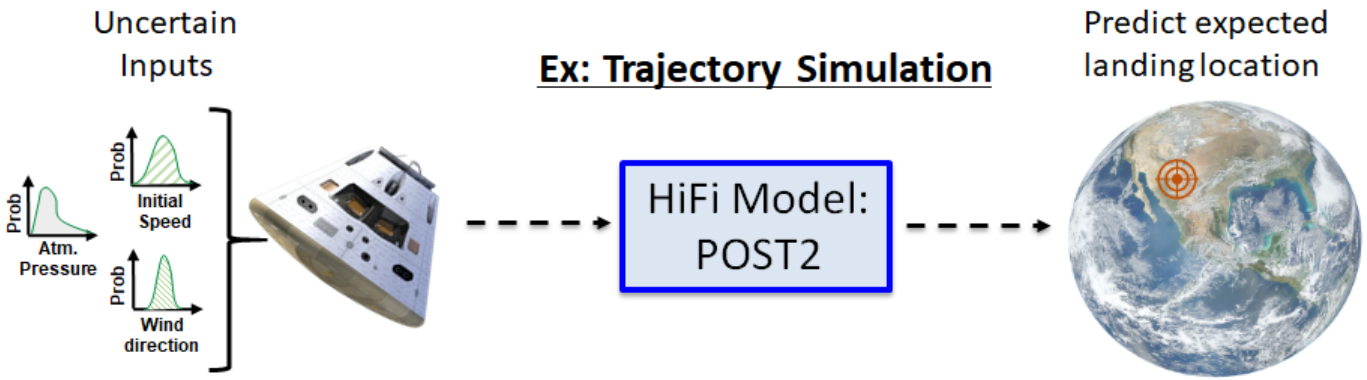
Trajectory Simulation Using Multi Model Monte Carlo with Python (MXMCPy)

Thomas Shera

Purdue University

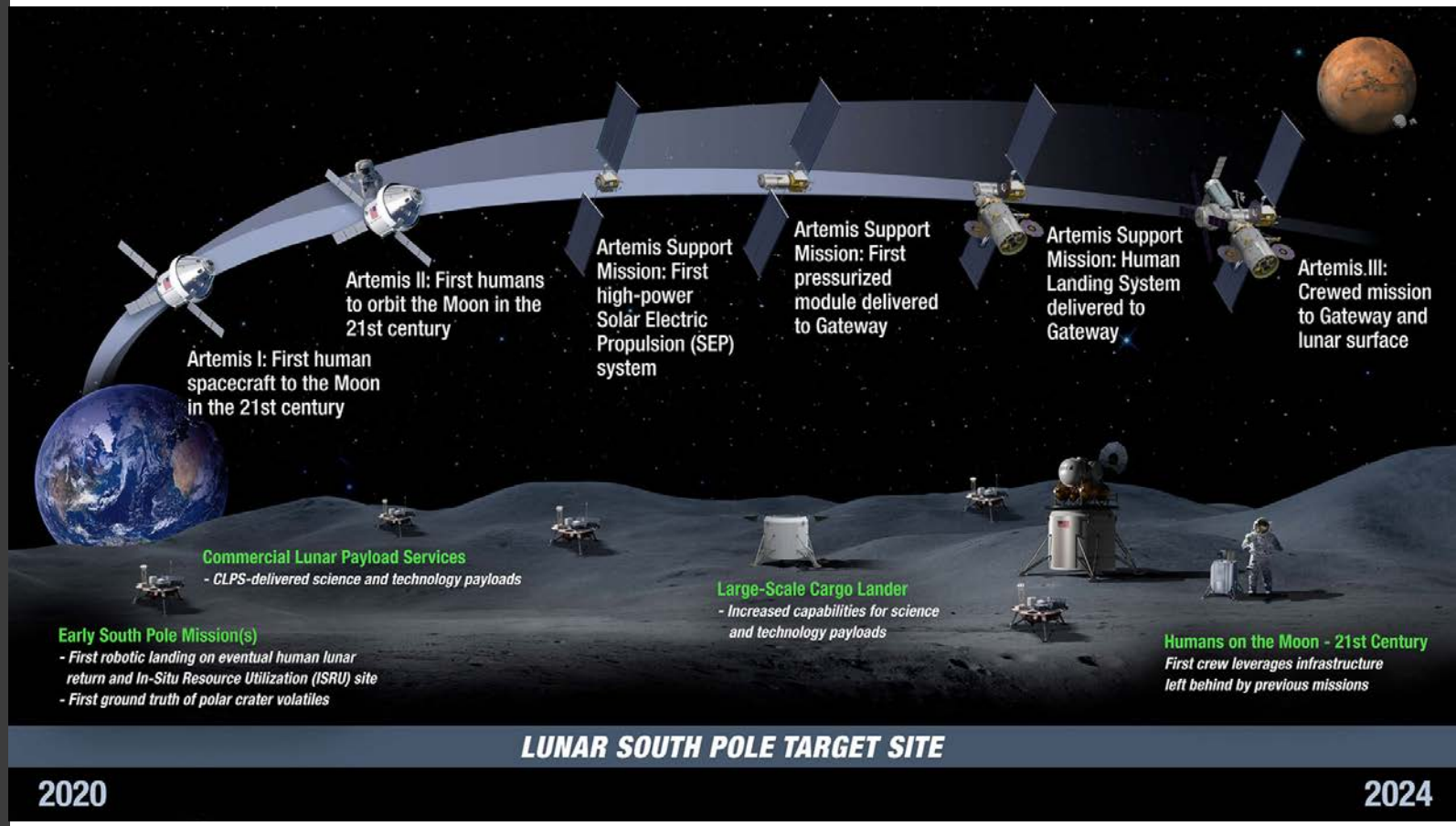
Introduction

- EDL (Entry, Descent and Landing): the stages from a vehicle approaching the surface to when it lands on the surface
- Existing NASA application POST2 (Program to Optimize Simulated Trajectories 2) is used by Langley as the primary EDL simulation tool
- POST2 is used in many Mars/Lunar missions NASA-wide



