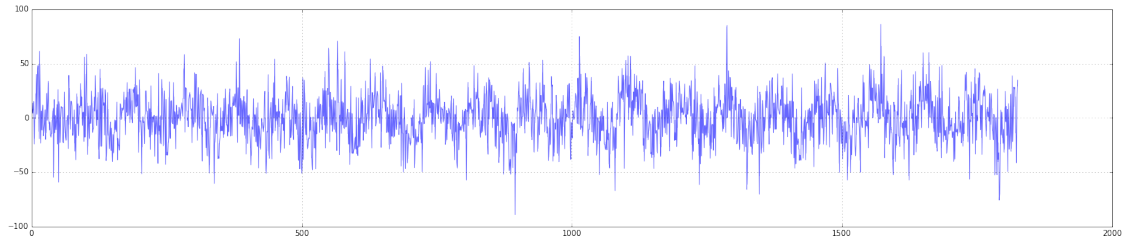
1.5.1 Quadratic Detrend Using PolyFit

In [84]:
```
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 [85]: plt.figure(figsize=(25, 5))

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

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



1.6 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.

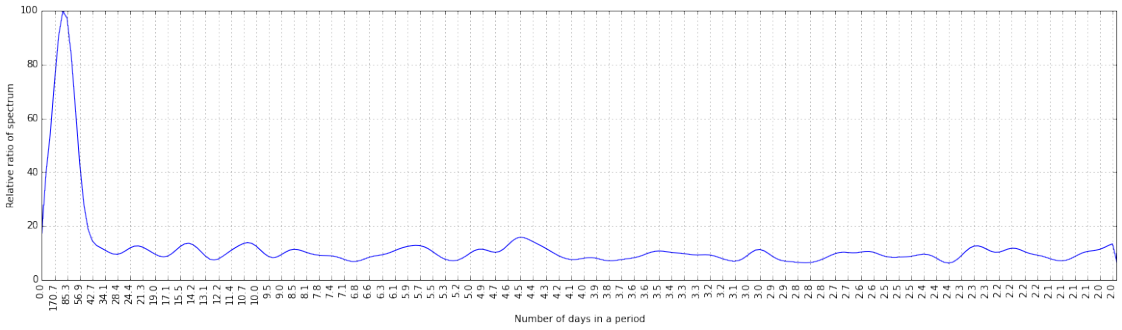
```
In [87]: import sys
         sys.path.append("/home/daniel/git/Python2.7/MRI/Modules")
         import SignalProcessTools

         sigtools = SignalProcessTools.SignalProcessTools()
```

1.6.1 Frequency Domain Analysis Using 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.

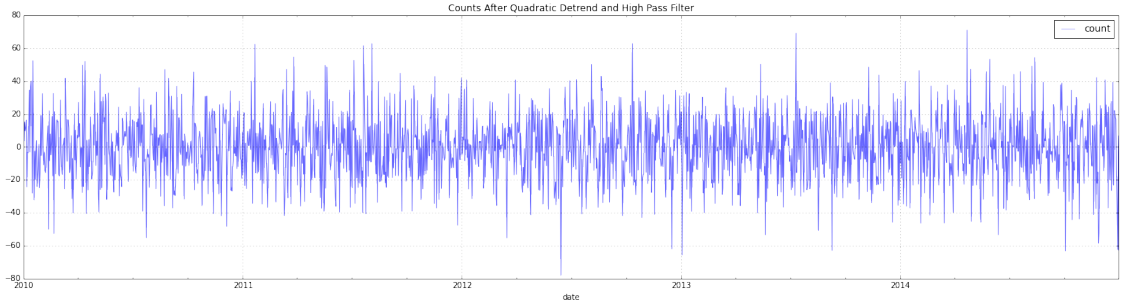
In [88]:
```
def fft(data):
    '''Plot FFT using Welch's method'''
    f, y = signal.welch(data, fs=1.0, nperseg=128, noverlap=64, nfft=512, scal

