Application of Modern Fortran to Spacecraft Trajectory Design and Optimization

Jacob Williams*
ERC Inc., Houston, TX, 77058

Robert D. Falck†
NASA Glenn Research Center, Cleveland, OH, 44135

Izaak B. Beekman‡
ParaTools Inc., Baltimore, MD, 21228

In this paper, applications of the modern Fortran programming language to the field of spacecraft trajectory optimization and design are examined. Modern Fortran (the latest standard is Fortran 2008, and the newer Fortran 2018 standard is due to be published next year) is a significant enhancement to the classical Fortran 77 language. Modern object-oriented Fortran has many advantages for scientific programming, although many legacy Fortran aerospace codes have not been upgraded to use the newer standards (or have been rewritten in other languages perceived to be more modern). NASA's Copernicus spacecraft trajectory optimization program, originally a combination of Fortran 77 and Fortran 95, has attempted to keep up with modern standards and makes significant use of the new language features. Various algorithms and methods are presented from trajectory tools such as Copernicus, as well as modern Fortran open source libraries and other projects.

I. Introduction

The original version of Fortran, developed by IBM in the late 1950s, was the first high-level programming language. It has continued to be updated regularly up to the present day. A brief overview of the Fortran language evolution is summarized in Table 1. For more details, see References [1–3]. The "classical" version of the language was first standardized in Fortran 66 and updated in Fortran 77. Significant expansions of the language were implemented in Fortran 90 and Fortran 2003. Fortran 2003 [4] was a very significant update that made Fortran an object-oriented language (the difference between Fortran 2003 and Fortran 77 is akin to the difference between C++ and C). The latest standard is Fortran 2008 [5], the upcoming Fortran 2018 standard (formerly known as Fortran 2015) [6] is due to be published next year, and planning has begun for the next standard (Fortran 202x). Fortran has maintained a high degree of backward compatibility, with each revision being mostly a superset of the previous revision (so older code can still be compiled with a modern compiler). The term "modern Fortran", as used in this paper, is intended to mean Fortran 2003 (and later), and implies free-form source, thread-safety, object-oriented (where appropriate), clearly written and well-documented code.

Fortran is a high-level general purpose programming language very well suited for scientific, technical and high performance computing (HPC) [7–10]. The syntax is fairly intuitive and includes built-in vector and matrix handling features, with less necessity than C-based languages to use potentially unsafe pointers, which lends itself to vectorization and parallelization. As an example, a basic Fortran function for computing third-body gravitational acceleration [11] is shown in Fig. 1. This function highlights the straightforward "close to the math" syntax of the language, including operations involving vectors, without the need for external libraries. The language also includes advanced high-level object-oriented features which are critical for the development of very complex codes. A standardized interoperability with the C programming language also allows Fortran to call C and for C to call Fortran, opening up the language to other libraries not written in Fortran (note that interoperability with C also means interoperability with any language that is also interoperable with C, such as C++ and Python).

*Senior Astrodynamics Engineer, JETS Engineering Department, AIAA Senior Member
†Aerospace Engineer, Mission Analysis and Architecture Branch, AIAA Member
‡HPC Scientist, AIAA Member

```fortran
pure function third_body_gravity(r,rb,mu) result(acc)

!! Third-body (pointmass) gravitational acceleration.

use iso_fortran_env, only: wp => real64 ! use "double precision" reals

implicit none

real(wp),dimension(3),intent(in) :: r  !! satellite position vector [km]
real(wp),dimension(3),intent(in) :: rb  !! third-body position vector [km]
real(wp),intent(in) :: mu  !! third-body gravitational parameter [km^3/s^2]
real(wp),dimension(3) :: acc  !! gravity acceleration vector [km/s^2]

real(wp),dimension(3) :: r_sc_b  !! vector from third-body to spacecraft [km]
real(wp) :: rb_mag  !! distance between origin and third-body [km]
real(wp) :: r_sc_b_mag  !! distance between spacecraft and third-body [km]

r_sc_b = rb - r
r_sc_b_mag = norm2(r_sc_b)
rb_mag = norm2(rb)
acc = (mu/r_sc_b_mag**3)*r_sc_b - (mu/rb_mag**3)*rb

end function third_body_gravity
```

Fig. 1 Example Fortran Function to Compute Third-Body Gravity (from the Fortran Astrodynamics Toolkit). This function highlights the "close to the math" syntax of Fortran, without the need to use external libraries. It has vector inputs and returns a vector output. The function is "pure" meaning that it has no side effects, which can allow the compiler greater possibilities for code optimization.

While there is a great deal of very high-quality freely available or open source Fortran 77 legacy code in existence (e.g., MINPACK, SLATEC, and MATH77 at Netlib*), it is not in a form that is very appealing to modern programmers. For well documented third-party libraries that do not need to be changed, this obsolescence is not really an issue, but it can present severe impediments if the legacy code needs to be modified. Sometimes, refactoring is quite straightforward and enables many improvements to the original code. Recent activities including the development of new open source modern Fortran libraries hosted on sites such as GitHub, are a cause for optimism for the future of the language. In the aerospace field, for example, the Fortran Astrodynamics Toolkit (a work in progress) is intended to be a modern open source library for the foundational algorithms of orbital mechanics[†].

The Copernicus spacecraft trajectory optimization program [12], developed at the Johnson Space Center (JSC) and distributed under a government use license[‡], is an example of an actively-developed modern Fortran application. Copernicus is capable of solving a wide range of trajectory design and optimization problems, including trajectories centered about any planet or moon in the solar system, trajectories influenced by two or more celestial bodies such as halo orbits or distant retrograde orbits, Earth-Moon and interplanetary transfers, asteroid and comet missions, and more. One of the core elements of the program is the "segment", which is the fundamental building block of mission design in Copernicus. Copernicus includes a full-featured Graphical User Interface (GUI) with interactive 3D graphics. The system is very flexible and has been used extensively at JSC (and other NASA centers) for a wide range of projects.

At NASA in general, many spacecraft trajectory optimization problems are solved using Fortran tools such as SORT [13], OTIS [14], MALTO [15], Mystic [16], and Copernicus. Other historic Fortran 77 tools have been replaced or rewritten in other programming languages. Examples include POST [17] (converted to C in the 1990s) and DPTRAJ/ODP (replaced with the C++/Python MONTE [18]). The JPL tool CATO [19] uses an "object-based" approach with Fortran 95, which is similar in some ways to how Copernicus was originally coded in the early-2000s. Unlike many of the legacy tools, the Copernicus code base (originally a combination of Fortran 77 and Fortran 95) has been kept up to date using modern Fortran features, including the incorporation of object-oriented design patterns [20]. This paper describes some of these modern Fortran concepts and their usefulness in trajectory design tools such as Copernicus.

*Netlib Repository. `http://www.netlib.org`

[†]A Modern Fortran Library for Astrodynamics. `https://github.com/jacobwilliams/Fortran-Astrodynamics-Toolkit`

[‡]Copernicus Trajectory Design and Optimization System (Version 4.x). `https://software.nasa.gov/software/MSC-25863-1`

Table 1 Major Fortran Language Milestones.

Version	Summary
Fortran I – IV	First high-level programming language, subscripted arrays, intrinsic functions, `GOTO`, `IF`, and `DO` statements, subroutines and functions. Various vendor-specific versions.
Fortran 66	First standardized programming language.
Fortran 77	Structured programming, `IF-ELSE-ENDIF`, `CHARACTER` variables, `PARAMETER` statement, generic names for intrinsic functions.
Fortran 90	Array language, derived data types, dynamic memory, pointers, modules, operator overloading, generic interfaces, numeric inquiry functions, recursion, free-form source.
Fortran 95	`PURE` and `ELEMENTAL` procedures, garbage collection of `ALLOCATABLE` arrays.
Fortran 2003	Object oriented programming, type extension and inheritance, polymorphism, type-bound procedures, procedure pointers, C interoperability, floating-point exceptions.
Fortran 2008	Coarray parallelism, submodules, `DO CONCURRENT` and `BLOCK` constructs, recursive allocatable components of derived types.
Fortran 2018	Further C/Fortran interoperability, additional parallel features (e.g., coarray teams).

II. Survey of Algorithms

In the following sections, various algorithms and codes are discussed, along with examples of how they are implemented and used in modern Fortran. The focus is on high-level, object-oriented algorithms that are used in trajectory design tools such as Copernicus. Many of the codes discussed here are open source, and the links are given in footnotes for reference.

A. JSON

While the Fortran language does include facilities for reading and writing text and binary files, the only high-level text file format built into the language is known as a "namelist". Many legacy codes use this format for input configuration files. Namelists are simple to read and write from Fortran code but have very severe limitations (such as limited error checking ability and strict variable type requirements). For modern applications, it is preferable to use a more standardized and flexible format with a high-level interface. JavaScript Object Notation (JSON) is a lightweight human-readable data exchange format that has a modern Fortran open source interface known as JSON-Fortran[§]. JSON can be used for configuration input files and data output files, as well as for communication between tools. Recent versions of Copernicus use JSON-Fortran for interfacing with user-defined plugins using a JSON-based application programming interface (API). An example of reading a JSON input file is shown in Fig. 2 using the high-level `json_file` class. Methods exits in the class for retrieving data from a file, inquiring about the contents of the file or the types and sizes of individual variables, and error checking. In the example shown in Fig. 2b, the `get()` method, which is overloaded for integer, real, character, and logical scalar and vector variables, is used to retrieve values from the file. Other methods are also included for creating and modifying a JSON file.

In a future release, the main Copernicus input file format will transition from namelists to JSON. The JSON format allows for portable storage of arbitrary data, and the flexibility of the JSON-Fortran API will eliminate various limitations in the program that exist due to the rigidity of the namelist format. The ubiquity of JSON also provides opportunities for scripting Copernicus (and interfacing with plugins) in a variety of programming languages (such as Python). The JSON-Fortran API can also be used for general-purpose collection of data internally that can then be transformed into a variety of other formats. Copernicus employs this strategy to generate data output files in a variety of different formats (e.g., CSV, HDF5, and SPK).

