(Preprint) AAS 15-606

A NEW ARCHITECTURE FOR EXTENDING THE CAPABILITIES OF THE COPERNICUS TRAJECTORY OPTIMIZATION PROGRAM

Jacob Williams*

This paper describes a new plugin architecture developed for the Copernicus spacecraft trajectory optimization program. Details of the software architecture design and development are described, as well as examples of how the capability can be used to extend the tool in order to expand the type of trajectory optimization problems that can be solved. The inclusion of plugins is a significant update to Copernicus, allowing user-created algorithms to be incorporated into the tool for the first time. The initial version of the new capability was released to the Copernicus user community with version 4.1 in March 2015, and additional refinements and improvements were included in the recent 4.2 release. It is proving quite useful, enabling Copernicus to solve problems that it was not able to solve before.

DRAFT

INTRODUCTION

Copernicus is a generalized spacecraft trajectory design and optimization software application.[1–3] Since the first release in 2006, Copernicus has become one of NASA's premier tools for spacecraft mission design. Development is ongoing, and a new major version release occurs every 2-3 years. Version 4.2 was released by Johnson Space Center (JSC) in July 2015. Copernicus is used extensively for design studies of manned missions at JSC,[4,5] as well as for a wide range of projects at other NASA centers. JSC makes Copernicus available free of charge to other NASA centers, government contractors, and universities, under the terms of a US government purpose license.

Since the initial release of Copernicus, the "segment" has been the basic building block for designing spacecraft missions.[1] Segments include all the optimization variables, constraints, and objective functions in the optimization problem. The user constructs a mission by defining any number of segments, which can represent different mission phases, different vehicles, or different stages of the same vehicle. Impulsive and finite burn maneuvers can be modeled with different maneuver control laws and propulsion systems. Copernicus includes a wide range of constraints for various variable types (e.g., time, mass, state, and engine parameters), as well as an "inherit" capability which is used to connect segments (e.g., by specifying that the initial time of one segment inherits the final time of another). A complicated mission can include many segments with complex interconnections. Segments can be propagated both forward and backward in time, and can be connected by inheritance, or initially discontinuous with continuity constraints imposed during the optimization problem to ensure continuity.

The purpose of the segment architecture was to enable any type of spacecraft mission to be designed, regardless of complexity. While it has proven very flexible, there are many limitations to the existing system. For example, only like variables can be inherited (e.g., a time variable can only

*Aerospace Engineer, ERC Inc. (JSC Engineering, Science, and Technology Contract), Houston, TX, 77058

directly inherit the value of another time variable). Also, only a finite set of constraints are available in the program (see Figure 1, which shows the state constraints GUI). The list of available state constraints is quite long, but there is no ability for the user to add additional constraints, and the grid-based GUI does not easily allow for anything other than constraints among like variables (for example, constraining r_p from one segment equal to r_p from another segment). The same applies for other types of constraints (time, mass, etc.).

Since Copernicus releases do not include the source code, end users have had few options for extending the tool. Only the features compiled into the tool are available to the users. Adding a new feature requires JSC to update the code (possibly including the GUI) and releasing a new version of Copernicus. While the segment approach is very flexible, a problem requiring some specialized constraint, for example, that is not present in Copernicus, is not solvable by Copernicus. Thus, a frequent request from Copernicus users has always been to have some ability to extend the tool in various ways. Examples include: (i) New types of constraints (for example, a constraint defined by a user-specified equation), (ii) An inherit defined by a user-specified equation (e.g., specifying that the x-position of one state is half-way between the x-position of two other states), (iii) Other methods for segment propagation such as user-defined guidance and control laws, (iv) Path constraints, and (v) Interfacing with another spacecraft trajectory software tool, passing information to and from Copernicus, and including its outputs into the Copernicus optimization problem.

The main motivation of the current work was an analysis need by the NASA Orion and Space Launch System (SLS) programs to produce optimized end-to-end (launch to Earth entry) trajectories.[6,7] Copernicus, primarily an in-space tool which does not include specific launch vehicle ascent models, is being used by JSC for Orion in-space trajectory design. Marshall Space Flight Center (MSFC) uses the Program to Optimize Simulated Trajectories (POST) tool for SLS ascent optimization. Having a capability for Copernicus to call an external tool (in this case, to generate an ascent trajectory, and manage the overall end-to-end optimization problem) was the genesis of the new plugin feature.

REQUIREMENTS AND APPROACH

To address the existing limitations in the program, and to enable incorporation of external algorithms and tools, it was decided that a new capability needed to be added to Copernicus. Rather than modifying or replacing the existing segment architecture, a new "plugin" mission attribute would be

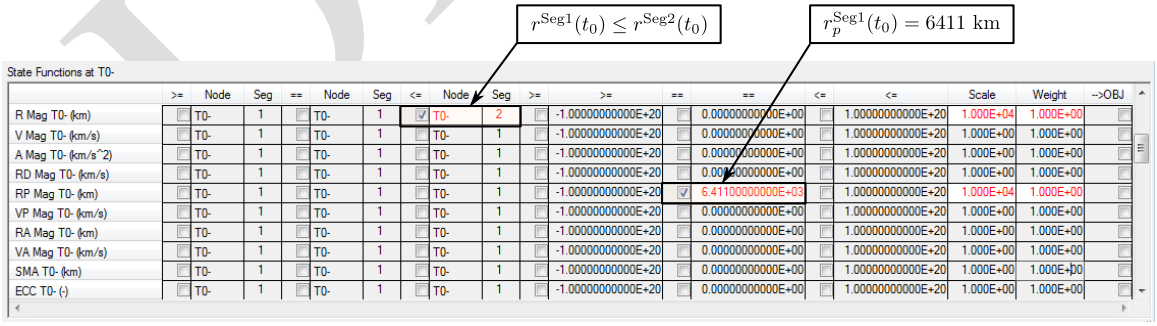


Figure 1: Copernicus GUI State Functions Grid. This screenshot shows how constraints are specified in the Copernicus GUI. In this case, the constraints are applied at t_0 of segment 1, and include a value equality constraint and an inequality constraint with segment 2.

DRAFT

added. The new capability would be an add-on to the tool, and existing users would not be required to use it to solve the problems that Copernicus is currently already capable of solving. Plugins would be user-created objects, separate from segments, that could be included in the mission and the optimization problem in various ways. When developing the plugin concept, several key goals and requirements were arrived at, including:

- The architecture should support different types of plugins, for different levels of complexity. A simple plugin that can be created quickly is allowed to sacrifice some computational efficiency. A more complex plugin may require a more complicated setup but be more computationally efficient.
- Basic plugins should be able to be created and used by non-expert programmers, without having to understand the inner workings of Copernicus.
- Plugins should not depend on a particular programming language or external third-party tool or application.
- The architecture should be object-oriented and extensible, so that new capability can be added to it at a later date, such as new plugin types, or new type of information that can be exchanged between Copernicus and plugins.