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

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

    plt.plot(f, y)
    #plt.semilogy(f, y)

    plt.grid(True)
    plt.ylabel("Relative ratio of spectrum")
    plt.xlabel("Number of days in a period")

    return f, y, frqs
```

In [89]:
```
plt.figure(figsize=(20, 5))
f, y, frqs = fft(det_curve)
```



In [90]:
```
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", grid=True, 
```

Out[90]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5a3254aa10>



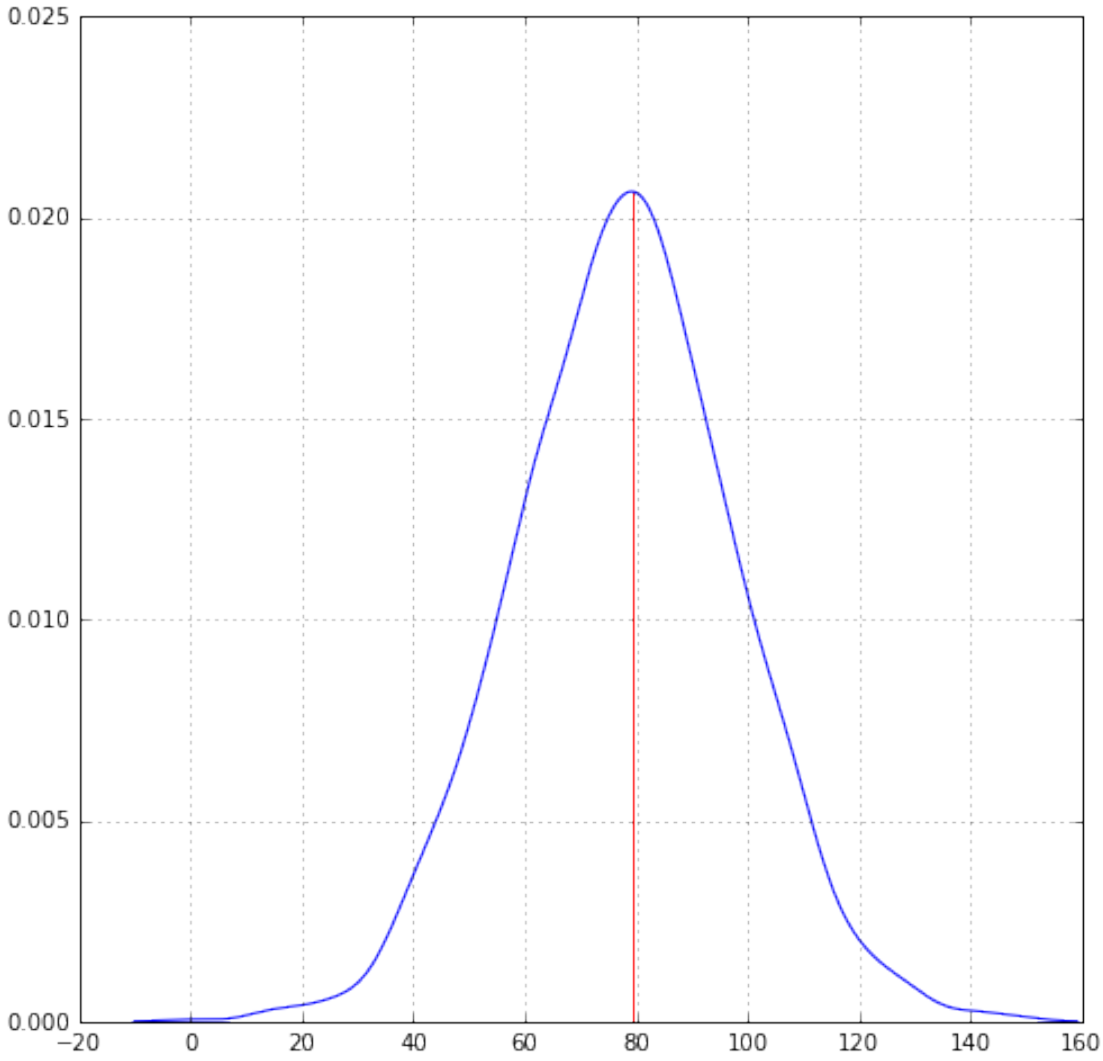
1.7 Kernel Density Estimate

http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gaussian_kde.html#scipy.stats.gaussian_kde

Sometime it makes sense to fit the data to a distribution and rely upon the distribution for the parameters like a mean and variance.

Let's assume that's a great thing to do here.