A variety of other file formats is also available for use in modern Fortran codes via other third-party libraries, including INI[¶], CSV[‖], and HDF5[**].

[§]JSON-Fortran: A Fortran 2008 JSON API. `https://github.com/jacobwilliams/json-fortran`

[¶]Fortran INI ParseR and Generator. `https://github.com/szaghi/FiNeR`

[‖]Read and Write CSV Files Using Modern Fortran. `https://github.com/jacobwilliams/fortran-csv-module`

[**]Object Oriented Fortran HDF5 Module. `https://github.com/rjgtorres/oo_hdf`

```
{
    "rp": 4500,
    "et": -6128265558.8176,
    "n_revs": 200,
    "generate_plots": true,
    "ephemeris_file": "JPLEPH.421"
}
```

(a) Example JSON Config File. This example contains a hypothetical set of input parameters.

```
subroutine read_config_file(filename)
use json_module
character(len=*),intent(in) :: filename
type(json_file) :: json
call json%load_file(filename)
call json%get('rp',              rp             )
call json%get('et',              year           )
call json%get('n_revs',          n_revs         )
call json%get('generate_plots',  generate_plots )
call json%get('ephemeris_file',  ephemeris_file )
if (json%failed()) error stop 'error loading file'
call json%destroy()
end subroutine read_config_file
```

(b) The Code to Read the Example JSON File.

Fig. 2 Reading a config file using the JSON library is fairly straightforward. Additional error checking can be done to check if each variable is found in the file and to verify the variable type (two features that are not easily done using Fortran namelists).

B. Dynamic Structures & Linked Lists

One of the limitations of Fortran 77 was the lack of any facility for creating and manipulating dynamic data structures. This was a significant impediment to the creation of very complicated interactive applications. Modern Fortran includes two components to facilitate dynamic data: `allocatable` and `pointer` variable attributes [2]. Both have their uses. Allocatable arrays are generally more efficient since they are guaranteed to be contiguous in memory, whereas pointers are allowed to point to non-contiguous array slices. Improper use of pointers can also lead to memory leaks, whereas allocatables will always be automatically "garbage collected" by the compiler when they go out of scope. A derived type can include an allocatable or pointer instance of the same type, thus allowing for the creation of many kinds of dynamic and recursive structures.

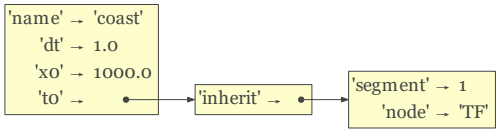
There are many example use cases for dynamic data structures in complex trajectory design software such as Copernicus. A primary example is the construction of the mission segments when reading the input file. A very simple example JSON snippet is shown in Fig. 3a, which could represent the initial conditions of a Copernicus segment (in this case, the initial time t_0 of the "coast" segment is inheriting the final time t_f from segment 1, whereas other variables Δt and x_0 are specified as real values). Data of this sort can be represented as a dictionary type structure containing integers, reals, booleans, strings, and other dictionary instances, as shown in Fig. 3b. In an interactive tool like Copernicus, the data layout of the mission can change during run time (for example, when a new segment is inserted, or if the selected gravity model is changed in a segment). Lists of different types of complex objects have many other applications in trajectory software (including frames, celestial bodies, and gravity models).

One of the basic building blocks of dynamic data structures is a linked list [21]. In earlier versions of Fortran, various workarounds were necessary to enable the creation of code to handle "generic" linked lists (i.e., a structure that can contain data of *any* type) [22]. The "unlimited polymorphic" variables introduced in Fortran 2003 make the creation of such structures much more straightforward (as shown in Fig. 4). Fig. 4a shows an example of a type that can be used to construct a polymorphic dictionary type structure using linked lists[††]. Here, the node content (`value`) is an unlimited polymorphic (`class(*)`) pointer variable, which can be allocated at run time to any variable type. The `key` is also polymorphic (e.g, to allow for integer, string, or other types of keys). Using these concepts, the data structure shown in Fig. 3 can be built as a polymorphic linked list using the code shown in Fig. 4d. There are a variety of flavors of this technique that can be used to build and manipulate various dynamics structures such as lists, stacks, and queues. It is important to note that a linked list constructed using pointer allocations in this manner must be properly destroyed in order to avoid memory leaks (the structure must be traversed and the pointers deallocated). The compiler will not do this automatically. Typically, a linked list could be encapsulated into a `list` derived type that includes all the public methods for creating and manipulating the data, which may not require the caller to use pointer variables (this is done in the JSON-Fortran library). The `list` type could also include a finalizer (a destructor-like type bound procedure called automatically when the variable goes out of scope) to clean up the memory. Dynamic structures using recursive allocatable variables are also possible (starting in Fortran 2008), which would be more memory safe, but perhaps less flexible depending on the use case [2].

[††]Modern Fortran Linked List. `https://github.com/jacobwilliams/flist`

```
{
    "name": "coast",
    "t0": {
        "inherit": {
            "segment": 1,
            "node": "TF"
        }
    },
    "dt": 1.0,
    "x0": 1000.0
}
```

(a) JSON Representation of Input Data. This is the type of data that could be read from a configuration input file.



(b) Linked List Representation of the Same Data.

Fig. 3 Example of Dictionary Type Structure. Using pointers, a variety of data structures can be created in Fortran. The example here is similar to the type of information read from a Copernicus input file.

```
type :: node
 private
 class(*),allocatable :: key
 class(*),pointer :: value => null()
 type(node),pointer :: next     => null()
 type(node),pointer :: previous => null()
 type(node),pointer :: parent   => null()
 type(node),pointer :: head     => null()
 type(node),pointer :: tail     => null()
end type node
```

(a) A Node in a Linked List. This type can be used to build tree structures. Both the `key` and `value` are unlimited polymorphic variables.

```
subroutine add_by_value(me,key,value)

type(node),pointer          :: me
character(len=*),intent(in) :: key
class(*),intent(in)         :: value

class(*),pointer :: p_value
allocate(p_value, source=value) !make a copy
call add_by_pointer(me,key,p_value)

end subroutine add_by_value
```

(b) Procedure for Adding a Node to a Linked List (by Value). In this method, the value is cloned using an `allocate` statement, which makes a copy to store in the list.

```
subroutine add_by_pointer(me,key,value)

type(node),pointer          :: me
character(len=*),intent(in) :: key
class(*),pointer            :: value

type(node),pointer :: p

if (associated(me%tail)) then !insert at end
    allocate(me%tail%next)
    p => me%tail%next
    p%previous => me%tail
else !first item in the list
    allocate(me%head)
    p => me%head
end if
me%tail => p
p%parent => me
allocate(p%key, source=key) ! the key
p%value => value ! the value

end subroutine add_by_pointer
```

(c) Procedure for Adding a Node to a Linked List (by Pointer). In this case, the pointer is directly added to the list and no copy is made.

```
program example

use linked_list_module

implicit none

type(node),pointer :: list
type(node),pointer :: inherit
class(*),pointer :: p_inherit

allocate(list)
call add_by_value(list,'name','coast')
call add_by_value(list,'dt',1.0_wp)
call add_by_value(list,'x0',1000.0_wp)

allocate(inherit)
call add_by_value(inherit,'segment',1)
call add_by_value(inherit,'node','TF')
p_inherit => inherit
call add_by_pointer(list,'t0',p_inherit)

end program example
```

(d) Constructing a Linked List. This is the data from Fig. 3, constructed as a linked list of `node` pointers.

Fig. 4 Linked List Example. A data type that contains pointers to instances of the same type can be used to construct a variety of different types of data structures using linked lists. The `next` and `previous` pointers can be used to build doubly-linked lists, while the `parent`, `head`, and `tail` pointers can be used for trees. See the JSON-Fortran library for a more comprehensive set of code for building these types of structures.

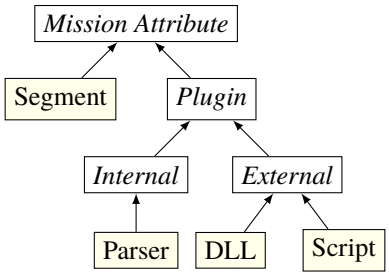


Fig. 5 Class Hierarchy of Copernicus Mission Attributes. Abstract types are italicized, which exist only to be extended. A mission consists of segments and plugins (either parser, DLL, or script).

```
type,abstract,public :: mission_attribute
contains
procedure(prop_func),deferred :: propagate
end type mission_attribute

type,extends(mission_attribute),public :: segment
contains
procedure :: propagate  => propagate_segment
end type segment

type,extends(mission_attribute),abstract,public ::
     plugin
end type plugin

type,extends(plugin),abstract,public ::
     internal_plugin
end type internal_plugin

type,extends(plugin),abstract,public ::
     external_plugin
end type external_plugin

type,extends(internal_plugin),public :: parser_plugin
contains
procedure :: propagate  => propagate_parser_plugin
end type parser_plugin

type,extends(external_plugin),public :: script_plugin
contains
procedure :: propagate  => propagate_script_plugin
end type script_plugin

type,extends(external_plugin),public :: dll_plugin
contains
procedure :: propagate  => propagate_dll_plugin
end type dll_plugin
```

Fig. 6 High-Level Mission Attribute Definitions. All concrete mission attribute types (which are all derived from the root `mission_attributes` type) must define a `propagate()` method. For segments, this will integrate the segment from t_0 to t_f. For plugins, this could be as simple as evaluating one user-defined equation. See also Fig. 5.

