Maine-Quebec Data Generation

October 4, 2016

1 Testing for a Generalized Conjecture on Sums of Coefficients of Cusp Forms

Let f be a weight k cusp form with Fourier expansion

$$f(z) = \sum_{n \geq 1} a(n) e(nz).$$

Deligne has shown that $a(n) \ll n^{\frac{k-1}{2}+\epsilon}$. It is conjectured that

$$S_f^1(n) := \sum_{m \leq X} a(m) \ll X^{\frac{k-1}{2}+\frac{1}{4}+\epsilon}.$$

It is known that this holds on average, and we recently showed that this holds on average in short intervals. (See HKLDW1, HKLDW2, and HKLDW3 for details and an overview of work in this area). This is particularly notable, as the resulting exponent is only 1/4 higher than that of a single coefficient. This indicates extreme cancellation, far more than what is implied merely by the signs of $a(n)$ being random.

It seems that we also have

$$\sum_{m \leq X} S_f^1(m) \ll X^{\frac{k-1}{2}+\frac{2}{4}+\epsilon}.$$

That is, the sum of sums seems to add in only an additional 1/4 exponent. This is unexpected and a bit mysterious.

The purpose of this notebook is to explore this and higher conjectures. Define the jth iterated sum as

$$S_f^j(X) := \sum_{m \leq X} S_f^{j-1}(m).$$

Then we numerically estimate bounds on the exponent $\delta(j)$ such that

$$S_f^j(X) \ll X^{\frac{k-1}{2}+\delta(j)+\epsilon}.$$

```
In [1]: # This was written in SageMath 7.3 through a Jupyter Notebook.
        # Jupyter interfaces to sage by loading it as an extension
        %load_ext sage

        # sage plays strangely with ipython. This re-allows inline plotting
        from IPython.display import display, Image
```

We first need a list of coefficients of one (or more) cusp forms. For initial investigation, we begin with a list of 50,000 coefficients of the weight 12 cusp form on $\mathrm{SL}(2,\mathbb{Z})$, $\Delta(z)$, i.e. Ramanujan's delta function. We will use the data associated to the 50,000 coefficients for pictoral investigation as well.

We will be performing some numerical investigation as well. For this, we will use the first 2.5 million coefficients of $\Delta(z)$

```
In [2]: # Gather 10 coefficients for simple checking
        check_10 = delta_qexp(11).coefficients()
        print check_10

        fiftyk_coeffs = delta_qexp(50000).coefficients()
        print fiftyk_coeffs[:10] # these match expected

        twomil_coeffs = delta_qexp(2500000).coefficients()
        print twomil_coeffs[:10] # these also match expected
```

```
[1, -24, 252, -1472, 4830, -6048, -16744, 84480, -113643, -115920]
[1, -24, 252, -1472, 4830, -6048, -16744, 84480, -113643, -115920]
[1, -24, 252, -1472, 4830, -6048, -16744, 84480, -113643, -115920]
```

```
In [3]: # Function which iterates partial sums from a list of coefficients

        def partial_sum(baselist):
            ret_list = [baselist[0]]
            for b in baselist[1:]:
                ret_list.append(ret_list[-1] + b)
            return ret_list

        print check_10
        print partial_sum(check_10) # Should be the partial sums
```

```
[1, -24, 252, -1472, 4830, -6048, -16744, 84480, -113643, -115920]
[1, -23, 229, -1243, 3587, -2461, -19205, 65275, -48368, -164288]
```

```
In [4]: # Calculate the first 10 iterated partial sums
        # We store them in a single list list, 'sums_list'
        # the zeroth elelemnt of the list is the array of initial coefficients
        # the first element is the array of first partial sums, S_f(n)
        # the second element is the array of second iterated partial sums, S_f^2(n)

        fiftyk_sums_list = []
        fiftyk_sums_list.append(fiftyk_coeffs) # zeroth index contains coefficients
        for j in range(10):                    # jth index contains jth iterate
            fiftyk_sums_list.append(partial_sum(fiftyk_sums_list[-1]))

        print partial_sum(check_10)
        print fiftyk_sums_list[1][:10]         # should match above

        twomil_sums_list = []
        twomil_sums_list.append(twomil_coeffs) # zeroth index contains coefficients
        for j in range(10):                    # jth index contains jth iterate
            twomil_sums_list.append(partial_sum(twomil_sums_list[-1]))

        print twomil_sums_list[1][:10]         # should match above
```

```
[1, -23, 229, -1243, 3587, -2461, -19205, 65275, -48368, -164288]
[1, -23, 229, -1243, 3587, -2461, -19205, 65275, -48368, -164288]
[1, -23, 229, -1243, 3587, -2461, -19205, 65275, -48368, -164288]
```

As is easily visible, the sums alternate in sign very rapidly. For instance, we believe tha the first partial sums should change sign about once every $X^{1/4}$ terms in the interval $[X, 2X]$. In this exploration, we are

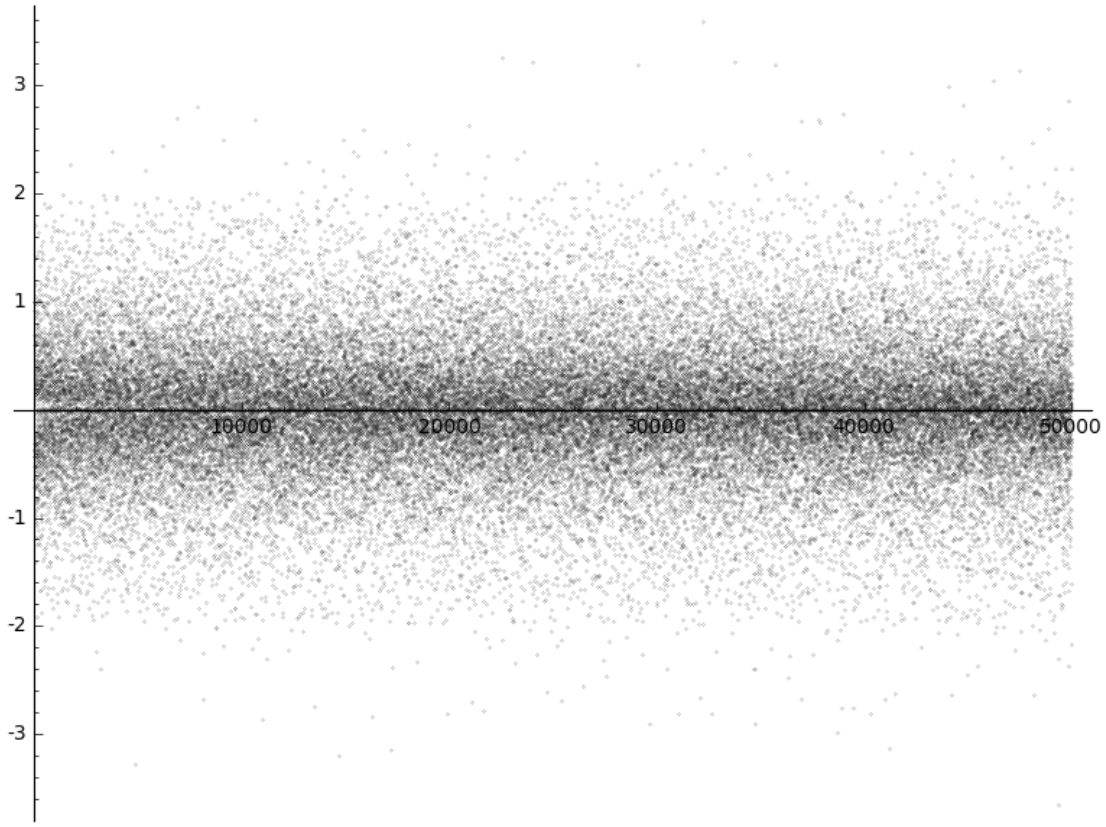
interested in the sizes of the coefficients. But in HKLDW3, we investigated some of the sign changes of the partial sums.

Now seems like a nice time to briefly look at the data we currently have. What do the first 50 thousand coefficients look like? So we normalize them, getting $A(n) = a(n)/n^{5.5}$ and plot these coefficients.

```
In [5]: norm_list = []
        for n,e in enumerate(fiftyk_coeffs, 1):
            normalized_element = 1.0 * e / (1.0 * n**(5.5))
            norm_list.append(normalized_element)
        print norm_list[:10]
```

```
[1.00000000000000, -0.530330085889911, 0.598733612492945, -0.718750000000000, 0.691213333204735, -0.3175
```

```
In [6]: # Make a quick display
        normed_coeffs_plot = scatter_plot(zip(range(1,60000), norm_list), markersize=.02)
        normed_coeffs_plot.save("normed_coeffs_plot.png")
        display(Image("normed_coeffs_plot.png"))
```

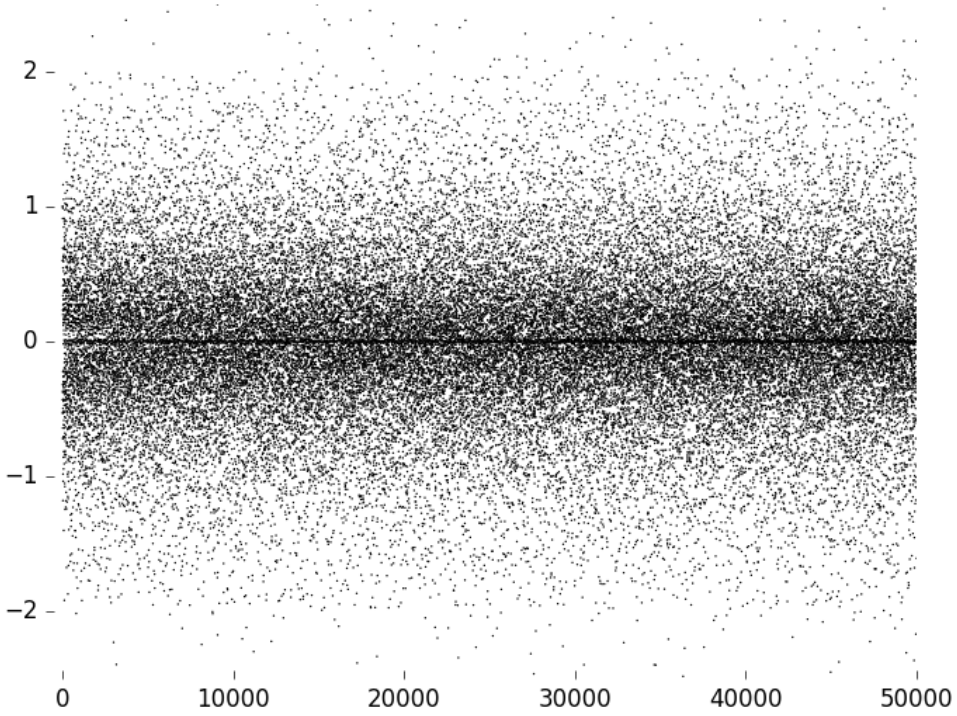


Since some figures will be featuring prominently in the talk I'm giving at Quebec-Maine, let us make high-quality figures now.

```
In [7]: # We will be using matplotlib itself to make some of our plots.
        # This import is necessary for us to interface between sage notebook and matplotlib.
        from matplotlib.backends.backend_agg import FigureCanvasAgg

        # We will use matplotlib's excellent pyplot
```

```
        import matplotlib.pyplot as plt

        # And numpy
        import numpy as np
        from scipy.optimize import curve_fit
```

