Design of an Application Programming Interface for the Program to Optimize Simulated Trajectories II

R. Anthony Williams[1] and Rafael A. Lugo[2]
NASA Langley Research Center, Hampton, Virginia, 23681, USA

James A. Hoffman[3]
Analytical Mechanics Associates, Hampton, Virginia, 23666, USA

A significant effort to upgrade the Program to Optimize Simulated Trajectories II (POST2), a heritage flight mechanics tool developed at NASA Langley Research Center, is ongoing to support current and future NASA missions. To meet mission requirements, it may be necessary for multiple specialized computational tools to interact to properly assess a system. An application programming interface for POST2 was developed to allow easier access for users and to enable communication between external applications. A demonstration of the POST2 application programming interface is presented by utilizing common engineering platforms such as MATLAB and Python.

I. Introduction

The premier flight mechanics trajectory simulation tool at the NASA Langley Research Center (LaRC) is the Program to Optimize Simulated Trajectories II (POST2). Due to its flight validated heritage, POST2 has become a widely used tool for atmospheric ascent and entry, descent, and landing (EDL) modeling and simulation [1-11]. POST2 first began its development in the 1970s as a Fortran program, and since has been maintained and updated at LaRC to the C programming language with some C++ elements. A significant recent improvement to POST2 enabled thread safety and added the capability to calculate optimization solutions in parallel [12].

As spacecraft and trajectories become more complex, the software and models used to assess the associated requirements need to be upgraded to sustain accurate and efficient performance. These models and programs are typically accessed by some type of user interface. For example, many tools have a command line interface (CLI) or graphical user interface (GUI). Currently, POST2 is executed with user provided options, commonly referred to as an input deck, via a CLI. However, to meet mission requirements it is often desired to have multiple applications interact with each other, allowing the strengths of each software to be leveraged. An application programming interface (API) provides a means for software to be stitched together and pass information back and forth [13].

The POST2 API began development to enable a more flexible experience for users, but also to allow for external applications to interact with POST2 in ways that were not possible before. The foundation of the API was completed during the thread safety update, and these changes were augmented to provide methods for users or other software to interface with POST2.

This work details significant elements of the POST2 API. Since POST2 is written primarily in C, the API is also written in C, which permits greater interoperability with external applications. One major consideration when designing the API was the inefficiencies due to file input and output (I/O). The API still relies on processing an initial input file processing, then afterwards requires no additional file I/O unless requested by the user. Currently, the API will not write or construct a POST2 input file for the user. Additionally, uniformity in function naming and argument

[1] Research Computer Scientist, Atmospheric Flight and Entry Systems Branch.
[2] Aerospace Engineer, Atmospheric Flight and Entry Systems Branch, AIAA Senior Member.
[3] Software Development Senior Manager, Atmospheric Flight and Entry Systems Branch.

structure was an important design feature (all POST2 API methods begin with `post2_`, and the arguments are all primitive C datatypes for simplicity and ease of interoperability). Lastly, where appropriate to return an error status, each method in the POST2 API will return one of two error codes: `POST2_OK` or `POST2_ERROR`.

A discussion of the hierarchy that a POST2 API user has control of will be discussed in Section II. Sections III and IV will outline the different aspects of user-controlled settings and data output. The levels of parallelism that are permitted and some example use cases are described in Section V. Specific use cases demonstrating some of the enabled capabilities in the API are then shown in Section VI.

II. User Controlled Hierarchy

The memory structure in POST2 is broken up into a hierarchy detailed in [12]. Following this framework, the user has control over two levels of objects within the API: a *workspace* and *trajectory*, with the trajectory being contained by a workspace as depicted in Fig. 1. A workspace is defined by the settings in a POST2 input deck, which is the fundamental set of information that determines how the simulation will execute. Since POST2 is a very general and modular flight mechanics software, input decks can vary widely in their complexity and structure. The input deck will contain information about which models to use, e.g., atmosphere, gravity, aerodynamics, and will detail how the simulation will evolve in time. POST2 is an event-based simulation, where an event during EDL might be main parachute deploy, so the event structure is defined in the input deck.

POST2 events are input deck constructs that control the order of input execution during trajectory propagation. The events in a simulation will typically occur in a sequence (e.g., atmospheric entry interface, drogue parachute deploy, main parachute deploy, touchdown), which permits the user to update simulation parameters. Events can be optional, occur out of numerical sequence (i.e., a roving event), and can be triggered using multiple criteria. For example, atmospheric entry interface can be triggered by the vehicle(s) reaching a specific geodetic altitude, at which point the atmospheric model can be activated. However, a parachute might be deployed by a barometric sensor measuring a specific pressure level, so the drogue parachute deploy event would be triggered based on the vehicle(s) experiencing that dynamic pressure level. The parachute inflation model can then be activated, as well as all the interactions the parachute has with the vehicle such as drag forces.

