Math 4500/6500 Final Project 2013

Taco-Related Mountain Navigation Challenge

While eating a delicious fish taco at the "El Chato" taco truck (1013 S. La Brea, Los Angeles), you overhear several shady characters discussing a desert location where they have buried a valuable taco-related item. They depart, presumably hoping to reclaim their treasure, and you (perhaps exhibiting salsa-related poor decision-making) decide to try to beat them to the prize.

Downloading topo data from the National Map and National Elevation Dataset to your laptop over your cellular data connection, you quickly parse, thin, rescale, and convert units, making a *Mathematica* `InterpolatingFunction` describing the landscape between you and the treasure, discovering that (in kilometers) you are at coordinates (9.57,10.23) and the treasure is at (20.12,22.3).

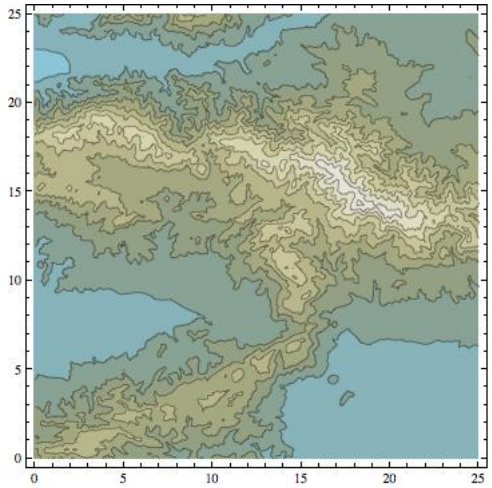
This means that (horizontally), you are only 16.03 km from the goal. However, there is a formidable obstacle in your way: the San Gabriel mountain range. Use *Mathematica*'s minimization functions to find the shortest overland route and win the race to the taco-treasure. The finder of the shortest route will receive a valuable taco-themed prize, as well as 2013 bragging rights.

Posted alongside this writeup on the course webpage, you'll find a Mathematica package file named `TerrainPackage.m`, as well as a notebook named `TerrainPackageTester.nb`. Download these and copy them into the same directory you'll use for your notebook.

With the commands

```
1 AppendTo[$Path, NotebookDirectory[]]
2 << TerrainPackage`
```

you should load the package and define two functions: `z[x,y]` and `GradZ[x,y]`. The `z` function defines the landscape shown by



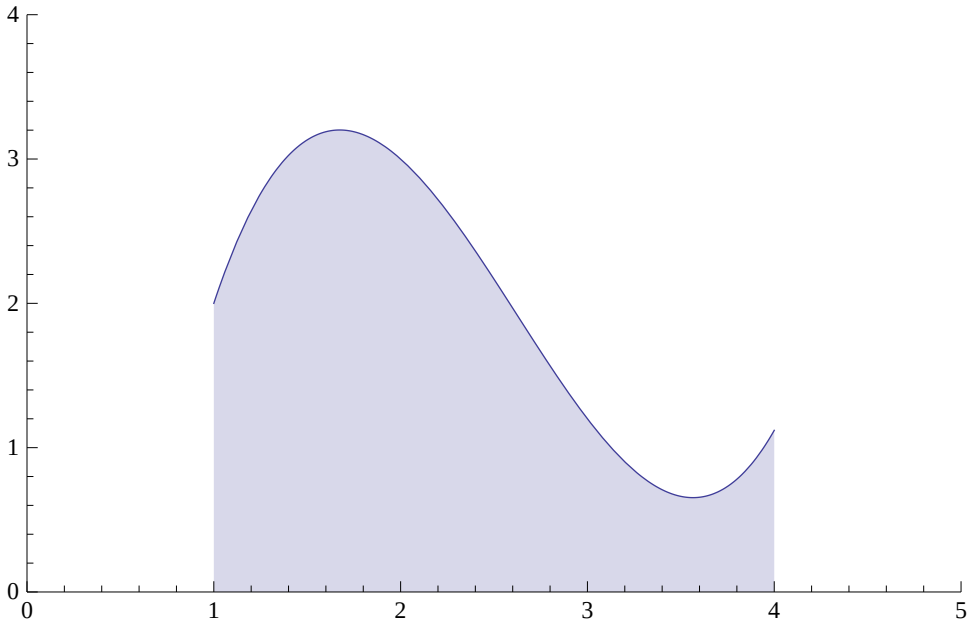
The `GradZ` function defines the gradient vector of the `z` function. In the function `z[x,y]`, `x`, `y` and the output are all in (consistent) units of km.