In [8]:
```
        fig = plt.figure(facecolor='0.97')      # I use just off-white backgrounds
        ax = plt.subplot(111)                   # create a plot
        ax.scatter(range(1, len(norm_list)+1), # x coords
                   norm_list,                   # y coords
                   edgecolor='',                # remove black outline of points
                   color='black',               # make inner points black
                   s=1)                         # set pointsize to 1

        ax.set_ylim([-2.5, 2.5])                # from quick figure above, most data is here
        ax.set_xlim([-1000, 50000])             # seperate left border from data
        ax.set_frame_on(False)                  # remove bounding box in plot
        ax.xaxis.tick_bottom()                  # only have x-ticks and labels on bottom
        ax.yaxis.tick_left()                    # only have y-ticks and labels on left

        # Now let's save it
        fig.set_canvas(FigureCanvasAgg(fig))
        fig.savefig('quebecmaine_normalized.png') # lossful but fast
        fig.savefig('quebecmaine_normalized.pdf') # lossless but heavy

        display(Image("quebecmaine_normalized.png"))
        # Doesn't that look better?
```



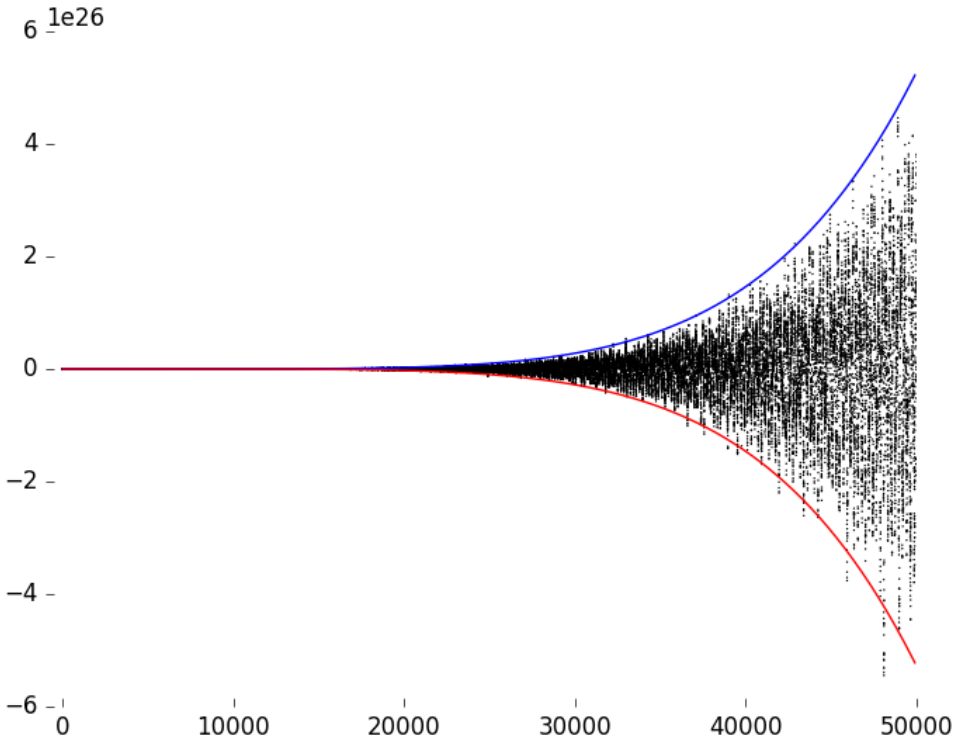
Notice the distribution is not actually random. There is a clear clustering near $y = 0$. This isn't a normal distribution, this is a circle distribution as in the Sato-Tate Conjecture.

I must admit I'm a bit surprised to not remember ever having seen this image, but I'm convinced I must have seen it before.

Let us now look at iterated partial sums. The first partial are expected to look like $S_f(n) \ll n^{5.5+0.25+\epsilon}$.

```
In [9]: ys = fiftyk_sums_list[1]
        print ys[:10]
        xs = range(1, len(ys)+1)

        fig = plt.figure(facecolor='0.97')
        ax = plt.subplot(111)
        ax.scatter(xs,
                   ys,
                   edgecolor='',
                   color='black',
                   s=1)

        # approximating best fits
        bf_xs = np.arange(0.0, 50000, 100)

        def bf_upper(t):
            return .5 * t**(5.5 + .25 + 0.001)

        def bf_lower(t):
            return -.5 * t**(5.5 + .25 + 0.001)

        ax.plot(bf_xs, bf_upper(bf_xs), ls='solid', c="blue")
        ax.plot(bf_xs, bf_lower(bf_xs), ls='solid', c="red")

        #ax.set_ylim([-2.5, 2.5])
        ax.set_xlim([-1000, 50000])
        ax.set_frame_on(False)
        ax.xaxis.tick_bottom()
        ax.yaxis.tick_left()

        # Now let's save it
        fig.set_canvas(FigureCanvasAgg(fig))
        fig.savefig('quebecmaine_sums1.png')
        fig.savefig('quebecmaine_sums1.pdf')

        display(Image("quebecmaine_sums1.png"))
```

```
[1, -23, 229, -1243, 3587, -2461, -19205, 65275, -48368, -164288]
```

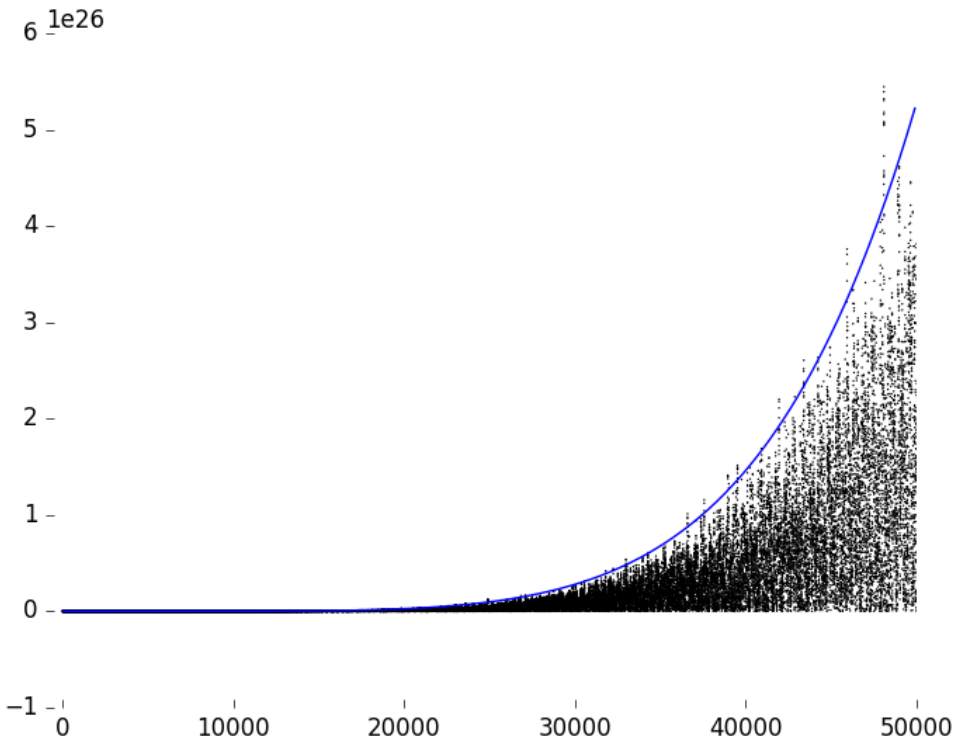


For comparison, it might be interesting to plot only the absolute values of the partial sums.

```
In [10]: ys = map(abs, fiftyk_sums_list[1])
         print ys[:10]
         xs = range(1, len(ys)+1)

         fig = plt.figure(facecolor='0.97')
         ax = plt.subplot(111)
         ax.scatter(xs,
                    ys,
                    edgecolor='',
                    color='black',
                    s=1)

         # approximating best fits
         bf_xs = np.arange(0.0, 50000, 100)

         def bf_upper(t):
             return .5 * t**(5.5 + .25 + 0.001)

         #def bf_lower(t):
         #    return -.5 * t**(5.5 + .25 + 0.001)

         ax.plot(bf_xs, bf_upper(bf_xs), ls='solid', c="blue")
         #ax.plot(bf_xs, bf_lower(bf_xs), ls='solid', c="red")
```

```
        #ax.set_ylim([-2.5, 2.5])
        ax.set_xlim([-1000, 50000])
        ax.set_frame_on(False)
        ax.xaxis.tick_bottom()
        ax.yaxis.tick_left()

        # Now let's save it
        fig.set_canvas(FigureCanvasAgg(fig))
        fig.savefig('quebecmaine_sums1_abs.png')
        fig.savefig('quebecmaine_sums1_abs.pdf')

        display(Image("quebecmaine_sums1_abs.png"))
```

```
[1, 23, 229, 1243, 3587, 2461, 19205, 65275, 48368, 164288]
```