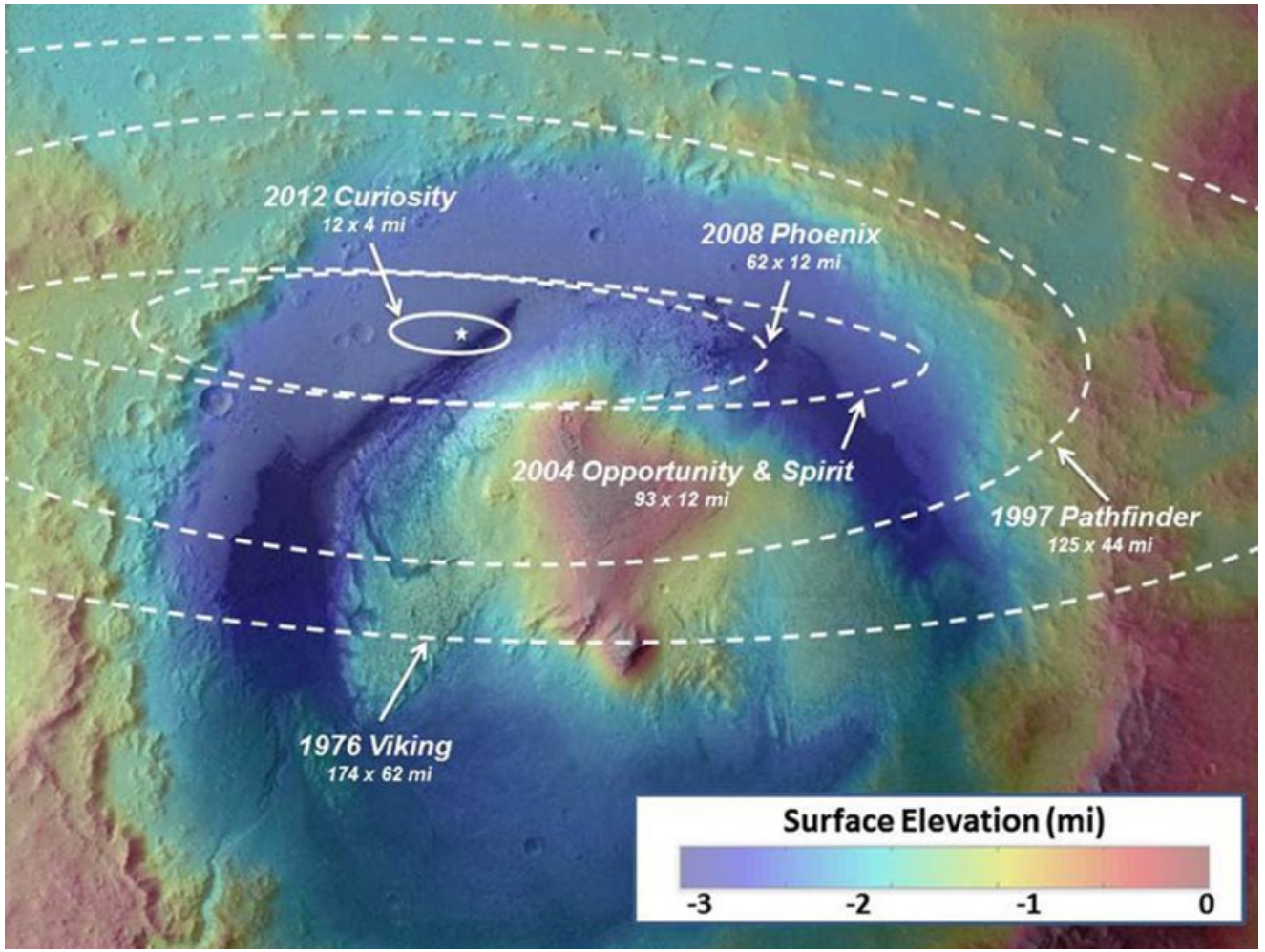
Motivation

- With more planned Moon/Mars landings in the near future, there is increased need for quicker EDL POST2 simulations
- Quicker simulations will allow for closer to real-time EDL predictions which will aid in more informed EDL vehicle control and decision making



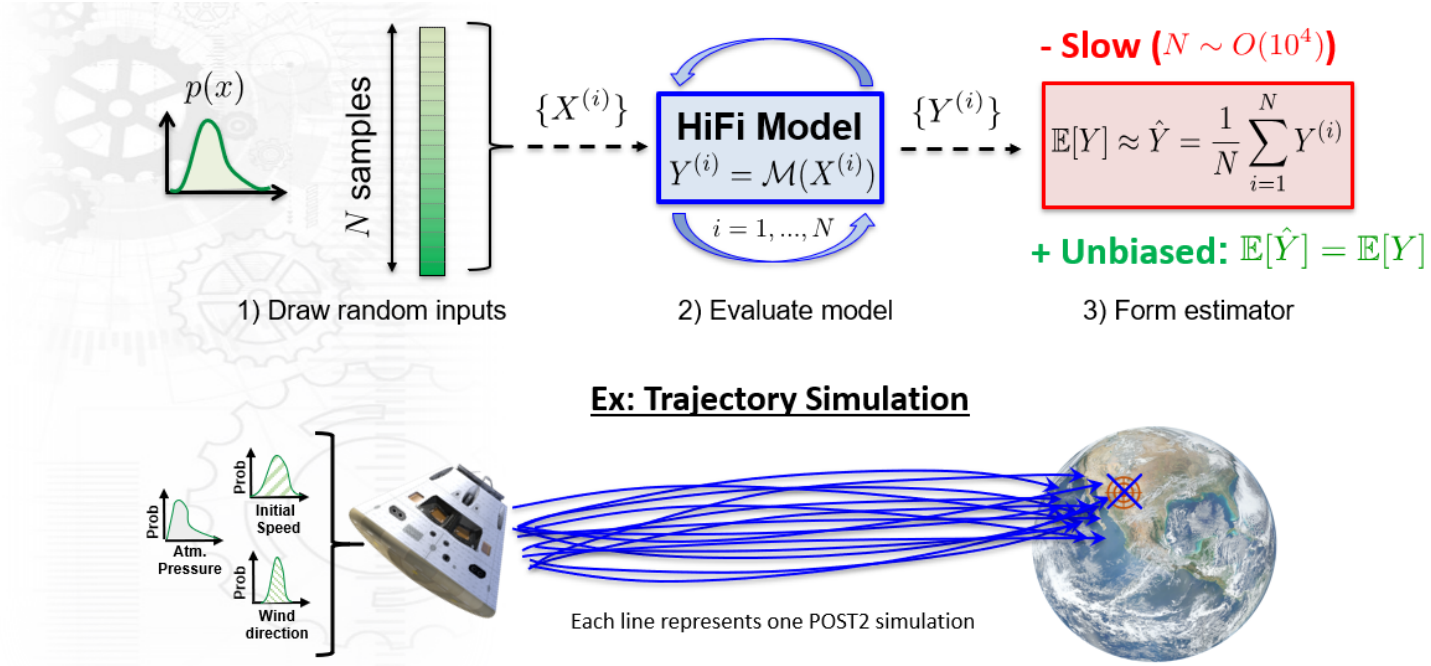
Motivation

- EDL landing ellipse estimation needs to be decreased from km to meters to meet mission plans to land supply vehicles near each other



Monte Carlo

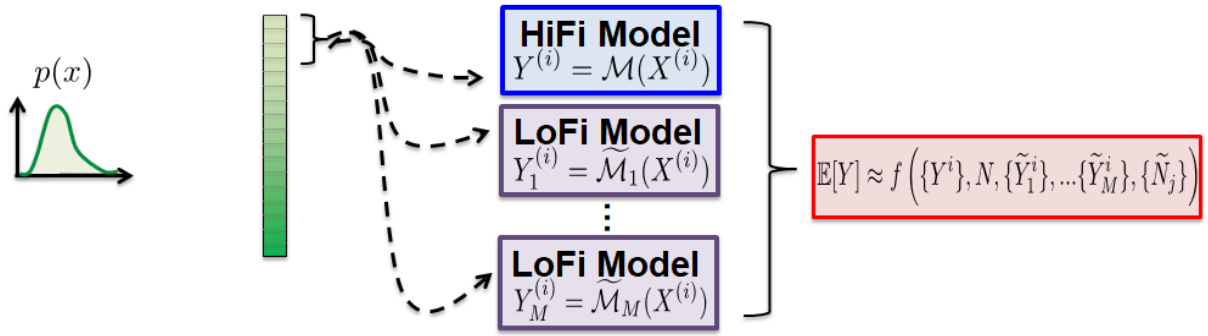
- Monte Carlo simulations are traditionally used to quantify the impact of uncertainties such as vehicle state and atmospheric conditions
- Traditional Monte Carlo (MC) POST2 simulations can take impractical lengths of time to deliver precision sufficient for the desired landing ellipse size
- Many time-consuming high fidelity MC iterations required for high precision



