Christoph Lüders — http://wrogn.com

Curve-fitting With Minimized Relative Error[1]

The Problem

I wrote a C++ function to multiply two large positive integers of the same length, say n 64-bit words, with the grade-school method. Let's call that function `omul_n()`.

Then, I wrote extensive benchmarking to assess the speed of my efforts. The resulting run-times for the multiplication of two numbers with n words look like this:

Words	Cycles
1	18
9	281
17	915
25	1 959
33	3 421
41	5 207
49	7 392
57	10 093
65	13 000
73	16 397
81	20 224
89	24 326
97	28 800
105	33 941
113	39 764
121	45 487
129	51 212
137	57 453
145	64 142
153	71 778

Figure 1: Measured run-times

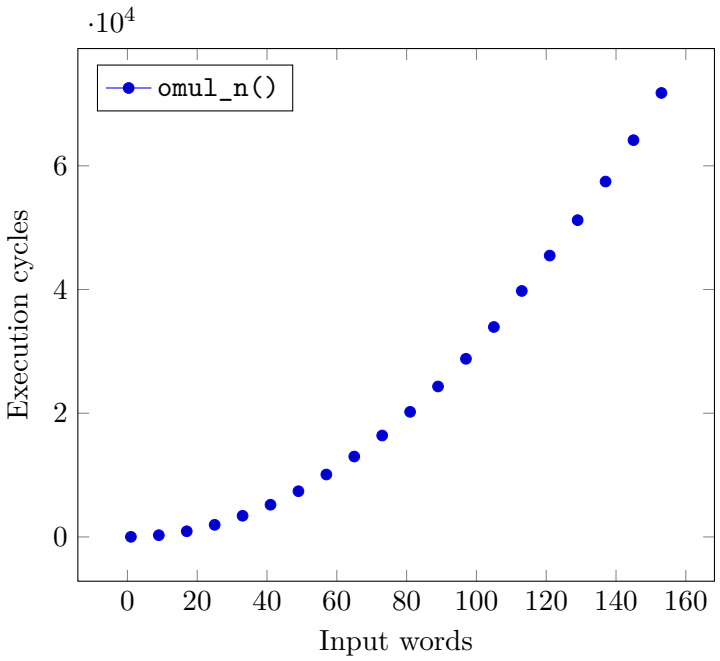

Figure 2: Plotted run-times

Now I wanted to find a closed function to most accurately describe the run-time of `omul_n()`. We know that to multiply two numbers of n digits each, we need to do n^2 digit-multiplications. So, most likely, the desired function will look something like

$$T(n) = c_0 + c_1 n + c_2 n^2.$$

The only question is: what values to use for c_0, c_1 and c_2? I like linear regression, but it only works for linear relationships, like $T(n) = c_0 + c_1 n$. We cannot use that here.

[1] v1.0, 19-Jan-2015

The First Solution

The solution to my question is curve-fitting. I used Python functions to do so, namely `scipy.optimize.curve_fit` from the *SciPy* package (a good starter article that inspired my use of curve-fittings is here.)

The program is really simple. You input your data plus the describing function (like $T(n)$ above) into the curve-fitting function and out pop the coefficients c_i that yield the $T(n)$ with the least squared error.

The Python script:

```
omul_str = open("omul-speed.txt", "r").read()    # read measured values
o = [float(i) for i in omul_str.split()]         # make one big list
os = o[0::2]                                     # slice out first column
ot = o[1::2]                                     # slice out second column

import numpy as np                               # imports
from scipy.optimize import curve_fit             # the magic function

xdata = np.array(os)                             # convert lists to np.array
ydata = np.array(ot)
def func(x, c0, c1, c2):                         # the modeled function
   return c0 + c1*x + c2*x*x

popt, pcov = curve_fit(func, xdata, ydata)       # and fit it!
print(popt)                                      # print optimized parameters
```

If you're not used to NumPy, `array` features an unfamiliar usage:

```
Python 3.4.1 |Anaconda 2.1.0 (64-bit)| ...
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> a
array([1, 2, 3])
>>> import math
>>> math.log(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
>>> np.log(a)
array([ 0.        ,  0.69314718,  1.09861229])
```

NumPy functions that are applied to an `array` again return an array with values of said function applied to every `array` element. That comes in pretty handy when handling larger sets of data.