0.1. **Defining the Path.** To define the problem, you'll need a way to describe a family of paths from `A = (9.57,10.23)` to `B = (20.12,22.3)` in terms of some variables. There are lots of ways to do this, but a very reasonable plan is to let the variables represent a series of waypoints `{x1,y1},{x2,y2},...,{xn,yn}` which you will pass through on your journey.

Mathematica can save you a lot of time here with the `Interpolation` command, which produces a special kind of function called an `InterpolationFunction` which interpolates between various data values. For instance, the commands

```
f = Interpolation[{{1, 2}, {2, 3}, {3, 1.2}, {4, 1.12}}];
Plot[f[x], {x, 1, 4}, Filling -> Axis, AxesOrigin -> {0, 0},
 PlotRange -> {{0, 5}, {0, 4}}]
```

gave me the plot



If we type `f`, *Mathematica* gives

```
InterpolatingFunction[{{1.,4.}},<>]
```

which tells us that the domain of the function is $[1, 4]$. We *can* ask for a value of the function at, say, 17, and *Mathematica* will take a best guess based on the data it has. But it will give us a warning that the results are probably worthless. Of course, this is because they probably are.

Problem 1. *Write a* Mathematica *function which takes a set of waypoints* {x1,y1}, ..., {xn,yn} *and returns a* ***pair*** *of* InterpolatingFunction *functions* x *and* y *so that*

- *The domain of* x[t] *and* y[t] *is* $[0, 1]$.
- x[0] = 9.57
- y[0] = 10.23
- x[1] = 20.12
- y[1] = 22.3.
- *At* $t = i/(n+1)$, x[t] = xi, y[t] = yi *(that is, the path* $(x(t), y(t))$ *passes through all the waypoints).*

Now define a "path" function

$$p(t) = (x(t), y(t))$$

0.2. **Arclength and the space path.** We now want to consider the length of various paths from the taco stand to the treasure. For each set of waypoints, you can now define an entire path $p(t)$ which passes through all the waypoints on the way. We now need to define the objective function to minimize in terms of these waypoints (that is, the length of the path).

The first thing to observe is that the path *in space* corresponding to the *planar* $p(t)$ is

$$\gamma(t) = (x(t), y(t), z(x(t), y(t)).$$

The formula for the length of this parametrized curve in space is given by the integral

$$\text{Length} = \int_0^1 \sqrt{x'(t)^2 + y'(t)^2 + \left(\frac{d}{dt} z(x(t), y(t))\right)^2}\, dt.$$

where I was a little bit careful in writing the derivative of the z coordinate with respect to t because in this case it's really a multivariable chain rule problem to figure out this derivative[1].

We can differentiate and integrate an InterpolationFunction in *Mathematica* just like any other function, but the results are new InterpolationFunction objects. We can evaluate them, but we can't see the formulae that are used to compute them. For instance, if we stick with the example f we defined earlier, then f' gives

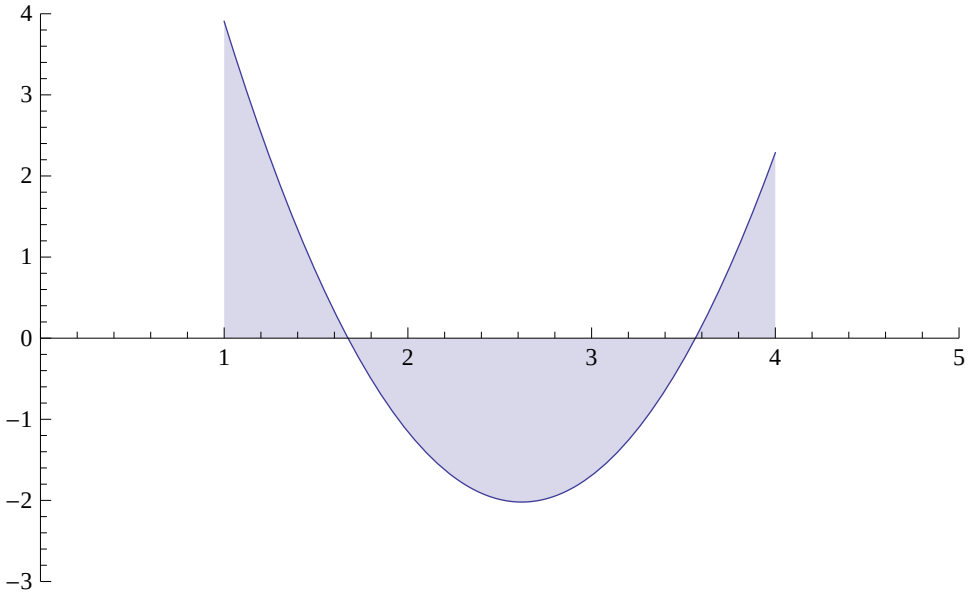
```
1 InterpolatingFunction[{{1.,4.}},<>]
```

