Stanford University

AA222/CS361: Engineering Design Optimization

Spring 2023

Prof. Mykel J. Kochenderfer • NVIDIA Auditorium • email: *mykel@stanford.edu*

PROJECT 2: CONSTRAINED OPTIMIZATION **Due date: May 5, 2023 (5:00pm PT)**

The purpose of this project is for students to code their own constrained optimization strategies. This project assumes students have either Julia1.7+ or Python3.6+ installed on their computer and have familiarized themselves with the basics of running scripts in these languages. Installation instructions, a project tutorial, and programming resources can be found on the Project 0 web page: `https://aa222.stanford.edu/projects/project-0/`.

1 Project Overview

In this project, you will be solving a set of constrained optimization problems. You must implement at least two constrained optimization algorithms in Julia/Python. Whatever methods you choose, they must find a feasible point at least 95% of the time for credit. In addition, the submissions that get closest to the global optimum value (within the allotted function and gradient evaluations), win the competition!

You will be implementing a function `optimize` that minimizes a function with a limited number of evaluations, subject to constraints. The rules are as follows:

1. We provide you a function $f(x)$, its gradient $g(x)$, a function $c(x)$ which evaluates the constraints, and a number of allowed evaluations n.
2. Each call to `f` or `c` counts as one evaluation while each call to `g` counts as two evaluations. Note that `f` returns a scalar, while `g` and `c` both return vectors.
3. The only external libraries allowed for your implementation of `optimize` are `numpy` and `scipy.stats` in Python and `Statistics` and `Distributions` in Julia. In addition to those, you may use any of the standard libraries of either language.
4. You can use different optimization strategies for each problem, since we pass you a string `prob` in the call to `optimize`.
5. You can base your algorithm on those found in the book or online, but you must give credit. The code you submit must be typed by you (please do not copy/paste).
6. You must use the initial point in some way in your algorithm (even if you use a sampling-based method). For sampling-based methods, you should use the initial point to define the distribution for your initial samples.
7. Although you may discuss your algorithm with others, you must not share code.

2 Project Instructions

The following instructions assume that the `AA222Project2` folder is your working directory.

2.1 Choose a programming language

Pick either Julia or Python as a programming language. Depending on your choice, go to `language.txt` and change `notalanguage` to either `julia` or `python`.

2.2 Complete the required code

If you chose Julia, go to `project2_jl/project2.jl` and complete the function `optimize`. If you chose Python, go to `project2_py/project2.py` and complete the function `optimize`. To get full credit on a given problem, your implementation must return a feasible point for at least 475 out of 500 different initial guesses (x_0). The autograder does not require that you use a different algorithm for each problem, but you are free to do so

2.3 Test your completed code

If you chose Julia test your completed code by running:

```
julia --color=yes localtest.jl
```

(see the file for more details) If you chose Python, test your completed code by running:

```
python3 localtest.py
```

You should see `Pass: optimize returns a feasible solution on [X]/500 random seeds.` for all the simple problems.

2.4 Prepare your `README.pdf`

In addition to the programming aspect, you are also required to submit (also on Gradescope) a PDF writeup, worth 50% of the assignment. In the README, you will be required to describe the algorithms called by `optimize`, as well as compare the performance of at least two distinct algorithms on the simple problems.

NOTE: you only need to pass the autograder one of your two algorithms.

2.4.1 Description of algorithms

For each of the five problems given, describe:

1. The algorithm you chose to solve it.
2. How you decided which hyperparameters to choose.
3. Why you think it works. (i.e. what are the specific mechanisms in the algorithm that allow it to find good solutions?)
4. At least one pro and one con to the chosen algorithm.

2.4.2 Comparison of algorithms

You must compare the performance of at least two algorithms by doing the following.

1. For `simple1` and `simple2`, plot the feasible region (where $c(x) \leq 0$) on top of a contour plot of $f(x)$. Show the path taken by the algorithm for at least three initial conditions on this plot. Make a separate plot for at least two distinct algorithms (four plots). For both problems, the axis limits should be $(-3, 3)$.
2. For only `simple2`, and for at least two distinct algorithms, plot the objective function versus iteration, and maximum constraint violation versus iteration, for at least three initial conditions. Plot the curves for different initial conditions on the same plot, but make a separate plot for the objective and constraint violation, and for each of the two algorithms compared (four plots).

2.5 Create the code submission

Create your `project2` submission by running

```
bash ./make_submission.sh
```

from the `starter_code` directory if you are on a Mac/Linux. On windows, run the following from a GitBash terminal (see Project 0 for download instructions):

```
bash ./make_submission_gitbash.sh
```

2.6 Submit on Gradescope

1. Submit the created zip file (`project2.zip`) on Gradescope/AA222/Project2
2. Submit your `README.pdf` on Gradescope/AA222/Project2 Writeup

3 Objective Functions

Three of the five objective functions are explicitly listed below and included in the starter code. Your algorithm will be tested and ranked on two additional secret functions, which are described below. Note: We are only revealing these problem descriptions to spark your interest. Knowledge of the problem domains is not necessary to be able to complete this project, and they can be solved with algorithms covered in this class.