C. Copernicus Mission Attributes

A spacecraft trajectory optimization problem in Copernicus is constructed using two types of mission attributes: *segments* and *plugins* (see Fig. 5). Segments are the original built-in mission components, which can represent a thrusting or coasting period, can include impulsive Δv maneuvers, and can represent any number of different vehicles [12]. Plugins are a recent addition to Copernicus, and allow for user-defined algorithms to be included in the mission and optimization problem [23]. Various types of plugins are available, as shown in Fig. 5. Copernicus also includes the concept of *groups*, which are collections of segments and/or plugins that define a specific optimization problem (a mission can include any number of groups). An outline of the basic code definitions for segments and plugins is shown in Fig. 6. The mission attributes use various object-oriented concepts including polymorphism, inheritance, encapsulation, and data hiding. An *abstract* `mission_attribute` type exists only to be extended (variables cannot be declared of an abstract type). An abstract type may contain procedures that must be defined in the extended types (for example the `segments` type). In the case of Copernicus mission attributes, each mission component defines a `propagate()` method (among others). Simulating the entire mission involves calling the `propagate()` method of each mission attribute.

D. Topological Sorting

A Copernicus mission composed of many segments and plugins can contain complex interdependencies. The diagram in Fig. 7 shows a representation of the dependencies among the numerous segments and plugins of the Orion EM-1 [24] mission, which was designed in Copernicus. A complex mission can include hundreds or even thousands of segments. To avoid forcing the user to define the segments in a specific order, Copernicus employs a topological sorting algorithm to determine the order in which they must be propagated (for example, when computing the gradients). Topological sorting is a recursive algorithm used to determine the order in which a series of interdependent tasks must be completed [21]. A simple example of this algorithm is shown in Fig. 8, and an example use case is shown in Fig. 9. In this example, elements of a mission are numbered $[1, 2, \ldots]$. Their dependencies are known (for example,

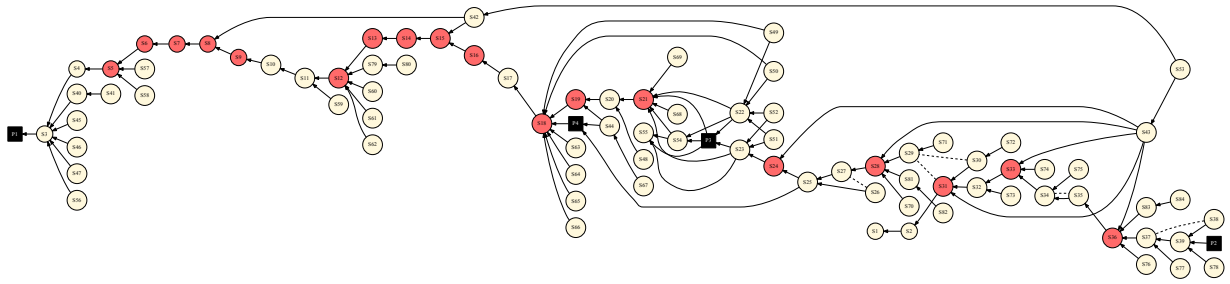


Fig. 7 Copernicus Mission Dependency Diagram. The dependencies of the attributes of a mission can be represented as a Directed Acyclic Graph (DAG). Copernicus dependencies arise, for example, when a segment inherits data from other segments. Before a segment (or plugin) can be propagated, all of its dependencies must be met (i.e., the segments it depends on must already have been propagated).

element 3 depends on elements 1 and 5). The `dag%toposort()` method produces the order: $[1, 2, 5, 3, 4]$, which is a valid propagation order that ensures that no segment is propagated before its dependencies are met. Another use of topological sorting in Copernicus is to ensure that a set of interdependent parser equations is evaluated in the correct order [23].

E. Calculation Expression Parser

General trajectory optimization tools such as OTIS and Copernicus allow users a great degree of freedom in defining their optimization problems. Much of this freedom comes from allowing the user to form their own objective functions and constraints as mathematical expressions based on an internal data dictionary of common variables and some available set of mathematical functions. Allowing the user to define these routines at run-time in infix notation extends the capability of such programs without the need to recompile the source code.

In this section, the implementation of a calculation expression parser is discussed. This parser converts mathematical expressions from infix notation into binary syntax trees. This parsing of mathematical expressions is a recursive operation, since expressions may be embedded in other mathematical expressions by wrapping them in parentheses. The parser uses several capabilities of modern Fortran to construct binary trees, which may then be efficiently evaluated.

1. Binary Syntax Trees

Binary syntax trees provide a data structure that can be used to represent complex mathematical expressions. As the name implies, a binary tree can have up to two children. Each node can represent a mathematical operator, function, literal numeric value, or variable value. The children represent arguments of operators or functions. Despite being limited to only two arguments, this framework accommodates most functions and operators commonly used. The algorithm presented here can be extended to a logical *if* function with *and* and *or* operators by allowing for a third *condition* child of each node, and by treating specific floating point values as booleans (1.0 = True, 0.0 = False, for instance). Fig. 10 depicts a few sample binary syntax trees. Note that the parsing algorithm must observe both the precedence rules and associativity of operators. All nodes in the tree are binary. In the case of unary functions, such as *sqrt*, the "right" child is an unassociated pointer. With the polymorphism supported by modern Fortran, the four types of node can each be a subclass derived from the base class, `cpNode` (shown in Fig. 11).

2. The Parsing Algorithm

The parsing algorithm is the most complex portion of the overall calculation expression capability. The algorithm must account for nested calculations embedded in parentheses, operator precedence and associativity, as well as some "corner cases" such as unary negation and redundant parentheses. In general, the calculation parsing algorithm involves the following steps:

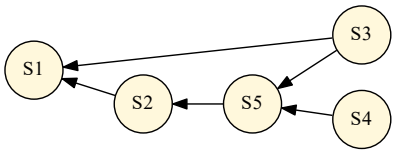
1) If the string begins with "(" and ends with ")", strip them and recursively call the parsing algorithm on the remaining string.
2) Call a subroutine `split_equation()` that splits the string by the operator of lowest precedence.

```
module toposort_module
implicit none
private

type :: vertex
!! a vertex of a directed acyclic graph (DAG)
  integer,dimension(:),allocatable :: edges
  integer :: ivertex = 0 !vertex number
  logical :: checking = .false.
  logical :: marked = .false.
contains
  procedure :: set_edges
end type vertex

type,public :: dag
!! a directed acyclic graph (DAG)
  type(vertex),dimension(:),allocatable :: vertices
contains
  procedure :: set_vertices => dag_set_vertices
  procedure :: set_edges => dag_set_edges
  procedure :: toposort => dag_toposort
end type dag

contains

subroutine set_edges(me,edges)
!! specify the edge indices for this vertex
  class(vertex),intent(inout) :: me
  integer,dimension(:),intent(in) :: edges
  allocate(me%edges(size(edges)))
  me%edges = edges
end subroutine set_edges

subroutine dag_set_vertices(me,nvertices)
!! set the number of vertices in the dag
  class(dag),intent(inout) :: me
  integer,intent(in) :: nvertices !! number of
     vertices
  integer :: i
  allocate(me%vertices(nvertices))
  me%vertices%ivertex = [(i,i=1,nvertices)]
end subroutine dag_set_vertices

subroutine dag_set_edges(me,ivertex,edges)
!! set the edges for a vertex in a dag
  class(dag),intent(inout) :: me
  integer,intent(in) :: ivertex !! vertex number
  integer,dimension(:),intent(in) :: edges
  call me%vertices(ivertex)%set_edges(edges)
end subroutine dag_set_edges

subroutine dag_toposort(me,order)
!! main toposort routine
  class(dag),intent(inout) :: me
  integer,dimension(:),allocatable,intent(out) ::
     order
  integer :: i,n,iorder
  n = size(me%vertices)
  allocate(order(n))
  iorder = 0  ! index in order array
  do i=1,n
    if (.not. me%vertices(i)%marked) &
        call dfs(me%vertices(i))
  end do

  contains

  recursive subroutine dfs(v)
  !! depth-first graph traversal
  type(vertex),intent(inout) :: v
  integer :: j
  if (v%checking) then
    error stop 'Error: circular dependency.'
  else
    if (.not. v%marked) then
      v%checking = .true.
      if (allocated(v%edges)) then
        do j=1,size(v%edges)
          call dfs(me%vertices(v%edges(j)))
        end do
      end if
      v%checking = .false.
      v%marked = .true.
      iorder = iorder + 1
      order(iorder) = v%ivertex
    end if
  end if
  end subroutine dfs

end subroutine dag_toposort

end module toposort_module
```

Fig. 8 Topological Sorting Module. This code can be used to determine the order in which to serially propagate a set of segments whose dependencies are specified. The main routine is `dag_toposort()`, which contains an internal recursive procedure that performs a depth-first traversal of the graph.

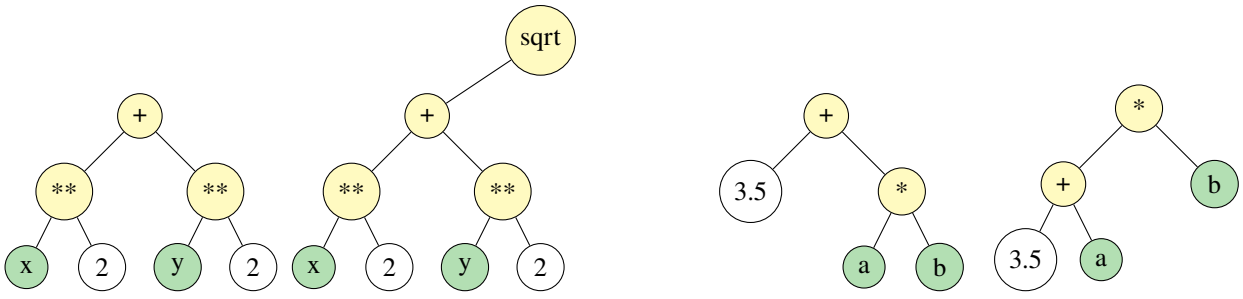
```
program main
use toposort_module
implicit none
type(dag) :: d
integer,dimension(:),allocatable :: order
call d%set_vertices(5)
call d%set_edges(2,[1])   !2 depends on 1
call d%set_edges(3,[5,1]) !3 depends on 5 and 1
call d%set_edges(4,[5])   !4 depends on 5
call d%set_edges(5,[2])   !5 depends on 2
call d%toposort(order)
write(*,*) order ! prints 1,2,5,3,4
end program main
```

(a) Topological Sorting Example. The module from Fig. 8 is used to compute the propagation order of a list of segments (in this case represented as integers) whose dependencies are known in advance.