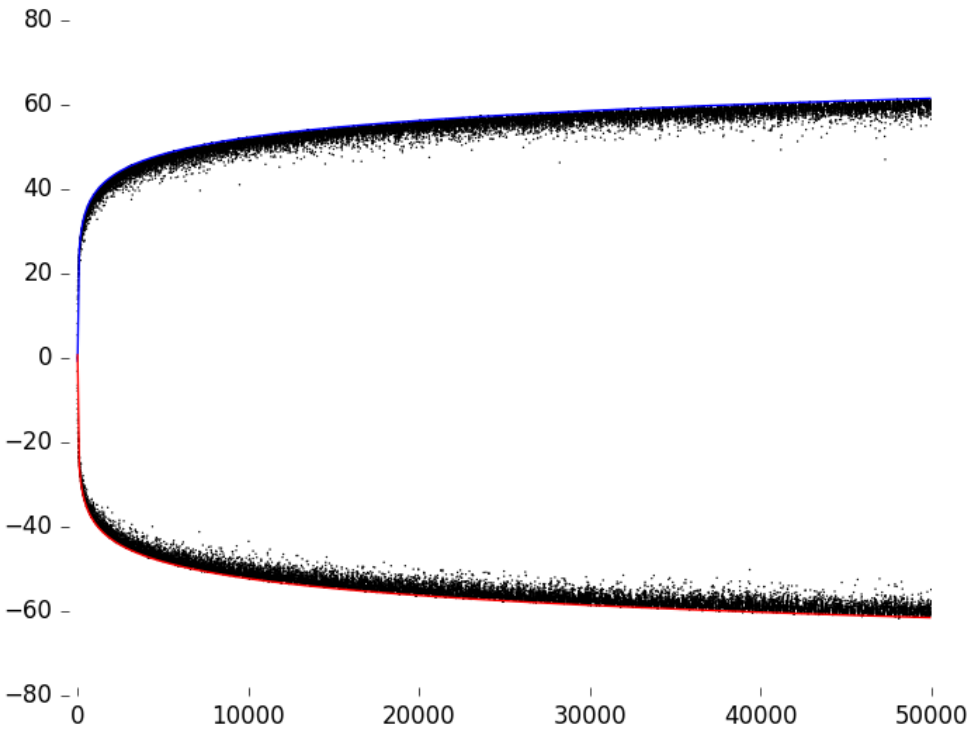


It might also be a good idea to look at a log-log plot to condense the data better. The bounding line $u(x) = 0.5 \cdot x^{5.5+.25+0.001} = 0.5 \cdot x^{5.751}$. In log-log plots, this becomes the relation $\log u(x) = 5.751 \log x + \log 0.5$.

```
In [11]: def log_sign(x):
             """
             Computes modified log function. Like log, except this remembers the sign
             of the input value, and returns 0 if 0 instead of blowing up.
             """
             if x == 0:
```

```
            return 0
        if x > 0:
            return log(x)
        return -1 * log(-x)
```

In [12]:
```
ys = map(log_sign, fiftyk_sums_list[1])
xs = range(1, len(ys)+1)

fig = plt.figure(facecolor='0.97')
ax = plt.subplot(111)
ax.scatter(xs,
           ys,
           edgecolor='',
           color='black',
           s=1)

# approximating best fits
bf_xs = np.arange(1.0, 50000, 100)

def bf_upper(t):
    return 5.751 * log(t) + log(0.5)

def bf_lower(t):
    return -5.751 * log(t) - log(0.5)

ax.plot(bf_xs, bf_upper(bf_xs), ls='solid', c="blue")
ax.plot(bf_xs, bf_lower(bf_xs), ls='solid', c="red")

#ax.set_ylim([-2.5, 2.5])
ax.set_xlim([-1000, 50000])
ax.set_frame_on(False)
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()

# Now let's save it
fig.set_canvas(FigureCanvasAgg(fig))
fig.savefig('quebecmaine_sums1_loglog.png')
fig.savefig('quebecmaine_sums1_loglog.pdf')

display(Image("quebecmaine_sums1_loglog.png"))
```



What an interesting graph! Let's see if we can make sense of it.

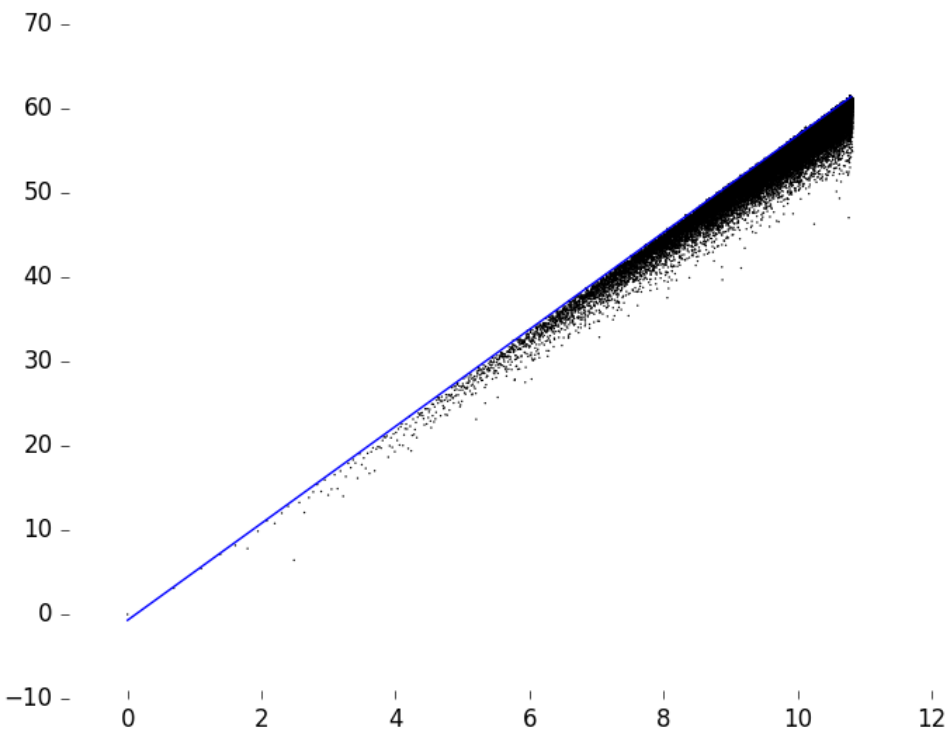
1. Firstly, the approximating lines seem pretty accurate.
2. Secondly, the sums are very rarely "near zero". This makes sense because each individual element is a sum of very many coefficients $a(n)$. And for large n, each $a(n)$ is pretty large — large enough to "cross the gap" from a bit negative to a bit positive.

For example for $n \approx 10000$, we expect $a(n) \approx 10000^{5.5}$. So $\log a(n) \approx 5.5 \log(10000) \approx 50$. Correspondingly, we shouldn't expect many terms less than 50 to appear for $n \geq 10000$, barring extreme and very perfect cancellation.

Let do the same graph, except with absolute values.

```
In [13]: ys = map(log, map(abs , fiftyk_sums_list[1]))
         xs = map(log, range(1, len(ys)+1))

         fig = plt.figure(facecolor='0.97')
         ax = plt.subplot(111)
         ax.scatter(xs,
                    ys,
                    edgecolor='',
                    color='black',
                    s=1)

         # approximating best fits
         bf_xs = np.arange(0.0, log(50000), 0.02)
```

```
def bf_upper(t):
    return 5.751*t + log(0.5)

#def bf_lower(t):
#    return -5.751 * log(t) - log(0.5)

ax.plot(bf_xs, bf_upper(bf_xs), ls='-', c="blue")
#ax.plot(bf_xs, bf_lower(bf_xs), ls='solid', c="red")

#ax.set_ylim([-2.5, 2.5])
ax.set_xlim([-1, 12])
ax.set_frame_on(False)
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()


# Now let's save it
fig.set_canvas(FigureCanvasAgg(fig))
fig.savefig('quebecmaine_sums1_loglog_abs.png')
fig.savefig('quebecmaine_sums1_loglog_abs.pdf')

display(Image("quebecmaine_sums1_loglog_abs.png"))
```



Now let us try to find good approximations for the lines of best fit for higher iterates. To do this, we will

consider increasing sublists (as these indicating the real growth), convert them to log-log plots, and then resolve the resulting linear best-fit problem.

```
In [14]: def give_increasing_sublist(input_list, cast_to_int=True):
             """
             Return a list whose nth component is the largest of the absolute values of the first
             n components in input_list.
             """
             ret_list = [input_list[0]]
             for elem in input_list[1:]:
                 a = int(abs(elem)) if cast_to_int else elem
                 if a >= ret_list[-1]:
                     ret_list.append(a)
                 else:
                     ret_list.append(ret_list[-1])
             return ret_list

         print give_increasing_sublist(check_10)
         print give_increasing_sublist(fiftyk_sums_list[0][:10])
         print
         print fiftyk_sums_list[1][:10]
         print give_increasing_sublist(fiftyk_sums_list[1][:10]) # Notice the repeated 3587
```

```
[1, 24, 252, 1472, 4830, 6048, 16744, 84480, 113643, 115920]
[1, 24, 252, 1472, 4830, 6048, 16744, 84480, 113643, 115920]

[1, -23, 229, -1243, 3587, -2461, -19205, 65275, -48368, -164288]
[1, 23, 229, 1243, 3587, 3587, 19205, 65275, 65275, 164288]
```

```
In [15]: fiftyk_inc_sums_lists = []
         for l in fiftyk_sums_list:
             fiftyk_inc_sums_lists.append(give_increasing_sublist(l))

         print fiftyk_inc_sums_lists[0][:10]
         print fiftyk_inc_sums_lists[1][:10]

         twomil_inc_sums_lists = []
         for l in twomil_sums_list:
             twomil_inc_sums_lists.append(give_increasing_sublist(l))
```

```
[1, 24, 252, 1472, 4830, 6048, 16744, 84480, 113643, 115920]
[1, 23, 229, 1243, 3587, 3587, 19205, 65275, 65275, 164288]
```

```
In [16]: var('m')
         my_model(x) = m*x
         fit_coeffs = []

         for j, ls in enumerate(fiftyk_inc_sums_lists):
             data = [[log(x), log(y)] for x,y in enumerate(ls, 1)]
             fit = find_fit(data, my_model, solution_dict=True)
             fit_coeffs.append(fit[m])
```

```
In [17]: for j, m in enumerate(fit_coeffs):
             print "The j = %2d sum is approximated by %1.5f x." % (j, m)
```

```
The j =  0 sum is approximated by 5.58936 x.
The j =  1 sum is approximated by 5.67706 x.
```

```
The j =  2 sum is approximated by 5.94356 x.
The j =  3 sum is approximated by 6.24293 x.
The j =  4 sum is approximated by 6.55078 x.
The j =  5 sum is approximated by 6.86176 x.
The j =  6 sum is approximated by 7.17432 x.
The j =  7 sum is approximated by 7.48790 x.
The j =  8 sum is approximated by 7.80214 x.
The j =  9 sum is approximated by 8.11676 x.
The j = 10 sum is approximated by 8.43152 x.
```