The author began part-time development of the plugin feature in January 2014. Undoubtedly, there are many different ways that it could have been done, though the solution arrived at (described in the remainder of this paper), was able to meet all of these requirements, as well as providing a foundation for future upgrades.

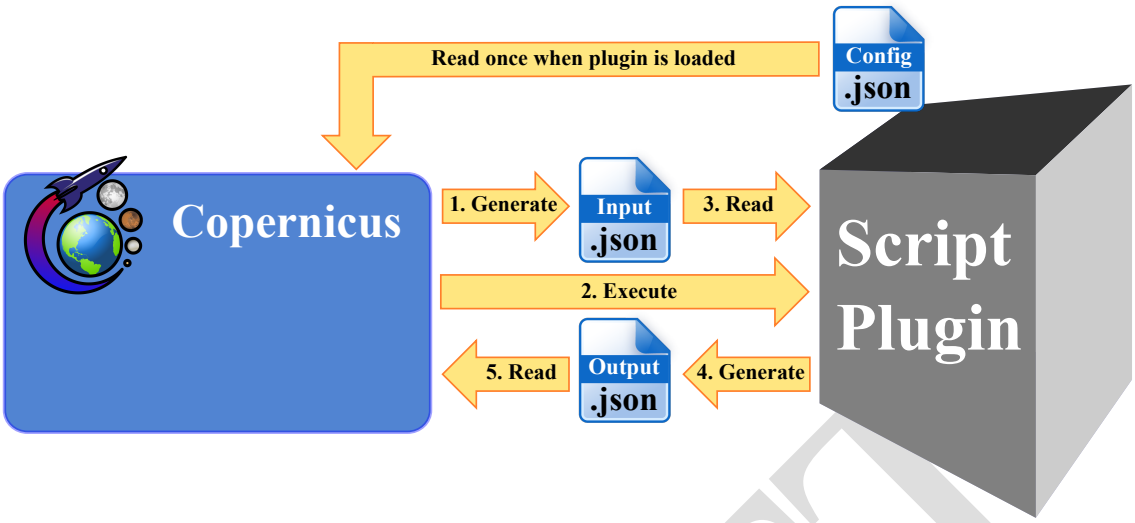
PLUGIN OVERVIEW

The new plugin architecture that was developed encompasses three types of plugins: script, DLL, and parser (refer to Figure 2 and Table 1). They are divided into two classes ("internal" and "external"), as shown in Figure 3. The fundamental concept of each plugin type is that Copernicus passes data (input variables) to the plugin and receives data back (output variables and trajectories) from it. The plugin simply computes the outputs as functions of the inputs. Script plugins pass this data using JavaScript Object Notation (JSON) files, while DLL plugins pass JSON structures (described in the following section). The plugin itself is a "black box" whose internal workings do not concern Copernicus, as long as the output data is properly formatted. Parser plugins (currently, the only instance of an internal plugin) are a bit different, and the data exchange is internal to Copernicus.

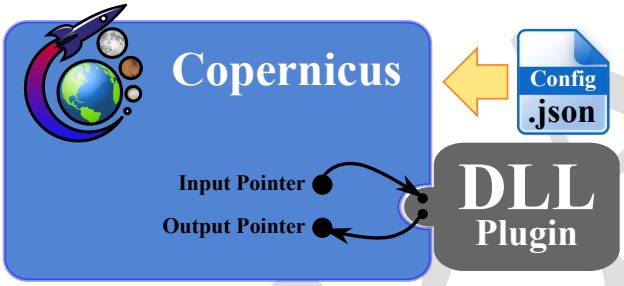
Table 1: Plugin Types and Data Exchange

Plugin Type	Config File	Plugin File	Input	Output
Parser	JSON file	None	None	None
Script	JSON file	Any executable file or script	JSON file	JSON file
DLL	JSON file	Compiled DLL (shared library)	`json_value` pointer	`json_value` pointer

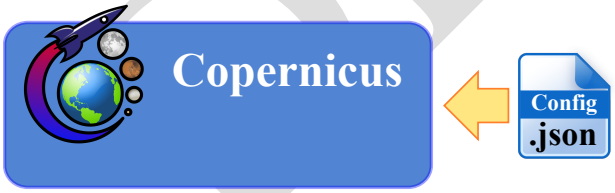
DRAFT



(a) Script Plugin Flow Chart. The plugin is a "black box" that is called by Copernicus. The plugin can be any file that can be executed by a system call (such as a Python script or an executable). It reads the input file and generates the output file, which is then read by Copernicus.



(b) DLL Plugin Flow Chart. A DLL plugin consists of a config file and a DLL file. The DLL is loaded by Copernicus and all data is exchanged using pointers, rather than files.



(c) Parser Plugin Flow Chart. A parser plugin consists only of a config file, and does not generate or use any additional files. This type of plugin can be used to define simple equations which are then evaluated internally within Copernicus.

Figure 2: Plugin Types. There are currently three types of plugin available: script, DLL, and parser. For each, the config file is used when the plugin is loaded to define the interface.

DRAFT

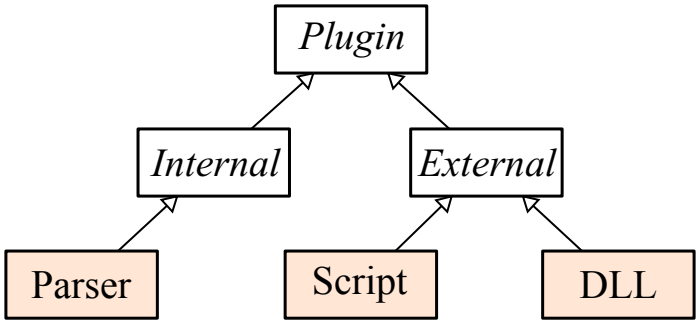


Figure 3: Plugin Class Hierarchy. Three types of plugins are available (parser, script and DLL). A parser is an instance of an "internal" plugin that does not require any external interaction once loaded. The other two are "external" plugins (which call either an external script or a DLL). There are potentially other types of both classes that could be added in the future.

All plugins require a JSON configuration file that is loaded by Copernicus. In this file, the user defines the input and output variables, and other information about the plugin required by Copernicus to use it. There can be any number of input and output variables, and the plugin can also export any number of trajectories (time tagged vectors of spacecraft state and mass). Variable names are assigned by the user, and their values can be associated with Copernicus segment variables (e.g, via the inheritance mechanism). An example plugin config file is shown in Figure 4. The details of this file are described fully in the Copernicus documentation.[3] Briefly, the format defines a way to specify the input variables, the output variables, and the exported trajectories. The `cop_var` flag provides Copernicus with the data type for plugin variables that can be associated with Copernicus variables (in this example, both the input and output variables are Δt variables). The format also allows for defining states using any Copernicus-supported reference frames and units.