Multi-Model Monte Carlo

- Fewer evaluations of the slow, high-fidelity model are coupled with many more evaluations of fast, lower-fidelity models

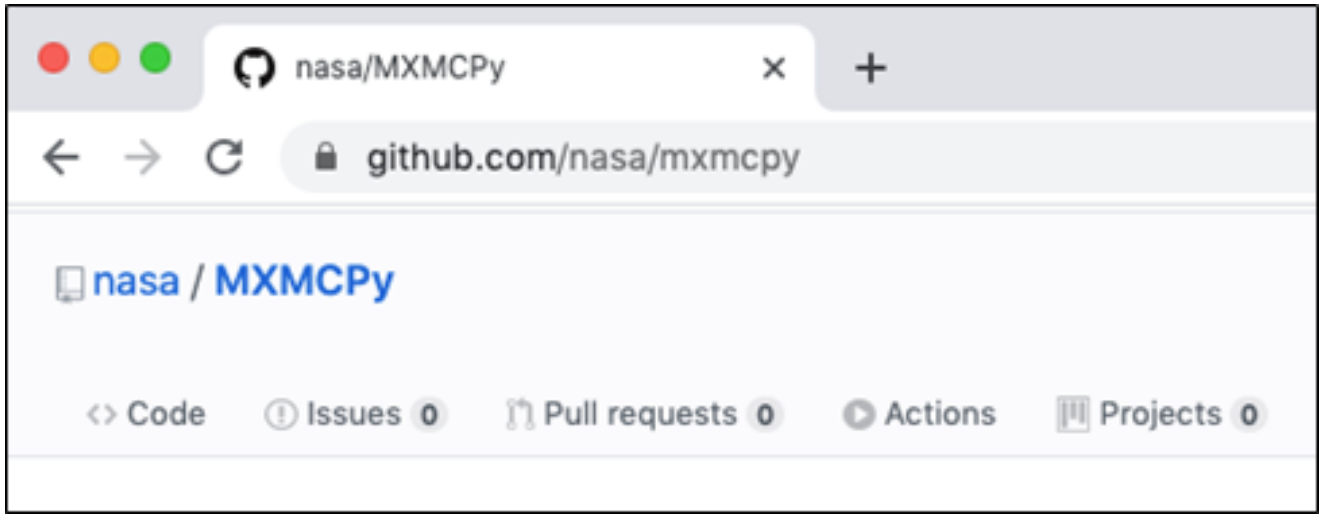
Multi-Model Monte Carlo Simulation

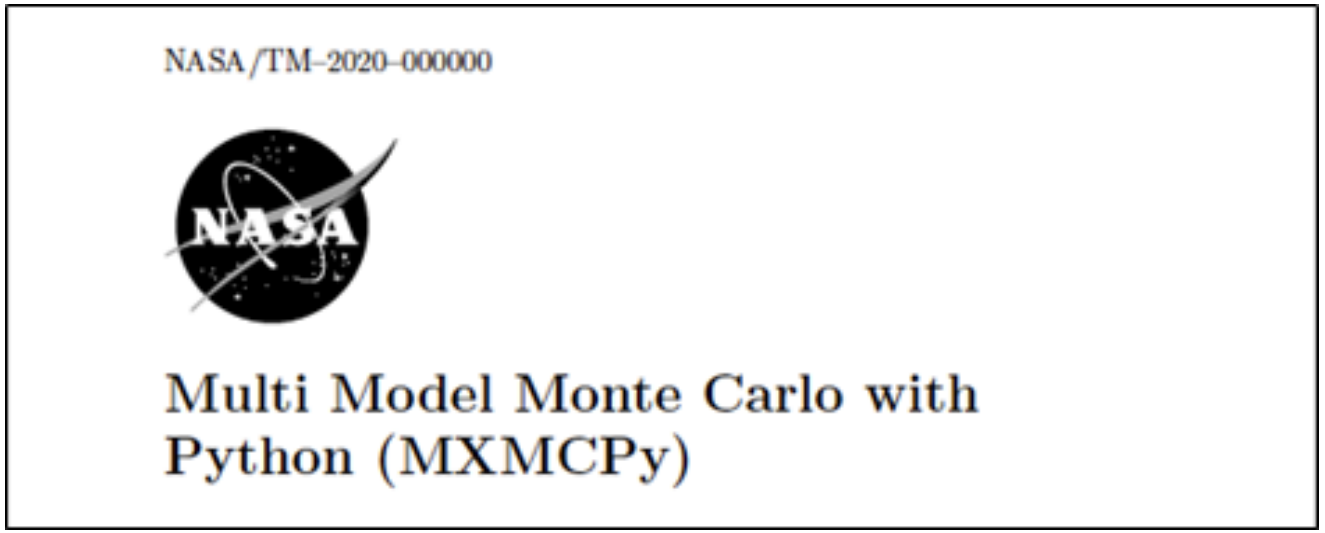


- *Low-fidelity models for computational speedup;* High fidelity model for *accuracy guarantees*
- These methods generalize to leverage an arbitrary collection of *M* low-fidelity models (analytical, data-driven, coarse discretization, simplified physics, etc.)
- The key to multi-model Monte Carlo approaches is the optimal allocation of computational resources across available models

MXMCPy

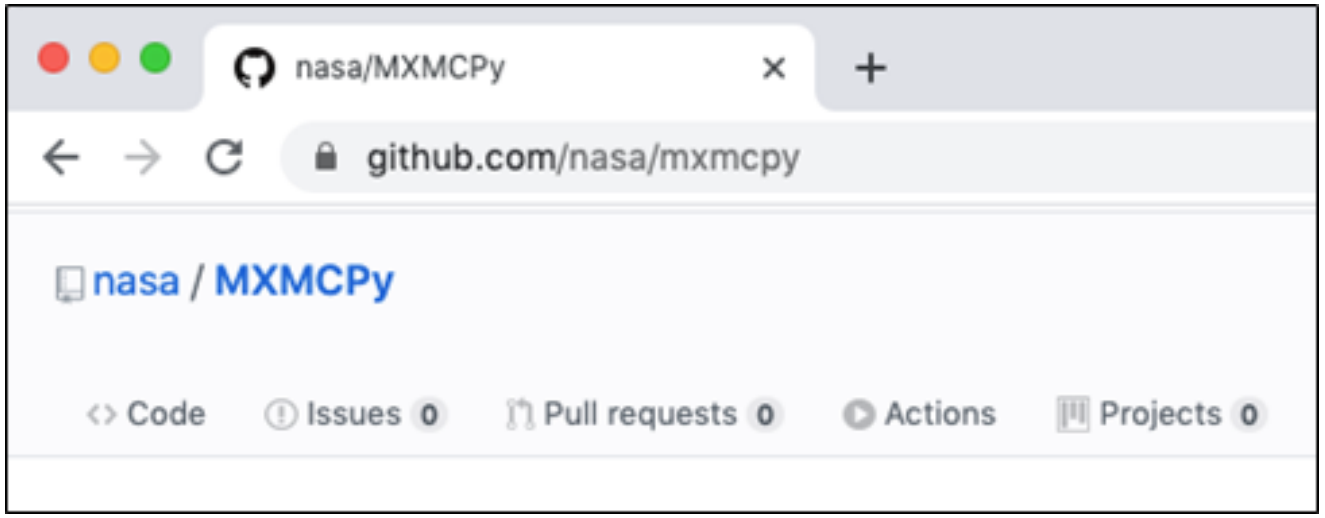
- MXMCPy implements several multi-model Monte Carlo methods so a user can easily compare performance