In any given POST2 API instance, there can be more than one workspace, where each is defined by its own input deck. For example, a simulation might be broken up into a deorbit and then a descent and landing segment. Each of these segments can be defined by different input decks and each is assigned its own unique workspace, described by a workspace key (an integer). The function to create and initialize a workspace based on an input deck is `post2_init()`, where the workspace key and the input deck path are arguments. This will allocate and set the value of all workspace data associated with the problem defined by the passed in input deck. The number of workspaces that can exist within a single POST2 API instance is only limited by the amount of available memory on the machine the API is executing on. When all work within the workspace is completed, the user may free the memory associated with that workspace by utilizing the `post2_free_workspace()` method.

Within each workspace there exists at least one trajectory; however, there can be numerous trajectories in each workspace. The trajectory object contains the bulk of the data that is needed for the simulation, including any vehicle(s). Similar to initializing a new workspace, the function to create and initialize a new trajectory is `post2_init_trajectory()`. Given that a trajectory is contained within a workspace, the arguments to the initialization function are the associated workspace key and trajectory key, where the trajectory key (an integer) describes the trajectory. Each trajectory within a workspace will follow the same simulation code path, and their data is assigned according to the input deck that defines the workspace. Methods of how to modify the data for each workspace and trajectory are described in Section III. When all work within a trajectory is completed, the user may free the memory associated with that trajectory by calling the `post2_free_trajectory()` function. If a workspace was freed already via a call to `post2_free_workspace()`, then all trajectories within that workspace are also freed for the user.

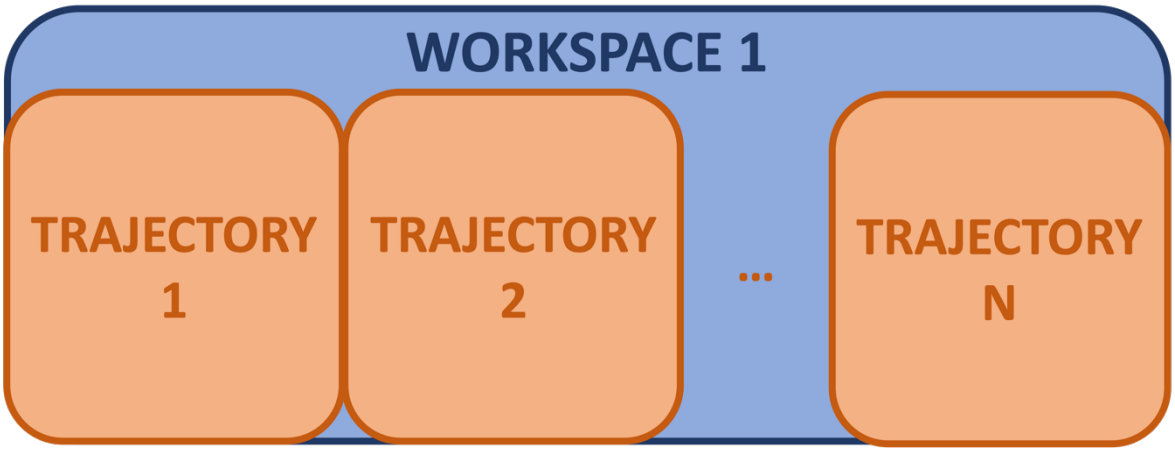

Fig. 1 A depiction of the hierarchy of objects that a user has control of in the POST2 API.

III. User Controlled Simulation Settings

The hierarchy described in Section II shows the two levels of API objects that a user can create, a workspace and a trajectory, and how these objects relate to each other. The settings that correspond to workspaces and trajectories are divided in a similar manner, and they are referred to as *inputs* and *values*, respectively, as is shown in Fig. 2. While the methods of how to modify inputs and values appear to be similar, their behavior is quite different.