```
In [91]: # undo the centering that occures fromt he previous procesing
         count = dff['count'] - dff['count'].min()

         def comp_kde(data):
             N = data.shape[0]
             min_ = data.min() - 10
             max_ = data.max()+10
             grid = np.linspace(min_, max_, 1000)

             kde = gaussian_kde(data, bw_method=None)
             out = kde.evaluate(grid)

             return out, grid

In [92]: FIT, grid = comp_kde(count)
         plt.figure(figsize=(8, 8))
         plt.plot(grid, FIT)

         max_ind = FIT.argmax()
         MU_ = grid[max_ind]

         ht = FIT.max()
         plt.vlines(MU_, 0, ht, 'r')
         plt.grid()

         print "Mean of distribution: %.2f" % MU_

Mean of distribution: 79.18
```



1.7.1 Variance as width to inflection point

A decent way to characterize the width, is the use the inflection point.

If you want to see how that is done, take a look at my notebook on Wikipedia Counts and time series. In that notebook I use a Savitzsky-Golay filter and a numerical adaption of the 1st and 2nd derivative tests from calculus.

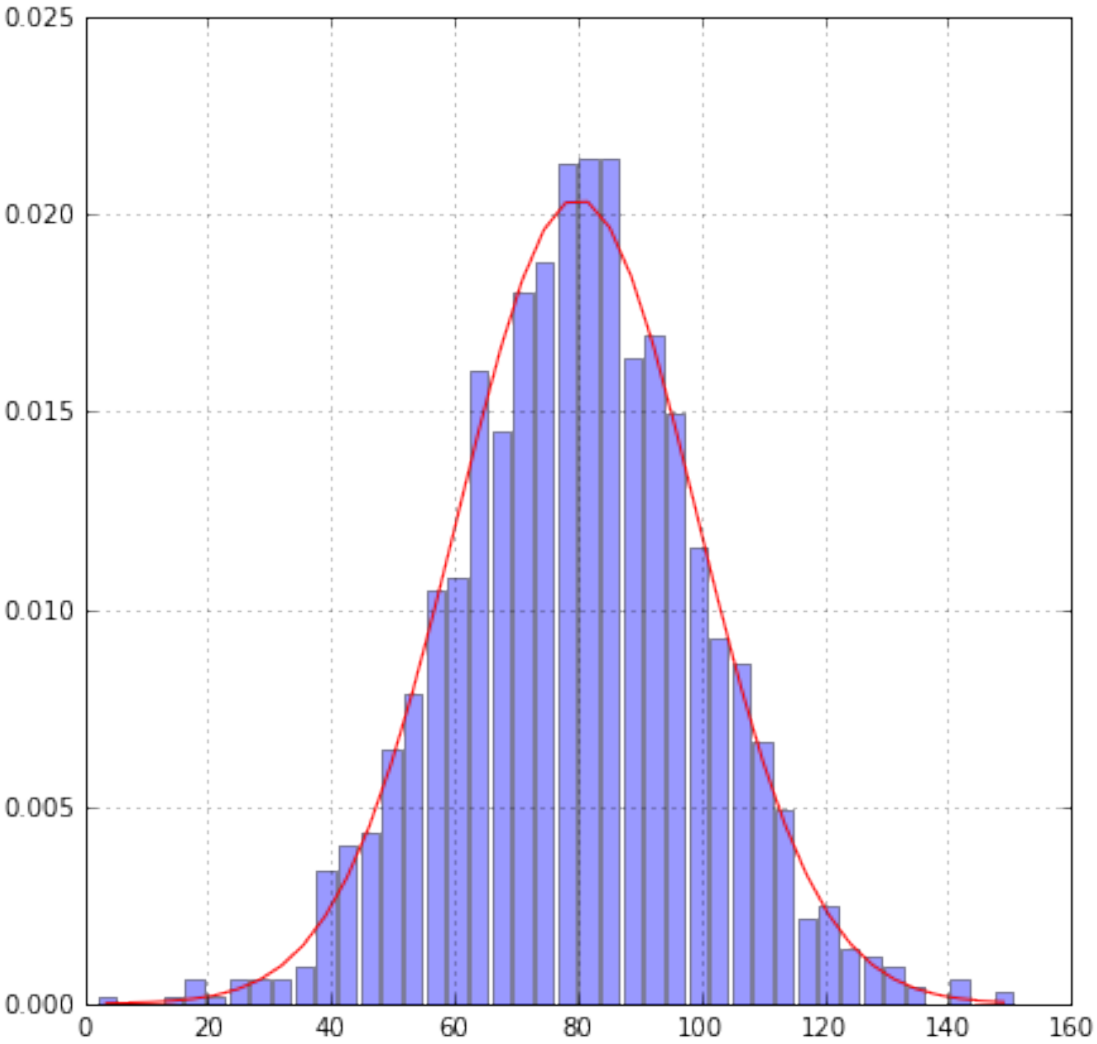
1.8 Gaussian Fit

In [93]:
```
from scipy.optimize import curve_fit

