Eric's Python Notes

Eric A. Applegate

ARTICLE HISTORY
Compiled December 16, 2019

1. Changes Eric made to Nathan's Python files

- in `allocate` function within `allocate.py`, updated input arguments to include the `WSFlag` parameter as in Matlab and added "equal" and "brute force ind" allocations (see next bullet points).
- in `allocate.py`, added the `equal_allocation` function for equal allocation
- in `allocate.py`, added the `bfind_allocation_smart` function for brute force independent allocation
- in `utils.py`, edited the `calc_phantom_rate` function (it was not working and needed some import statements)
- in `utils.py`, added the `calc_bf_rate` function based on my Matlab code and how Nathan coded `calc_phantom_rate` in Python
- added `python_timing.py`. This file contains the functions from the `Timing` folder of the Matlab implementation that read in the problems we used in the journal article timing tables, calculate times for each allocation, calculate rates for each problem, record the times and rates in `.pickle` files, and summarize the data to re-create the timing tables. The file is set up such that it can be called from the terminal (i.e. `python python_timing.py`) to start executing (and be left to run however long it takes).

2. Work that needs to be done

- Find all places (Solve, TestSolve, Allocate) where `allocate` is called and update to include `WSFlag` where appropriate
- Debug Brute Force allocations. After running the $d = 3, r = 10$ timing table row in Table 1 (below), it seems as though there's something wrong with the Python Brute Force calculations. If we ignore the times (we expected them to be longer than Matlab), the phantom rates in Table 1 are approximately what we saw in the corresponding table in the journal article. However, the brute force rates are different when comparing to the journal article and the "MVN True" and "MVN Ind." allocation columns are also quite different from the journal article table.

  I have compared some of the Matlab brute force rate computations to the Python rate computations and have found that they are not significantly different, in my opinion. For instance, in comparing Matlab vs Python rates for problems in the $d = 3, r = 10$ row, I found that the rates would differ such as

0.0041 compared to 0.0046. But, as we multiply these "raw" rates by 10^5 for Table 1, it appears to be a bigger difference. Now, one may argue that a difference between 0.0041 and 0.0046 IS significant. If that is the case, I would encourage the next researcher to look into the quadratic solver settings in both Matlab and Python to see if tweaking settings may help.

It is curious to see that the brute force rates and phantom rates are different for the $d = 3, r = 10$ timing table row in Table 1, in particular for the Equal allocation column. This again seems to be that some of the quadratic solver outputs for the brute force computations are just slightly different than those for the phantom rate computations, such as the 0.0041 and 0.0046 mentioned above. Note that in looking at the brute force vs phantom rates for individual problems within that row, several are indeed the same. The ones that are different are just different enough to affect the 50th percentile computation for the table (when factoring in the multiplication by 10^5). I have compared the Python and Matlab code closely to ensure that each of them is setting up the brute force rate computation the same way (same objective functions and constraints passed to the solver), so any difference in the output must be due to a difference in the solver algorithm or stopping criterion. Again, if this continues to be something that needs looking into, I would encourage the next researcher to look into quadratic solver setting differences between Matlab and Python.

Finally, it is obvious from comparing the journal article table to Table 1 that the Brute Force allocations are not correct. Based on the above two paragraphs, it appears that the rate computations are set up correctly and result in values that are in the same "neighborhood" of the Matlab rates. But, I see that the resulting allocations from the Python Brute Force functions are not similar to the Matlab allocations. For example, the Matlab Brute Force allocation for problem number 3 in the $d = 3, r = 10$ timing table row is

$$0.2025, 0.04293, 0.20297, 0.22499, 0.044882, 0.00035268, 0.22086, 0.013698, 0.044836, 0.0019841$$

while the Python allocation is

$$0.108517, 0.10552, 0.09897, 0.0998, 0.1123, 0.09156, 0.096245, 0.09666, 0.09773, 0.09268$$

which appears to be close to equal allocation. Thus, I would encourage the next researcher to look closely at the solver settings for Brute Force in Python to ensure we are getting correct optimal allocations.