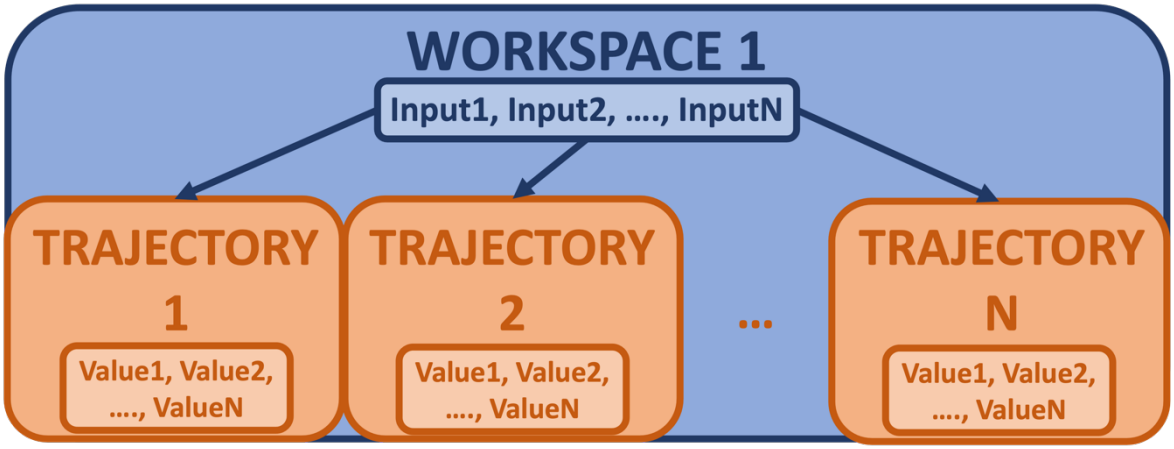

Fig. 2 A brief display of the settings that a user has control of in the POST2 API, and which level of the hierarchy they correspond to.

A. Workspace Inputs

Setting a workspace input is akin to opening a POST2 input deck in a text editor and modifying it directly. However, usage of the API requires no file input or output (I/O) outside of the initial processing of the input deck. Instead of modifying a file and rereading it, the API allows for a user to update the POST2 memory directly and programmatically, enabling the alteration of POST2 data from external applications in an efficient manner. The methods of setting inputs for a given workspace take the form of `post2_set_*_input()`, where the wildcard character, `*`, is replaced by the type of input to be set. The user has the ability to modify scalar inputs, specific indices of an array input (one or two dimensional), string inputs, and specific characteristics of look-up tables (y-values, multipliers, and biases).

An example of a scalar workspace input that a user may want to modify could be the value of a variable where a specific event is triggered. For instance, for an EDL mission at Earth that requires a parachute, it may be of interest to analyze the effect of main parachute deploy altitude on the landing location of the entry vehicle, so the user may want to change the specific altitude where the main parachute deploys. Elements of an initial position vector may be modified by setting an array input value. String inputs are used for multiple purposes, such as the variable to examine

to determine whether an event has been triggered. Instead of triggering main parachute deploy at a specific altitude, the user can designate to trigger main parachute deploy based on Mach number using a string input, and then declare the Mach number using a scalar input. The inputs set can work with each other, like in the previous Mach number example; however, if the same variable is set multiple times, the most recent input will be used.

Just as trajectories are contained within a workspace, all trajectories will be affected by any of the inputs set for a given workspace.

B. Trajectory Values

While workspace inputs affect all trajectories within a workspace, trajectory values are targeted to affect specific trajectories. Multiple trajectories may be initialized, via `post2_init_trajectory()`, and simulated within the API, by utilizing the `post2_run()` method. The `post2_run()` method executes a single trajectory, defined by the trajectory key argument, from a specific workspace, defined by the workspace key argument. This trajectory will follow the exact settings laid out in the input deck and any workspace inputs, as well as the trajectory values set. For any trajectory value set that conflicts with a workspace input, the trajectory value will have precedence and will be used.

The methods of setting trajectory values take the form `post2_set_*_value()`, where the wildcard character, `*`, is replaced by the type of value to be set. Trajectory variables that can be modified are scalar values, specific indices of an array value (one or two dimensional), and string values. Table data is not able to be modified at the trajectory level as tables are readable by all trajectories, hence table data can only be modified via a workspace input.

Modifying trajectory values is especially useful for situations where specific parameters might want to be varied but the majority of settings for the simulation should remain the same. An example of this would be a parametric sweep where only a single parameter is changed between trajectories, such as the entry flight path angle (EFPA), and the effect this variation has on the peak heating of an entry vehicle. Also, a sensitivity study or Monte Carlo analysis could be employed by using trajectory values where each Monte Carlo sample is a single trajectory with only the dispersed parameter modified. Additionally, investigating the effect that multiple different guidance schemes have on the performance of a system could be done via trajectory values.

IV. User Controlled Data Output

Prior to the API development, the process of getting output data from the POST2 simulation was in the form of parsing an output file (either ASCII text or MATLAB file format). However, as described previously, the POST2 API was designed with limited file I/O in mind. Therefore, methods were defined to allow a user to access POST2 data directly from memory without writing to an output file.

