twiddling twiddle - to speed up twiddle by 400%, replace 1.1 and 0.9 by 1+/- 0.723273755703

Goldsong Mar '12

There is something funny about the result from twiddle. The 'gradient ascent' gives you the impression that you are finding 'optimal' answers but it appears that you are not really doing that.. you are just finding 'ok' answers, definitely not bad choices but not 'optimal' either.

Consider the results from the routine given in the quiz..

```
Twiddle # 1 [1.0, 1.0, 0.0]  ->  4.33323043444
Twiddle # 2 [1.0, 2.100000000000, 0.0]  ->  3.03664661762
Twiddle # 3 [1.990000000000, 2.1000000000000005, 0.0]  ->  0.75991875878
Twiddle # 4 [1.990000000000, 2.1000000000000005, 0.0]  ->  0.75991875878
Twiddle # 5 [1.990000000000, 3.0801000000000007, 0.0]  ->  0.75950826029
Twiddle # 6 [2.872090000000, 3.0801000000000007, 0.0]  ->  0.362175999306
Twiddle # 7 [3.842389000000, 4.0503990000000014, 0.0]  ->  0.203737026633
Twiddle # 8 [4.909717900000, 5.117727900000002, 0.0]  ->  0.126098387291
Twiddle # 9 [6.083779690000, 6.2917896900000025, 0.0]  ->  0.0819731323449
Twiddle # 10 [6.083779690000, 6.2917896900000025, 0.0]  ->  0.0819731323449
Twiddle # 11 [7.246100862100, 7.4541108621000038, 0.0]  ->  0.0577718679842
Twiddle # 12 [8.524654151410, 8.7326641514100061, 0.0]  ->  0.041988949944
Twiddle # 13 [8.524654151410, 10.139072769651008, 0.28242953648100]  ->  0.0102123913128
Twiddle # 14 [7.258886394993, 11.68612224971611, 0.593102026610100]  ->  0.000186304633521
Twiddle # 15 [5.866541862934, 13.387876677787723, 0.59310202661010]  ->  8.84676066983e-05
Twiddle # 16 [4.334962877670, 15.259806548666496, 0.28553626138229]  ->  6.02443098625e-05
Twiddle # 17 [2.650225993879, 13.200683690699844, 0.28553626138229]  ->  1.05681735463e-09
Twiddle # 18 [2.650225993879, 10.935648546936529, 0.28553626138229]  ->  1.31653164258e-13
Twiddle # 19 [2.650225993879, 10.935648546936529, 0.28553626138229]  ->  1.31653164258e-13
Twiddle # 20 [2.650225993879, 8.6932637546108467, 0.28553626138229]  ->  7.24578396495e-14
Twiddle # 21 [2.650225993879, 8.6932637546108467, 0.28553626138229]  ->  7.24578396495e-14
Twiddle # 22 [2.650225993879, 8.6932637546108467, 0.28553626138229]  ->  7.24578396495e-14
Twiddle # 23 [2.650225993879, 10.69122860457303, 0.285536261382291]  ->  1.75676386146e-14
Twiddle # 24 [2.650225993879, 10.69122860457303, 0.285536261382291]  ->  1.75676386146e-14
Twiddle # 25 [2.650225993879, 10.69122860457303, 0.285536261382291]  ->  1.75676386146e-14
Twiddle # 26 [2.650225993879, 10.691228604573029, 0.28553626138229]  ->  1.75676386146e-14
Twiddle # 27 [2.650225993879, 9.0890605913883533, 0.28553626138229]  ->  8.44381076479e-15
Twiddle # 28 [2.650225993879, 9.0890605913883533, 0.28553626138229]  ->  8.44381076479e-15
Twiddle # 29 [2.650225993879, 9.0890605913883533, 0.28553626138229]  ->  8.44381076479e-15
Twiddle # 30 [2.650225993879, 10.516592291135899, 0.28553626138229]  ->  2.93660921281e-15
...
Twiddle # 100 [2.12691242842, 8.5600988540336935, 0.25103453458344]  ->  1.21709360256e-18
```

it takes 18 iterations to reduce the error to 1e-13. This looks pretty good.. twiddle must be doing its job. However, if the function that twiddle is working on is full of local maxima, all twiddle did was to found one of the local maxima and happily converge to it, giving a reasonable solution but certainly not an optimal one.