These give best-estimate guesses for the exponents, based on the first 50000 coefficients.
Let us now perform this analysis on the first 2.5 million coefficients. [This takes a long time].

```
In [18]: ## This is too slow as written - this needs to be optimized significantly
         #var('m')
         #my_model(x) = m*x
         #two_mil_fit_coeffs = []
         #
         #for j, ls in enumerate(twomil_inc_sums_lists):
         #    print j # for time tracking
         #    data = [[log(x), log(y)] for x,y in enumerate(ls, 1)]
         #    fit = find_fit(data, my_model, solution_dict=True)
         #    two_mil_fit_coeffs.append(fit[m])
         #    print str(fit[m])
         #
         #for j, m in enumerate(two_mil_fit_coeffs):
         #    print "The j = %2d sum is approximated by %1.5fx." % (j, m)
```

It might be a good idea to rewrite inc_sums to keep just the values **and indices** of the maximal elements. This should give approximately the same line, and will vastly reduce the overall size of the arrays considered.

```
In [19]: def keep_maximal_elements(input_list, cast_to_int=True):
             ret_list = [[0, input_list[0]]]
             for i, elem in enumerate(input_list[1:], 1):
                 a = int(abs(elem)) if cast_to_int else elem
                 if a >= ret_list[-1][1]:
                     ret_list.append([i, a])
             return ret_list
```

```
In [20]: fiftyk_red_inc_sums_lists = []
         for l in fiftyk_sums_list:
             fiftyk_red_inc_sums_lists.append(keep_maximal_elements(l))

         print fiftyk_red_inc_sums_lists[0][:10]
         print fiftyk_red_inc_sums_lists[1][:10]

         twomil_red_inc_sums_lists = []
         for l in twomil_sums_list:
             twomil_red_inc_sums_lists.append(keep_maximal_elements(l))
```

```
[[0, 1], [1, 24], [2, 252], [3, 1472], [4, 4830], [5, 6048], [6, 16744], [7, 84480], [8, 113643], [9, 1
[[0, 1], [1, 23], [2, 229], [3, 1243], [4, 3587], [6, 19205], [7, 65275], [9, 164288], [10, 370324], [1
```

```
In [21]: #var('m')
         #var('b')
         #my_model(x) = m*x + b
```

```
        #red_fit_coeffs = []
        #
        #for j, ls in enumerate(fiftyk_red_inc_sums_lists):
        #    data = [[log_sign(arr[0]), log_sign(arr[1])] for arr in ls]
        #    fit = find_fit(data, my_model, solution_dict=True)
        #    red_fit_coeffs.append([fit[m], fit[b]])
        #    #print str(fit[m])
        #
        #for j, arr in enumerate(red_fit_coeffs):
        #    print "The j = %2d sum is approximated by %1.5fx + %1.5f." % (j, arr[0], arr[1])
```

In [22]:
```
var('m')
my_model(x) = m*x
red_fit_coeffs = []

for j, ls in enumerate(fiftyk_red_inc_sums_lists):
    data = [[log_sign(arr[0]), log_sign(arr[1])] for arr in ls]
    fit = find_fit(data, my_model, solution_dict=True)
    red_fit_coeffs.append(fit[m])
    #print str(fit[m])

for j, m in enumerate(red_fit_coeffs):
    print "The j = %2d sum is approximated by %1.5fx." % (j, m)
```

```
The j =  0 sum is approximated by 5.57895x.
The j =  1 sum is approximated by 5.66199x.
The j =  2 sum is approximated by 5.92205x.
The j =  3 sum is approximated by 6.21501x.
The j =  4 sum is approximated by 6.51347x.
The j =  5 sum is approximated by 6.81805x.
The j =  6 sum is approximated by 7.12444x.
The j =  7 sum is approximated by 7.43294x.
The j =  8 sum is approximated by 7.74646x.
The j =  9 sum is approximated by 8.06513x.
The j = 10 sum is approximated by 8.37913x.
```

In [23]:
```
var('m')
my_model(x) = m*x
twomil_red_fit_coeffs = []

for j, ls in enumerate(twomil_red_inc_sums_lists):
    print "\rBeginning number %d out of %d" % (j+1, len(twomil_red_inc_sums_lists)),
    data = [[log_sign(arr[0]), log_sign(arr[1])] for arr in ls]
    fit = find_fit(data, my_model, solution_dict=True)
    twomil_red_fit_coeffs.append(fit[m])
    #print str(fit[m])

print "\r"

for j, m in enumerate(twomil_red_fit_coeffs):
    print "The j = %2d sum is approximated by %1.5fx." % (j, m)
```

```
Beginning number 11 out of 11
The j =  0 sum is approximated by 5.59287x.
The j =  1 sum is approximated by 5.69824x.
```

```
The j =  2 sum is approximated by 6.02557x.
The j =  3 sum is approximated by 6.37382x.
The j =  4 sum is approximated by 6.73043x.
The j =  5 sum is approximated by 7.08986x.
The j =  6 sum is approximated by 7.45110x.
The j =  7 sum is approximated by 7.81358x.
The j =  8 sum is approximated by 8.17457x.
The j =  9 sum is approximated by 8.53548x.
The j = 10 sum is approximated by 8.89763x.
```

```
In [24]: # Idle curiosity on a different model
         var('m')
         var('b')
         my_model(x) = m*x + b
         twomil_red_fit_model2_coeffs = []

         for j, ls in enumerate(twomil_red_inc_sums_lists):
             print "\rBeginning number %d out of %d" % (j, len(twomil_red_inc_sums_lists)),
             data = [[log_sign(arr[0]), log_sign(arr[1])] for arr in ls]
             fit = find_fit(data, my_model, solution_dict=True)
             twomil_red_fit_model2_coeffs.append([fit[m], fit[b]])
             #print str(fit[m])

         print "\r"

         for j, arr in enumerate(twomil_red_fit_model2_coeffs):
             print "The j = %2d sum is approximated by %1.5fx + %1.5f." % (j, arr[0], arr[1])
```

```
Beginning number 10 out of 11
The j =  0 sum is approximated by 5.61624x + -0.25946.
The j =  1 sum is approximated by 5.78298x + -1.02602.
The j =  2 sum is approximated by 6.26099x + -3.09721.
The j =  3 sum is approximated by 6.75518x + -5.02891.
The j =  4 sum is approximated by 7.24850x + -6.86260.
The j =  5 sum is approximated by 7.74531x + -8.70554.
The j =  6 sum is approximated by 8.24323x + -10.54464.
The j =  7 sum is approximated by 8.74027x + -12.36027.
The j =  8 sum is approximated by 9.23824x + -14.18239.
The j =  9 sum is approximated by 9.73723x + -16.01435.
The j = 10 sum is approximated by 10.23596x + -17.84025.
```

```
In [25]: # Yet a different model
         # a x^b (log x)^c
         var('a')
         var('b')
         var('c')
         my_model(x) = a + b*x + c*log(x)
         twomil_red_fit_model3_coeffs = []

         for j, ls in enumerate(twomil_red_inc_sums_lists):
             print "\rBeginning number %d out of %d" % (j, len(twomil_red_inc_sums_lists)),
             data = [[log_sign(arr[0]), log_sign(arr[1])] for arr in ls[1:] if log_sign(arr[0]) > 0 and
             fit = find_fit(data, my_model, solution_dict=True)
             twomil_red_fit_model3_coeffs.append([fit[a], fit[b], fit[c]])
             #print str(fit[m])
```

```
        print "\r"


        for j, arr in enumerate(twomil_red_fit_model3_coeffs):
            print "The j = %2d sum is approximated by %1.5f + %1.5fx + %1.5flog(x)." % (j, arr[0], arr
```

```
Beginning number 10 out of 11
The j =  0 sum is approximated by 0.28814 + 5.71037x + -0.67253log(x).
The j =  1 sum is approximated by -0.27220 + 5.87357x + -0.74899log(x).
The j =  2 sum is approximated by -1.66834 + 6.37580x + -1.14449log(x).
The j =  3 sum is approximated by -3.01253 + 6.90286x + -1.54145log(x).
The j =  4 sum is approximated by -4.41762 + 7.42351x + -1.84879log(x).
The j =  5 sum is approximated by -5.84772 + 7.94286x + -2.12511log(x).
The j =  6 sum is approximated by -7.23868 + 8.46575x + -2.42752log(x).
The j =  7 sum is approximated by -8.52978 + 8.99478x + -2.79586log(x).
The j =  8 sum is approximated by -9.74138 + 9.52864x + -3.21718log(x).
The j =  9 sum is approximated by -11.19005 + 10.04743x + -3.46740log(x).
The j = 10 sum is approximated by -12.77113 + 10.55766x + -3.62116log(x).
```