Within the API, there is no default set of data to collect for a given execution of a trajectory within a workspace. To instruct the API to collect specific data, a call to the `post2_set_output_variable()` function must be made for each variable that is desired by passing the name of the variable as the argument. The requested data is associated with a workspace, so all trajectories within a workspace will log the same variables.

The default behavior of logging data is to capture the value of any requested variables at each time step throughout the entire simulation, which utilizes the most memory of the output options. There are three additional options available that modify how often data is logged. A call to the `post2_set_output_at_intervals()` will capture the time history of all variables at a specified interval that is a multiple of the simulation time step. The `post2_set_output_at_events()` method will instruct the API to only log data at the defined POST2 events in the input deck. This decreases the amount of memory needed because there are usually far fewer events than there are time steps in a simulation. The last option for modifying output cadence is the `post2_set_output_at_final_state()`, which is the least memory-intensive option since it will only capture data at the simulation's final state. It is important to note that this is the simulation's final state, which is not guaranteed to be the end of the designed trajectory, but it could be when an error was encountered during the trajectory propagation.

Once the trajectory propagation has completed, the user has many options to inspect simulation information and any data that was logged. The number of time steps and events that occurred can be accessed via calls to `post2_get_number_of_timesteps()` and `post2_get_number_of_events()`, respectively. The methods of getting logged data from a trajectory within a workspace take the form of `post2_get_value_*()`, where the wildcard character, `*`, is replaced by the type of output to be gathered. A user can request a value at a specific time step that occurred, e.g., the initial or final time step. Data can be gathered at a specific POST2 event, referred to either by event number or event name. Lastly, the entire time history of a requested variable can be returned

as a reference to a memory location. For this particular method, it is important to use the `post2_free_time_history()` method when the data is no longer needed for analysis.

A brief example usage of the API is shown in the Appendix. More complex examples that utilize different frameworks is included in the documentation that is provided as part of the POST2 distribution.

V. Parallelism with the API

The work completed as part of the thread safety improvement ensured thread safe execution of the POST2 trajectories within a simulation. Leveraging this foundation, the additional updates were implemented to allow for thread safe execution of API workspaces. Thread safe execution of workspaces and trajectories allows for multiple layers of concurrent calculations within the API. The first layer of parallelism corresponds to running multiple workspaces in parallel. Utilizing this capability, and with the proper simulation set up, a user may split their overall simulation structure into multiple segments or phases of flight. Fig. 3 depicts an example simulation that has two distinct segments being simulated in parallel: ascent and booster return. An additional potential use case of parallel workspaces could be the use of a trajectory collocation optimization scheme, provided the simulation could be divided into segments where each trajectory is a collocation node.

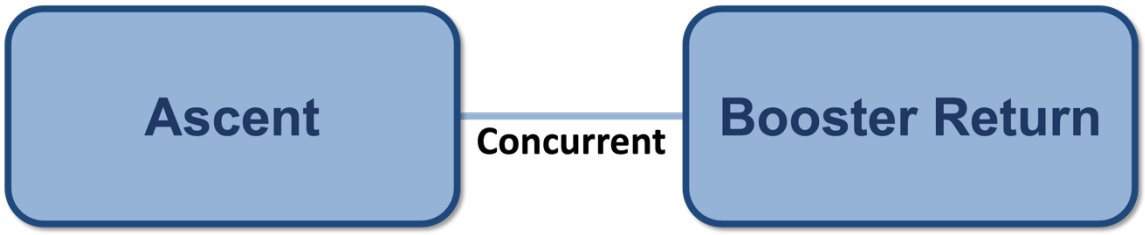

Fig. 3 Example simulation with concurrent workspaces.

At the second level of the API hierarchy, a user has the ability to simulate trajectories in parallel. This level of parallelism currently can be exploited for more efficient optimization solutions with POST2, where each trajectory simulation represents a perturbation of a control variable. Another use case for parallel trajectories is a side-by-side comparison of models or integration methods. An analysis of the effect of varying the fidelity of flight software, such as sensor models, could be completed more efficiently in parallel. Alternatively, the sensitivity to integration method, a two step Runge-Kutta versus a four step Runge-Kutta for example, can be analyzed simultaneously. Additionally, a parametric sweep can be simulated concurrently, where EFPA is the modified parameter for example, as shown in Fig. 4. An area of interest for EDL is uncertainty quantification (UQ), which is a rapidly changing field of study. The standard practice for UQ is to utilize Monte Carlo Simulations, which results in many samples that are independent of one another. These samples, or trajectories in the case for POST2, can be simulated in parallel by leveraging the API.