because the derivative is defined on the same domain as the original InterpolatingFunction and

```
1 Plot[f'[x], {x, 1, 4}, Filling -> Axis, AxesOrigin -> {0, 0},
2  PlotRange -> {{0, 5}, {-3, 4}}]
```

yields the perfectly nice plot

[1]Hint: The gradient of the z function GradZ is involved in the computation. Which is why you have it.



With this in hand, we are ready for the next

Problem 2. *Write a* Mathematica *function called* `PathLength[x1_,y1_,x2_,y2_,x3_,y3_]` *which does the following things:*

- *Constructs a path built from* `InterpolatingFunction` *objects* `x[t]` *and* `y[t]` *as above which passes through the waypoints* {`x1,y1`},{`x2,y2`},{`x3,y3`}.
- *Defines a new function* γ`[t]` *which gives the 3-space path which follows* `x[t]` *and* `y[t]` *across the landscape with height function* `z[x,y]`.
- *Numerically computes the* Length *integral above (using* `NIntegrate`*) for this path. You should set the* `AccuracyGoal` *low enough that this is fast*[2]*. You're going to be calling this function a lot of times.*

This is the objective function for our problem, and the variables are `x1,y1,x2,y2,x3,y3`. This is a six-dimensional optimization problem. Be a little bit careful that you don't construct the `InterpolatingFunction` defining the path inside the call to `NIntegrate`. If you do, *Mathematica* will have to rebuild the interpolation every time it evaluates it, which will make this whole process unbearably slow. If this takes more than a few hundredths of a second, you're doing it wrong.

0.3. **Breaking out the minimization algorithm.** The first thing to say here is that you are going to learn to use *Mathematica*'s `NMinimize` function, which includes implementations of both Nelder-Mead and Simulated Annealing which are better than anything you can write by hand. You are not going to write your own optimization algorithm here. The second thing to say is that documentation for `NMinimize` (at least in Mathematica 8) all seems to be hidden in the tutorial "Numerical Nonlinear Global Optimization". You'll want to refer to that as we go.

[2] `(stuff) // Timing` will measure the time it takes to evaluate (stuff) in seconds

You want to call `NMinimize` on your `PathLength` function. There are two points to consider

- `NMinimize` actually[3] requires you to set up an interval of reasonable values for each of your input variables. You can certainly get cleverer with this, but the largest intervals which would make sense would be $[1, 25]$ for both the xi and yi variables since this is pretty close to the entire domain of the `z[x,y]` function.
- `NMinimize` will try to evaluate your input `PathLength` function with symbolic inputs before starting up. This will be a big disaster, since the internal `NIntegrate` is not going to like trying to integrate an `InterpolatingFunction` which doesn't even really exist yet.

  Therefore, you have to surround your `PathLength` function with `Hold` in order to prevent it being evaluated until there are actual numbers for the `xi` and `yi`.

An example of a similar call to `NMinimize` which actually works[4] is

```
NMinimize[Hold[PathLength[x1,y1,x2,y2,x3,y3]],{{x1,-10,10},{y1
   ,-10,10},{x2,-10,10},{y2,-10,10},{x3,-10,10},{y3,-10,10}},
   Method->{"SimulatedAnnealing"}, MaxIterations->500,
   AccuracyGoal->2]
```

Here all the variables are limited to the rectangle $[-10, 10] \times [-10, 10]$ *which is, of course, completely inappropriate for your problem*, but it gives you an example of how to set limits on your variables. This runs in 30 seconds or so on my laptop, but with a much simpler z function.