(b) DAG Diagram for Segment Dependencies. Here, (S2)→(S1) indicates that segment 2 depends on segment 1 (i.e., segment 2 is inheriting some data from segment 1).

Fig. 9 Use Case for Topological Sorting to Determine Segment Propagation Order.



(a) `x**2 + y**2` (b) `sqrt(x**2 + y**2)` (c) `3.5+a*b` (d) `(3.5+a)*b`

Fig. 10 Binary Syntax Trees Associated with Various Infix Expressions.

```
type, public :: cpnode
    !! represents a single cpnode of a binary calculator tree.
    !! in general type(cpnode) variables should always be pointers.
    integer :: id = 0
    !! id tag for this cpnode, used for debugging
    class(cpnode), pointer :: left
    !! pointer to the left child cpnode
    class(cpnode), pointer :: right
    !! pointer to the right child cpnode
    class(cpnode), pointer :: parent
    !! points to the parent cpnode (null for root)
 contains
    procedure :: eval => eval_node
    !! returns the scalar floating point value for this node.
    procedure :: set_left => set_left
    !! sets the left child node
    procedure :: set_right => set_right
    !! sets the right child node
    procedure :: set_parent => set_parent
    !! sets the parent node
    procedure :: print => print_node
    !! set the print bound method for debugging
end type cpnode

type, public, extends(cpnode) :: literal_node
    !! implementation of cpnode that represents literal numeric values
    real(wp) :: value = 0.0_wp
    !! the value represented by this node
contains
    procedure, public :: eval  => eval_literal_node
    !! evaluation bound method specific to literal nodes
end type literal_node

type, public, extends(cpnode) :: variable_node
    !! implementation of cpnode that represents a variable value
    integer :: varindex = 0
    !! index in the array of variable values for this node
contains
    procedure, public :: eval  => eval_variable_node
    !! evaluation bound procedure specific to variable nodes
end type variable_node

type, public, extends(cpnode) :: func_node
    !! implementation of cpnode that represents a function or operator
    character(len=op_len) :: func = repeat(' ',op_len)
    !! function or operator name for this node
    procedure(cpFunc), nopass, pointer :: f
    !! the function represented by this node
contains
    procedure, public :: set_function
    !! bound procedure to set the function represented by this node
    procedure, public :: eval  => eval_func_node
    !! evaluation bound procedure specific to function nodes
end type func_node
```

Fig. 11 Base Class and Derived Classes of Nodes for a Binary Syntax Tree.

- Proceed through the string, keeping track of parentheses "depth".
- If an operator is found and the parentheses depth is zero, store it.
- If we find an operator with lower precedence, it becomes the "splitting operator"
- If the operator found has the same precedence as the current splitting operator but it associates left-to-right, override the current splitting operator.
- If a splitting operator is found then return the following: the operator, the portion of the string left of the operator, and the portion of the string right of the operator.

3) If `split_equation()` returns a splitting operator, then
   - Form a new *FunctionNode* associated with the splitting operator
   - Recursively call `parse_calc()` on the left and right substrings.
     - The nodes associated with the left and right substrings become the left and right children of the *FunctionNode*.
4) If `split_equation()` finds no splitting operator, then test for one of the following conditions:
   - Attempt to read the contents of the calculator string into a floating point variable. If no error is raised, the text represents a literal numeric value and `parse_calc()` returns a *LiteralNode* object.

- Attempt to match the text to the known function names. If it matches, return a *FunctionNode* associated with the function and recursively call `parse_calc()` on the arguments to the function, setting them to the left and right children of the *FunctionNode*.
- Attempt to match the text to the known variable names. If it matches, `parse_calc()` returns a *VariableNode* object.

3. Evaluation of Syntax Trees

Once an expression has been parsed and stored as a tree structure, it can be evaluated with a relatively simple algorithm. The polymorphism available in modern Fortran simplifies the logic by allowing evaluation functions that are specific to each class of node:

- Nodes that represent literal numeric values simply return the floating point value which they represent.
- Nodes that represent variables, *VariableNodes*, return the floating point value of their associated variable.
- Nodes that represent functions or operators first evaluate their child nodes and then pass those return values as arguments to the function with which they are associated.

The evaluation algorithm is a recursive, depth-first traversal of the tree in pre-order. If a given node has child nodes, they are evaluated first before evaluating the node itself. Figure 12 demonstrates the evaluation algorithm on a sample tree.

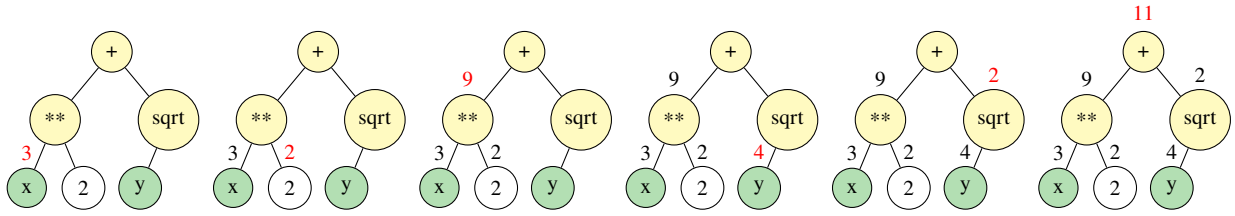


Fig. 12 Example of a recursive evaluation of a tree for the expression `"x**2 + sqrt(y)"` where $x = 3$ and $y = 4$.

F. Gravity Models

Often, a significant computational component of a high-fidelity spacecraft simulation is the evaluation of the spherical harmonic gravitational field. In Copernicus, the nonsingular algorithm of Pines is used [25], which has been refined by various others over the years [26]. The original published Fortran 77 code [27] is fairly straightforward and does not require much effort to modernize (see, for example, the `geopotential_module` in the Fortran Astrodynamics Toolkit). For small, irregularly-shaped bodies such as asteroids, a polyhedral gravity model may also be used, where the acceleration **a** is computed by the following equation (see Reference [28] for details):

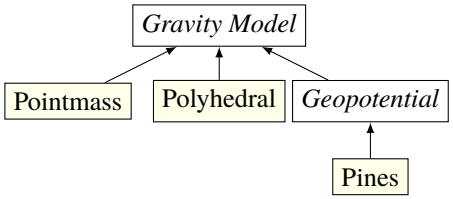
$$\mathbf{a} = \nabla U = G\sigma \left(\sum_{f \in \text{faces}} \mathbf{F}_f \cdot \mathbf{r}_f \cdot \omega_f - \sum_{e \in \text{edges}} \mathbf{E}_e \cdot \mathbf{r}_e \cdot L_e \right) \tag{1}$$

For a very complex polyhedron with many faces, computation of the acceleration is very computationally intensive, but fortunately, the sums are easily parallelizable with OpenMP [29] as shown in Fig. 14, where Eq. (1) is evaluated in a set of two loops (one for the edge terms and another for the face terms).

In Copernicus, the various types of central-body gravity models are extensions of an abstract `gravity_model` class, part of a general force model for trajectory segments (which can include other gravitating bodies, solar radiation pressure, and atmospheric drag) when they are propagated. Fig. 13 shows the basic concept, where the polyhedral model is an extension of the abstract type, and contains the acceleration routine (`get_acc_polyhedral()`) shown in Fig. 14). The `polyhedral_model` class also contains all the various variables and methods required to compute the acceleration.

```
type,abstract,public :: gravity_model
 !! The base abstract class for the various models
 contains
 ! each has to define this method:
 procedure(acc_function),deferred,public :: get_acc
end type gravity_model

type,extends(gravity_model),public :: polyhedral_model
 !! Polyhedral gravity model
 contains
 procedure,public :: get_acc => get_acc_polyhedral
end type polyhedral_model

abstract interface
 subroutine acc_function(me,r,a)
  import :: wp, gravity_model
  class(gravity_model),intent(inout)  :: me
  real(wp),dimension(3),intent(in)    :: r
  real(wp),dimension(3),intent(out)   :: a
 end subroutine acc_function
end interface
```

(a) Abstract Gravity Model and Extension.



(b) General Gravity Model Class Hierarchy

Fig. 13 The `polyhedral_model` is an extension of the abstract `gravity_model` class. The extended class also includes an initialization method, and any other methods and data that are needed for the model (not shown here). It inherits all the data and other methods in the base class. For example, while the pointmass model only contains the gravitational parameter μ, the polyhedral model contains the polyhedral mesh coordinates and other ancillary data.

G. ODE IVP Solvers

Copernicus includes a large collection of numerical methods for segment propagation, from fixed-step Runge-Kutta and Nyström methods of various orders, to variable-step variable-order Adams methods. There are numerous publicly-available Fortran 77 variable-step variable-order Adams-type codes that work well for the orbit problem (e.g., DLSODE [30], DVODE [31], DDEABM [32], and DIVA[‡‡]). Very few of these have been updated by the original authors in decades (CVODE, the C++ successor to DVODE, continues to be developed at LLNL). However, many of the original Fortran 77 codes are fairly easy to improve using modern Fortran concepts. Copernicus includes modernized versions of some of these codes. An open source modernized update to DDEABM is also available[§§], which includes new features not available in the original version made possible by the modern language features. The refactored version is object-oriented and thread safe. It also includes a new event finding capability (which incorporates the well-known ZEROIN algorithm for finding a root on a bracketed interval [33]), and the ability to export intermediate points via a user-defined method. Use of the algorithm is via a single class, which can be extended to include the data to be passed into the derivative function.