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

1.8.1 Make a histogram

```
In [94]: nbins = np.floor(np.sqrt(count.shape[0]))
         y_counts, bin_x = np.histogram(count, nbins, normed=True);
         bin_x = bin_x[1:]# drop the first bin to match the array lengths

         # fit the histogram to a Gaussian
         coeff, var_matrix = curve_fit(gauss, bin_x, y_counts, p0=[0.001, 0.0001, 20.0])
         print "Amplitude:%f  mean:%f   std:%f" %(coeff[0], coeff[1], coeff[2])
         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');
         plt.grid()
```

```
Amplitude:0.020397  mean:79.954072   std:13.724083
```



1.9 Jackknife Bias Estimation

Maybe we decide that the data looks Gaussian enough for us and we just compute the mean and standard deviation. You can certainly fit a Gaussian with Scipy but I'll avoid it here so that I can use the Jackknife method on something simple.

```
In [95]: def jk_params(data):
             mu = np.zeros_like(data)
             sig = np.zeros_like(data)
             n = data.shape[0]
             for i in range(n):
                 sample = np.delete(data, i)
                 mu[i] = sample.mean()
                 sig[i] = sample.std()

             return np.array([mu, sig])
```

```
In [96]: count = np.array(count) # cast from dataframe/series into numpy array
         jk_mu, jk_sig = jk_params(count).mean(axis=1)
```

1.9.1 Bias Calculation

Means are pretty robust estimator, but the standard deviation is not for low N.

```
In [97]: n = count.shape[0]
         mu_bias = (n - 1) * (jk_mu - count.mean())
         sig_bias = (n - 1) * (jk_sig - count.std())
```

```
In [98]: print "Original STD:              %.10f"    % count.std()
         print "Mean STD from Samples:    %.10f\n"  % jk_sig
         print "Bias:                     %e"       % sig_bias
```

```
Original STD:              19.9345480188
Mean STD from Samples:     19.9345433278

Bias:                      -8.561164e-03
```

```
In [99]: MU = count.mean() - mu_bias
         SIG = count.std() - sig_bias

         print "Mean:          %.2f" % MU # rounding to three sig figs.
         print "STD from KDE:  %3.2f" % SIG
```

```
Mean:          78.03
STD from KDE:  19.94
```

I'm choosing to round to a whole number since these are counts and I don't see the helpfulness of a fractional count for this data set.

$\mu_{count} = 78 \pm 20$