Problem 3. *Write a modified version of your `PathLength` function for testing which uses the landscape height function `z[x,y] = 0` (and `GradZ[x,y] = {0,0}`). This is the flat plane, and the shortest path should be the straight line.*

- *Write a function* `PathPlot[x1_,y1_,x2_,y2_,x3_,y3_]` *which plots the path given by the waypoints* `{x1,y1},{x2,y2},{x3,y3}` .
- *Run `NMinimize` on the modified version of the `PathLength` function.*
- *Debug the above until*
  a *It runs.*
  b *When you plug the resulting values into `PathPlot`, the path is a straight line.*

Now return to the original `PathLength` function (which gives the length of the path through the mountains) and try to `NMinimize` it using various settings.

It's worth setting `AccuracyGoal` very low for `NMinimize` until you know what you're doing. Similarly, you can set `MaxIterations` very low. These should make the `NMinimize` runs fast enough that you can see if things are working at all. I would strongly consider working out an intermediate version of the `PathLength` function where the `z[x,y]` function is the surface of a sphere or something and doing a 3d plot to make sure that the results look reasonable.

[3] though the documentation doesn't mention it

[4] for me, using *Mathematica* 8 on my laptop

Also, I'd recommend reading through the entire tutorial and trying various other options (including Nelder-Mead and the "DifferentialEvolution" minimizer as well) to see what works best.

0.4. **Scaling Up.** Of course, a path with only 3 waypoints is not going to be the optimum path through the mountains, or anything particularly close to it. After all, we might have to make many twists and turns in order to get to the treasure first, and the path with only 3 waypoints just has a limited amount of turning it can do. On the other hand, more flexibility in the path means adding more waypoints, and adding more waypoints means adding more variables, and adding more variables means the computation will take more time.

One of the major obstacles to experimenting with different numbers of waypoints is that you have to type lines like `PathPlot[x1_,y1_,\dots,x20_,y20_]` and it's really painful and error prone. *Or do you?* Here's the principle:

Write interesting code which writes boring code for you.

It's not actually that hard to use *Mathematica*'s string operation code to produce the text strings needed to define all of this stuff in an automated way. But to convert a string called `CodeString` to *Mathematica* code *and evaluate that code in Mathematica* requires only one line:

```
1 ToExpression[CodeString];
```

This is really spooky the first time you try it. But kind of awesome immediately afterwards. On the other hand, if this terrifies you, you're welcome to cut-and-paste, too. If you stick with the debugging long enough, it should eventually work out.

Another principle to keep in mind is that if you optimize with, say, 10 waypoints, and you get a pretty nice path but want to improve it by adding more waypoints, a really reasonable first guess for the next run would be simply to distribute the new waypoints along the current best path, and use that as an initial guess[5] for `NMinimize`. The other thing you could do is simply use those waypoints as starts and ends for 10 smaller optimization problems, and then stick all the resulting paths back together at the end. This is probably going to require more coding, though.

Problem 4. *Scale up to at least 10 waypoints, and find the best solution you can. You'll probably want to try various initial conditions with the hope of finding several different local minimum solutions and then getting to choose the best one.*

0.5. **Turning in your work.** I want you to turn in **two** *Mathematica* notebooks for this project.

a `YourLastName_work.nb`
In this notebook, you should go through all the work above of building an objective function and optimizing it with `NMinimize`. This should include a plot of your solution and its length (as calculated by you).

[5]Yes, you can set the initial guess– the directions are in the tutorial.

b `YourLastName_solution.nb`

This notebook should define a single function `CodenameSolution[t]`. For `t` between 0 and 1, this function should return a 2 element list $\{x, y\}$ of coordinates, with

`CodenameSolution[0] = {9.57,10.23}`
`CodenameSolution[1] = {20.12,22.3}`.

This function should use `Module` for any internal variables and be completely self-contained, since I'll be cutting and pasting it into the class notebook. The results will be visible to the class by `Codename`, so please choose an appropriate alias if you don't want your solution to be associated with your real name.

A great way to do this would be to take the `InterpolatingFunction` objects corresponding to your best path and print them out with[6] `// InputForm`, then copy the resulting glob of text into the solution notebook.

Unfortunately, I can't extend the due date past **midnight, December 3, 2013**. But I'll try to help out as best I can between now and then.

[6]The `x // y` syntax in Mathematica means that you want to evaluate `x` and then pass the results to the filter `y` for further processing. For instance, `x // CForm` gives you the result of `x` in a form which you could cut and paste into a `C` program, while `x // TeXForm` gives the results in a form which you could cut and paste into a TeX document. `x // InputForm` gives you a version which you could cut and paste back into Mathematica itself in a `Code` cell. There's a lot more about this in the documentation.