H. Ephemeris

A celestial body ephemeris is another necessary component of a spacecraft trajectory design and optimization program. Copernicus uses the ephemeris system provided by the Fortran 77 SPICELIB from JPL/NAIF [34], but also allows for pre-splining the ephemeris using various order interpolating B-Splines[¶¶]. This technique eliminates the SPICELIB overhead and can result in much faster execution time [35], and also provides a workaround for the fact that SPICELIB is not thread safe. Unfortunately, many classic Fortran codes such as SPICELIB have never moved beyond Fortran 77 and contain all the limitations of a programming language that was superseded almost thirty years ago. SPICELIB is severely restricted by the constraints of Fortran 77 (e.g., maximum number of kernels that can be loaded, maximum number of pool variables, lack of thread safety, impossible to have more than one SPICELIB instance at a time, and a coding style that uses `ENTRY` statements that were made obsolete by modules in Fortran 90). This is unfortunate since SPICELIB is exceptionally well-written and well-documented code. A modern object-oriented and thread safe SPICELIB could be produced by refactoring the code using modern Fortran techniques, and C-interoperability could be

[‡‡] JPL MATH77 Library. `http://netlib.org/math/`

[§§] Modern Fortran implementation of the DDEABM Adams-Bashforth algorithm. `https://github.com/jacobwilliams/ddeabm`

[¶¶] Multidimensional B-Spline Interpolation of Data on a Regular Grid. `https://github.com/jacobwilliams/bspline-fortran`

```
pure subroutine get_acc_polyhedral(me,r,a)

use iso_fortran_env, only: wp => real64 ! using double precision reals

implicit none

class(polyhedral_model),intent(in) :: me
real(wp),dimension(3),intent(in)  :: r   !! Spacecraft position vector (in body-fixed model frame) [km]
real(wp),dimension(3),intent(out) :: a   !! Acceleration vector in body-fixed model frame [\( km/s^2 \)]

real(wp),dimension(3) :: a_sumedge,a_sumface
real(wp),dimension(3) :: r1vec,r2vec,r3vec
real(wp),dimension(3) :: temp_vec
real(wp)              :: l_e,r1mag,r2mag,r3mag,wf,numer,denom,term
integer               :: i

a_sumedge = 0.0_wp; a_sumface = 0.0_wp ! initialize

!$omp parallel do default(shared) private(i,r1vec,r2vec,r1mag,r2mag,numer,denom,term,l_e) reduction(+:a_sumedge)
do i = 1, me%n_edges     !Edge loop

    r1vec = me%v(:,me%edges(1,i)) - r   !vectors from the field point to the edge endpoints
    r2vec = me%v(:,me%edges(2,i)) - r
    r1mag = norm2(r1vec)
    r2mag = norm2(r2vec)
    term = r1mag + r2mag   !calculate the logarithm expression, l_e
    numer = term + me%en(i)
    denom = term - me%en(i)
    l_e = log(numer/denom)

    !sum the edge contributions:
    a_sumedge = a_sumedge + matmul(me%ee(i)%matrix*l_e,r1vec)    ! acceleration

end do
!$omp end parallel do

!$omp parallel do default(shared) private(i,r1vec,r2vec,r3vec,r1mag,r2mag,r3mag,numer,denom,term,wf,temp_vec) reduction(+:a_sumface)
do i = 1, me%n_plates     !Face loop

    r1vec = me%v(:,me%p(1,i)) - r    ! vectors from field point to face vertices
    r2vec = me%v(:,me%p(2,i)) - r
    r3vec = me%v(:,me%p(3,i)) - r
    r1mag = norm2(r1vec)
    r2mag = norm2(r2vec)
    r3mag = norm2(r3vec)
    numer = dot_product(r1vec,cross(r2vec,r3vec))
    denom = r1mag*r2mag*r3mag + &
            r1mag*dot_product(r2vec,r3vec) + &
            r2mag*dot_product(r3vec,r1vec) + &
            r3mag*dot_product(r1vec,r2vec)
    wf = 2.0_wp * atan2( numer , denom )     !calculate the solid angle term, wf

    !sum the face contributions
    temp_vec  = matmul(me%ff(i)%matrix*wf,r1vec)
    a_sumface = a_sumface + temp_vec   ! acceleration

end do
!$omp end parallel do

a = me%gdensity*(a_sumface - a_sumedge)    ! acceleration (see Equation 1)

end subroutine get_acc_polyhedral
```

Fig. 14 Polyhedral Gravity Acceleration Core Routine. This is a method in the `polyhedral_model` class (an extension of the abstract `gravity_model` class shown in Fig. 13). The class contains all the necessary data such as the vertex coordinates (`me%v`), plate indices (`me%p`), edge indices (`me%edges`), edge lengths (`me%en`), edge matrices (`me%ee`), and the face normal outer products (`me%ff`). All of these quantities are computed when the class is initialized. See [29] for a detailed description of this code.

```
subroutine ballistic_derivs(me,et,x,xdot)

class(segment),intent(inout)      :: me
real(wp),intent(in)               :: et   !! ephemeris time [sec]
real(wp),dimension(:),intent(in)  :: x    !! state [r,v] in inertial frame (moon-centered)
real(wp),dimension(:),intent(out) :: xdot !! derivative of state [dx/dt]

reaL(wp),dimension(6) :: rv_earth_wrt_moon,rv_sun_wrt_moon
real(wp),dimension(3,3) :: rotmat
real(wp),dimension(3) :: r,rb,v,a_geopot,a_earth,a_sun
logical :: status_ok

r = x(1:3); v = x(4:6)  ! get state
rotmat = icrf_to_iau_moon(et)  ! rotation matrix from inertial to body-fixed Moon frame
rb = matmul(rotmat,r)  ! r in body-fixed frame

call me%grav%get_acc(rb,a_geopot)  ! get the acc due to the geopotential
a_geopot = matmul(transpose(rotmat),a_geopot)  ! convert acc back to inertial frame

call me%eph%get_rv(et,body_earth,body_moon,rv_earth_wrt_moon,status_ok)  ! 3rd body vecs (inertial wrt moon)
call me%eph%get_rv(et,body_sun,body_moon,rv_sun_wrt_moon,status_ok)

a_earth = third_body_gravity(r,rv_earth_wrt_moon(1:3),mu_earth)  ! 3rd body perturbations
a_sun = third_body_gravity(r,rv_sun_wrt_moon(1:3),mu_sun)

xdot = [v, a_geopot + a_earth + a_sun]  ! total derivative vector

end subroutine ballistic_derivs
```

Fig. 15 Ballistic Equations of Motion for a Spacecraft in the Vicinity of the Moon. In this example Fortran Astrodynamics Toolkit usage, a `segment` class contains an instance of a ephemeris (`eph`) as well as a geopotential gravity model (`grav`) for the Moon. The ephemeris is used to compute the perturbations from the Earth and Sun.

used to provide interfaces to other programming languages. Virtually none of the mathematical code would need to be modified. Be that as it may, NAIF has recently announced that a modern edition of SPICELIB will be produced by rewriting the entire library in C++.

A modern Fortran object-oriented celestial body ephemeris system has been created that is based on the legacy Fortran 77 JPLEPH code [36]. This system is included in the Fortran Astrodynamics Toolkit, and is a fairly light refactoring of the original code (as none of the mathematics have changed). The ephemeris is a class, and it allows for thread-safe use and multiple instantiations, neither of which are possible in either the original code or SPICELIB. The ephemeris class can be employed for the trajectory segment propagation problem by having a segment include an instance of an ephemeris. An example use case for this is shown in Fig. 15, where the equations of motion of a spacecraft at the Moon include the third-body perturbations from the Earth and Sun. An ephemeris is also necessary for certain state transformations (for example conversion between an inertial frame and an Earth-Moon rotating frame), so a general-purpose state transformation class also requires an instance of the ephemeris to be input to the transformation method.

I. Optimizers & Nonlinear Equation Solvers

Historically, much of spacecraft trajectory optimization at NASA has been done using Fortran 77 solvers such as VF13AD [37], SLSQP [38], NPSOL, SNOPT [39], and IPOPT [40] (IPOPT originally was written in Fortran, and though eventually converted to C++, still includes many third-party Fortran components). These general solvers can be used to solve nonlinear programming problems to minimize a scalar objective function subject to general equality and inequality constraints and to lower and upper bounds on the variables. Most complex problems in Copernicus are solved using SNOPT, which has proven very effective for very large and sparse trajectory problems. A new modern Fortran version of SNOPT has been under development for some time. SLSQP (suitable only for smaller problems) is also available in Copernicus and OTIS (as well as the Python SciPy package), and a new open source modern Fortran refactoring is also available[***].

For some problems not requiring optimization, a very simple differential corrector solver can often be used [41].

[***]Modern Fortran Edition of the SLSQP Optimizer. `https://github.com/jacobwilliams/slsqp`

```
subroutine nle_solver(me,x)

class(nle_solver_class),intent(inout) :: me
real(wp),dimension(:),intent(inout)  :: x  !! control variable vector

integer :: iter
real(wp),dimension(me%m,me%n) :: fjac  !! jacobian matrix
real(wp),dimension(me%n)      :: p     !! search direction
real(wp),dimension(me%m)      :: fvec  !! function vector

fvec = me%func(x)    !evaluate the function at the initial point
do iter = 1,me%max_iter
    fjac = me%grad(x)                     ! compute the jacobian matrix
    p = linear_solver(fjac,-fvec)         ! compute the search direction p by solving linear system
                                          ! [this is just a wrapper for DGELS]
    x = x + me%alpha * p                  ! compute new x
    fvec = me%func(x)                     ! evaluate the function at the new point
    if (maxval(abs(fvec)) <= me%ftol) exit ! check for convergence in f
end do

end subroutine nle_solver
```

Fig. 16 Overview of a Differential Corrector Solver. The `func()` and `grad()` functions in the `nle_solver_class` are user-supplied functions defined when the base class is extended (all the data necessary to compute the functions is contained in the class, thus the solver code can be very generic).