When the config file is loaded into Copernicus, the file is parsed and the plugin data structures are initialized to prepare for calling the plugin. When a plugin call is required (say, during optimization), Copernicus first populates the input variable values. For script and DLL plugins, the input JSON data is then generated by Copernicus and passed to the plugin (script plugins via a JSON file, and DLL via a pointer to a JSON structure). After a script or DLL plugin runs, the corresponding JSON output (generated by the plugin) is read by Copernicus. The output variable values are then updated, along with any segment variables receiving data from them. A caching system is also implemented, so that if the input variables have not changed since the last time the plugin was run, the plugin is not actually called, and the output variables retain the values they had from the previous call (the plugin is assumed to be a consistent function generator that executes the same computations for the same values of the inputs[8]).

JSON

JSON[10] is a lightweight human-readable language-independent data interchange format. This format was selected to be used for the communication between Copernicus and the plugins. An example JSON file is shown in Figure 5a (this is a simple script plugin output file). JSON files use a {`"name": value`} syntax with six valid JSON value data types: number, string, boolean, array, object, and null. Objects are delimited using curly brackets and arrays are delimited using square brackets. Whitespace is not significant in a JSON file (except for within strings). Some special characters in a string require an escape sequence. A comma is used to separate each variable and

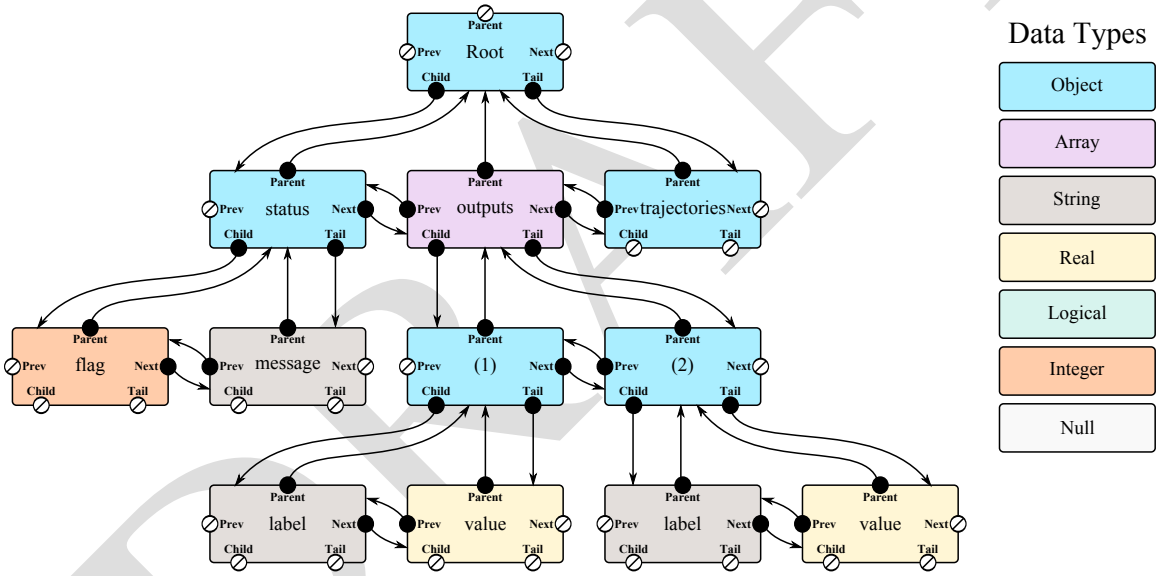
DRAFT

```
{
    "info": {
        "name": "Example",
        "description": "Example plugin dll",
        "type": "dll",
        "dll": "../support_files/plugins/dll_example/example.dll"
    },
    "inputs": [
        { "label": "dt1",
          "cop_var": "DT",
          "units": "day",
          "assume_choice": 1,
          "inherit_seg": 1 }
    ],
    "outputs": [
        { "label": "dt2",
          "units": "day",
          "cop_var": "DT",
          "assume_choice": 1,
          "inherit_seg": 2 }
    ],
    "trajectories": {
        "label": "orbit1",
        "traj": [
            { "label": "et", "cop_var": "ET" },
            { "label": "rv",
              "cop_var": "STATE",
              "frame": { "frame_type": "J2000",
                         "main_body": "EARTH" },
              "time_units": "s",
              "distance_units": "km",
              "params": [
                  { "label": "Rx", "cop_var": "Rx" },
                  { "label": "Ry", "cop_var": "Ry" },
                  { "label": "Rz", "cop_var": "Rz" },
                  { "label": "Vx", "cop_var": "Vx" },
                  { "label": "Vy", "cop_var": "Vy" },
                  { "label": "Vz", "cop_var": "Vz" }
              ]
           }
        ]
    }
}
```

Figure 4: Example DLL Plugin Config File. The config file is created by the user, and defines the interface to the plugin. The simple example shown here has one input variable and one output variable. The input variable inherits the Δt from segment 1, and the output variable pushes a Δt to segment 2. It also exports a trajectory time history (using ephemeris time, and the Cartesian state specified in a J2000-Earth frame). The "`info.type`" variable specifies the plugin type ("`parser`", "`script`", or "`dll`"). For script plugins, an "`info.script`" variable specifies the location of the plugin, while for DLL plugins the "`info.dll`" variable (shown here) is used instead.

```
{
  "status": {
    "flag": 0,
    "message": "converged"
  },
  "outputs": [
    { "label": "x", "value": 0.0 },
    { "label": "y", "value": 0.0 }
  ],
  "trajectories": { }
}
```

(a) This example script plugin output JSON file includes the "status", "outputs", and "trajectories" structures (which is empty in this case). Note that whitespace and line breaks are not significant in JSON files.



(b) The JSON data can be represented as a tree structure using a Fortran linked list.[9] Each node (which can be an object, array, integer, real, string, logical, or null value) contains five pointers (`parent`, `child`, `previous`, `next`, and `tail`). Here, ● represents an associated pointer, and ⊘ represents a null pointer.

Figure 5: JSON Example. JSON (either as a file or as a data structure) is used for all data exchange between Copernicus and plugins.

DRAFT

each element of an array. A JSON file can contain any number of variables. In the example shown in Figure 5, the file contains three variables: `status` (an object containing the two variables `flag` and `message`), `outputs` (an array of two elements, each element containing the variables `label` and `value`), and `trajectories` (an empty object).