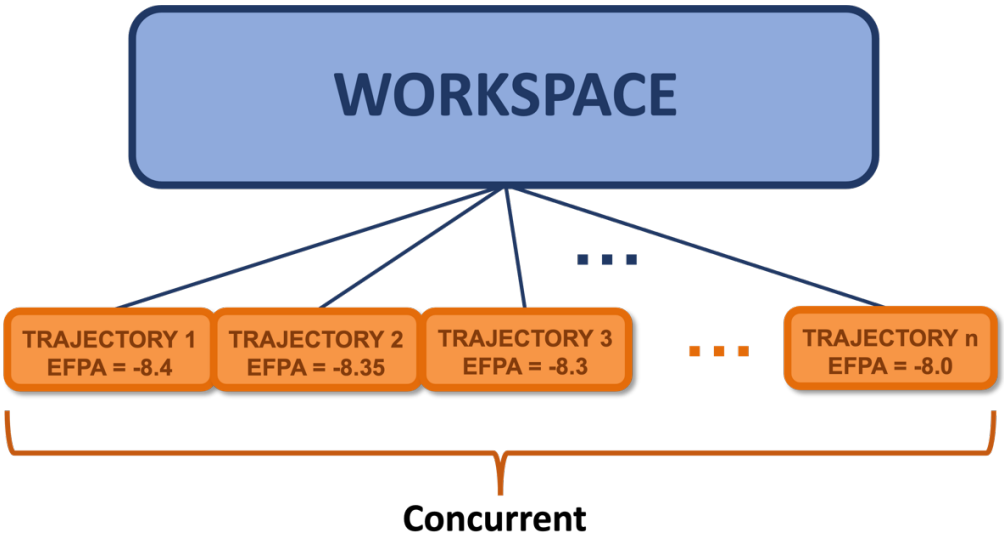

Fig. 4 Simple depiction of multiple trajectories running concurrently within a workspace, each with a different value for entry flight path angle.

A combined third level of parallelism is permitted in the API, where there are concurrent workspaces that contain concurrent trajectories as depicted in Fig. 5. Ultimately, the user chooses which levels of parallelism might serve their use case best.

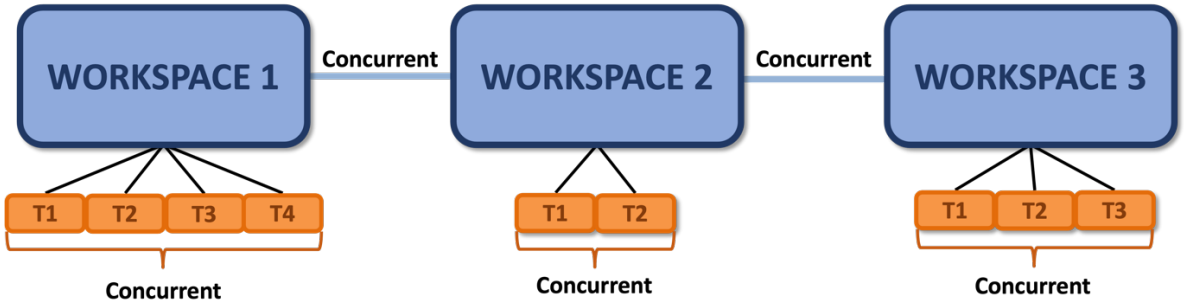

Fig. 5 The combined multi-level parallelism that is permitted in the POST2 API. All workspaces operate concurrently, and all trajectories in a given workspace are simulated concurrently.

VI. Example use cases of the API

The POST2 API enables many use cases that are not detailed in this work, and the examples shown in this section will not be an exhaustive list. Since the API is written in C, many languages and frameworks have methods to interface with external libraries, so the potential options for interoperability are extensive. In this work, two common platforms used in engineering applications calling the POST2 API will be highlighted: Python and MATLAB. As part of the API development, classes and methods were designed in Python and MATLAB to allow for more seamless utilization of POST2.

A common use case of the API might be a design space exploration or a parametric sweep. For this case, a Python framework that leverages the C Foreign Function Interface (CFFI) module [14] to call the POST2 API was utilized. The class defined for the developed framework has methods that closely resemble the direct API calls. Since the API is executed through a module created by CFFI, all data pertaining to the API instance resides within the same Python process. Thus, features such as data visualization can be directly leveraged within Python, through the use of common third-party packages such as matplotlib, as depicted in Fig. 6. Each curve represents an independent trajectory that only differs from its counterparts by the entry flight path angle. Additionally, parallelism through existing Python modules can be leveraged to run this problem, since the trajectories are independent from each other. When executing the same set of simulations in parallel, a speedup of approximately 5X was achieved.