Solvers of this sort use a Newton-style iteration step from a previous control vector $\mathbf{x}_k$ to the next one $\mathbf{x}_{k+1}$, using:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \mathbf{J}^{-1}\mathbf{f}(\mathbf{x}_k) \tag{2}$$

where $\mathbf{x}$ is the $n \times 1$ vector of control variables, $\mathbf{f}$ is the $m \times 1$ vector of constraint violations, and $\mathbf{J}$ is the $m \times n$ Jacobian matrix. If $n > m$ then $\mathbf{J}^{-1}$ is the minimum norm pseudoinverse of the underdetermined linear system, which can be computed, for example, using the LAPACK `dgels()` subroutine. See Fig. 16 for a very basic overview of this type of code. A robust, general purpose solver is much more complex than the example shown here, and includes error checking, other types of convergence and singularity checks, as well as various options for computing the step size α (for example, by using a line search). For square systems ($n \times n$) there are many publicly available high-quality Fortran 77 solvers, such as HYBRJ from MINPACK, an implementation of the Powell hybrid method [42].

J. Gradients

Gradient-based solvers and optimizers (such as SNOPT) require computation of the derivatives of the objective function and the constraints with respect to the control variables (i.e., the Jacobian). Some solvers (such as IPOPT) also allow or require the input of second derivative information as well (i.e., the Hessian). Computing the gradients can be the most time-consuming part of the optimization problem, and accurate gradients are critical to successful convergence.

The classical method to compute gradients is via finite differences. Finite differences have the advantage of being simple to compute, and they can be used if the function contains third-party or "black-box" components. Copernicus primarily uses finite differences to compute gradients (although other methods are also available such as differentiation of interpolating polynomials such as cubic splines). If $f_n = f(x+nh)$ is a function evaluation given a control variable x and a perturbation h, then the set of three-point finite differences approximating the derivative at x are $(-3f_0 + 4f_1 - f_2)/(2h)$, $(-f_{-1} + f_1)/(2h)$, and $(f_{-2} - 4f_{-1} + 3f_0)/(2h)$. In practice, the second one (the classical central difference method) is often used, but the others are useful if the central difference would violate the variable bounds (for example, if the function is undefined beyond a certain bound). Copernicus includes a full set of formulas from two to eight points [43], as well as algorithms for tuning the step size h (which is critical when using finite differences) [44]. A new open source object-oriented finite-difference library (NumDiff) is also available with a variety of user-selectable methods[†††] including from two to six point formulas as well as Neville's algorithm [45]. NumDiff also includes an implementation of a graph coloring algorithm for efficient computation of the Jacobian by taking advantage of the sparsity pattern [46]. Another Fortran numerical differentiation library also exists that employs OpenMP and MPI for parallelization [47].

For applications where the user has full control of the problem function and all the source code, other types of gradient methods are also possible. Since complex numbers are natively supported in Fortran, complex-step differentiation

[†††] Modern Fortran Numerical Differentiation Library. `https://github.com/jacobwilliams/NumDiff`

```
module operator_overloading

use iso_fortran_env, only: wp => real64
implicit none

private

type,public :: a_real
    private
    real(wp),public :: value = 0.0_wp
    real(wp),public :: grad = 0.0_wp
end type a_real

interface operator(*) ! overload multip. operator
    module procedure :: aa_multiply
end interface
public :: operator(*)

interface sin ! overload sin() function
    module procedure :: a_sin
end interface
public :: sin

contains

pure elemental function aa_multiply(a,b) result(c)
    !! a_real * a_real
    type(a_real),intent(in) :: a
    type(a_real),intent(in) :: b
    type(a_real) :: c
    c = a_real(a%value * b%value,&
               b%value*a%grad + a%value*b%grad)
end function aa_multiply

pure elemental function a_sin(a) result(c)
    !! sin(a_real)
    type(a_real),intent(in) :: a
    type(a_real) :: c
    c = a_real(sin(a%value), cos(a%value)*a%grad)
end function a_sin

end module operator_overloading

program operator_overloading_test

use iso_fortran_env, only: wp => real64
use operator_overloading

implicit none

type(a_real) :: x,z

x = a_real(2.0_wp,1.0_wp) ! get derivative w.r.t. x
z = x*sin(x) ! computes value and dz/dx

write(*,*) z%value, z%grad ! 1.818594, 0.077003

end program operator_overloading_test
```

Fig. 17 Operator Overloading to Compute Gradients. In this example, we define a new `a_real` type to replace all `real` variables in a set of calculations where gradients are required. For the most general case, all operators, assignments, and intrinsic functions must be overloaded. In this example, it is shown how to overload the multiplication operator and the `sin()` function to compute the function value (z) and gradient (dz/dx) for the function $z(x) = x \sin x$ when $x = 2$.

is an option [48]. Operator overloading (see Fig. 17 for a basic example) can also be used to compute the gradients analytically. Various modern Fortran libraries are available for differentiation using operator overloading[‡‡‡] [49]. Finally, quadruple precision (`real128`, which is natively supported in Fortran), or even arbitrary precision (available via third-party libraries [50]) can be employed to reduce roundoff error in computations and produce more accurate finite difference gradients.

III. Coarray Fortran

Thanks to the introduction of coarrays, Fortran is a partitioned global address space (PGAS) language, allowing programmers to write parallel programs in standard conforming Fortran without relying on third party libraries or compiler directives and extensions. Coarray Fortran (CAF) [51, 52] has been part of the Fortran language since the 2008 standard.

A defining characteristic of PGAS languages is the ability to perform operations with *global* memory, regardless of whether the machine is a leadership-class, petascale HPC cluster, or a shared-memory laptop. This presents the application programmer with a consistent programming model across different computer architectures, including distributed memory, shared memory, and heterogeneous systems. Another feature of PGAS languages is the ability to manipulate remote memory through fetching remote values, "gets," or storing remote values, "puts," without directly involving the processing element (PE) that "owns" the memory in question. This presents the programmer with a simplified programming model reminiscent of shared memory paradigms, even when the program is running across many nodes of a distributed and/or heterogeneous system. In PGAS languages the burden of explicitly coordinating communication between the sender(s) and receiver(s) is removed; one-sided communication is the default, and less time is spent waiting and coordinating with remote PEs relative to the more common two sided message passing paradigms. A one-sided communication strategy is achievable with the Message Passing Interface (MPI), as of MPI-3, but only at the cost of substantial additional complexity and programmer effort. An advantage of one-sided communication is that whenever hardware support for remote direct memory access (RDMA) is present, the remote PE need not be involved in the data transfer at all, both at the program and operating system (OS) level [53]. Computations on the remote image

[‡‡‡] Automatic Differentiation (1st order) for Fortran 95. `https://github.com/pv/adjac`

may proceed uninterrupted while data is delivered or retrieved, thus overlapping communication with computation.

While all PGAS languages present some access to global memory, they may differ in other, substantial ways. Some PGAS languages, such as Chapel [54] do their best to provide high-level interfaces that can hide and abstract all details about parallelism from the application programmer. While Chapel does provide a means to take more explicit control of the parallelism when required, many programs can be written in such a way that they appear no different than serial programs. Other PGAS languages, or language extensions such as XcalableMP [55], provide "global view" objects where the array objects appear to be indexed conventionally, but ranges of indices reside on physically distinct distributed memories. For example, a "global view" matrix is represented as a two-dimensional array with non-overlapping portions residing in physically distinct memories. While such a "global view" object could be emulated with CAF, the CAF programming model is closer to a more traditional single program, multiple data (SPMD) paradigm.

The parallelism in CAF derives from two key concepts: *images* and *coarrays*. When a CAF program is launched, it is replicated N times. Each copy is called an *image* and is approximately analogous to an MPI "rank." The intrinsic function `this_image()` returns the image index of the current image, allowing each image to distinguish itself, and the intrinsic function `num_images()` provides access to the total number of images, N, spawned. Each variable is local to the image, and cannot be accessed by remote images unless it is declared as a coarray. Arrays and scalars, both statically and dynamically allocated, of intrinsic or user defined type, may be declared as coarrays, with some restrictions. Coarrays differ from other variables in that they are declared with a codimension having cobounds. The codimension utilizes square brackets to distinguish it from traditional array dimensions denoted by parentheses (see Fig. 18). The number of images spawned is fixed before the program is launched, usually specified by an environment variable or a job launcher script on distributed memory systems. Once the program is launched, the number of images cannot be altered. As a consequence, the last codimension is never explicitly specified in the program and must be syntactically specified with an asterisk when a coarray variable is declared or allocated. Each image in a CAF program corresponds to a single partition of a global address space. Coarray variables are entities in this global address space that can be retrieved or modified from any partition, when a coindex is specified.

Coordination and reasoning about parallel CAF programs is achieved through the concept of *execution segments* and *image control statements*. Image control statements are either explicit or implicit coordinations and synchronizations between different images. A list of CAF image control statements is given in Table 2. Image control statements separate *execution segments*, which are the mechanism that allows reasoning about global program state. If an execution segment m on image P (P_m) is terminated by an image control statement that matches a corresponding image control statement demarcating the beginning of an execution segment n on image Q (Q_n) then execution segment P_m is said to be ordered with respect to Q_n and to precede it. In such a case, a "get," performed in segment Q_n of a coarray variable residing on image P and modified in execution segment P_m yields the expected value. Similarly a "put" of a coarray variable where segment Q_n writes to a variable residing on image P ensures that the write happens after segment P_m is finished executing. If images are *un-ordered* with respect to each-other, then the behavior of "gets" and "puts" is undefined if the coarray variable in question is also locally modified or used. In un-ordered segments there is no guarantee that a "get" of a coarray variable that is locally modified during the un-ordered segments will yield the value before or after the modification, and, similarly, a "put" of a coarray variable read locally in the segments may or may not overwrite the variable before, after, or while it is being read. Ordered segments provide a means to guarantee the state of remote coarray variables. A few lines of source code demonstrating these principles is shown in Fig. 18.