- After making the above changes and verifying that brute force is correct, move on to update/test/debug `solve` and `testsolve`. Updates may include changing any calls to `allocate` (see first bullet), ensuring that previous alpha-hat vectors are not used in the sequential iterations (a change we made in Matlab around the time Nathan left the project), and ensuring that the output is saved in some fashion (text or pickle files).
- Figure out how to incorporate all of this into PyMOSO via the command line. After discussing with Kyle Cooper at WSC, I believe this will involve re-coding the sequential algorithm to "work like" an RA iteration of R-PERLE or R-MGSPLINE. So, in a `Solve` implementation, a line should be written to the output (text) file each time we estimate the Pareto set, starting with the first estimated Pareto set after initial samples are taken for each system. Then, in a `TestSolve`, we would write a separate text file for each macro replication as if it

were an individual `Solve`. The "metric" in PyMOSO for these types of problems would be the misclassification statistics (PMC, PMCE, PMCI) at each iteration.

Additionally, the next researcher will need to work with Kyle Cooper to figure out how a user will specify all the input parameters (initial samples δ_0, samples between allocations Δ, parallel processing, CRN, α_ϵ, etc...). Some of these parameters will be similar to what's already in PyMOSO, but others will need to be defined. Also, we'll need to figure out how a user should specify which allocation to use - should we create a general MOSCORE sequential algorithm and use an input parameter to define the allocation, or should we define individual solvers for each allocation?

Table 1. PYTHON WITHOUT WARM-STARTS For 10 MORS problems generated by the fixed Pareto method, each with $d \geq 3$ objectives and $r \geq 10$ systems, the rest of the table reports: the median number of Paretos and phantoms, sample quantiles of the wall-clock time T to solve for each allocation $\boldsymbol{\alpha}$ in minutes (m) and seconds (s); the median rate of decay of the $\mathbb{P}\{\text{MC}\}$ calculated by brute-force ($Z^{\text{bf}}_{0.5}(\boldsymbol{\alpha})\times 10^5$) or by the phantom approximation ($Z^{\text{ph}}_{0.5}(\boldsymbol{\alpha})\times 10^5$).

d	r	p	Med. $\lvert\mathcal{P}^{\text{ph}}\rvert$	Metric	MVN True	MVN Phantom	MO-SCORE	MVN Ind.	iMO-SCORE	Equal
3	10	5	11	Median T	6m 31s	3.75s	0.71s	7m 52s	0.26s	0s
				75th %-ile T	10m 32s	5.62s	1.05s	15m 13s	0.38s	0s
				$Z^{\text{bf}}_{0.5}(\boldsymbol{\alpha})\times 10^5$	545.830	950.225	924.567	524.921	720.966	426.498
				$Z^{\text{ph}}_{0.5}(\boldsymbol{\alpha})\times 10^5$	510.551	950.211	924.543	519.867	695.679	393.455
	500	10	21	Median T	–	–	–	–	–	–
				75th %-ile T	–	–	–	–	–	–
				$Z^{\text{ph}}_{0.5}(\boldsymbol{\alpha})\times 10^5$	–	–	–	–	–	–
	10,000	10	21	Median T	–	–	–	–	–	–
				75th %-ile T	–	–	–	–	–	–
				$Z^{\text{ph}}_{0.5}(\boldsymbol{\alpha})\times 10^5$	–	–	–	–	–	–
4	5,000	10	42	Median T	–	–	–	–	–	–
				75th %-ile T	–	–	–	–	–	–
				$Z^{\text{ph}}_{0.5}(\boldsymbol{\alpha})\times 10^5$	–	–	–	–	–	–
	10,000	10	44	Median T	–	–	–	–	–	–
				75th %-ile T	–	–	–	–	–	–
				$Z^{\text{ph}}_{0.5}(\boldsymbol{\alpha})\times 10^5$	–	–	–	–	–	–
5	10,000	10	90	Median T	–	–	–	–	–	–
				75th %-ile T	–	–	–	–	–	–
				$Z^{\text{ph}}_{0.5}(\boldsymbol{\alpha})\times 10^5$	–	–	–	–	–	–

Computed in MATLAB R2017a on a 3.5 Ghz Intel Core i7 processor with 16GB 2133 MHz LPDDR3 memory.