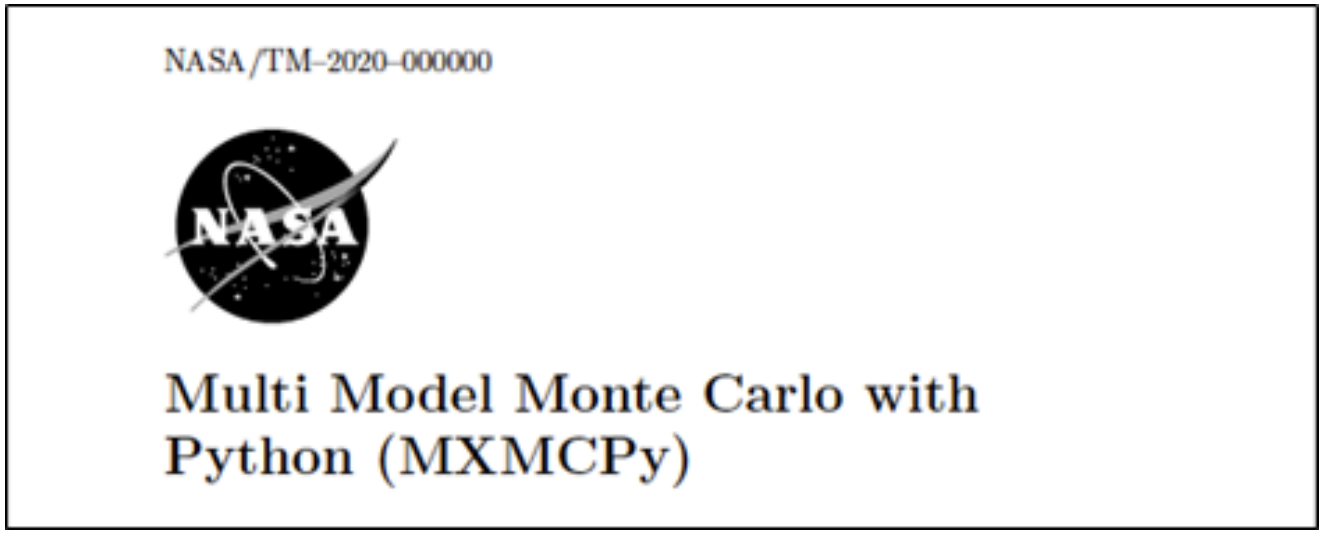




MXMCPy

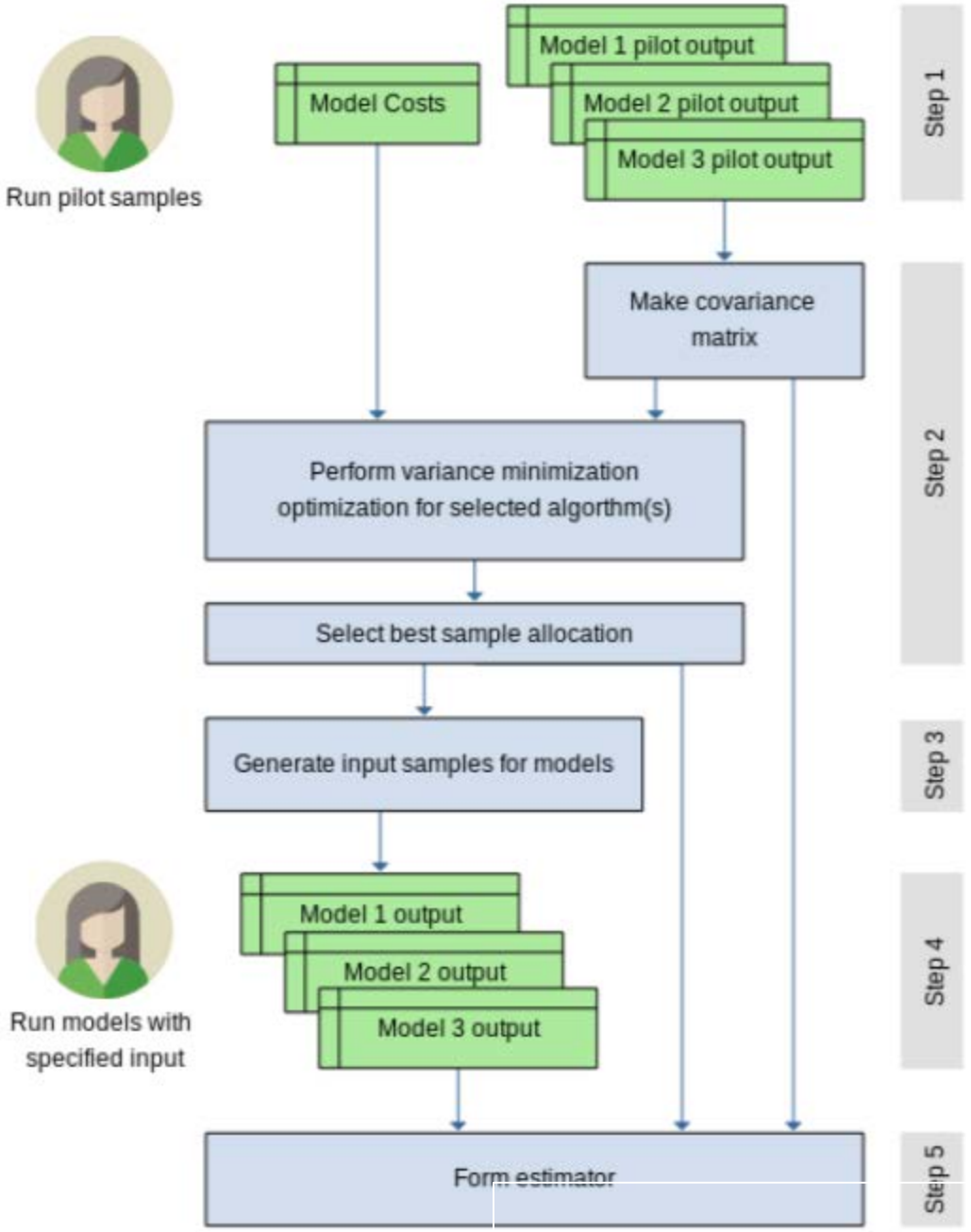
- Can be used with POST2 to optimize the allocation of POST2 runs for several different model fidelities, combining long, precise simulations with quick, imprecise simulations for optimal results
- Informs the user on what method of sample allocation is best to deliver the most precise solution for a specific target cost of computer runtime