The chief benefits of choosing CAF over other parallel runtime solutions are that CAF provides a small but powerful and flexible API that is easy to learn, is architecture and implementation agnostic, and is portable and performant. The programmer is not tied to any particular architecture (shared vs. distributed memory) or implementation technology. As technologies change, the same CAF program can be run on new systems, using new network fabrics or back end transport layers without any modification required by the programmer. With a suitable implementation, such as GFortran with OpenCoarrays,[§§§] [56] it may even be possible to switch the underlying communication technology by simply re-linking the program [57]. This is useful if a bug or defect is discovered in an underlying component. CAF has a much more concise API than many other alternatives like MPI, which has hundreds of procedure calls to learn. Therefore, learning CAF is less of a burden on programmers, and enables performant one-sided communication characteristic of PGAS languages. The one-sided nature allows programmers to overlap communication and computation, thus hiding the latency associated with communication. Compilers are free to make arbitrary optimizations including out of order execution and code movement within an execution statement, so a smart enough compiler can perform some optimizations on both "puts" where the coindex appears on the left hand side of an assignment, as well as "gets" where

[§§§] OpenCoarrays: A transport layer for coarray Fortran compilers. `https://github.com/sourceryinstitute/OpenCoarrays`

Table 2 List of CAF Image Control Statements and Their Meanings.

Image Control Statement	Summary
`sync all`	A global synchronization between all images. Each image waits until all images have reached a `sync all` statement, analogous to an MPI barrier.
`sync images (` *image-set* `)`	*image-set* is a rank 1 integer array of distinct image indices or an asterisk to denote all other images. This provides coordination between one or more individual images.
`lock` or `unlock`	User defined coordination using `lock_type` provided by `iso_fortran_env`.
`critical` and `end critical`	Serialize the execution segment defined between `critical` and `end critical`.
Coarray allocation & deallocation	Global synchronization when a coarray variable is allocated or deallocated, either explicitly via `allocate` and `deallocate` or when a coarray variable goes out of scope.
`sync memory`	Ensure global memory views are consistent when using some user defined segment ordering.
`event wait` and `event post`	An implementation of counting semaphores using the `event_type` provided by `iso_fortran_env`.
`end program`, `stop`, `error stop`	Terminate program execution.

```
! ...
real(wp) :: a, x, local_var1, local_var2 ! variables local to the image
real(wp),codimension[*] :: co_var1, co_var2 ! coarray variables

! ...
! Perform some local computations with coarray variables on all images
co_var1 = a*x + co_var2
! Use the sync images image control statement to coordinate image P and image Q
if (this_image() == P) then
   sync images(Q) ! Image P waits for image Q, end segment P_m
else if (this_image() == Q) then
   sync images(P) ! Image Q waits for image P, begin segments Q_n and P_n
   local_var1 = co_var1[P] ! "get" co_var1 from image P defined in segment P_m
   ! do some work with local_var1
   co_var2[P] = local_var1 ! "put" co_var2 without any danger of overwriting the value needed in segment P_m
end if
! Problematic un-ordered assigment:
co_var2 = local_var2 ! Causes ambiguous value on image P
! co_var2 on image P may have the value local_var2 OR the value local_var1 from image Q due to assignment of
! co_var2 residing on P from unordered execution segments P_n and Q_n
! ...
```

Fig. 18 Code Snippet Demonstrating Execution Segment Ordering and Basic Coarray Syntax. Execution segment P_m uses local coarray variable `co_var2` to compute a value assigned to local coarray variable `co_var1`. A `sync images` image control statement is then used to ensure that image Q can "get" the correct value `co_var1` from image P after the computation in segment P_m has taken place, and that segment Q_n does not prematurely overwrite `co_var2` while the old value is still needed on image P when a "put" is performed by segment Q_n.

the coindex appears on the right hand side of an assignment. Necessarily, due to the semantics of the language, execution cannot proceed until the remote data transfer has completed in the case of a "get." While, in the case of a "get," the compiler may be able to initialize the transfer earlier in the current execution segment, with a "put" the local image sends the coarray data to the remote image, and is immediately able to proceed once the data has been dispatched. Therefore it is likely that, in cases were data motion is a bottleneck, improved parallel scaling is achievable by preferring "puts" over "gets" and "sending" the data via an assignment with a coindexed variable on the right-hand side as early in the computation as possible.

The coarray syntax has some additional benefits over other paradigms. The square bracket coindex notation serves as a visual cue to indicate whenever communication is occurring between images. When a coarray variable appears without square brackets, it is a local reference and may be treated as any other local object. This makes the parallelization of serial Fortran programs straightforward, since syntactic modifications required to create a coarray variable from an extant one are limited to the variable declaration, explicit allocation (where applicable) and any places where references to the variable on a remote image are required. Otherwise, the variable may be treated as unchanged when performing local computations, and passed to procedures without modification, so long as the procedures do not require remote coarray references.

Some of the challenges when parallelizing programs with coarrays include restrictions on what types of variables may be coarrays. In particular, no references to polymorphic subobjects of a coindexed object, or to a coindexed object that has a polymorphic allocatable subcomponent are allowed. Furthermore, coarray derived types may have type-bound procedures and procedure pointers, but procedure pointer references through coindexed objects are not allowed. (I.e., procedures cannot be called via a procedure pointer of a coindexed object, since the procedure being called may be different, or undefined between images.) Furthermore, for allocatable coarray arrays and components, the allocation status and shape must agree to avoid implicit reallocation on assignment.

Despite these limitations, CAF is a convenient way to parallelize extant serial Fortran programs, to quickly prototype parallel programs that may later be implemented with a more complicated parallel runtime implementation, and to write portable and performant new parallel applications. Like any programming language or paradigm, CAF is better suited for some problems than others, but is general and flexible enough that its performance oriented design and ease of use make it an attractive candidate for a wide set of problems.

IV. Refactoring Legacy Code

Modern Fortran is almost entirely backward compatible with Fortran 77 (exceptions include various deleted features from the language, but in practice, most compilers still allow them to be used, although special compiler flags may be necessary). Thus, many, if not most, legacy Fortran 77 codes can be used without modification in modern applications (e.g., the Copernicus use of SNOPT and SPICELIB). However, refactoring can often be advantageous, improving readability as well as laying the groundwork for other modernizations, as demonstrated in this paper and others [58, 59]. A simple example of a refactored legacy subroutine is shown in Fig. 19. This code is a basic binary search routine of a sorted integer array taken from the NASTRAN program originally written in the early 1970s [60]. The original is classic "spaghetti code" and very difficult to follow, while the modernized version is quite straightforward. While the fixed-form to free-form conversion can be done with any number of automated tools that are available, unravelling the "spaghetti" can take more effort (although there are tools to automate this process as well, such as the commercial SPAG tool for Fortran code restructuring, which was used for this example).

V. Interoperability with Other Programming Languages

Modern object-oriented Fortran can be integrated with other programming languages (such as C, C++, and Python) through the standardized C-interoperability language feature. An example of this is shown in Fig. 20a. This module provides a C interface to the geopotential gravity module that can be called, for example, from Python (as shown in Fig. 20b). This technique uses a private `container` type in the Fortran module that exists only to contain the gravity model class (since it uses an unlimited polymorphic `class(*),pointer` variable it can be used to contain any variable). A C pointer to a `container` variable can be passed back to the C code. Access to the class methods is achieved using this pointer as a subroutine argument. When the Fortran routine is called, the gravity model in the container is accessed and its methods can be called. Interoperability with C also allows Fortran to make use of external libraries that have a C interface (e.g., system calls or GUI toolkits). Current Copernicus development work includes rewriting the GUI in Python using the PyQt toolkit, requiring heavy use of the interoperability of variables and procedures.

```
      SUBROUTINE BISLOC (*,ID,ARR,LEN,KN,JLOC)
C-----
C BINARY SEARCH - LOCATE KEY WORD 'ID' IN ARRAY 'ARR', 1ST ENTRY
C IF FOUND, 'JLOC' IS THE MATCHED POSITION IN 'ARR'
C IF NOT FOUND, NON-STANDARD RETURN
C
C I.E.
C ID  = KEY WORD TO MATCH IN ARR.      MATCH AGAINST 1ST COL OF ARR
C ARR = ARRAY TO SEARCH.
C       ARR(ROW,COL)
C LEN = LENGTH OF EACH ENTRY IN ARRAY.
C       LEN=ROW
C KN  = NUMBER OF ENTRIES IN THE ARR.
C       KN =COL
C JLOC= POINTER RETURNED - FIRST WORD OF ENTRY. MATCHED ROW
C-----
C
      INTEGER  ARR(1)
      DATA     ISWTCH / 16 /
C
      JJ = LEN - 1
      IF (KN .LT. ISWTCH) GO TO 120
      KLO = 1
      KHI = KN
   10 K   = (KLO+KHI+1)/2
   20 J   = K*LEN - JJ
      IF (ID-ARR(J)) 30,90,40
   30 KHI = K
      GO TO 50
   40 KLO = K
   50 IF (KHI-KLO -1) 100,60,10
   60 IF (K .EQ. KLO) GO TO 70
      K   = KLO
      GO TO 80
   70 K   = KHI
   80 KLO = KHI
      GO TO 20
   90 JLOC = J
      RETURN
  100 JLOC = KHI*LEN - JJ
      J    = KN *LEN - JJ
      IF (ID .GT.ARR(J)) JLOC = JLOC + LEN
  110 RETURN 1
C
C     SEQUENTIAL SEARCH MORE EFFICIENT
C
  120 KHI = KN*LEN - JJ
      DO 130 J = 1,KHI,LEN
      IF (ARR(J)-ID) 130,90,140
  130 CONTINUE
      JLOC = KHI + LEN
      GO TO 110
  140 JLOC = J
      GO TO 110
      END
```

(a) Fortran 66 Binary Search Routine from NASTRAN.

```
pure function bisloc(id,arr) result(jloc)

!! binary search of a sorted array.

implicit none

integer,intent(in) :: id
    !! key word to match in `arr`
integer,dimension(:),intent(in) :: arr
    !! array to search (it is
    !! assumed to be sorted)
integer :: jloc
    !! the first matched index in 'arr'
    !! (if not found, 0 is returned)

integer :: j,k,khi,klo,n
integer,parameter :: iswtch = 16

n = size(arr)
jloc = 0
if ( n<iswtch ) then
    ! sequential search more efficient
    do j = 1 , n
        if ( arr(j)==id ) then
            jloc = j
            return
        else if (arr(j)>id) then
            return ! error
        end if
    end do
else
    klo = 1
    khi = n
    k = (klo+khi+1)/2
    do
        j = k
        if ( id<arr(j) ) then
            khi = k
        else if ( id==arr(j) ) then
            jloc = j
            return
        else
            klo = k
        end if
        if ( khi-klo<1 ) then
            return ! error
        else if ( khi-klo==1 ) then
            if ( k==klo ) then
                k = khi
            else
                k = klo
            end if
            klo = khi
        else
            k = (klo+khi+1)/2
        end if
    end do
end if

end function bisloc
```

(b) Modern Fortran Binary Search Routine.