JSON APIs exist for many different programming languages.[11] Copernicus is programmed in Fortran, so a Fortran API was necessary for this project. Since a feature-complete production-ready modern Fortran JSON solution did not exist, a new opensource project (called *JSON-Fortran*) was initiated to provide one.[9] Based on an older Fortran 95 project, the new library incorporates modern features from the Fortran 2003/2008 standards.[12] Figure 5b shows the JSON-Fortran representation of JSON output data, which is a tree structure using a Fortran linked list.[13,14] Each node contains five pointers (`parent`, `child`, `previous`, `next`, and `tail`). Copernicus incorporates the JSON-Fortran library to provide JSON file I/O as well as manipulation of the linked-list JSON data structures. A plugin written in Fortran can also use the library. For other implementations, such as Python, the details of how the data is represented internally is not necessary required. For example, the same JSON data could be constructed using a Python dictionary, and then converted to JSON using a call to `json.dumps`.[15] The wide range of programming languages supporting JSON means that a plugin can be written in any of them, without having to write a new parser in order to accomplish the data exchange. The flexibility of the JSON format also means that future additions to the types of data that can be exchanged between Copernicus and plugins can be easily incorporated into the data structures if needed.

SEGMENT AND PLUGIN VARIABLES

The plugin architecture was designed to serve as an augmentation to the existing segment architecture. The power and utility of plugins comes from the ability to interface plugin variables and segment variables (say, to use a plugin to impose a constraint in the optimization problem, or propagate a trajectory which is then subsequently inherited by a Copernicus segment).

Figure 6 shows a schematic of the relationship between Copernicus segment variables (such as the segment initial time or mass), and plugin variables (both input and output variables). While Copernicus considers a plugin as a black box completely separate from all segments, data can be exchanged between plugins and segments. Plugin input variables can *inherit* the value of segment variables, and plugin output variables can *push* their value back to segment variables. The main Copernicus optimization problem can include plugin input variables as optimization variables, and plugin output variables as constraints or objective functions. Plugin input variables can also be *updated* by plugin output variables after the plugin is run (for example, if the plugin is solving its own optimization problem and some or all of the input variables are the optimization variables for this problem). A cache scheme is also included, so that if the same inputs are passed to a plugin multiple times, then the previously-computed outputs are returned without having the call the plugin again (this can make a big difference if calling the plugin if very computationally expensive).

With the inclusion of plugins in the tool, a Copernicus mission can now be composed of any number of segments and any number of plugins. The order in which the segments and plugins are propagated is significant, and must be determined by their variable interconnectivity. See the example shown in Figure 7. For example, if plugin P1 is inheriting data from segment S2, then Copernicus must propagate S2 before P1 is called. Conversely, if P1 is pushing data to S3, then it must be called before S3 can be propagated. The propagation order is determined

DRAFT

automatically by Copernicus using a topological sorting algorithm.[13] Thus the user does not need to be concerned with specifying mission phases or attributes in any particular order, since the program will automatically handle this, and propagate everything in the correct order.

SCRIPT PLUGINS

The first type of external plugin is known as a "script plugin". For this type, the plugin itself is an external script or application. Data is transferred between Copernicus and the script by reading and writing JSON files. A flow chart of the script plugin concept of operations is shown in Figure 2a. When a call is required to the plugin (e.g., during optimization), Copernicus generates a JSON input file and executes the plugin. An example input file is shown in Figure 9. The plugin reads the input file, runs, and generates the output file (an example output file was shown in Figure 5a). Copernicus then reads the output file and continues.

A script plugin can be any file that can accept command line arguments and be executed by a system call. Examples include executable files, Python scripts, and shell scripts. The input file is passed to the plugin in the system call as the first command-line argument. Copernicus also redirects the plugin's standard output and standard error streams to the current mission log file. This is so any messages printed by the plugin will be visible in the Copernicus GUI. The precise call syntax used is:

```
> <executable> <plugin> <input file> >> <log> 2>&1
```

For example, a call to a Python plugin would look like:

```
> python my_plugin.py input.json >> LOG.OUT 2>&1
```

It is the responsibility of the plugin to retrieve the input file name from the command line argument. A script plugin can be written in any programming language (it is only necessary for it to be able to read and write JSON files). Basic outlines in Python, C++, and Fortran 2003 are shown in Figure 8.

Note that, when solving the optimization problem, Copernicus may call the plugin many times (e.g., during computation of the Jacobian matrix using finite-difference gradients). If the plugin is a simple computation, then the overhead of calling the external program (say, Python) and the file I/O could be a significant percentage of the plugin run time. However, for applications where evaluation of the plugin outputs has a significant computational burden, this may not be an issue. It is up to the user to decide this. For maximum computational efficiency, the DLL type plugins can be used (described in the following section), although these are (perhaps) more difficult to create than a simple Python script.

DLL (SHARED LIBRARY) PLUGINS

Shared library (or DLL) external plugins (see Figure 2b) are the most powerful and computationally efficient type of Copernicus plugin. For these, the user-created code is compiled and dynamically linked to Copernicus when the plugin is loaded. It runs in the same memory space as Copernicus, and can be as efficient as the main Copernicus code. In addition, all information is passed through the DLL interface rather than by reading and writing files. This can allow it to run with much less overhead than a script plugin. The config file for a DLL plugin is very similar to

DRAFT

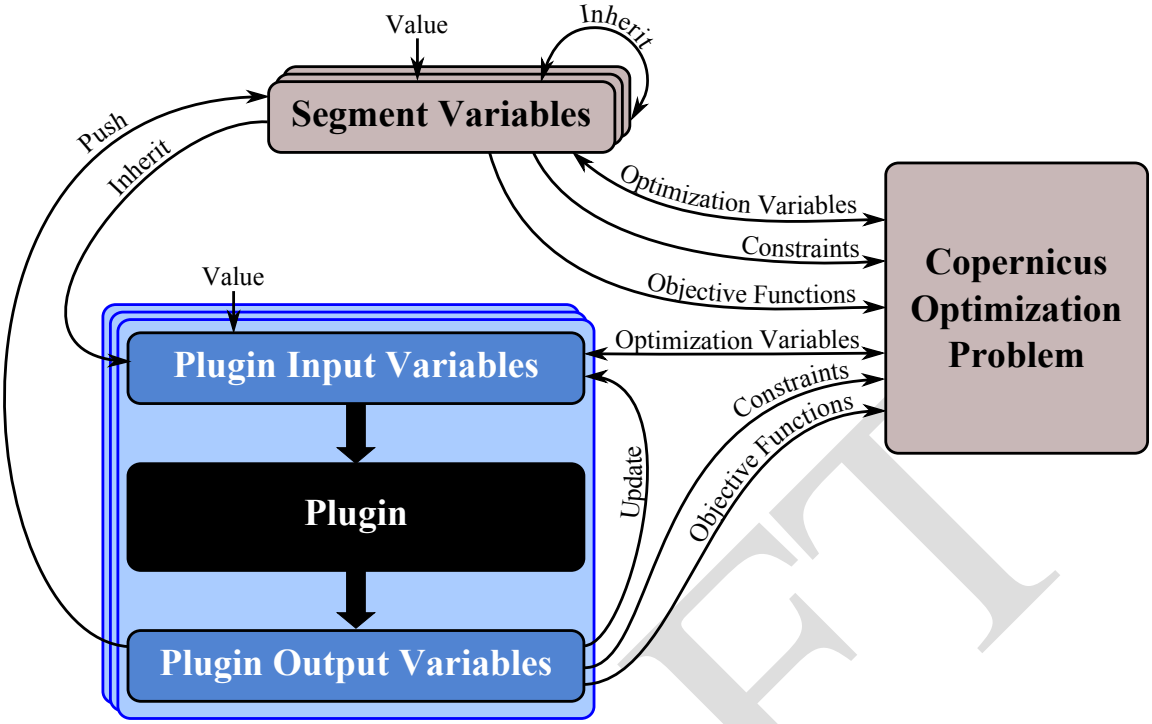


Figure 6: Segment and Plugin Variables. This diagram shows the relationship between Copernicus segment variables and plugin variables, and how they can be interconnected and/or included in the Copernicus optimization problem.

