PIC 16, Winter 2018 – Assignment 8W

Assigned 2/28/2018. Code (a single `.py` file) due by the end of class 3/5/2018 on CCLE. Hand in a printout of this document with the self-assessment portion completed by the end of class on 3/5/2018.

In this assignment, you will solve the oldest known problem in the calculus of variations, Dido's problem.

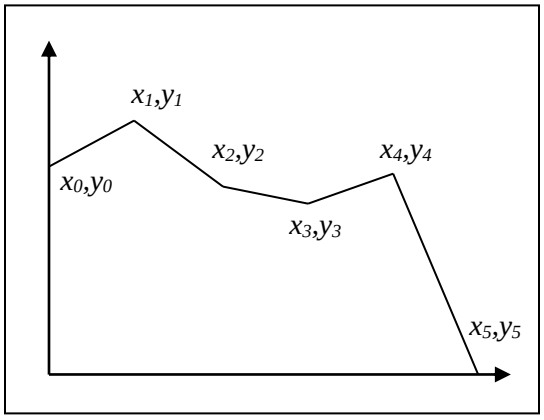
Task

Find the shape of maximum area for a given perimeter.
(The answer is a circle. Let's show it numerically!)

This is not the simplest introduction to optimization I could assign, but I believe in you. The problem is worth the effort, because it gives you a taste of the variety of problems to which numerical optimization can be applied. It doesn't require you to punch in a bunch of data or copy a bunch of equations manually. It hints at optimal control, my favorite application of optimization (since I like robots). It also illustrates how we must often massage problems to fit the form of problems we know how to solve. Finally, it's fun to throw the might of modern technology at ancient questions. We don't have to be nearly as clever to find solutions as they did!

"Nonlinear programming" is a term that encompasses many approaches to numerical optimization. The general problem is to find the list of *decision variables* x that minimizes some scalar (single-valued) *objective function* $J(x)$ while satisfying a set of *constraints* like $C(x) \leq 0$. It's actually not restricted to minimization, since maximization of one function can always be expressed as minimization of the negative of the same function. Likewise, it's not restricted to inequality constraints, since an equality constraint on an expression can be expressed as two inequality constraints: the expression is less than or equal to zero, and its negative is less than or equal to zero. It turns out that this is quite a general problem statement indeed, and a *lot* of problems can be solved in this way - after some massaging to make the problem fit the specified form. After all, this assignment is about finding a *shape* that maximizes area, while nonlinear programming can only give us a list of numbers. How can we use nonlinear programming to tell us find a shape?

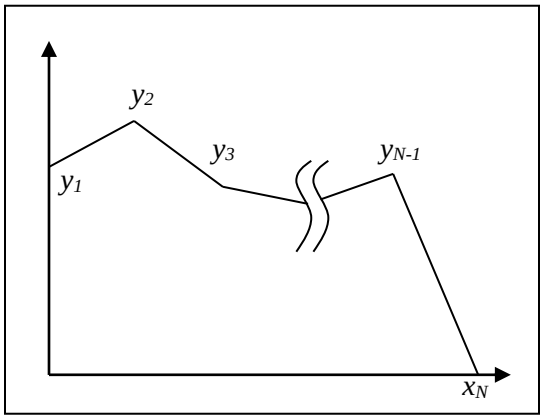
It's sort of like curve-fitting from the preparation. We assume some form of the shape's curve, describing the specific shape with a list of parameters, and we optimize those parameters as our decision variables. Since we're allowed to have as many decision variables as we want (within practical limitations), assuming some form of the curve is not much of a limitation. For this problem, we'll assume that our curve is a connected series of many straight lines parameterized by the points joining them, like:



We'll use nonlinear programming – specifically `scipy.optimize.minimize` – to choose the *x*s and *y*s that enclose the most area under the constraint that the sum of the line lengths is the specified perimeter. With a sufficient number of lines – 100, 1000, a million – we can approximate the true solution, a circle, quite closely.

Let's start by making the problem just a little simpler. We'll only solve a quarter of the problem, that is, we'll find the curve of given perimeter that encloses the most area between itself and the positive *x* and *y* axes (in the first quadrant). The solution to that problem is a quarter-circle, and this implies that the solution to the original problem is a full circle.

Let's also simplify the choice of *x*-coordinates. Instead of requiring `minimize` to pick a complete list of *x*s, let's just have it pick only the rightmost *x*-value (where the corresponding *y*-value has to be zero in order to "enclose" the area with the *x*-axis), and assume that the other *x* values are equally-spaced between zero and it. `minimize` will pick the corresponding *y*-value for each of these *x*-values, too. Therefore, our decision variables are the list of *N* values $[x_N, y_1, y_2, \ldots, y_{N-2}, y_{N-1}]$:



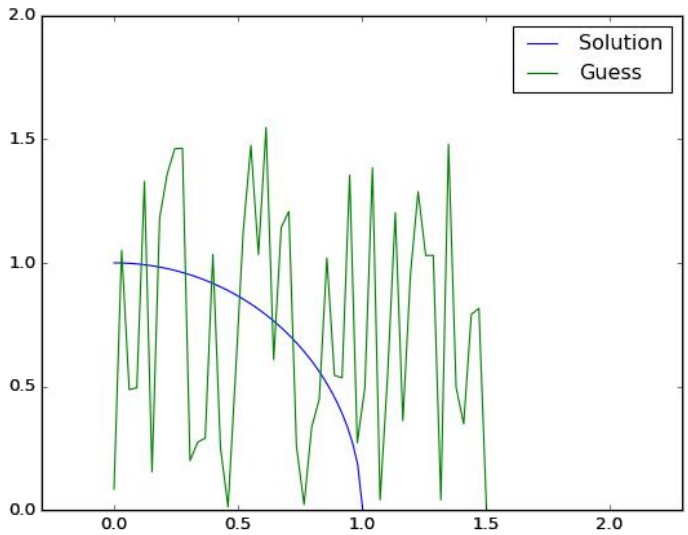
1. In your code, let *N* be 50. That is, you will have 50 *decision variables*: the *x*-coordinate of the 50th point in our set of lines, and the *y*-coordinates of the 49 other points.
2. Let the perimeter $P = \frac{\pi}{2}$.
3. We need to provide SciPy with a *guess* of the solution (values of the decision variables) for it to get started. Since we know the correct answer, a quarter circle of unit radius, let's base our guess on that. After all, if `minimize` can't solve the problem when we give it the correct answer as the guess, we'll know that something is wrong with our code!
    a. Generate arrays `x` ordered like $[x_1, x_2, \ldots, x_{N-1}, x_N]$ and `y` ordered like $[y_1, y_2, \ldots, y_{N-1}, y_N]$ to represent the unit quarter-circle with *N* points. Plot them to check that they are correct.
    b. From these, generate an array `z0`, ordered like $[x_N, y_1, y_2, \ldots, y_{N-2}, y_{N-1}]$, that will be your (perfect) guess of the *N* decision variables.
4. Write a function `z2xy` that calculates and returns the array of all the *x*s, $[x_1, x_2, \ldots, x_{N-1}, x_N]$, and all the *y*s ,$[y_1, y_2, \ldots, y_{N-1}, y_N]$, given (as a parameter) an array `z` of decision variable values, assuming they are ordered like $[x_N, y_1, y_2, \ldots, y_{N-2}, y_{N-1}]$. You will want to call this function in all the subsequent functions below, and it will be useful for plotting the solution returned by `minimize`. Hint: use `linspace` to generate your *x*s based on x_N from `z`, and append y_N to the list of *y*s from `z`. (Of course, you can choose to order your decision variables in `z` in any way you want, but your code would be a bit different depending on your choice. If you might want me to help you debug, please write this function assuming this ordering.)

5. Test your function `z2xy` on `z0`. It should return the original `x` and `y` arrays representing the unit quarter-circle because `z2xy` is the inverse of whatever you did to generate `z0` from `x` and `y`. Plot the returned arrays to confirm. If your `z2xy` doesn't return the `x` and `y` that you started with and plot a complete unit quarter-circle, you should fix your code before moving on.
6. Define the *objective* function, the function you are trying to minimize. Our goal is to *maximize* the area under the curve, but since SciPy's `minimize` function will find the values of the decision variables in `z` that *minimize* our objective function, it needs to return the *negative* of the area under this curve. So write this function `obj` that accepts your decision variables `z`, uses `z2xy` to calculate the corresponding *x*s and *y*s, then computes and returns the negative of the area (a single number) under the curve. (You will find `scipy.integrate.trapz` useful. Watch the order of the arguments! Your function should return a negative number…)
7. What should your objective function return when you pass it `z0` as an argument? Test it! If it doesn't return what you expect, fix it before moving on.
8. Define the *constraint* function. We didn't see this in the preparation, but it's no harder than defining the objective function. Here, we are going to specify that the total length of the curve must equal *P*. So first, compute the length *l* of the curve parameterized by the array `z`. This is not so hard, since `z` defines all the points of all the lines, you know how to find the length of a line, and the length of the curve *l* is the sum of the lengths of all its component lines. You can actually calculate this in just one line (no loops) using NumPy's ability to perform operations on all items of an array at once! SciPy is going to try to minimize the objective while making the constraint function return the value zero, so your function should return $l - P$. That way, this function will return 0 when the constraint is satisfied, that is, the length of the curve $l = P$. Like `obj`, this function `con` must accept only one argument, `z`.
9. What should your constraint function return when you pass it `z0` as an argument? Test it! If it doesn't return what you expect, fix it before moving on.
10. Specify *bounds* on the decision variables: a list of tuples containing the minimum and maximum permissible values for each decision variable. In our case, all the decision variable values must be positive (since our curve must lie in the first quadrant), and it's not possible for any of them to be greater than *P* (else the curve would certainly be longer than *P*), so return a list of 50 tuples (0, *P*). Use list comprehension, of course!
11. See the documentation for `scipy.optimize.minimize`. It says that we need to specify our constraint function in a dictionary with keys `type`, the type of constraint (equality or inequality), and `fun`, the constraint function itself (from 5). Create a dictionary containing this information for our problem.
12. Call `minimize`, providing as arguments the objective function from 4, the initial guess `z0` from 3, the list of bounds from 6, and the constraints dictionary from 7. Print the result returned by `minimize` to the console. If all went well, you'll see `success: True`, the `message: 'Optimization terminated successfully.'`, and that the objective function value `fun` is the negative of 0.785 (close to the true value of $\frac{\pi}{4}$, the area contained within a quarter-circle of the given perimeter). `minimize` has improved your guess `z0` for the decision variables and reports the optimal values of the decision variables in the field called `x` (not to be confused with the array of x coordinates).
13. Now let's see if SciPy can find the answer even when we give it a very bad guess. Make an array `z0` of 50 *random* numbers of reasonable magnitude for the problem. That is, since *P* is reasonable length scale for the problem, scale your array of 50 random floating point numbers such that the minimum is 0 and the maximum is *P*.
14. Run minimize with your new guess. Plot the guess and solution curve as shown on the next page.

Self-Assessment

Print the second and third page of the assignment (double-sided, preferably) and check off the steps you completed successfully. Give yourself 10 points each for all steps *except* 1, 2, 7, and 8. Indicate the total below.

Example solution:



Note that I used an option in `matplotlib` to make the *x*- and *y*-axis (visual size) scales equal….