The best-fits above are clearly not particularly useful. Ideally, one would restrict the best fits to positive coefficients of log, and most likely positive coefficients of the contstant term. As it is here, the log term and the constant term are both very negative attempting to make up for the excessive size of the coefficients of x.

```
In [26]: # That this doesn't work feels like an interaction between sage and scipy.
         # One should export the data to a regular python setup and use numpy+scipy from there

         #def model(x, e, f, g):
         #    return e + f*x + g*math.log(x)
         #
         #xs = [math.log(arr[0]) for arr in fiftyk_red_inc_sums_lists[1][1:]]
         #ys = [math.log(arr[1]) for arr in fiftyk_red_inc_sums_lists[1][1:]]
         #print xs[:10]
         #print ys[:10]
         #
         #curve_fit(model, xs, ys)
         #
```

```
In [27]: # a x^b (log x)^c
         #import math
         #var('a')
         #var('b')
         #var('c')

         #def test_model(x,a,b,c):
         #    return a + b*x + c*math.log(x)

         #fiftyk_red_fit_coeffs = []

         #for j, ls in enumerate(fiftyk_red_inc_sums_lists):
         #    print "Beginning number %d out of %d" % (j+1, len(fiftyk_red_inc_sums_lists))
         #    xs = [math.log(arr[0]) for arr in ls[1:]]
         #    ys = [math.log(arr[1]) for arr in ls[1:]]
             #xs, ys = zip(*[[log_sign(arr[0]), log_sign(arr[1])] for arr in ls])
```

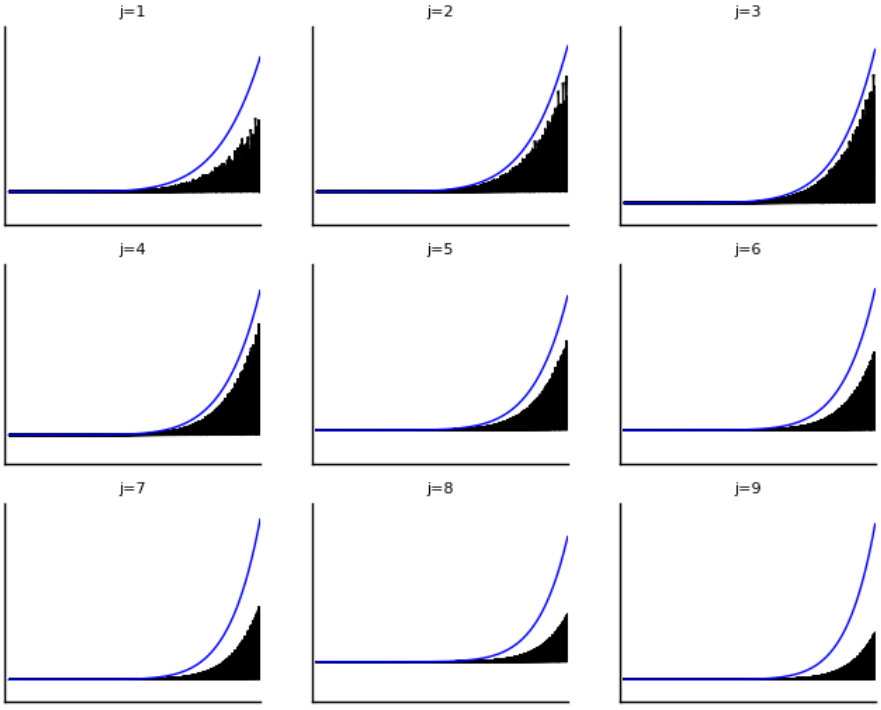
```
        #data = [[log_sign(arr[0]), log_sign(arr[1])] for arr in ls[1:]]
        #fit = find_fit(data, my_model, solution_dict=True)
        #fiftyk_red_fit_coeffs.append([fit[a], fit[b], fit[c]])
        #print str(fit[m])
#    popt, pcov = curve_fit(model, xs, ys)
#    print popt

#print "\r"

#for j, arr in enumerate(fiftyk_red_fit_coeffs):
#    print "The j = %2d sum is approximated by %1.5f + %1.5fx + %1.5flog(x)." % (j, arr[0], ar
```

Let us create the last several plots. I know that I am going to put these in 3x3 arrays on a 11cmx9cm area. I do not want there to be labels for the axes — I mean for these to be much more qualitative than quantitative.

With that in mind, let's create them.

```
In [28]: def scatter_array(j):
             ys = map(abs, fiftyk_sums_list[j+1])    # skip the first set of data
             xs =range(1, len(ys)+1)
             # I assume a fig is already created
             ax = plt.subplot(3,3,j)
             ax.scatter(xs,
                        ys,
                        edgecolor='',
                        color='black',
                        s=1)
             bf_xs = np.arange(1.0, 50000, 100)
             def bf_upper(t):
                 coeff = twomil_red_fit_coeffs[j+1]
                 return 1.0/j * t**(coeff)
             ax.plot(bf_xs, bf_upper(bf_xs), ls='-', c="blue")
             ax.set_xlim([-1000,50000])
             ax.spines['right'].set_visible(False)
             ax.spines['top'].set_visible(False)
             ax.xaxis.set_ticklabels([])
             ax.yaxis.set_ticklabels([])
             ax.xaxis.set_ticks([])
             ax.yaxis.set_ticks([])
             ax.title.set_text("j=%d" % (j))
             ax.title.set_size(8)

In [29]: fig = plt.figure(facecolor='0.97')

         for j in range(1,10):
             print "\rBeginning %d out of 10" % (j+1)
             scatter_array(j)

         # Now let's save it
         fig.set_canvas(FigureCanvasAgg(fig))
         fig.savefig('quebecmaine_9_plots.png')
         fig.savefig('quebecmaine_9_plots.pdf')

         display(Image("quebecmaine_9_plots.png"))
```

```
Beginning 2 out of 10
Beginning 3 out of 10
Beginning 4 out of 10
Beginning 5 out of 10
Beginning 6 out of 10
Beginning 7 out of 10
Beginning 8 out of 10
Beginning 9 out of 10
Beginning 10 out of 10
```

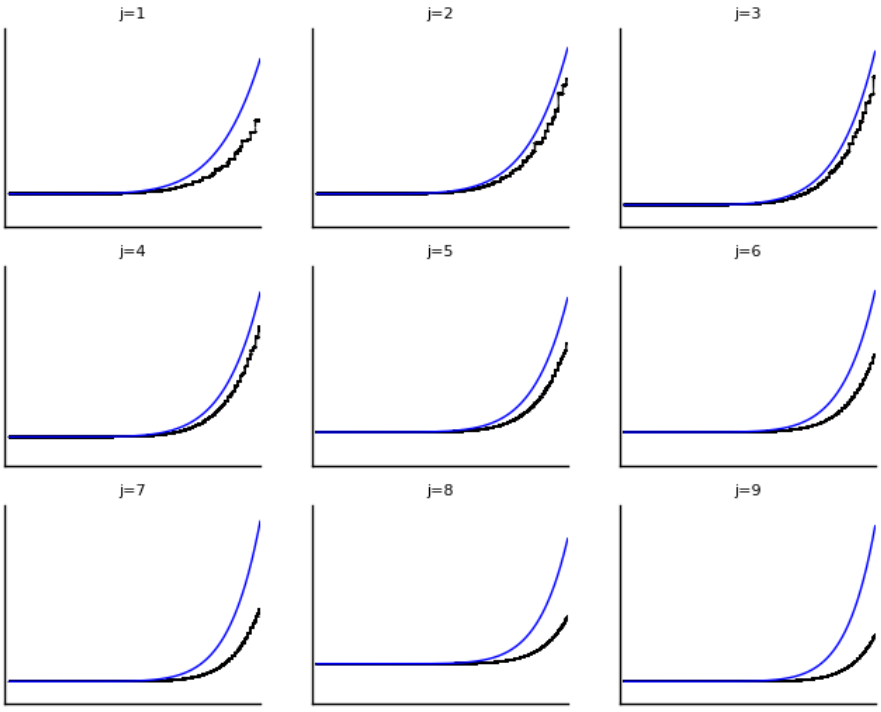



```
In [30]: def inc_scatter_array(j):
             ys = map(abs, fiftyk_inc_sums_lists[j+1])
             xs =range(1, len(ys)+1)
             # I assume a fig is already created
             ax = plt.subplot(3,3,j)
             ax.scatter(xs,
                        ys,
                        edgecolor='',
                        color='black',
                        s=1)
             bf_xs = np.arange(1.0, 50000, 100)
             def bf_upper(t):
                 coeff = twomil_red_fit_coeffs[j+1]
                 return 1.0/j * t**(coeff)
```

```
            ax.plot(bf_xs, bf_upper(bf_xs), ls='-', c="blue")
            ax.set_xlim([-1000,50000])
            ax.spines['right'].set_visible(False)
            ax.spines['top'].set_visible(False)
            ax.xaxis.set_ticklabels([])
            ax.yaxis.set_ticklabels([])
            ax.xaxis.set_ticks([])
            ax.yaxis.set_ticks([])
            ax.title.set_text("j=%d" % (j))
            ax.title.set_size(8)
```

In [31]:
```
fig = plt.figure(facecolor='0.97')

#plt.rc('text', usetex=False)

for j in range(1,10):
    print "\rBeginning %d out of 10" % (j+1)
    inc_scatter_array(j)

fig.set_canvas(FigureCanvasAgg(fig))
fig.savefig('quebecmaine_9_inc_plots.png')
fig.savefig('quebecmaine_9_inc_plots.pdf')

display(Image("quebecmaine_9_inc_plots.png"))
```