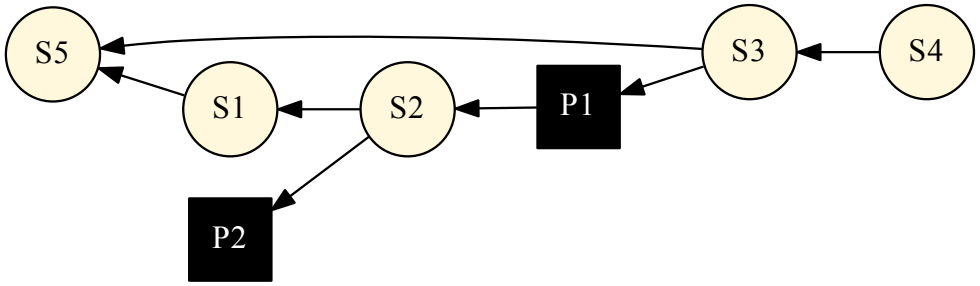


Figure 7: Segment and Plugin Dependencies. The interconnectivity of segment and plugins can be represented by a Directed Acyclic Graph (DAG) such as this. Here, the circles represent segments and the squares represent plugins. (S2)→(S1) indicates that segment 2 depends on segment 1 (i.e., it is inheriting data from it). In this case, the proper propagation order is: {S5, P2, S1, S2, P1, S3, S4}.

DRAFT

```python
import sys
n_args = len(sys.argv)
if (n_args>=2):
  input_file = str(sys.argv[1])
  # ... main plugin code ...
else:
  raise Exception('Error')
```

(a) Python Example

```cpp
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
  if (argc > 1) {
    string input_file = argv[1];
    // ... main plugin code ...
  }
  else { cout << "Error" << endl; }
}
```

(b) C++ Example

```fortran
program my_plugin
  implicit none
  integer :: ilen
  character(len=:),allocatable :: input_file
  ilen = command_argument_count()
  if (ilen>=1) then
   call get_command_argument(1, length=ilen)
   allocate(character(len=ilen) :: input_file)
   call get_command_argument(1, input_file)
   ! ... main plugin code ...
  else
   stop 'Error'
  end if
end program my_plugin
```

(c) Fortran 2003 Example

Figure 8: Example Script Plugin Snippets. A script plugin must first read the input file name from the first command line argument. Examples for Python, C++, and Fortran 2003 are shown here (script plugins can be written in any programming language). The main plugin code reads the input file, performs the appropriate computations, and generates the output file.

```
{
  "info": {
    "config": "path/to/PLUGIN_CONFIG.json",
    "output": "path/to/PLUGIN_OUTPUT.json",
    "dir": "path/to/script/",
    "need_trajectories": true
  },
  "inputs": [
    {
      "label": "dt1",
      "value": 0.1E+2
    }
  ]
}
```

Figure 9: Example Script Plugin Input File. The JSON input file is generated by Copernicus, and must be read by a script plugin. This is the input file that corresponds to the config file shown in Figure 4. The main purpose of this file is to pass the values of the input variables into the plugin. In this case, there is only a single variable ("dt1"). Other information is also provided to the plugin via the info variable.

the one for a script plugin, the only difference being in the info structure (see Figure 4). Here, the type variable must be set to "dll", and the dll variable must specify the location of the DLL file.

Communication between Copernicus and the user-created DLL plugin is only through a set of specified procedures that must be exported by the DLL. The main two are:

- The DLL must export an execute() subroutine which is called by Copernicus. The execute() subroutine has the interface shown in Figure 10a (data is exchanged using the input and output pointers). The json_value type is defined in the JSON-fortran library,[9] which must be linked with the DLL. While script plugins can be written in any programming language that can read and write JSON files, the DLL plugins must be written in modern Fortran and compiled with the Intel Fortran compiler (which is used to compile Copernicus). This is because the json_value types use Fortran pointers and allocatable variables which, in general, are not interoperable with C (and may not be interoperable among different Fortran compilers).
- A DLL plugin can also include a procedure that is meant to be called only once when the DLL is first loaded. This is done by exporting an initialize() subroutine (the interface is shown in Figure 10b). For example, this routine can be used to load database files that are subsequently used by the plugin when the execute() routine is called.

Copernicus can dynamically load any number of DLL plugins. On Windows, this process uses the operating system APIs LoadLibrary(), GetProcAddress(), and FreeLibrary().[16] The Linux version of Copernicus can load shared library plugins (.so files) in the same manner, using dlopen(), dlsym(), and dlclose().

As with script plugins, DLL plugins do not require any knowledge of the internal workings of Copernicus. It is enough to simply provide the two procedures in the DLL that Copernicus will call. The advantage of this is ease of use, however it also has its disadvantages. A plugin cannot access information from Copernicus which might be useful (say, the force models, reference frames,

DRAFT

or propagation methods). This capability could be added in a future revision, either by adding additional data to the JSON structures that are passed back and forth, or potentially by exporting additional types or procedures from Copernicus that the plugin can access.

The DLL plugins provide a "power user" solution for extending Copernicus. It is the most computationally efficient option, but is more complicated compared to the script plugin option. The next plugin type to be discussed is the simplest option.

PARSER PLUGINS

A parser plugin is an instance of the internal plugin type that allows for easily introducing user-specified equations into the mission and the optimization problem. It does not call external scripts or DLLs and does not use any input or output files (see Figure 2c). A parser plugin consists only of a JSON config file. The basic premise is that the set of output variable are computed as functions of the set of input variables, based on user-specified equations. The equations are evaluated by Copernicus using a function parser internal to the program.[17] Thus, for simple equations, a parser plugin can be much faster to execute than a script plugin, while being much easier to create than a DLL plugin. Once the plugin is loaded, all computations take place internally within Copernicus. Parser plugins are the easiest to use plugin type, and will perhaps be the ones used the most by Copernicus users.