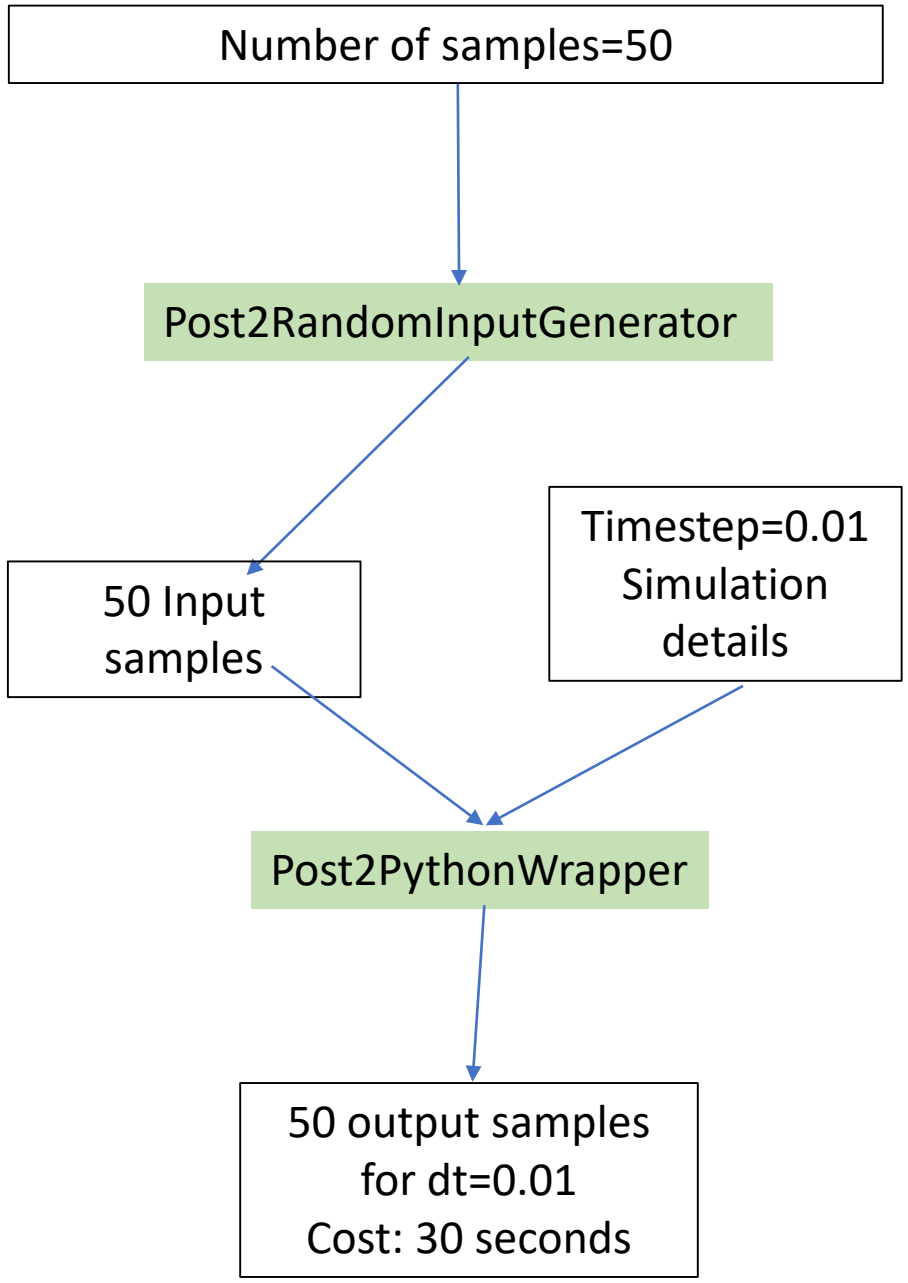
MXMCPy

- 5 steps to form an estimate on a quantity of interest



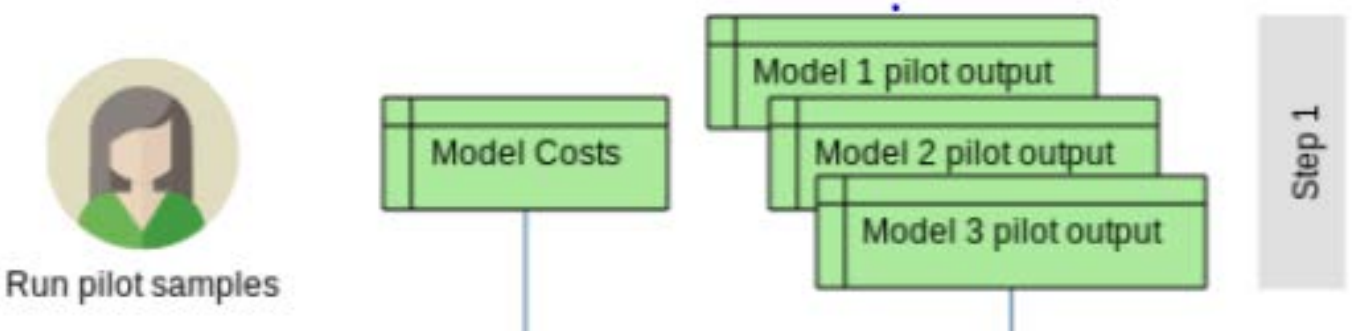
Terminology

- Input samples: randomly generated values for 80+ POST2 attributes
- Timestep: time lapse between POST2 simulation iterations
- Output samples: results generated by POST2 from input samples
- Model: POST2 instance
- Cost: runtime of POST2 model with specific timestep



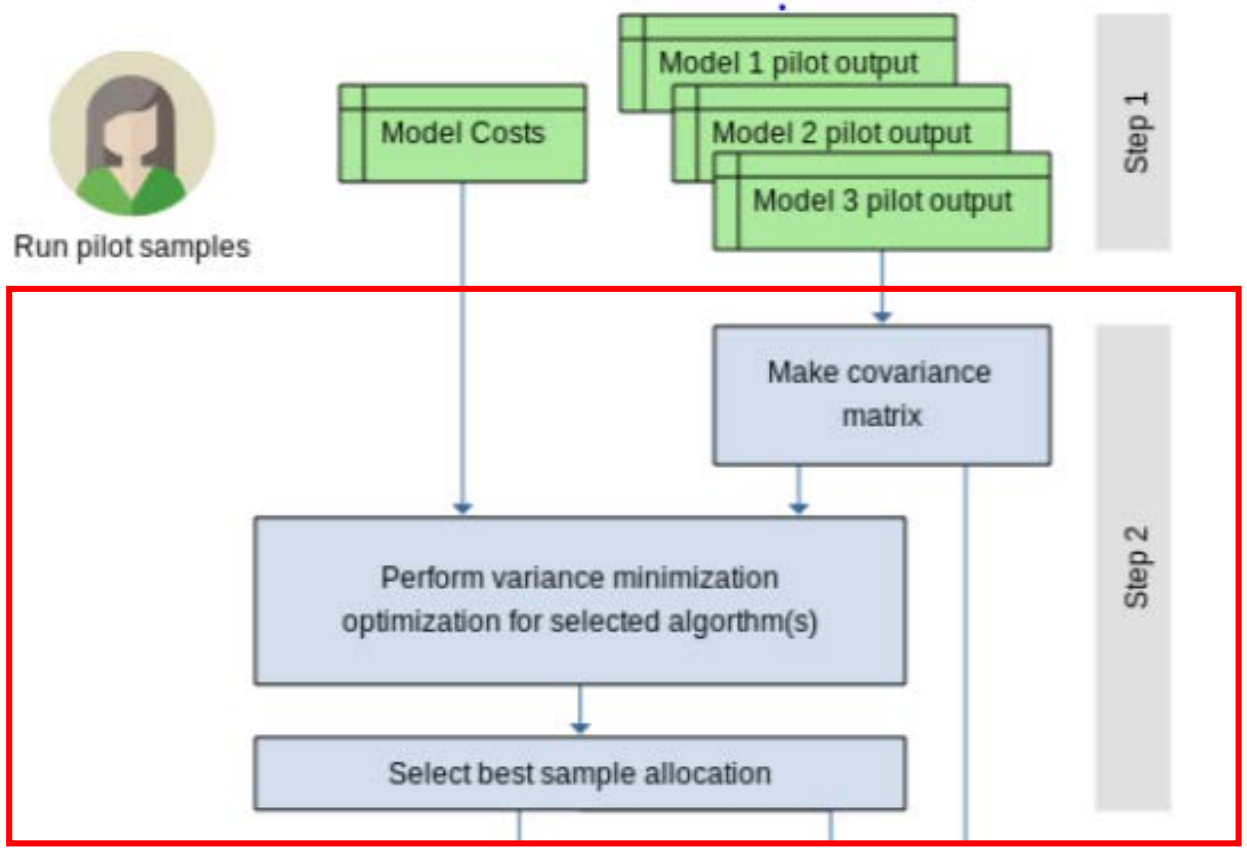
MXMC Step 1: generate model costs/outputs