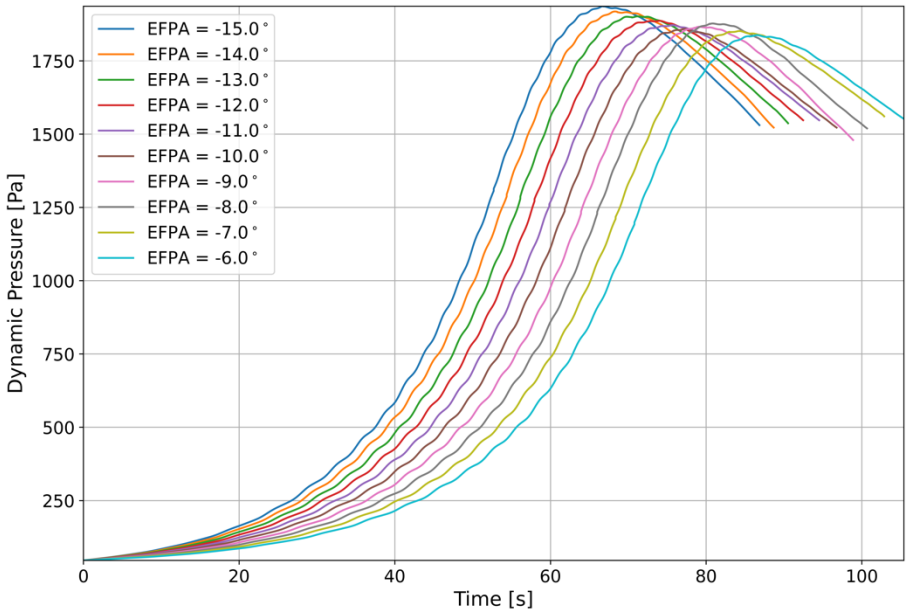

Fig. 6 A parametric sweep, completed using the POST2 API in Python, demonstrating the effect of entry flight path angle on dynamic pressure.

A key POST2 capability is trajectory optimization. While POST2 includes robust optimizers, there is no universal optimization algorithm for all situations. Therefore, it may be of interest to use off-the-shelf optimizers from external frameworks like Python or MATLAB. Such external frameworks provide a much larger set of algorithms than POST2, and the trajectory being optimized might be better suited to a different optimization approach than what is available in POST2.

A class with methods that closely resemble the API was defined to make the interface between MATLAB and POST2 more seamless. While many frameworks offer optimizers, this work will focus on those optimization algorithms within MATLAB. An additional class was created to allow for easier use of these optimization algorithms, and the algorithm that is leveraged here is MATLAB's `fmincon`. A targeting problem for a powered descent segment was solved using engine throttle, time of flight, and flight path angle as controls, and vertical velocity, horizontal velocity, and geodetic altitude as constraints. The visualization capability was also leveraged to provide real-time feedback while the optimizer was running, which can alert the user if the optimization problem is ill-posed or if the optimizer appears to diverge. Additionally, post-convergence feedback can be gathered to determine how well constraints were met and how saturated controls may have been. This information can be obtained from the POST2 native output file; however, it is not visualized or as easily readable as what is shown in Fig. 7.