An example parser plugin in shown in Figure 11. The JSON config file specifies a parser plugin by setting the `type` variable in the `info` structure to "`parser`" (rather than "`dll`" or "`script`"). Also note that the output variable structure includes an `expression` string variable. The expression is an equation which can include any of the input or output variables, referenced by the variable labels. For a parser plugin, each of the output variables must include an expression. Since the output variables can be used as constraints in the Copernicus optimization problem, this type of plugin makes it easy to create simple user-defined constraints. It can also be used for more general inherits, since output variable values can also be pushed to segment variables. For example, a parser plugin could be used to specify that the semimajor axis of one segment should be equal to some fraction of the semimajor axis from another segment. Previously in Copernicus, such an inherit (or even a nonlinear constraint) was not possible. In the example shown in Figure 11, the following constraint is defined:

$$m_0 = \left(1 - \frac{\mathrm{ContPer}}{100}\right)\left(ae^{-C_3/b} + c\right) \tag{1}$$

where a, b, and c are user-specified coefficients, and $\mathrm{ContPer}$ is a contingency % factor. The variable C_3 is the characteristic energy of a state (which can be inherited from a segment in Copernicus). The constraint can then be applied to the initial mass (m_0) of the segment. Before the plugin feature was available, a new constraint like this would have required a code change to Copernicus and a new release. Now, it can be easily implemented by the user.

A full set of arithmetic operators (`+`, `-`, `*`, `/`, `^`), logical operators (`>`, `<`, `>=`, `<=`, `==`, `/=`), and basic intrinsic functions (`abs`, `sin()`, `exp()`, `log()`, etc.) are available for constructing expressions.[3] Logical `IF()` statements are also allowed (`IF(x,y,z)` being evaluated as $(x\ ?\ y : z)$). For logical expressions, values $= 0$ are considered false, and values $\neq 0$ are considered true. Any number of equations can be defined, thus allowing for potentially very complicated parser plugins (each expression can be arbitrarily complicated and include any number of variables). The parser code is written in modern Fortran and uses various Fortran 2003 features such as deferred-length character strings in order to make this possible.[12]

DRAFT

```
abstract interface
 subroutine execute(input,output,status_ok)
 !DEC$ ATTRIBUTES DEFAULT, DLLEXPORT, ALIAS:"execute" :: execute
   import :: json_value
   implicit none
   type(json_value),pointer :: input      !input data structure
   type(json_value),pointer :: output     !output data structure
   logical,intent(out)      :: status_ok  !false if there was an error
 end subroutine execute
end interface
```

(a) A DLL plugin must export the `execute()` subroutine with this interface. It reads the input data structure and generates the output data structure.

```
abstract interface
 subroutine initialize(config,dll,status_ok)
 !DEC$ ATTRIBUTES DEFAULT, DLLEXPORT, ALIAS:"initialize" :: initialize
   implicit none
   character(len=*),intent(in) :: config    !config file path
   character(len=*),intent(in) :: dll       !dll file path
   logical,intent(out)         :: status_ok !false if there was an error
 end subroutine initialize
end interface
```

(b) A DLL plugin may contain an `initialize()` subroutine with this interface. If present in the DLL, Copernicus will call this subroutine when the DLL is first loaded. This can be used to perform initialization operations that only need to be done once.

Figure 10: DLL Subroutine Interfaces. A DLL plugin must contain the `execute()` subroutine, and can optionally contain the `initialize()` subroutine. The subroutines must have the exact interfaces shown here (on Windows, they must include the `!DEC$` Intel compiler directives so they are exported by the DLL).

DRAFT

A few simple example output variables with expressions are shown in Figure 12, where x, y, and z are input variables. Expressions are not case sensitive, and whitespace is not significant. Each of the input and output variables must have a unique label in a parser plugin, and only include valid variables names containing alphanumeric characters and underscores. An invalid expression will generate an error message. For parser plugin outputs that depend on other outputs, a topological sorting algorithm[13] is used to determine the proper evaluation order (and also check for and warn about circular dependencies). Thus, the expressions can be defined in any order. This can be seen in Figure 12, where the "`f2`" variable depends on variables defined after it.

EXAMPLES

The usefulness of plugins is that they can be used to augment the existing capabilities of Copernicus, allowing solutions of trajectory optimization problems that could not otherwise be solved using the tool. A schematic of this is shown in Figure 13, where a mission is shown incorporating ten segments and two plugins. Various plugin examples have been given in the previous sections. Additional possibilities for Copernicus plugins include:

- A Python script plugin has been created to enable Copernicus to call the POST ascent optimization program. This is represented by `P1` in Figure 13, and is the subject of a companion paper.[6] A Copernicus screenshot of this in action is shown in Figure 14.
- Implementation of a wide range of constraints are now possible using plugins. The simple parser plugins makes it easy to impose constraints that can be expressed as a equation or set of equations (see `P2` in Figure 13). More sophisticated constraints can also be created using script or DLL plugins. For example, Copernicus currently includes a very specific type of polynomial Entry Interface (EI) constraint.[3] However, a user can now create a plugin for any such constraint (say, using a different type of polynomial, or even by querying a database of EI states).
- If a trajectory sub-problem can be parameterized into a set of variables, and solved for a range of those variables, then this can be used to build of database that can be interpolated (rather than solving the problem again every time it is needed).[18] A DLL plugin could then be created which simply evaluates the outputs as interpolated functions of the inputs (say, using a multivariate b-spline interpolation method[19]). This plugin can then be included into a larger optimization problem. For example, a database of optimized POST ascent trajectories could be constructed for a particular launch vehicle.
- It is now possible to introduce additional methods of spacecraft propagation into Copernicus using plugins. For example, collocation (not currently present in the tool) could be added as a plugin.[20] The collocation variables and constraints would be included into the main Copernicus optimization problem along with the existing segment-based ones.