The 'point' of the quiz was to implement a twiddle that converged as fast as possible. The algorithm is so simple that is difficult to see what can be done to make it faster. However, one possibility is to alter the k = 0.1 factor that is it being added or substracted from 1.0 to give the 1.1 or the 0.9 multiplication factors. The k = 0.1 is pulled out of a hat. Would it converge faster if it is 1.15? or 1.2? It kinds of makes sense if you you think about it. For example, shell sort reduces the complexity of insertion sort from O(n^2) to O(n^1.25) by bracketing the search space in thirds and twiddle tunes its findings in the same way: bracketing. Anyways.. it makes sense.. so I twiddled the k parameter of twiddle to see if there was a better k and, consistent with the idea that the function has many local maxima, there are many many k's better than 0.1.

The following are some of the errors for various k values on iteration 30:

k	error	
0.1	2.93660921281e-15	This is the quiz, i.e., our reference
0.10754566	2.51850526313e-15	
0.18994287	1.77720910106e-19	

Mar 2012

1 / 10
Mar 2012

Mar 2012

```
   0.2579829     4.36698060631e-19

   0.723733427   1.56823144816e-25  WOW!!! 10 orders of magnitude smaller
```

wow.. using k=0.19 or k=0.257 or k=0.7237 goes to zero way faster that using k = 0.1, even though the algorithm is the same. Taking as a benchmark the number of iterations that it took them to reduce the error to 1e-15 we have:

```
k = 0.1:

Twiddle # 27 [2.6502259938, 9.08906059138, 0.285536261382]  ->  8.4438107647e-15

k = 0.1075..:

Twiddle # 27 [2.0925992492, 9.05057335241, 0.189403695121]  ->  2.5185052631e-15

k = 1899..:

Twiddle # 20 [2.2099918950, 9.20148208670, 0.233712385794]  ->  1.0030051631e-15

k = 0.2579..:

Twiddle # 16 [1.3630329894, 6.79211137054, 0.135971917539]  ->  2.7357715375e-15

k = 0.723733427551

Twiddle # 7 [1.35742696378, 7.10993239875, 0.112669093538]  ->  5.4295767237e-16
```

wow wow.. so.. simply replacing 1.1 for 1.7237 and 0.9 for 1-0.7237.. = 0.2763 makes twiddle converge about 400% faster!!!, in 7 iterations (vs. the 27 needed by the version of the quiz)

Let's see the convergence:

```
 Twiddle # 1 [1.0, 1.0, 0.0]  ->  4.33323043444
 Twiddle # 2 [1.0, 2.7237334275510001, 0.0]  ->  3.01996776794
 Twiddle # 3 [1.47620992584528, 5.69499035680771, 0.0763232190527185]  ->  2.06930805594e-06
 Twiddle # 4 [1.47620992584528, 5.69499035680771, 0.0763232190527185]  ->  2.06930805594e-06
 Twiddle # 5 [1.24943403237171, 7.10993239875633, 0.1126690935380868]  ->  7.17186831035e-11
 Twiddle # 6 [1.24943403237171, 7.10993239875633, 0.1126690935380868]  ->  7.17186831035e-11
 Twiddle # 7 [1.35742696378625, 7.10993239875633, 0.1126690935380868]  ->  5.42957672372e-16
 Twiddle # 8 [1.35742696378625, 7.10993239875633, 0.1126690935380868]  ->  5.42957672372e-16
 Twiddle # 9 [1.35742696378625, 7.05850509289560, 0.1126690935380868]  ->  3.59584222366e-17
 Twiddle # 10 [1.3574269637862, 7.05850509289560, 0.1130340479217356]  ->  1.65265724504e-17
 Twiddle # 11 [1.3613520613167, 7.05850509289560, 0.1130340479217356]  ->  1.91231482489e-18
 Twiddle # 12 [1.3613520613167, 7.05850509289560, 0.1130340479217356]  ->  1.91231482489e-18
 Twiddle # 13 [1.3613520613167, 7.06037426329953, 0.1130820616430607]  ->  9.81185045045e-20
 Twiddle # 14 [1.3613520613167, 7.06037426329953, 0.1131648244994897]  ->  6.31488180931e-20
 Twiddle # 15 [1.3613520613167, 7.06037426329953, 0.1131648244994897]  ->  6.31488180931e-20
 Twiddle # 16 [1.3613126488230, 7.06062017301018, 0.1131648244994897]  ->  5.43558686586e-20
 Twiddle # 17 [1.3613805854559, 7.06062017301018, 0.1131539361449372]  ->  1.33163345524e-20
 Twiddle # 18 [1.3614976901010, 7.06073727765525, 0.1131539361449372]  ->  3.45573551115e-22
```

Hum.. but notice that the parameters for each 'optimal' k found by twiddle are not the same... they vary. Particularly the ones with k=0.25 and k=0.72 have parameters that differ a lot from the results from the quiz. Clearly, the parameters found by twiddle are not optimal.. there are better parameters that can also be found with twiddle using different k's. So what is happening here?