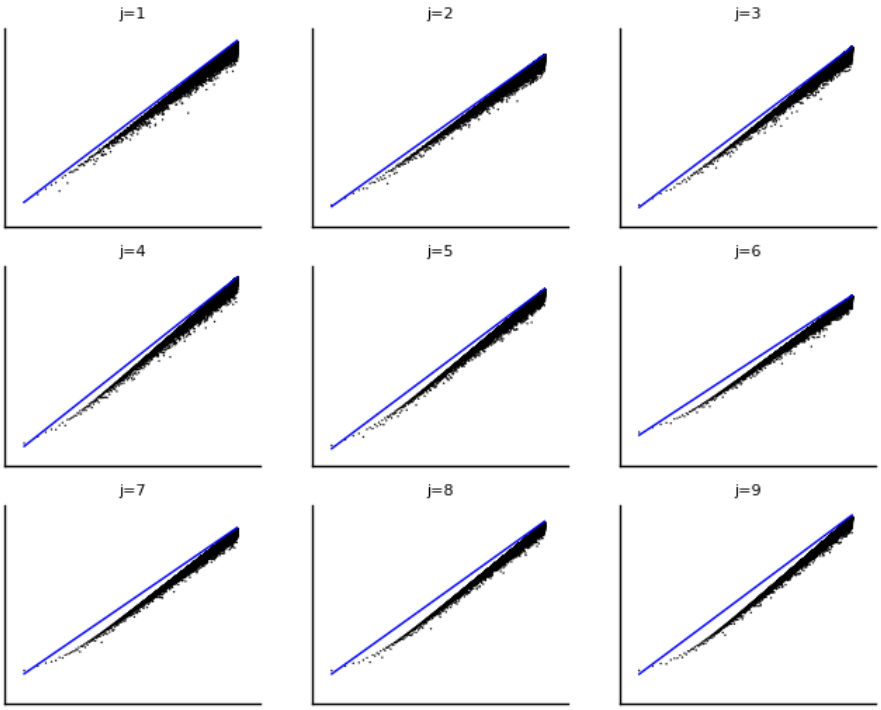
```
Beginning 2 out of 10
Beginning 3 out of 10
Beginning 4 out of 10
Beginning 5 out of 10
Beginning 6 out of 10
Beginning 7 out of 10
Beginning 8 out of 10
Beginning 9 out of 10
Beginning 10 out of 10
```




```
In [32]: def log_scatter_array(j):
             ys = map(log, map(abs, fiftyk_sums_list[j+1]))
             xs = map(log, range(1, len(ys)+1))
             # I assume a fig is already created
             ax = plt.subplot(3,3,j)
             ax.scatter(xs,
                        ys,
                        edgecolor='',
                        color='black',
                        s=1)
             bf_xs = np.arange(0.0, log(50000), 0.02)
             def bf_upper(t):
                 coeff = twomil_red_fit_coeffs[j+1]
                 return coeff * t + log(1./j)
             ax.plot(bf_xs, bf_upper(bf_xs), ls='-', c="blue")
             ax.set_xlim([-1,12])
             ax.spines['right'].set_visible(False)
             ax.spines['top'].set_visible(False)
             ax.xaxis.set_ticklabels([])
             ax.yaxis.set_ticklabels([])
             ax.xaxis.set_ticks([])
             ax.yaxis.set_ticks([])
             ax.title.set_text("j=%d" % (j))
             ax.title.set_size(8)
```

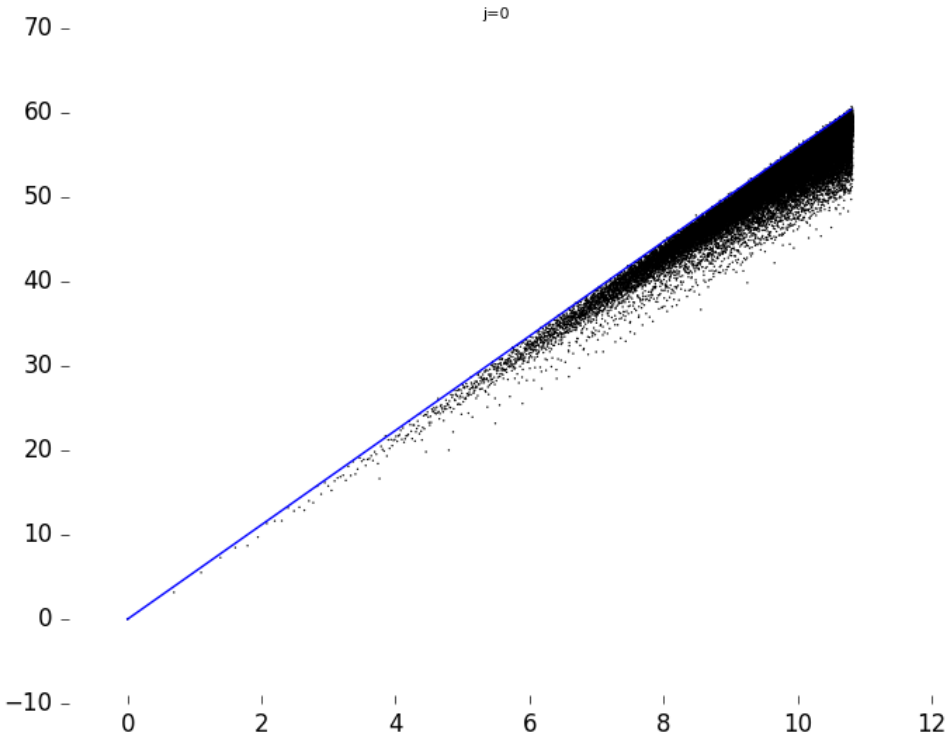
In [33]:
```
fig = plt.figure(facecolor='0.97')

for j in range(1,10):
    print "\rBeginning %d out of 10" % (j),
    log_scatter_array(j)

fig.set_canvas(FigureCanvasAgg(fig))
fig.savefig('quebecmaine_9_log_plots.png')
fig.savefig('quebecmaine_9_log_plots.pdf')

display(Image("quebecmaine_9_log_plots.png"))
```