- Generate input samples and corresponding output samples/model costs
- Input samples generated with Python wrapper for POST2's input sample generator
- Output samples generated by feeding the input samples to a Python wrapper for the POST2 output generator
- Model costs generated by finding the average runtime for each POST2 model
- POST2-driven step



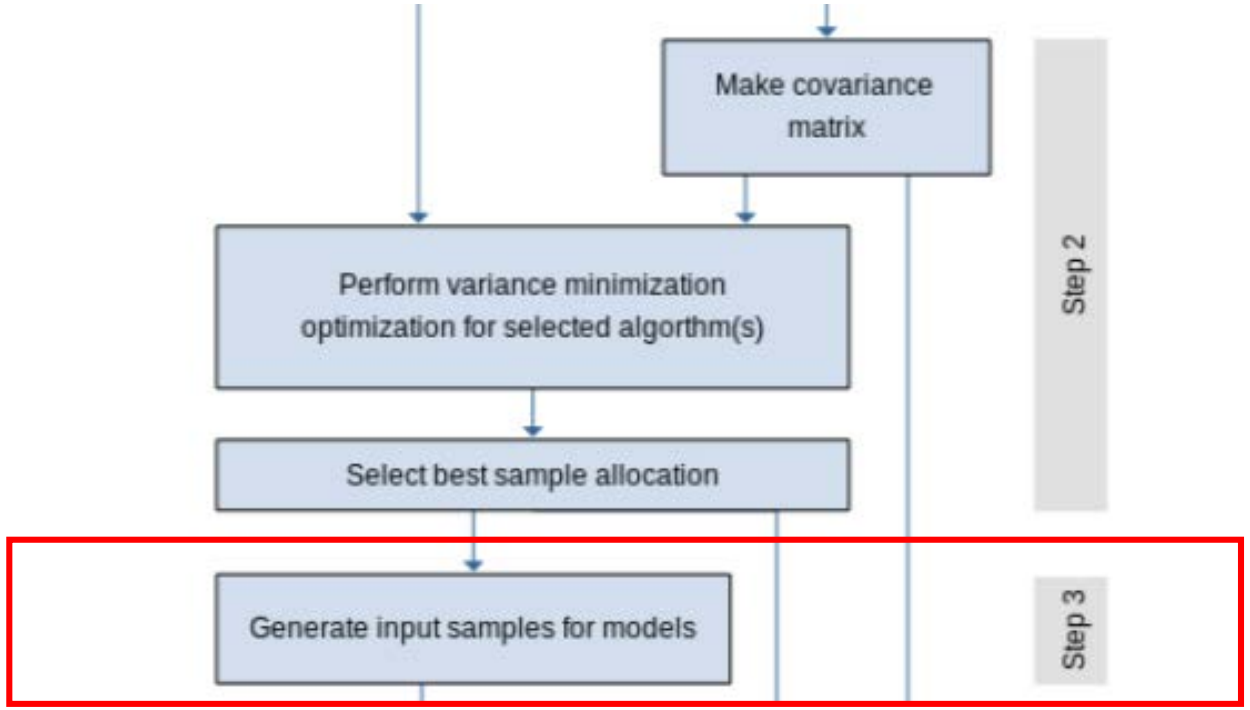
MXMC Step 2: generate cov matrix

- Generate covariance matrix from the outputs generated in step 1, find optimal sample allocation for the user selected algorithms by minimizing variance.
- MXMC-driven



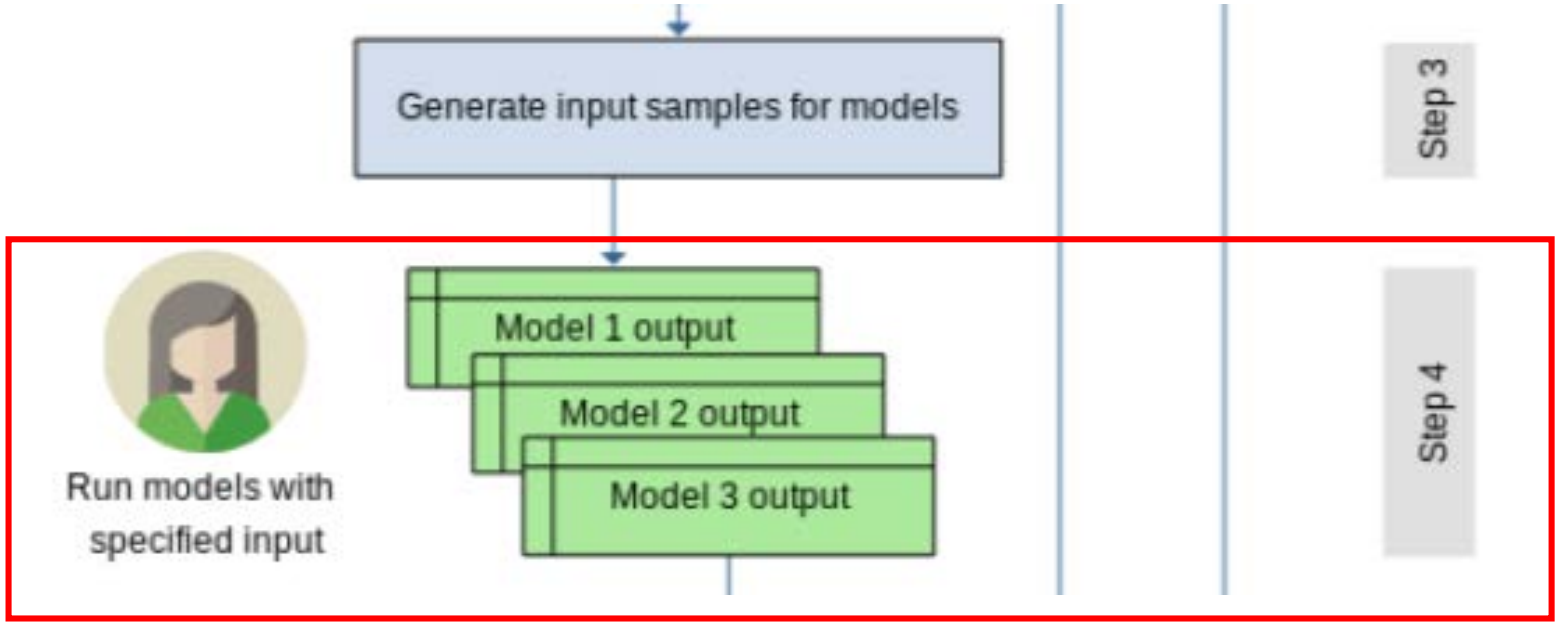
MXMC Step 3: generate input samples

- Generate input samples for models based on the sample allocation calculated by MXMC in step 2
- For POST2, the input samples will be the same regardless of model fidelity, so we can just find the total number of input samples required.
- For example: for sample allocation [1, 10, 50], generate 1+10+50=61 POST2 input samples
- POST2-driven