My impression is that twiddle is finding a set of Tau_p, Tau_d and Tau_i that work well together.. but they are not optimal.. they just happen to be at the cusp of a local maxima. Looking for a better maxima would require the usual approaches: simulated annealing and the like. However, it appears that there is little point in finding the absolute best. You just need a good set and that is it and probably a reasonable set is found starting the search with k=0.1.

Likewise, k alters the product of dparams*k so thus altering k should be equivalent to altering the initial dparams. There have been some postings indicating that people have arrived to a different set of parameters when they slightly vary the initial conditions. For example, one person arrived to a slightly different set of parameters simply reversing the order in which the params were bracketed, i.e., first Tau-I, then Tau_d and finally Tau_p instead of the standard order. Hence, the whole thing indicatates that we are dealing with a function with many local maxima and the results of twiddle are sensitive to the starting conditions. These conditions might be the value of k (as shown here), the initial values of params and dparams, and even the order of evaluation of the paramters.

There is a second possibility, though, that I have been thinking about but have not checked into and would have to do with twiddle itself. I learned about this a long time ago but I would have to unpack some boxes in the attic to find the reference and really I rather not do that. Still.. here I mention it. Twiddle is treating all the variables the same but they are not. Tau_p is about 10 times smaller than Tau_d and 10 times bigger than Tau_i. I cannot come up with a good analogy but I think it is like finding the parameters of a scale that will measure at the same time an elephant, a person and a mouse. In this case, the contribution of the error by the elephant shades that of the person and dwarfs that of the mouse. Thus, the elephant finds the hill to climb and the person and the mouse find the cusp. I know that there are some special optimization techniques to deal with variables of different orders of magnitude. I am not sure whether that applies here or not.

Finding parameters that go together well might indeed be enough. Finding the absolute maxima, might take you to the goal too fast and incur in undesirable overshootings that could have been avoided using a different set of parameters that, while taking longer to zero the CTE, approach the goal in a more gentle way. For example, the trajectory with k=0.72 shown above is great if the sole goal is to reduce the CTE fast. The CTE goes to zero in 4 iterations and stays there with little overshoot. However, nice as it is, it might or might not be an actually physically possible or desirable trajectory because we have modeled the car as a massless point that faithfully follows our control law.

In practice, whether we can approximate our car to a massless particle is a function of the car's inertia, i.e., its velocity and mass. If we are driving a small car at low speeds (like a mindstorms lego model or even a Prius at 10mph), we might get away with it, but if we are driving a truck or at 60mph, we will not: the fast zeroing of the CTE means a sharp turn so if the inertia is high the car will turn-over or tail-spin. If by some miracle it manages to track the trajectory correctly, the driver would be splattered against the passenger window. Hence, satisfying the goal did not really take us to a desired result.

Maybe like Aristotle said, the virtue is in the middle: even if we had used a physical model it's probably best to look for a nice compromise of performance (reduce the CTE fast) and safety (no overshootings or sharp turns); k=0.1 (and many others) probably offer such a compromise. Having the zeroing of the CTE as the sole goal is probably worthy if we are using a pysical model of the car, which would have automatically tempered the trajectories. My guess is that the Google cars do have such model.

For those of you who want to try the runs, here is the code:

```
def twiddle2(t_param, tol = 0.1):
    n_params = 3
    dparams = [1.0 for row in range(n_params)]
    params = [0.0 for row in range(n_params)]

 best_error = run(params)
 n = 0
 while best_error &gt; 1.0e-20 and n &lt;= 30:
     for i in range(len(params)):
         params[i] += dparams[i]
         err = run(params)
         if err &lt; best_error:
             best_error = err
             dparams[i] *= (1.0+t_param)
         else:
             params[i] -= 2.0 * dparams[i]
             err = run(params)
             if err &lt; best_error:
                 best_error = err
                 dparams[i] *= (1.0+t_param)
             else:
                 params[i] += dparams[i]
                 dparams[i] *= (1.0-t_param)
     n += 1
     print 'Twiddle #', n, params, ' -&gt; ', best_error
 print ' '
 return params, n



def twiddling_twiddle(dparams, tol = 0.1):

params = 0.1   # starting point.



 p, iter = twiddle2(params)
 best_error = run(p)
 n = 0
 while best_error &gt; 1.0e-25 and n &lt; 100:
     params += dparams
     p, iter = twiddle2(params)
     err = run(p)
```

```
    err1 = err
    if err &lt; best_error:
        best_error = err
        dparams *= 1.1
    else:
        params -= 2.0 * dparams
        p, iter = twiddle2(params)
        err = run(p)
        err2 = err
        if err &lt; best_error:
            best_error = err
            dparams *= 1.1
        else:
            params += dparams
            dparams *= 0.9
    n += 1
    print 'Twiddle #', n, params, dparams, ' -&gt; ', best_error
print ' '
return params, best_error, n
```