FUTURE WORK

The plugin capability was first included in the 4.1 release of Copernicus, and additional refinements were included in the 4.2 release. Future work includes:

- Allow a plugin to also optionally return gradient information if it is known. For example, if some of the plugin output variables are included in the optimization problem, and they have

DRAFT

```
{
  "info": {
    "type": "parser",
    "name": "MASSCOMPLEX",
    "description": "Parser plugin example."
  },
  "inputs": [
    { "label": "a",       "value": 20000 },
    { "label": "b",       "value": 60 },
    { "label": "c",       "value": 10 },
    { "label": "ContPer", "value": 10 },
    { "label": "m0", "cop_var": "M", "units": "kg"},
    { "label": "x0", "cop_var": "STATE",
        "frame": {
          "frame_type": "J2000",
          "main_body": "EARTH"
        },
        "time_units": "s",
        "distance_units": "km",
        "angle_units": "deg",
        "params": [
          { "label": "C3",   "cop_var": "C3"   },
          { "label": "Rp",   "cop_var": "Rp"   },
          { "label": "Inc",  "cop_var": "Inc"  },
          { "label": "RAAN", "cop_var": "RAAN" },
          { "label": "AOP",  "cop_var": "AOP"  },
          { "label": "Tau",  "cop_var": "Tau"  }
        ]
    }
  ],
  "outputs": [
    {
      "label": "Mfunc",
      "units": "kg",
      "expression": "m0 - ((1.0-ContPer/100.0)*(a*exp(-C3/b)+c))"
    }
  ]
}
```

Figure 11: Example Parser Plugin Script Plugin. A parser plugin consists only of the JSON config file. This example has six input variables (one of which is a state of six elements), and one output variable. It constrains an initial mass m_0 to an exponential decay function of C_3. The value of C_3 can be obtained from a Copernicus segment, while the coefficients (a, b, c, and $\mathrm{ContPer}$) can be specified by the user, or even included as part of the optimization problem. Note that some information (e.g., specification of inherits, segments numbers, optimization and constraint scale factors, etc.) is done in the GUI and is not shown here.

```
"outputs": [
  {"label": "f1", "expression": "2.0 * x^2 + sin(x*y) + (1.0 / z)" },
  {"label": "f2", "expression": "f1 + f3 + f4" },
  {"label": "f3", "expression": "if ( sin(x)>0.5, 1.0, exp(y) )" },
  {"label": "f4", "expression": "abs(x) * pi() - tan(y^(2+z))" }
]
```

Figure 12: Example Parser Expressions. This example (which are output variables from a parser configuration file), define four functions of the input variables x, y, and z. The output functions can be used as constraints in the optimization problem, or can be pushed to segment variables. Any number of output variables can be defined in any order.

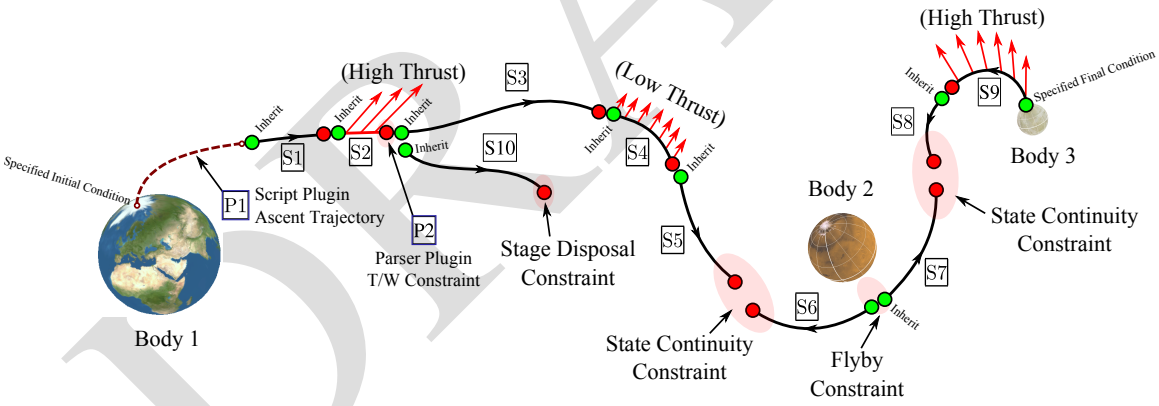


Figure 13: Example Copernicus Mission Using Segments and Plugins. The mission shown here includes ten segments and two plugins. The segments can represent multiple vehicles with multiple propulsion systems, and can be propagated both forward and backward in time. In this example, the script plugin P1 exports an Earth ascent trajectory that begins the mission, while the parser plugin P2 implements a T/W constraint at the conclusion of a finite burn phase.

DRAFT

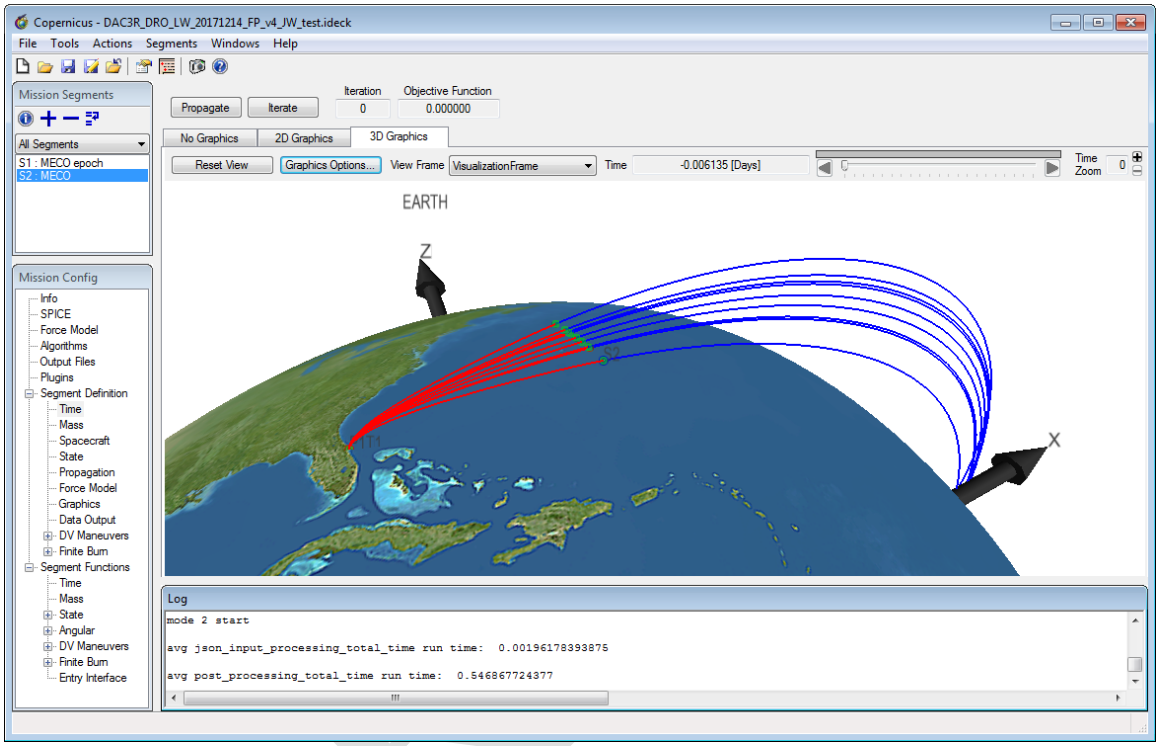


Figure 14: Example Copernicus Mission Screenshot. In this simple example, the red trajectory is an ascent trajectory produced by POST, imported into Copernicus via a Python script plugin. Copernicus manages the optimization problem (multiple iterations are shown here), so POST is used in this case as only a simulator. The blue segment is a normal Copernicus segment that inherits the time, mass, and state at the end of the ascent phase.