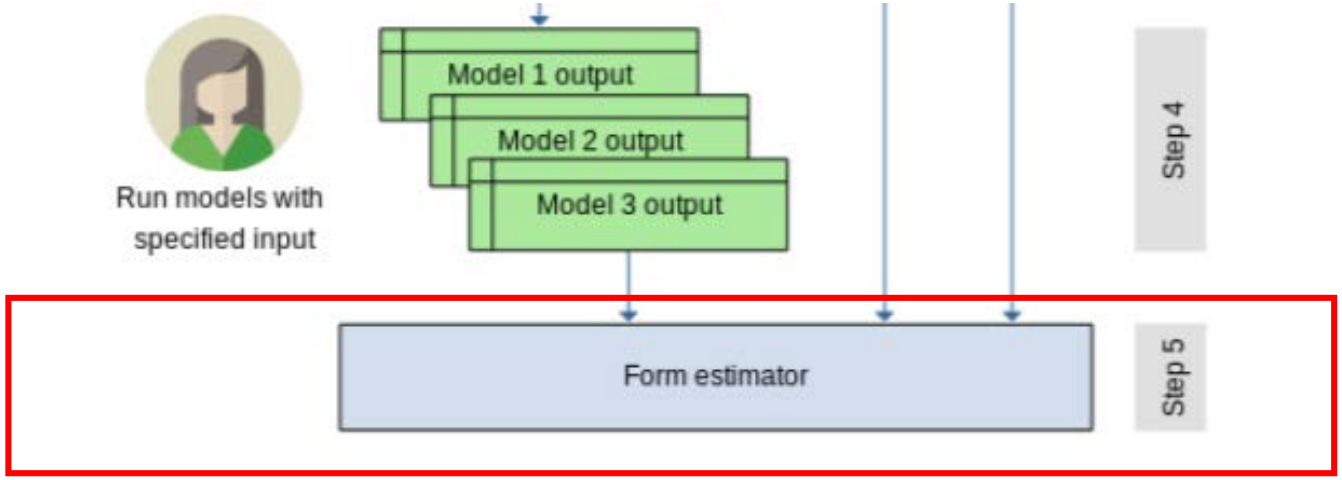
MXMC Step 4: generate output samples

- Run models with input samples specified in step 3 to generate output samples
- POST2-driven