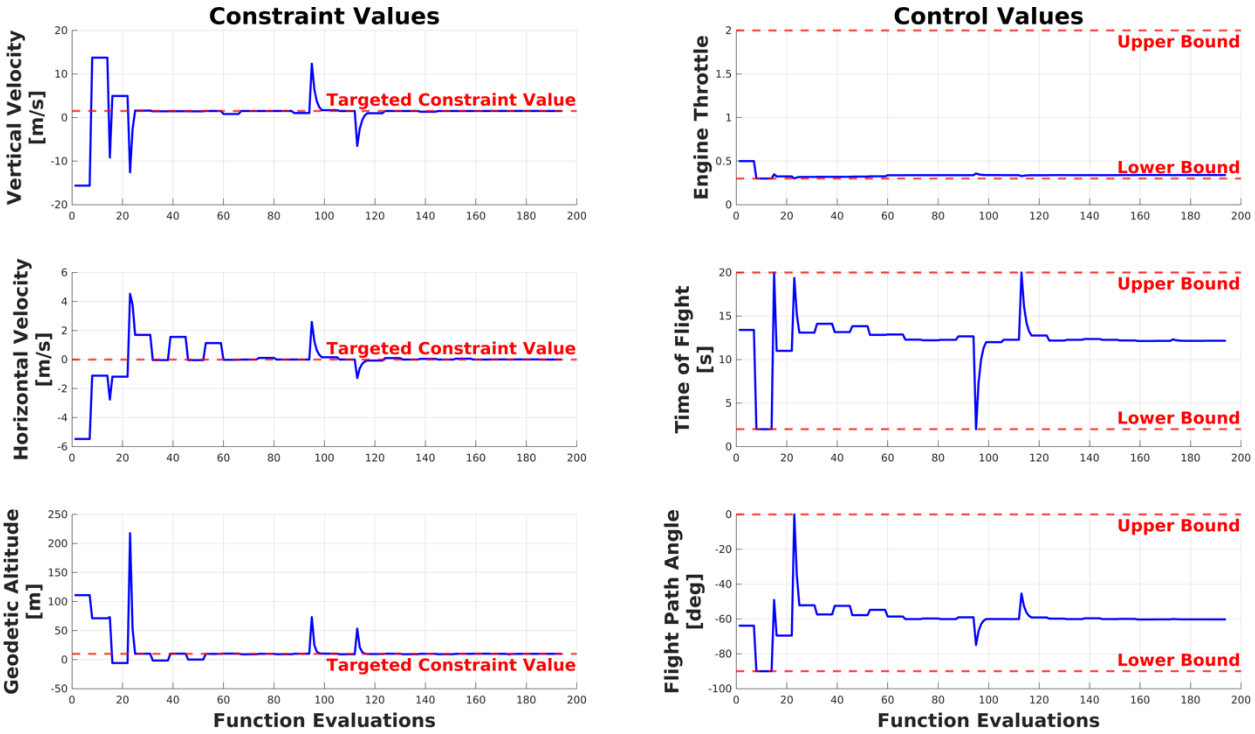

Fig. 7 Output after a successful convergence to a targeted solution from MATLAB's `fmincon` optimizer.

VII. Summary

An application programming interface was developed for a widely used flight mechanics tool to allow for interfacing with external applications, which followed a recent upgrade to allow for parallel trajectory simulations. This API for POST2 was developed with very limited I/O in mind outside of the initial input deck processing, to maintain efficient execution. Currently, the API is reliant on the input deck for the overall simulation structure, and it will not create an input deck for the user. One main reason why an API is useful is to allow coupling of multiple tools to solve a complex problem, where the strengths of each tool can be leveraged accordingly. All of the methods within the API use primitive C data types to allow for simpler interoperability with external applications. Additionally, a user might have the most productive workflow within a given framework or language, and the API allows them the option to remain within this framework while leveraging POST2.

The objects that a user has control over, a workspace and trajectory, and the methods associated with their instantiation and freeing were discussed. Functionality to enable a user to modify POST2 memory directly, either a workspace input or a trajectory value, leading to efficient processing through bypassing file I/O was also discussed. While nearly all POST2 supported data types can be altered via API calls, there are additional capabilities needed to support the full set of data. Output of data is another crucial element in the API design. Which data and at what cadence the data are logged can be controlled by the user. This allows the user to limit the memory consumption of the API by selecting to record data at their desired rate.

Additional modifications were made to POST2 enabling parallel workspace simulations based on previous upgrades that enable parallel trajectory simulations. Both levels of parallelism can be exploited via the API, as well as a combination of the two. A demonstration of parallel computation while utilizing Python was showcased in an example of a parametric sweep. The parallel execution of all the independent trajectories lead to a computation time approximately five times faster than the sequential execution. Additionally, the visualization packages available in Python allow for more immediate feedback of analysis.

Another showcase of utilizing the API from an external application was displayed in MATLAB. The `fmincon` optimizer was used to solve a targeting problem, which involved directly modifying POST2 memory to alter control values to ensure constraints are met. Real-time and post-convergence feedback of the optimizer's process is more readily visualized within the MATLAB framework, making for useful information about how well constraints are met or how saturated controls may be. The use of external optimizers can be powerful to ensure state of the art algorithms are employed for the most optimal and efficient solutions.

Appendix

A simple example demonstrating how a user might leverage the POST2 API in C is provided below as reference. Additional detail is provided in the documentation that is included with the POST2 distribution.

```
// Initializing Workspace 1 based on information in input deck
post2_init(1, "/path/to/input/deck");

// Initializing Trajectory 1 within Workspace 1
post2_init_trajectory(1, 1);

// Requesting Workspace 1 to track geodetic altitude
post2_set_output_variable(1, "gdalt");

// Requesting Workspace 1 to track planet-relative velocity magnitude
post2_set_output_variable(1, "velr");

// Executing Trajectory 1 in Workspace 1
post2_run(1, 1);

// Getting time history of geodetic altitude from Trajectory 1 in Workspace 1
double * gdalt = post2_get_value_time_history(1, 1, "gdalt");

// Getting value of planet-relative velocity magnitude at event 100 from
// Trajectory 1 in Workspace 1
double velr_100 = post2_get_value_at_event(1, 1, "velr", 100);

// Getting value of planet-relative velocity magnitude at event named "PDI"
// from Trajectory 1 in Workspace 1
double velr_pdi = post2_get_value_at_event_name(1, 1, "velr", "PDI");

// Getting final state value of geodetic altitude from Trajectory 1 in
// Workspace 1
double gdalt_final = post2_get_value_at_final_timestep(1, 1, "gdalt");

// … do analysis with gathered data

// Cleanup
// Freeing time history of geodetic altitude gathered earlier
post2_free_time_history(gdalt);

// Freeing all memory associated with Workspace 1, including Trajectory 1
post2_free_workspace(1);
```