to test a given k:

```
tolerance = 0.1
t_param = 0.723733427551    # or 0.1, 0.199 or whatever.. this
params, n = twiddle2(t_param, tolerance)
err = run(params)
print '\nFinal parameters: ', params, ' -> ', err, ' n: ', n
```

to find an 'optimal' set of params using twiddle starting with a given dparam

```
dparams = 0.2    # 0.05, 0.1., 0.2,.. or anything else
                 # 0.2 will converge to 0.723733427551
params, best_error, n = twiddling_twiddle(dparams, tolerance)
params, n = twiddle2(params, tolerance)
err = run(params)
print '\nFinal parameters: ', params, ' -> ', err, ' n: ', n
```

I am not using tolerance values. I set the stop conditions to number of runs or as functions of the best_error.

---

albertocides **Mar '12**

Very good!

I also tried changing the 1.1 and the 0.9

The 0.9 was critical because the stop condition depends on the sum of the dp ... so, if we make dp decrease faster then the algorithm will stop earlier! (but, if we make it too small, the error won't be as small as we want)

First, I tried with 0.5 (one half seemed a good compromise value) then I tested 0.618 = 1/Phi ... I just like the Phi number... but, after that I found that the best results were with 0.3
On the other hand, 1.1 seemed too small ... I tried 2.0, then I tried Phi = 1.618 ... and it was a good value.
Moreover, there's a third value, the case of "-dp", that is, after

```
params[i] -= 2.0 * dparams[i]
```

In that case I chose -1.1 (I thought that if going on the opposite direction gives a better value of the error then I continue with that direction, that's why I chose the minus sign... Also, I thought that chosing a number in "-dp" different than "dp" was a good idea to avoid repeating values. p1, then p2 = p1+dp ... if p3 = p2 -dp then p3 = p1 !!! we repeat values and can enter a cycle, so I tried to avoid that)
But, reviewing this third value again I find that in the case of 1.618 and 0.3 that third value is never used.
Also, if one dp can be negative then the out condition must be changed from a simple sum to something else (maybe a norm: the square root of sum of squares... or the sum of absolute values)

Well, I have to think further, to see if I find more ideas, or other points of view for this problem. I find this Twiddle algorithm too "random" ... and I think it should be improved.

This is the output:

```
Tolerance (tol): 0.1
Twiddle # 1 [1.000, 1.000, 0.000] -> 4.33323043444
Twiddle # 2 [1.000, 2.618, 0.000] -> 3.02119985706
Twiddle # 3 [1.485, 5.236, 0.090] -> 3.2117584078e-07
Twiddle # 4 [1.485, 9.472, 0.090] -> 6.01131747748e-08
Twiddle # 5 [1.485, 9.472, 0.134] -> 1.75461801427e-08
Twiddle # 6 [1.556, 7.416, 0.134] -> 2.22428777912e-15

Final parameters: [1.5560839480000004, 7.4156672110672011, 0.133686]

-> 2.22428777912e-15
```

Kevin_Crosby **Mar '12**

When I use the multipliers 1.1 and 0.9 and a tolerance of 0.001, my solution converges to this local minimum:

```
Final parameters:  [2.9229268964347743, 10.326767087320677, 0.4932708323372665] -> 3.61146228557224
```

However, when I change the multipliers to the golden ratio and its reciprocal, respectively, my solution converges faster and to an even smaller local minimum:

```
Final parameters:  [1.9968943799848557, 8.237987356225288, 0.2224592561495802] -> 4.316630352108431
```

However, when I submit my obviously superior solution, which as an even smaller error than the original problem by about 5 orders of magnitude, the website reports that my solution just isn't good enough. ;)