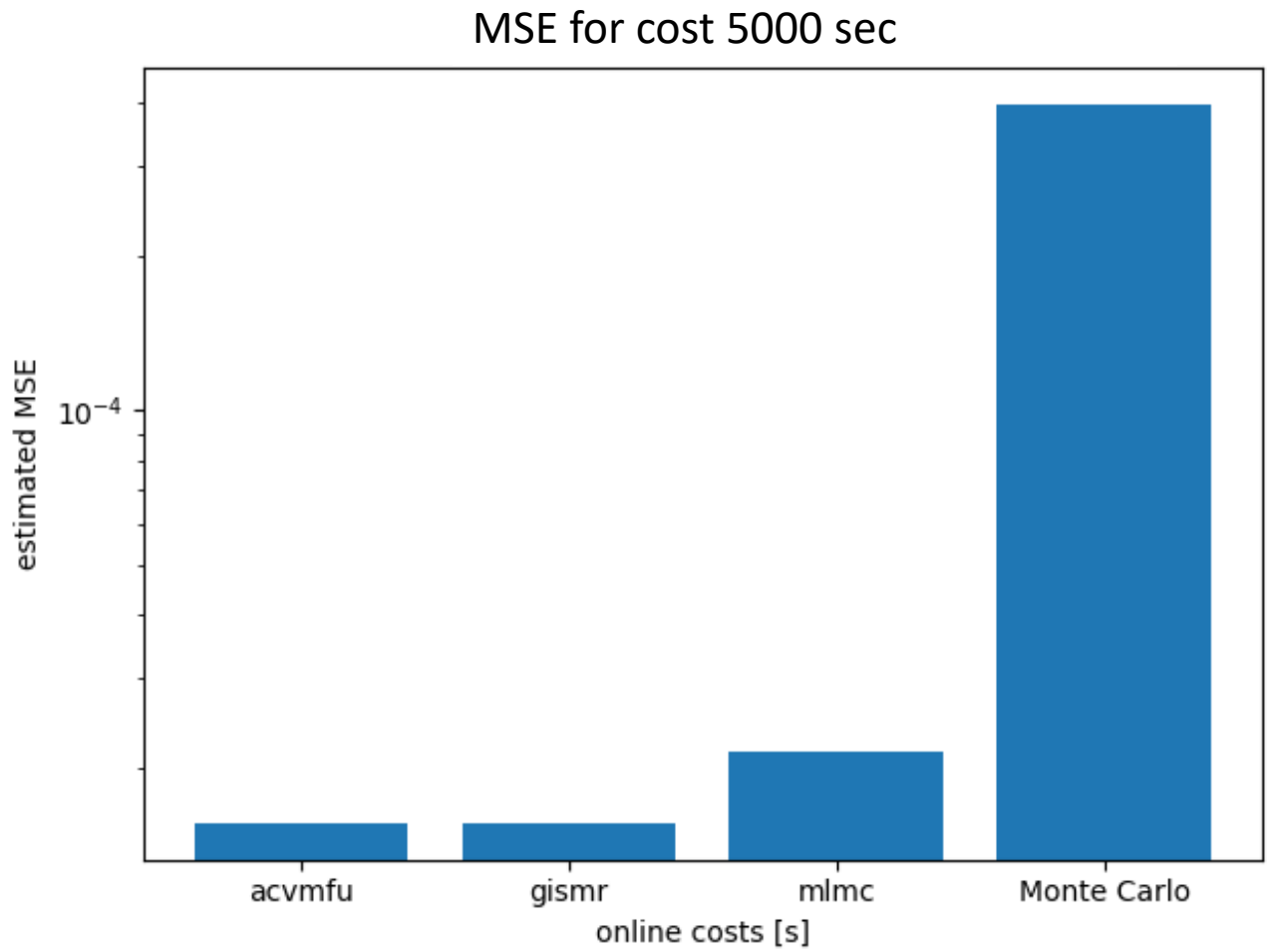
MXMC Step 5: form estimator

- Form estimator from outputs generated in step 4
- MXMC-driven



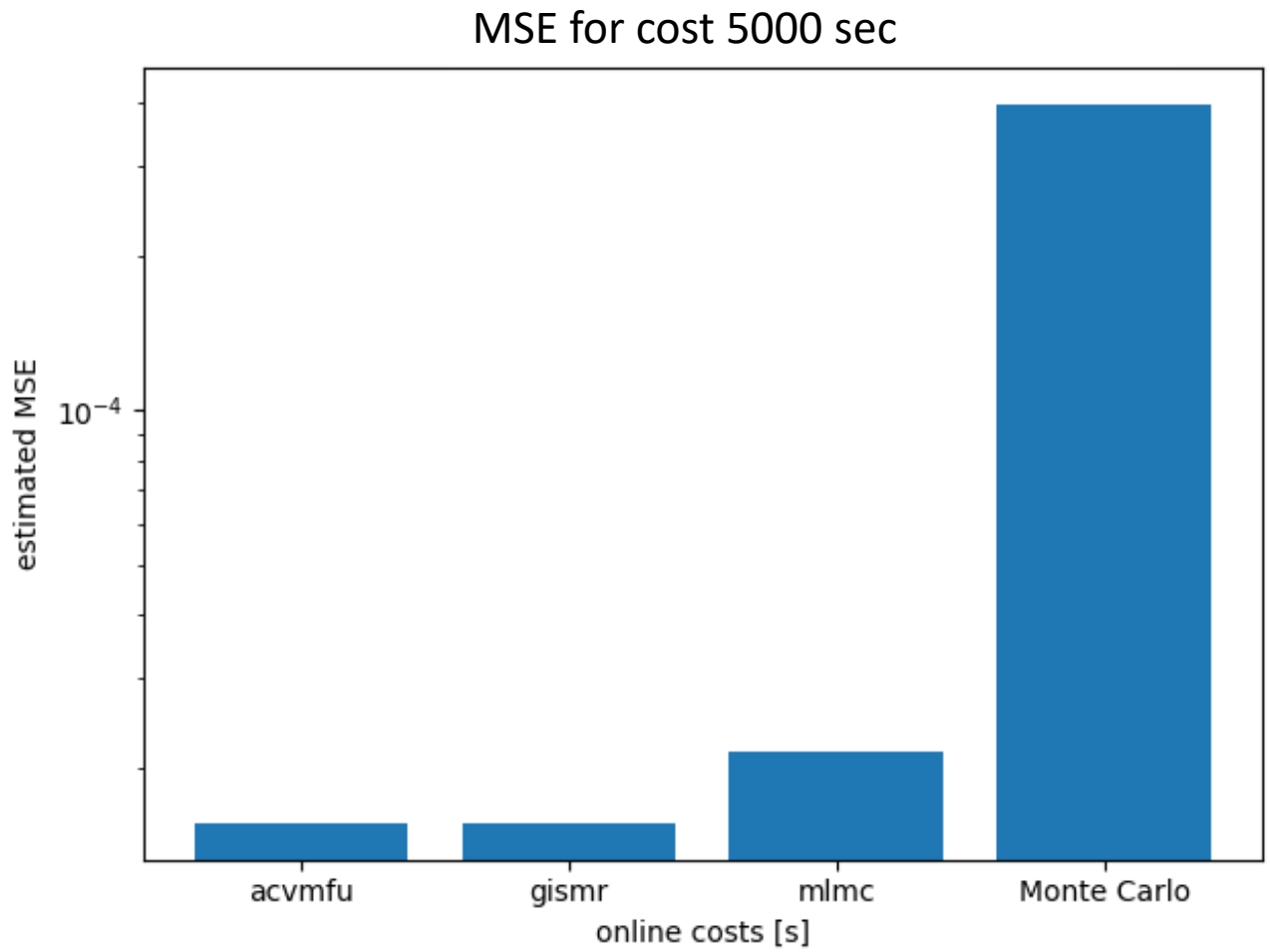
Mean Squared Error (MSE) for runtime of 5000 seconds

- MC POST2 timestep: 0.001 seconds
- MXMCPy methods use timesteps (0.1, 0.01, 0.001)
- MXMCPy methods tested: ACVMFU, GISMR, MLMC
- Prediction: landing location



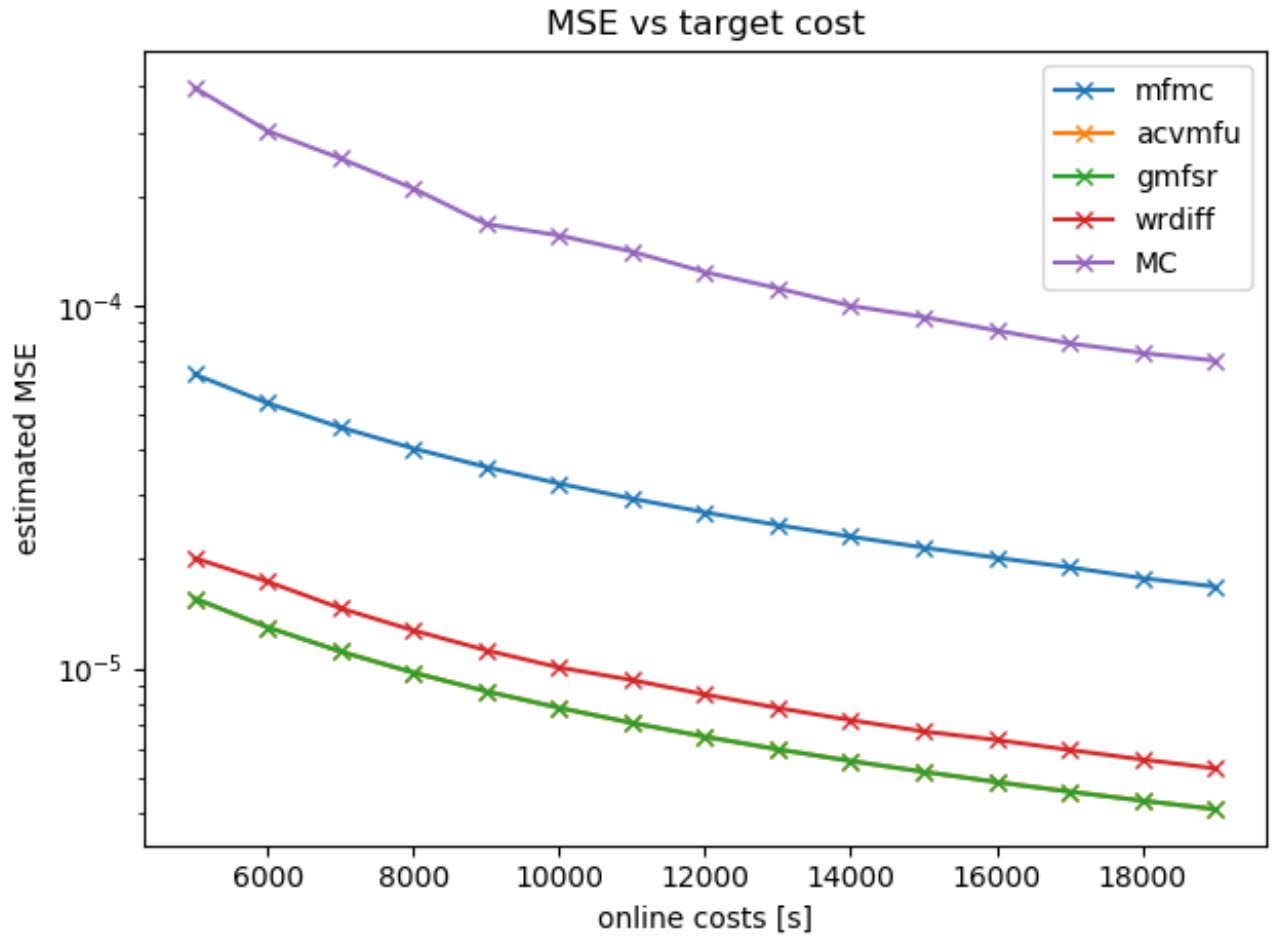
MSE for cost 5000 sec

- Conclusion: at cost 5000 sec, all 3 tested MXMCPy methods provide a lower error estimate than traditional Monte Carlo.



MSE vs target cost for target costs from 5000-20000 sec

- Conclusion: at costs 5000-20000 sec, all 4 tested MXMCPy methods provide a lower error estimate than traditional Monte Carlo.



Accomplishments

- All 5 steps of MXMCPy were automated with Python/Bash programming

Learned About

- Learned how to apply Test-Driven Development (TDD) framework: writing computer-automated tests for small units of code
- Uncertainty Quantification (UQ)

Conclusion

- Each MXMCPy step was automated with Python code.
- MXMCPy methods can provide much lower variance at the same target cost than traditional Monte Carlo.

- MXMCPy performs fast and accurate UQ by combining predictions from multiple models of varying speed/accuracy

References

- https://www.nasa.gov/mission_pages/msl/multimedia/pia16039.html