Acknowledgments

This project was supported by Entry Systems Modeling, which is a program under NASA's STMD.

References

[1] R. D. Braun, R. W. Powell, W. C. Engelund, P. A. Gnoffo, K. J. Weilmuenster and R. A. Mitcheltree, "Mars Pathfinder Six-Degree-of-Freedom Entry Analysis," *Journal of Spacecraft and Rockets,* vol. 32, no. 6, pp. 993-1000, 1995.

[2] P. N. Desai, J. L. Prince, E. M. Queen, M. Schoenenberger, J. R. Cruz and M. R. Grover, "Entry, Descent, and Landing Performance of the Mars Phoenix Lander," *Journal of Spacecraft and Rockets,* vol. 48, no. 5, pp. 798-808, 2011.

[3] P. N. Desai, M. Schoenenberger and F. M. Cheatwood, "Mars Exploration Rover Six-Degree-of-Freedom Entry Trajectory Analysis," *Journal of Spacecraft and Rockets,* vol. 43, no. 5, pp. 1019-1025, 2006.

[4] R. A. Lugo, A. M. Dwyer-Cianciolo, S. Dutta, R. A. Williams, A. Pensado and P. Chen, "Integrated Precision Landing Performance Results for a Human-Scale Mars Landing System," in *AIAA SciTech Forum*, San Diego, CA, 2022.

[5] R. A. Lugo, A. M. Dwyer-Cianciolo, S. Dutta, R. A. Williams, J. S. Green, P. Chen and S. D'Souza, "Precision Landing Performance and Technology Assessments of a Human-Scale Lunar Lander Using a Generalized Simulation Framework," in *AIAA SciTech Forum*, San Diego, CA, 2022.

[6] D. W. Way, J. L. Davis and J. D. Shidner, "Assessment of the Mars Science Laboratory Entry, Descent, and Landing Simulation," in *23rd AAS/AIAA Space Flight Mechanics Meeting*, Kauai, HI, 2013.

[7] D. W. Way, S. Dutta, C. Zumwalt and D. Blette, "Assessment of the Mars 2020 Entry, Descent, and Landing Simulation," in *AIAA SciTech Forum*, San Diego, CA, 2022.

[8] R. W. Maddock, A. M. Dwyer-Cianciolo, A. M. Korzun, D. K. Litton, C. H. Zumwalt and C. D. Karlgaard, "InSight Entry, Descent, and Landing Postflight Performance Assessment," *Journal of Spacecraft and Rockets,* vol. 58, no. 5, pp. 1530-1537, 2021.

[9] S. A. Striepe, D. W. Way, A. M. Dwyer and J. Balaram, "Mars Science Laboratory Simulations for Entry, Descent, and Landing," *Journal of Spacecraft and Rockets,* vol. 43, no. 2, pp. 311-323, 2006.

[10] S. Dutta and J. S. Green, "Flight Mechanics Modeling and Post-Flight Analysis of ADEPT SR-1," in *AIAA Aviation Conference*, Dallas, TX, 2019.

[11] S. Dutta, A. L. Bowes, S. A. Striepe, J. L. Davis, E. M. Queen, E. M. Blood and M. C. Ivanov, "Supersonic Flight Dynamics Test 1 - Post-Flight Assessment of Simulation Performance," in *AAS/AIAA Space Flight Mechanics Meeting*, Williamsburg, VA, 2015.

[12] R. A. Williams, R. A. Lugo, S. M. Marsh, J. A. Hoffman, J. D. Shidner and J. T. Aguirre, "Enabling Thread Safety and Parallelism in the Program to Optimize Simulated Trajectories II," in *AIAA SciTech Forum*, National Harbor, MD, 2023.

[13] M. Biehl, API Architecture, CreateSpace Independent Publishing Platform, 2015.

[14] A. Rigo and M. Fijalkowski, "CFFI: C Foreign Function Interface for Python," [Online]. Available: https://cffi.readthedocs.io/en/stable/.