DRAFT

analytic gradients with respect to some of the optimization variables. That would mean these gradients would not have to computed by Copernicus using finite differences, which would speed up the overall solution process.

- Expand the DLL plugin feature to allow for interfacing plugins written in programming languages other than Fortran. This may require the "Further Interoperability with C" features expected in the Fortran 2015 standard.[21] However, in practice, this may not be an issue, since this is currently already possible with the inclusion of a bridge DLL written in Fortran that can interface with both Copernicus and the user-created DLL written in another language.

- Allow for sending an entire segment trajectory to a plugin. In this mode, Copernicus would propagate a segment as normal, and send all the points to a plugin. This could be used to implement various path type constraints, without requiring the plugin to perform the propagation.

- Exporting of additional data and/or procedures from Copernicus to DLL plugins, so that they have access to some of the internal types and procedures of Copernicus (such as force models, reference frames, celestial body ephemerides, and propagators). This would make more sophisticated DLL plugins possible without having to include basic algorithms that are already present in Copernicus.

- Implement a more comprehensive hash table caching scheme for plugin input and output variable values. This would provide a more comprehensive framework for reducing the number of unnecessary plugin calls (if, for example, the plugin is called multiple times with the same input).

- Expand the GUI to allow for more control over aspects of plugin configuration that currently require manually editing the config file. Ideally, there would be a GUI for creating plugins from scratch.

CONCLUSIONS

The new plugin feature is one of the most significant updates to Copernicus since the initial release. For the first time, it enables users to expand the tool in a variety of ways. Allowing different types of plugins (from simple to complex) also enables use by analysts who may not be expert programmers, while still enabling "power users" to code up very sophisticated plugins that have the potential to add significant new capabilities to the tool. It is expected that this new ability will greatly increase the utility of Copernicus, and expand the tool into new realms of spacecraft trajectory design and optimization. It is already being used for Orion and SLS analysis, and will continue to expand and evolve with the needs of NASA and by incorporating feedback from users.

ACKNOWLEDGMENTS

This work was funded by NASA JSC under contract NNJ13HA01C. The author wishes to acknowledge Ryan Whitley and Roland Martinez for their support of this work, Cesar Ocampo and David Lee for their brainstorming assistance during the plugin feature development, and Izaak Beekman for his assistance with JSON-Fortran development.

DRAFT

NOTATION

API	Application Program Interface
DAG	Directed Acyclic Graph
DLL	Dynamic-Link Library
EI	Entry Interface
GUI	Graphical User Interface
JSC	Johnson Space Center
JSON	JavaScript Object Notation
MSFC	Marshall Space Flight Center
NASA	National Aeronautics and Space Administration
POST	Program to Optimize Simulated Trajectories
SLS	Space Launch System

REFERENCES

[1] C. Ocampo, "An Architecture for a Generalized Trajectory Design and Optimization System," *Proceedings of the Conference: Libration Point Orbits and Applications* (G. Gómez, M. W. Lo, and J. J. Masdemont, eds.), World Scientific Publishing Company, June 2003, pp. 529–572. Aiguablava, Spain.

[2] J. Williams, J. S. Senent, and D. E. Lee., "Recent Improvements to the Copernicus Trajectory Design and Optimization System," *Advances in the Astronautical Sciences*, Vol. 143, January 2012.

[3] J. Williams, *Copernicus Version 4.2 User Guide*. NASA Johnson Space Center, July 2015. JETS-JE23-15-AFGNC-DOC-0052.

[4] J. P. Gutkowski, T. F. Dawn, and R. M. Jedrey, "Trajectory Design Analysis over the Lunar Nodal Cycle for the Multi-Purpose Crew Vehicle (MPCV) Exploration Mission 2 (EM-2)," *Advances in the Astronautical Sciences: Guidance, Navigation and Control*, Vol. 151, 2014. AAS 14-096.

[5] J. Williams and G. L. Condon, "Contingency Trajectory Planning for the Asteroid Redirect Crewed Mission," *AIAA SpaceOps 2014*, May 2014. AIAA 2014-1697.

[6] R. Whitley, J. Gutkowski, S. Craig, T. Dawn, J. Williams, B. Stein, D. Litton, R. Lugo, and M. Qu, "Combining Simulation Tools for End-to-End Trajectory Optimization," *AAS/AIAA Astrodynamics Specialist Conference*, August 2015. AAS 15-662.

[7] D. Litton, "Creating an End-to-End Simulation for the Multi-Purpose Crewed Vehicle," *AAS/AIAA Astrodynamics Specialist Conference*, August 2015. AAS 15-641.

[8] J. T. Betts, *Practical Methods for Optimal Control Using Nonlinear Programming*. Society for Industrial and Applied Mathematics, 2001.

[9] J. Williams, "JSON-Fortran: A Fortran 2008 JSON API," `https://github.com/jacobwilliams/json-fortran`.

[10] *The JSON Data Interchange Format*. ECMA International, October 2013.

[11] "JSON Website," `http://www.json.org`.

[12] M. Metcalf, J. Reid, and M. Cohen, *Modern Fortran Explained*. Oxford University Press, 2011.

[13] H. R. Lewis and L. Denenberg, *Data Structures and their Algorithms*. Harper Collins, 1991.

[14] E. Akin, *Objected-Oriented Programming via Fortran 90/95*. Cambridge University Press, 2003.

[15] "Python 3.3.5 Documentation: JSON Encoder and Decoder," `https://docs.python.org/3.3/library/json.html`.

[16] Microsoft Windows Dev Center, "Using Run-Time Dynamic Linking," `https://msdn.microsoft.com/en-us/library/windows/desktop/ms686944%28v=vs.85%29.aspx`.

[17] R. Falck, "CALCPARSER," March 2009. Originally developed for Optimal Trajectories by Implicit Simulation, version 4 (OTIS4).

[18] J. Williams, E. C. Davis, D. E. Lee, G. L. Condon, T. F. Dawn, and M. Qu, "Global Performance Characterization of the Three Burn Trans-Earth Injection Maneuver Sequence over the Lunar Nodal Cycle," *AAS/AIAA Astrodynamics Specialist Conference*, August 2009.

[19] J. Williams, "Bspline-Fortran: Multidimensional B-Spline Interpolation of Data on a Regular Grid," `https://github.com/jacobwilliams/bspline-fortran`.

[20] C. R. Hargraves and S. W. Paris, "Direct Trajectory Optimization Using Nonlinear Programming and Collocation," *Journal of Guidance, Control, and Dynamics*, Vol. 10, July 1987, pp. 338–342.

[21] ISO/IEC TS 29113:2012, *Information Technology – Further Interoperability of Fortran with C*. International Organization for Standardization, Geneva, Switzerland, 2012.

DRAFT