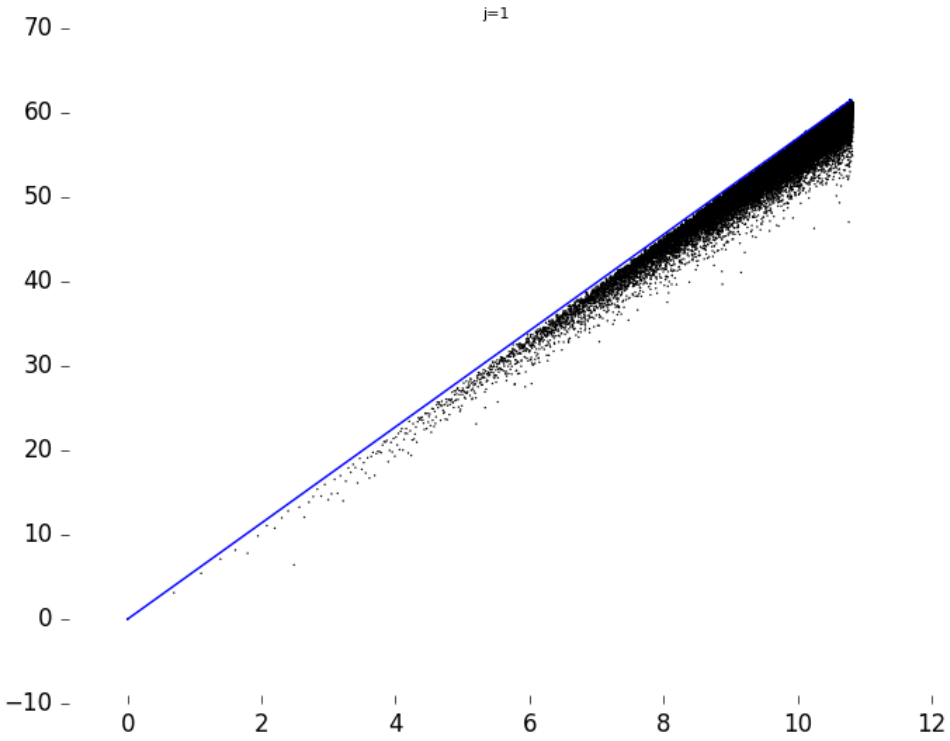
```
Beginning 9 out of 10
```

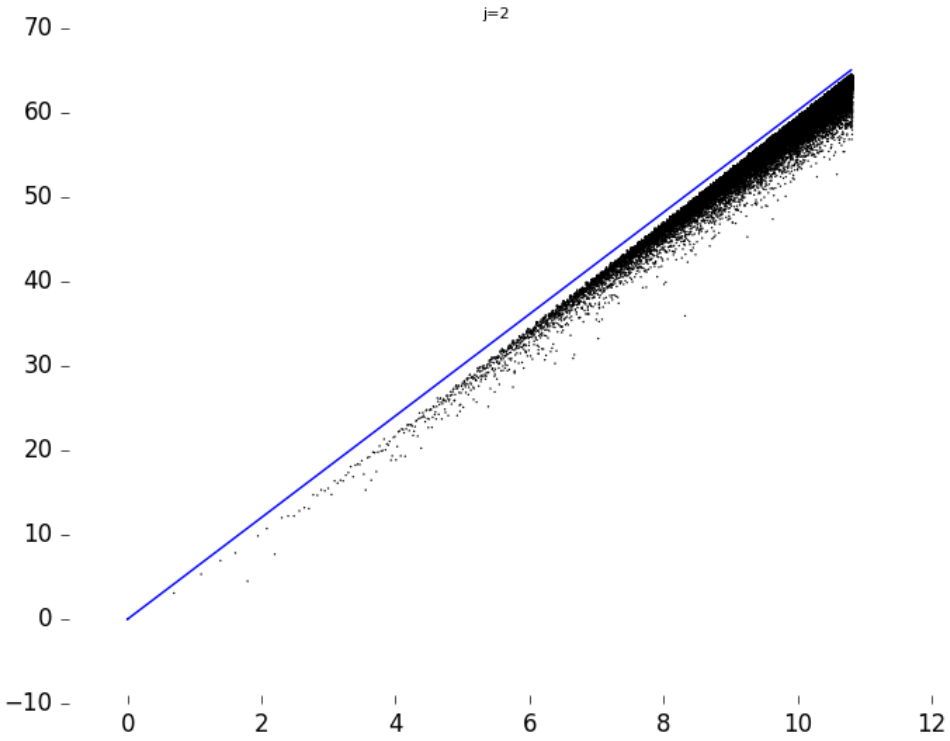


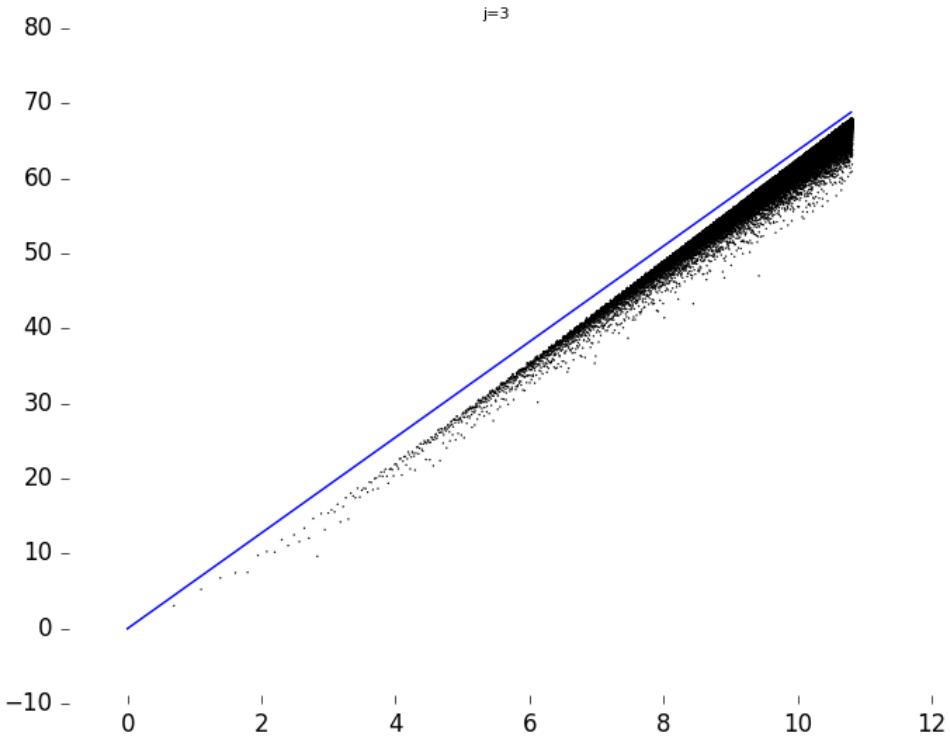
In [34]:
```
for j in range(11):
    ys = map(log, map(abs , fiftyk_sums_list[j]))
    xs = map(log, range(1, len(ys)+1))

    fig = plt.figure(facecolor='0.97')
    ax = plt.subplot(111)
    ax.scatter(xs,
               ys,
               edgecolor='',
```

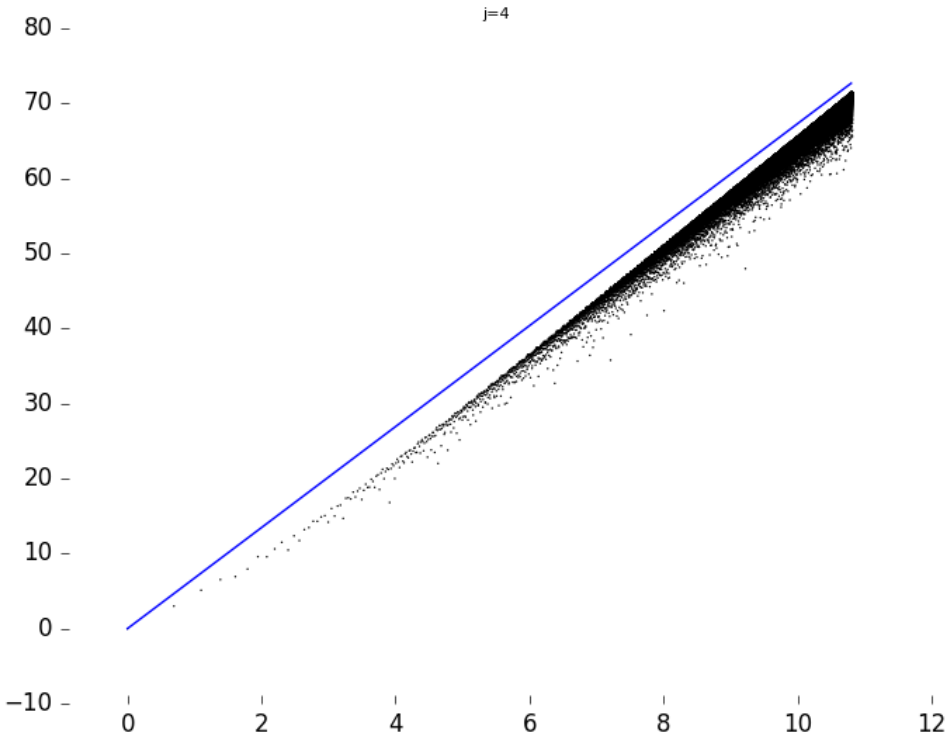
```
            color='black',
            s=1)

# approximating best fits
bf_xs = np.arange(0.0, log(50000), 0.02)

def bf_upper(t):
    coeff = twomil_red_fit_coeffs[j]
    return coeff*t

#def bf_lower(t):
#    return -5.751 * log(t) - log(0.5)

ax.plot(bf_xs, bf_upper(bf_xs), ls='-', c="blue")
#ax.plot(bf_xs, bf_lower(bf_xs), ls='solid', c="red")

#ax.set_ylim([-2.5, 2.5])
ax.set_xlim([-1, 12])
ax.set_frame_on(False)
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()

ax.title.set_text("j=%d" % (j))
ax.title.set_size(8)


# Now let's save it
fig.set_canvas(FigureCanvasAgg(fig))
fig.savefig('quebecmaine_logsum_detail_%d.png' %(j))
fig.savefig('quebecmaine_logsum_detail_%d.pdf' %(j))

display(Image("quebecmaine_logsum_detail_%d.png" %(j)))
```

