HAYLEY R. O. SOHN

Boulder, CO
(636) 208-0743 :: hayley.r.o.sohn@gmail.com :: personal website

January 13, 2019

Approach
In this write-up, I will first explain the approaches and ideas I used to solve the challenge problem, then dive into the specific execution and explanation of the deliverable attached. From the prompt, I categorized two distinct methods to generate solutions, which I'll refer to as the "Bottom-Up" and "Top-Down" approaches.
My first preference was the Bottom-Up approach, in which I planned to use the constraint equations as input to generate a set of vectors representing all possible solutions by means of some kind of solving or minimization functions in NumPy or SymPy. Another option I had in mind for generating the feasible solutions using this method was to define the boundaries of the valid space one dimension at a time, then use the minimum/maximum values from each dimension to find the space where all the valid regions overlap to form a composite valid space. I spent a couple hours digging into these ideas for the Bottom-Up approach and concluded that this method would incur additional constraints in order to solve for all possible solutions, including limiting assumptions about the shape of the valid space, and many of these steps would only work up to a few dimensions.
For the sake of submitting something that worked for higher dimensions without additional constraints or assumptions, I turned my attention to the Top-Down approach, in which I generate a full set of all possible numerical points in the space for each dimension, then create vectors by iterative combination of those points, and apply the constraints to systematically eliminate the vectors that lie outside of the valid space. Once a feasible set of vectors is determined, I selected *n_results* = 1000 vectors evenly-spaced throughout the list of all solutions to write to the output file. I will explain this approach in detail below.

Execution
The *sampling.py* file that I've submitted attached is a Python class that has been expanded from the *constraints.py* source provided in the challenge folder. At the end of the file are a few lines that are used to define input variables and build/run the code.

The functions within the class are:
__init__ *:* Constructs a new object from the input file, parses the input file, and assigns variables/lists for the dimensions, example point, and constraints.
get_example : Returns list containing the example feasibility point.
get_ndim : Returns variable containing the dimensionality of the example.
get_allvectors : Calculates all possible vector solutions that meet the multi-dimensional constraints. First, volume-fraction points for each dimension are generated using *linspace*, with values between 0 and 1. For each dimension the value provided in the example feasibility point (*self.example*) is evaluated to define the precision (number of decimal points) of the solution set (*self.all_solns).* The purpose of this step is to generate the minimum number of points to be evaluated while avoiding unnecessary precision/computation time.
An embedded function based on *apply* in *constraints.py* is defined and called within a nested for-loop structure to iteratively evaluate each vector created by systematically combining the solution points for each dimension in *self.all_solns*. These nested for-loops are sufficient to analyze files up to 12 dimensions, since the largest example file provided was 11 dimensions. If-statements are included to evaluate dimensionality and avoid unnecessary iterations. Each vector that fulfills the constraints is saved in *self.all_vectors*.
get_nresults_solns *:* Selects a number (*n_results*) of requested vectors from the complete solution set (*self.all_vectors*) by using *linspace* to define evenly-spaced query points within the list of solutions. The if-statement allows solution sets with fewer vectors than the requested number to pass straight to the writing stage, with all vectors printed in the output file.

HAYLEY R. O. SOHN

Boulder, CO
(636) 208-0743 :: hayley.r.o.sohn@gmail.com :: personal website

Results & Testing
I would have preferred to avoid using nested for-loops, which significantly slow down the code and cause an exponential increase in the run-time because of the combinatorics nature of my approach. I am confident that the code I'm submitting can reliably produce a solution set within the valid space for dimensions higher than four, but I have only been able to run up to four dimensions because of time constraints. The table below shows the run-time for each sample file provided. For the *alloy.txt* example file with 11 dimensions, I initiated the build and left for lunch for ~ 1 hour and when I came back it hadn't finished. That being said, there might be some errors for larger dimensions that I haven't found yet, but I suspect the hurdle is a long run-time like I mentioned above.

Input file	Dimensions	Time to complete (s)
example	4	0.8
formulation	4	232.8
mixture	2	0.3
alloy	11	over 1 hour

A few limitations of the code include the exponential increase in run-time and a bug at the end when the *self.output_vectors* set is generated, which tends to generate 999 (= *n_results*-1) vectors instead of 1000. I think this has something to do with the final for-loop, but I haven't ironed that out in the few minutes I've spent on it.
Another limiting assumption I haven't addressed for the sake of time is that, for some lower dimensional cases like the *mixture.txt* file in which I generate less than *n_results* = 1000 vectors, the *get_nresults_solns* writes fewer solutions than requested. To address this, I would add some kind of try/catch-type of switch to create more possible solutions (with increased precision) in the *get_allvectors* stage.

Conclusions
I enjoyed working on this challenge and hope I've been able to articulate my level of expertise and areas of potential growth through these files. I am interested in learning best practices for approaching this kind of higher-dimensional challenge outside of my typical toolbox for 2-3D problems, in which I'd usually rely on meshing functions to generate evenly-distributed solutions. It was an intriguing puzzle to solve and my goal/metric of success in this challenge was to deliver something reliable and robust (but nevertheless clunky) that generated an output file in the correct format for various input dimensions. I'm confident the progress I've made towards that goal is representative of my skill set.