Back to our curve-fitting. The above listed script generates this output:

```
[-60.37910437    5.09798716    3.03566267]
```

That means that the best fitting function is about

$$T_{\text{abs}}(n) = -60 + 5.1 \cdot n + 3.04 \cdot n^2.$$

Pretty neat, eh? Plotted it looks like this. The red line is not the connection of the dots, but our model:

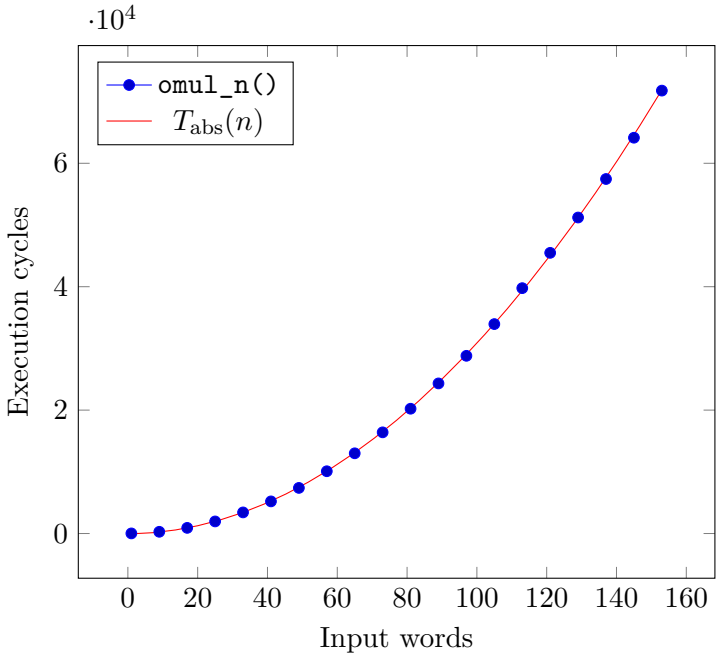

Figure 3: Modeled run-time with minimized absolute error

My Discontent

So far, so very cool. An issue arises when we look at the relative errors between data points and model. That is, $|T(n)/T_n|$, where $T(n)$ is our model and T_n is the measured run-time. In contrast, the above curve-fitting minimized the absolute error $|T(n) - T_n|$.[2]

Some additional lines of Python code added to the end of our script will print the relative errors and their average:

```
relerr = abs(1 - ydata / func(xdata, *popt))     # relative errors
np.set_printoptions(suppress=True)               # switch off sci. notation
print(relerr * 100)
avgrel = sum(relerr) / len(ydata) * 100          # calc average
print("avgrel:", avgrel)
```

Which does produce this extra output:

```
[ 134.45275796   21.43922899    1.26238363    0.27284922    0.21410507
    0.84902538    1.15067892    0.00073479    0.73808731    0.55686398
    0.22467506    0.46166589    0.6782697     0.00615863    1.23715341
    1.07859592    0.19226999    0.28013432    0.56064524    0.00479269]
avgrel: 8.28305380503
```

[2] Actually, it minimized the *squared* absolute error, but I let that slide here and focus on absolute vs. relative error.

So, we have an average relative error of 8 %, which seems rather high for me. Obviously, the relative error is extremely high with the two starting values: 134 % and 21 %. Can we improve that? That is, can we model so that the average and maximum relative error is lower?