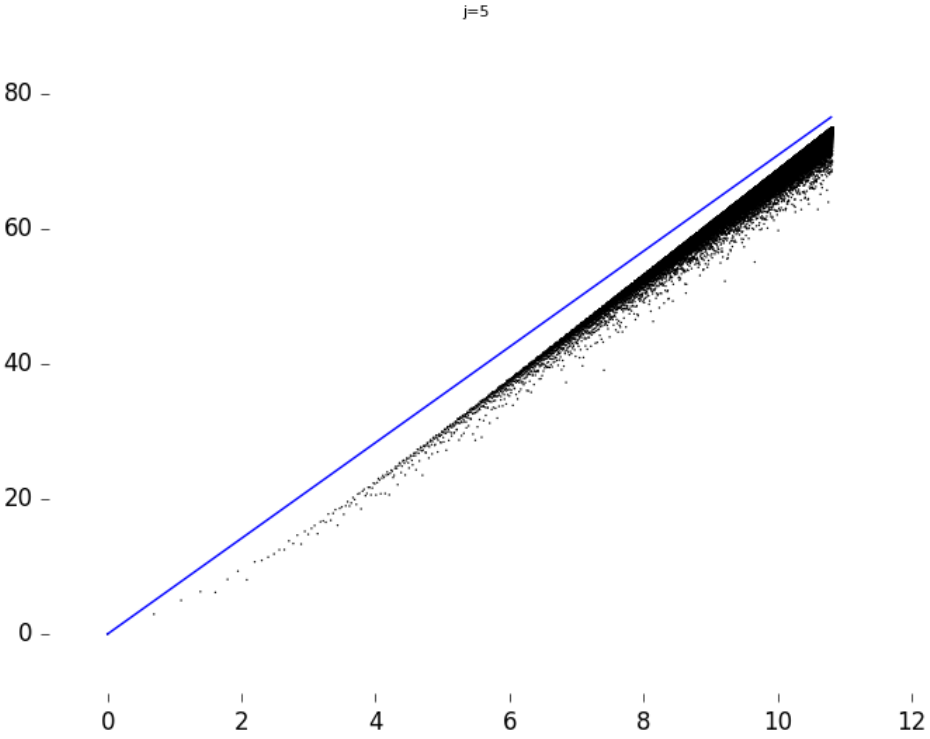

j=0
70
60
50
40
30
20
10
0
−10
0
2
4
6
8
10
12

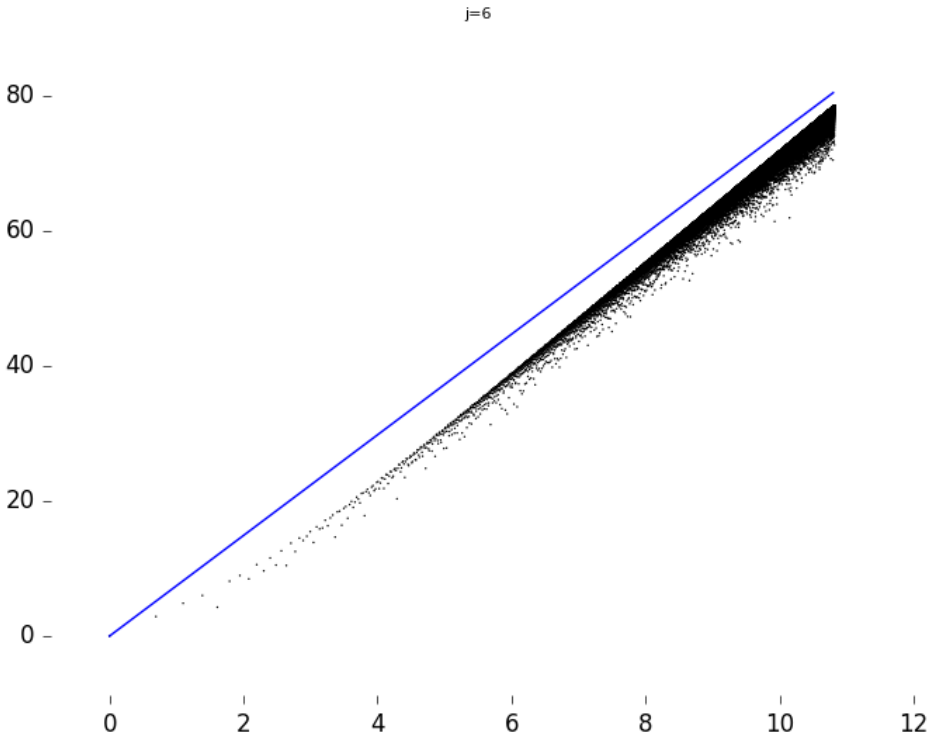

j=1
70
60
50
40
30
20
10
0
−10
0
2
4
6
8
10
12

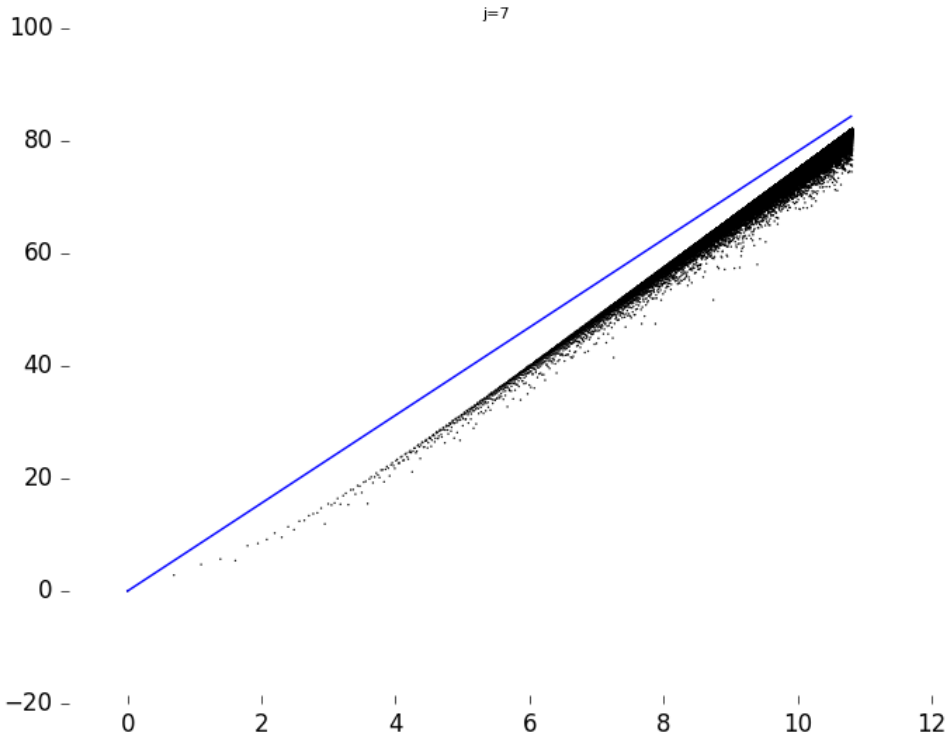

j=2
70
60
50
40
30
20
10
0
−10
0
2
4
6
8
10
12

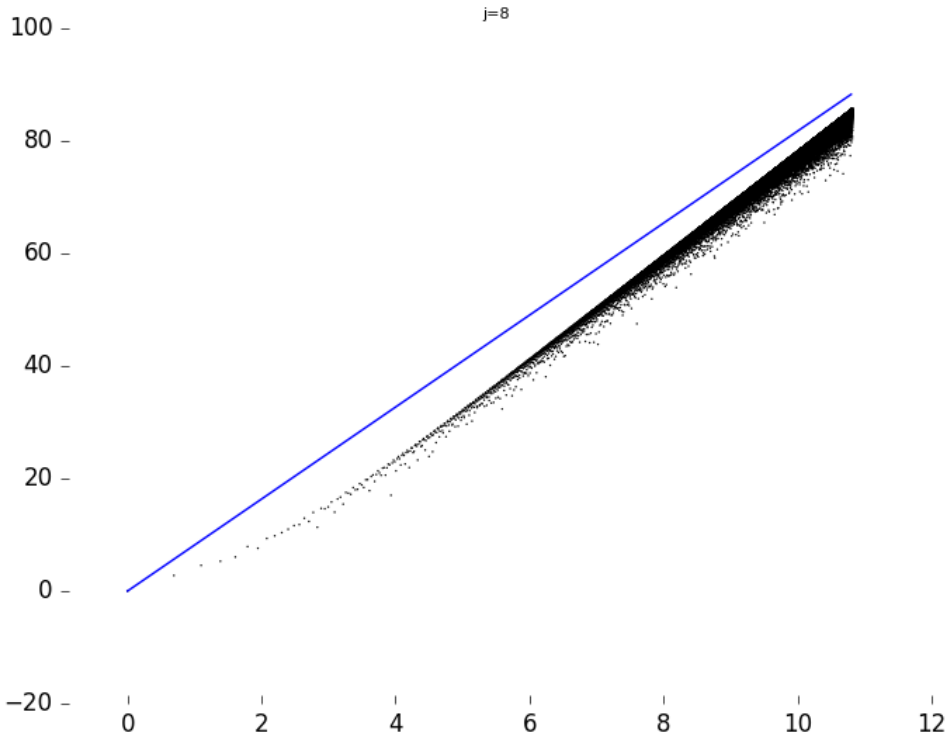

j=3
80
70
60
50
40
30
20
10
0
−10
0
2
4
6
8
10
12

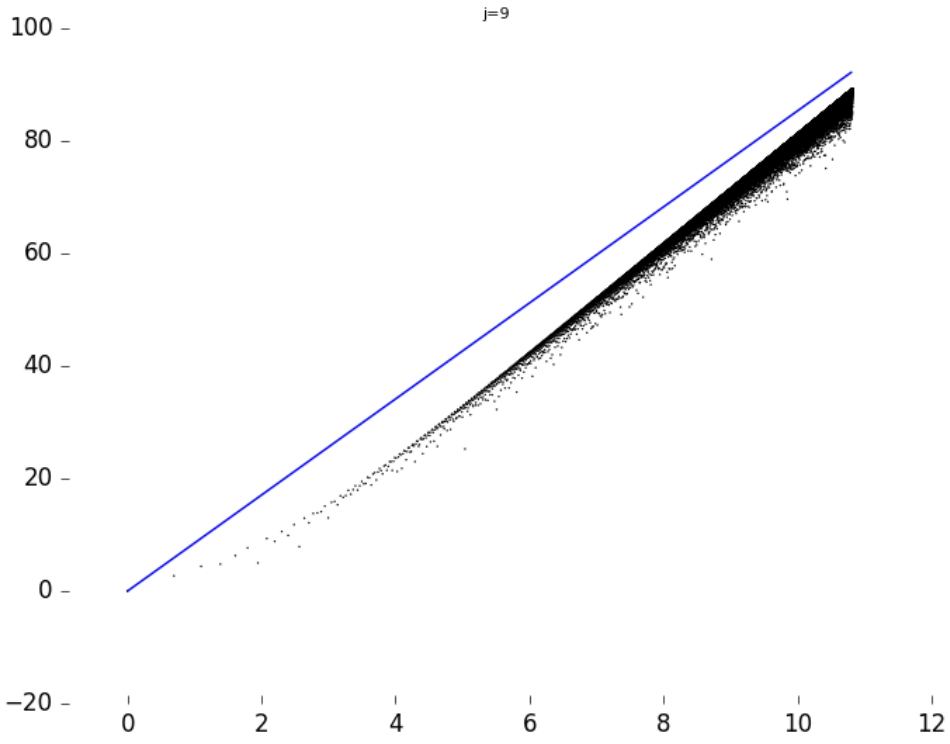

j=4
80
70
60
50
40
30
20
10
0
−10
0
2
4
6
8
10
12

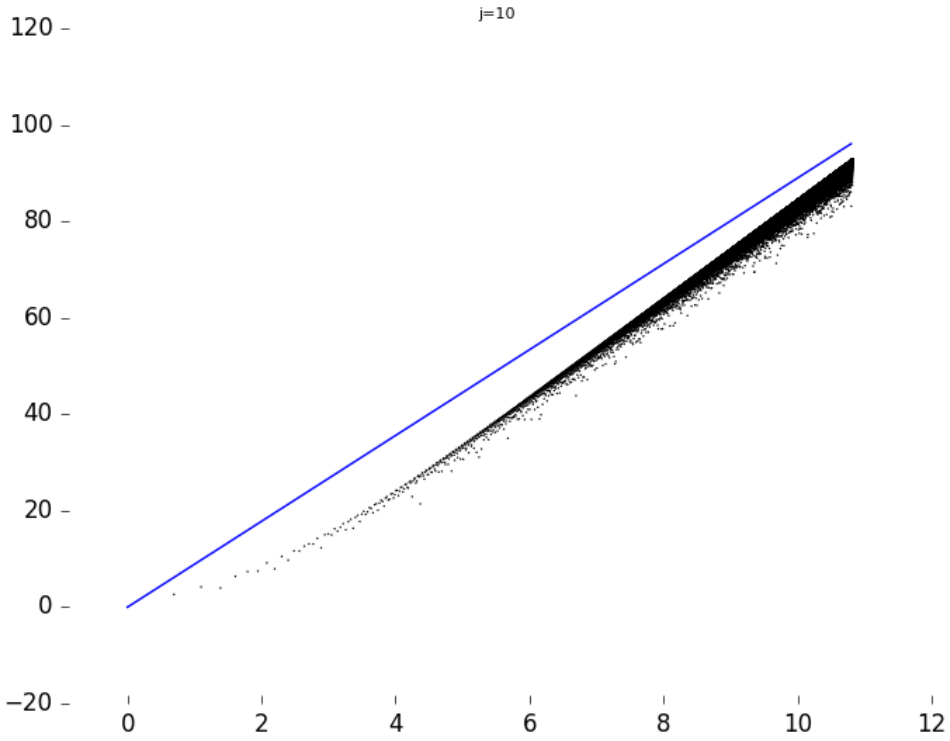

j=5
80
60
40
20
0
0
2
4
6
8
10
12


j=6
80
60
40
20
0
0
2
4
6
8
10
12


j=7
100
80
60
40
20
0
−20
0
2
4
6
8
10
12


j=8
100
80
60
40
20
0
−20
0
2
4
6
8
10
12


j=9
100
80
60
40
20
0
−20
0
2
4
6
8
10
12




```
In [35]: print "Run Completed"
```

```
Run Completed
```

```
In [ ]:
```