1. Simple 1:
   - $f = -x_1 x_2 + \frac{2}{3\sqrt{3}}$
   - $c_1 = x_1 + x_2^2 - 1 \leq 0$
   - $c_2 = -x_1 - x_2 \leq 0$
   - $x^* = (\frac{2}{3}, \frac{1}{\sqrt{3}})^\top, \quad f^* = 0$
   - Max number of evaluations = 2000
2. Simple 2:
   - $f = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$
   - $c_1 = (x_1 - 1)^3 - x_2 + 1 \leq 0$
   - $c_2 = x_1 + x_2 - 2 \leq 0$
   - $x^* = (1, 1)^\top \quad f^* = 0$
   - Max number of evaluations = 2000
3. Simple 3:
   - $f = x_1 - 2x_2 + x_3 + \sqrt{6}$
   - $c_1 = x_1^2 + x_2^2 + x_3^2 - 1 \leq 0$
   - $x^* = (-\sqrt{\frac{1}{6}}, \sqrt{\frac{2}{3}}, -\sqrt{\frac{1}{6}})^\top, \quad f^* = 0$
   - Max number of evaluations = 2000
4. Secret 1:
   - In this problem, must find the minimum of an objective with many local optima within an elliptical boundary.
   - $x \in \mathbb{R}^4$, and there is one constraint $c(x) \in \mathbb{R}$.
   - Max number of evaluations = 2000

5. Secret 2:

   - In this problem, we must build ramps that minimize the time for a particle under gravity to traverse them. No part of those ramps can be taller than the starting point.
   - $x \in \mathbb{R}^{100}, \quad c(x) \in \mathbb{R}^{100}$.
   - Max number of evaluations = 2000

4 Frequently Asked Questions

What counts as "two distinct algorithms"?

Two algorithms are considered distinct if they differ in any more than their selection of hyperparameters. For example, a penalty method with a quadratic penalty and count penalty are distinct, while two quadratic penalty methods with different choices of ρ are not.

Can I compute the gradient of the constraint functions?

Sure! However, if you want to get the gradient of the constraints, you must implement a finite difference method. You may NOT hard-code an analytical solution, and you may NOT use any automatic differentiation packages (see rule number 3). These rules ensure that your number of function calls is accurate.

My strategy involves randomness. Are the scores averaged with different random seeds?

Yes! The scores are averaged over 500 runs with different seeds.

Where are the Hessians?

We are not providing the Hessian function for you to use. But feel free to estimate it with calls to `f` and `g`. The cost would depend on the number of calls to `f` and `g` you end up making. Note that you are not allowed to implement and use the analytical Hessians for problems.

How many submission can we make?

Unlimited!

Can we exceed the max number of evaluations when making the plots for the README?

Yes! In python, you can get around the assertion error by calling the `problem.nolimit()` method to allow infinite evaluations.

In Julia, you can pass in `n = Inf` to `optimize`.

How long does the autograder take to grade?

It shouldn't take more than 10 minutes to grade. If your submission times-out during grading, please contact us on Ed.

How are leaderboard scores computed?

All of the problems are designed to have an optimal value near 0. The closer you are to 0, the closer you are to winning! The total score is the sum of all 5 problems (all of the problems are weighted the same). Infeasible solutions are still given finite scores, but are heavily penalized ($\mathcal{O}(1e7)$), so you cannot win unless your method always finds a feasible point.

Can I write code outside of the optimize function?

Yes, you can organize your code however you want as long as, at the end of the day, optimize works as described. Note that if you decide to create additional files, please make sure to import/include (python/julia respectively) them in your `project2` file, or else they won't be available to the autograder.

Can I change the starter code files?

Yes, they are not used in the autograder, so you have complete ownership over them.

Do I have to manually keep track of how many times the functions have been called?

Nope, we've decided to be very generous and provide a method for doing just that.

In Julia: `count(f)` will return how many times the function `f` has been called. For convenience, `count(f, g, c)` will give `count(f) + 2*count(g) + count(c)`. Example:

```
function optimize(f, g, c, x0, n, prob)
  while count(f, g, c) < n
    # ... do some optimization
  end
  return x_best
end
```

In Python: the optimize function has additional input argument count, which takes no arguments and evaluates to f + 2g + c. Example:

```
def optimize(f, g, c, x0, n, count, prob):
  while count() < n:
    # ... do some optimization
  return x_best
```

My README plots require an optimization path, but `optimize` only returns the optimum itself!

`optimize` is geared towards the autograder's evaluation of your method, so it only returns the final point. However, if you're coding with the `README` in mind (which is a good idea!), you may want to design your code in a way that is conducive to both requirements by writing methods that collect the optimization history. See the following julia example:

```
function optimize(f, g, c, x0, n, prob)
  if prob == "simple1"
    x_history = some_method(f, g, c, x0, n)
  else
    x_history = some_other_method(f, g, c, x0, n)
   end
   return last(x_history)
end
```