The Improved Solution

Least squares optimization with minimized absolute error is used very widely, but unfortunately, there is no easy way to switch the functions performing this to minimize the relative error. But I found this forum post that was very helpful. It's on some other math software system, but we can borrow the idea: *"Usually the best way to do relative error is to log your model. This changes a proportional error structure into an additive one, which is exactly what you want"* (with "*log*" as in *logarithm*).

Luckily, that is very easy to accomplish in Python. This is a changed version of the earlier script:

```
omul_str = open("omul-speed.txt", "r").read()    # read measured values
o = [float(i) for i in omul_str.split()]         # make one big list
os = o[0::2]                                     # slice out first column
ot = o[1::2]                                     # slice out second column

import numpy as np                               # imports
from scipy.optimize import curve_fit             # the magic function

xdata = np.array(os)                             # convert lists to np.array
ydata = np.array(ot)
def func(x, c0, c1, c2):                         # the modeled function
   return c0 + c1*x + c2*x*x
def logfunc(x, c0, c1, c2):                      # ... and the log of it
   return np.log(func(x, c0, c1, c2))

popt, pcov = curve_fit(logfunc, xdata, np.log(ydata))  # and fit it!
print(popt)                                      # print optimized parameters

relerr = abs(1 - ydata / func(xdata, *popt))     # relative errors
np.set_printoptions(suppress=True)               # switch off sci. notation
print(relerr * 100)
avgrel = sum(relerr) / len(ydata) * 100          # calc average
print("avgrel:", avgrel)
```

And now the output looks like this:

```
[ 12.98237958   1.9705695    3.05332744]
[ 0.03485745  1.06567529  1.49572472  0.58745607  0.52644119  0.37155771
  0.65289914  0.47219135  0.31728127  0.18879885  0.09165894  0.19598836
  0.45928895  0.171688    1.37775523  1.18298158  0.26311364  0.23936649
  0.54721279  0.01646704]
avgrel: 0.512920201737
```

Awesome! The average relative error is down to 0.5 % with a maximum of 1.5 %.

The linear plot looks largely the same, because the absolute differences are too small to see. But if we switch to a double-logarithmic plot, we can see them clearly:

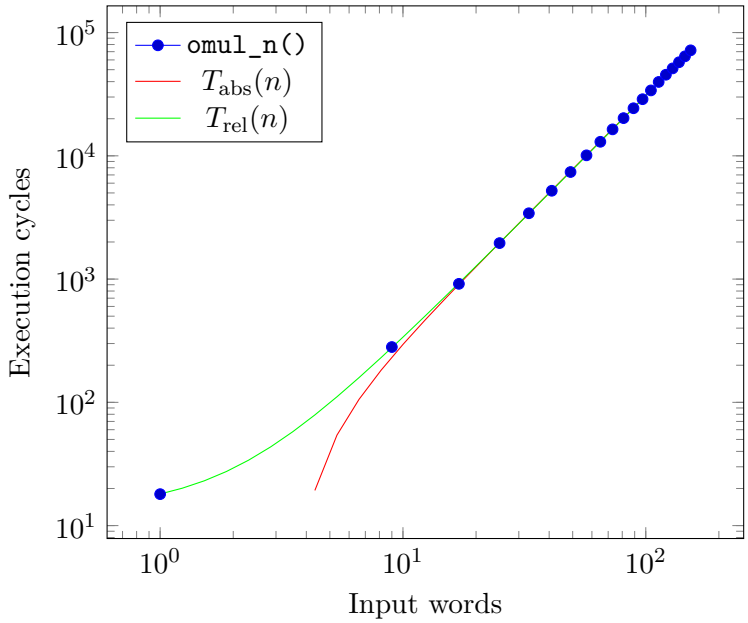

Figure 4: Modeled run-times with both minimized absolute and relative error

Clearly, the smaller the values are, the larger the difference is between the red graph (minimized absolute errors model) and the data points, whereas the green graph (minimized relative errors) is much closer to the data points for small n.

□