Fig. 19 Example of Refactoring Legacy Code into Modern Fortran. The original unstructured code is very hard to follow, while the modern version is very straightforward. GOTO statements and line numbers are almost entirely unnecessary in modern Fortran.

```fortran
module c_interface_module

use iso_c_binding
use geopotential_module
implicit none
private

type :: container
    class(*),pointer :: model
end type container

contains

subroutine c_ptr_to_f_string(cp,fstr)
!! Convert a c_ptr to a string into a Fortran string
type(c_ptr),intent(in) :: cp
character(len=:),allocatable,intent(out) :: fstr
integer :: ilen  !! string length
ilen = strlen(cp) !! C library function
block
    character(kind=c_char,len=ilen+1),pointer :: s
    call c_f_pointer(cp,s)
    fstr = s(1:ilen) ! exclude the '\0' char
end block
end subroutine c_ptr_to_f_string

function initialize(gravfile,n,m) &
    result(cp) bind(c,name='initialize')
!! Initialize the gravity model
type(c_ptr),intent(in),value :: gravfile
integer(c_int),intent(in),value :: n  !! degree
integer(c_int),intent(in),value :: m  !! order
type(c_ptr) :: cp
type(container),pointer :: grav_container
class(geopotential_model),pointer :: grav
logical :: status_ok
character(len=:),allocatable :: gravfile_f
allocate(grav_container)
allocate(geopotential_model_pines :: grav_container%
    model)
select type (g => grav_container%model)
class is (geopotential_model_pines)
    call c_ptr_to_f_string(gravfile,gravfile_f)
    call g%initialize(gravfile_f,n,m,status_ok)
    cp = c_loc(grav_container)
end select
end function initialize

subroutine destroy(cp) bind(c,name='destroy')
!! Destroy the gravity model
type(c_ptr),intent(in),value :: cp
type(container),pointer :: grav_container
call c_f_pointer(cp,grav_container)
select type (g => grav_container%model)
class is (geopotential_model)
    call g%destroy()
end select
deallocate(grav_container)
end subroutine destroy

subroutine get_acc(cp,rvec,acc) bind(c,name='get_acc')
!! Compute the acceleration vector
type(c_ptr),intent(in),value :: cp
real(c_double),dimension(3),intent(in) :: rvec
real(c_double),dimension(3),intent(out) :: acc
type(container),pointer :: grav_container
call c_f_pointer(cp,grav_container)
select type (g => grav_container%model)
class is (geopotential_model)
    call g%get_acc(rvec,acc)
end select
end subroutine get_acc

end module c_interface_module
```

(a) Example of C Interoperability. The main procedures here are `initialize()`, `get_acc()`, and `destroy()`. These routines are marked as interoperable with C using the `bind` attribute.

```python
from ctypes import *

grav = CDLL('libgrav.so')  # c_interface_module compiled as a shared library

initialize = grav.initialize  # define all the functions and their arguments
initialize.argtypes = [c_char_p,c_int,c_int]
initialize.restype = POINTER(c_int)
get_acc = grav.get_acc
get_acc.argtypes = [POINTER(c_int),POINTER(c_double),POINTER(c_double)]
get_acc.restype = None
destroy = grav.destroy
destroy.argtypes = [POINTER(c_int)]
destroy.restype = None

VecType = c_double*3 # 3x1 vector
gravfile = c_char_p(b'GGM03C.GEO') # gravity coefficient file to load
n = c_int(8)  # degree
m = c_int(8)  # order
acc = VecType(0.0,0.0,0.0)
rvec = VecType(10000.0, 10000.0, 10000.0) # point to evaluate the gravity field

# now we can call the Fortran procedures:
cp = initialize(gravfile,n,m)
get_acc(cp,rvec,acc)
print(acc[0],acc[1],acc[2])
destroy(cp)
```

(b) Example of Calling the Module from Python. The Python `ctypes` module provides a mechanism for calling functions in shared libraries. It is used here to wrap the object-oriented Fortran code in a Python interface.

Fig. 20 Example Use of Fortran Code in Python. The C-Interoperability feature of modern Fortran provides a standardized way to interface with C (and thus other languages compatible with C such as Python).

```
subroutine constraint_violations(me,x,f,funcs_to_compute)

!! Compute the constraint violation vector for the mission.

implicit none

class(mission_type),intent(inout) :: me
real(wp),dimension(:),intent(in)  :: x !! opt var vector for the mission [n]
real(wp),dimension(:),intent(out) :: f !! constraint violation vector for the mission [m]
integer,dimension(:),intent(in),optional :: funcs_to_compute !! the indices of f to compute

integer :: i  !! counter
integer,dimension(:),allocatable :: isegs !! segments to be propagated

call me%put_x_in_segments(x) ! extract data from the opt var vector and populate the segments
call me%segs_to_propagate(funcs_to_compute,isegs) ! get the list of segments that needs to be propagated
do i = 1, size(isegs)  ! propagate the segments:
    call me%segs(isegs(i))%propagate()
end do
call me%get_problem_arrays(f=f) ! mission class procedure to compute the constraint violations

end subroutine constraint_violations
```

Fig. 21 NRHO Problem Function. The input to this function is the control vector (x) and (optionally) the indices of the function vector that need to be computed. All variables and methods are contained within the `mission_type` class, which includes the segment array (`segs`). All the necessary segments are propagated and the constraint violations are then computed.

VI. Test Case: NRHO Solver

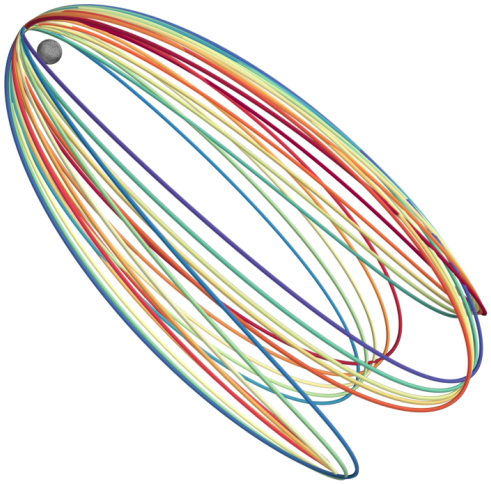
In addition to general trajectory optimization tools such as Copernicus, it is often advantageous to build stand-alone tools that solve only one problem (or variants of a problem). In many cases, this allows a reduction in overhead that may be present in a more comprehensive tool, which can be significant if the problem is to be solved numerous times. Using the various open source libraries described here (as well as other algorithms from Copernicus), a stand-alone solver was created for the computation of long-term ballistic Near Rectilinear Halo Orbits (NRHOs) in the Earth-Moon system, using the "forward/backward shooting" algorithm described in Reference [61].

In this solver (which is planned to be released publicly at a later date), the mission segments are extensions of the abstract DDEABM integration class. The segment class contains all the data necessary to propagate the segment by the integration method (t_0, t_f, x_0, the force model, etc.). The Fortran Astrodynamics Toolkit is used to define the force model, which is an 8×8 GRAIL model [62] for the Moon, with the Earth and Sun included as pointmass third bodies. The equations of motion for this problem are shown in Fig. 15. The object-oriented JPLEPH ephemeris is used (each segment contains an instance of the force model and ephemeris). Gradients are computed by finite differences using the NumDiff library (graph coloring is used to efficiently compute the Jacobian with as few function evaluations as possible). Finally, a basic differential corrector (see Fig. 16) is used to ensure state continuity (the CR3BP solution is used as an initial guess for the ephemeris model).

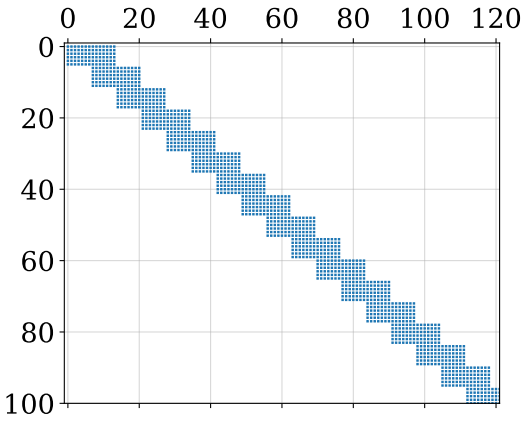
A simplified input file for this tool is shown in Fig. 2a, which includes the orbit size (defined by the initial lunar periapsis radius), initial epoch, and number of revolutions to compute. Fig. 21 shows the constraint violation function passed to the solver. Note that the mission segments are defined as an array of `segs` in the main `mission_type` class, and are propagated serially in this example (lines 17–19). Using task-based parallelism to propagate mission segments whose data dependencies have already been satisfied has not yet been implemented, but provides an appealing opportunity for accelerating the convergence time for missions with many segments—even when complex segment dependencies are present. Section VII discusses two possible implementation strategies. An example output of the NRHO solver is shown in Fig. 22.

VII. Proposed Task-Based Parallelization of Mission Segments

In Copernicus, or in the NRHO test case described in Section VI, an opportunity for coarse grained parallelism exists in the mission segment propagation. For missions with many segments, some may be computed concurrently rather than looping through the topologically sorted segments sequentially as discussed above. For missions with many segments this may provide a large speedup in the time to solution, if the dependencies provide sufficient opportunity for concurrent computation.



(a) NRHO Solution (Moon-Centered J2000 Frame). This is 20 revolutions of a ballistically propagated L_2 4,500 km r_p (southern family) NRHO.



(b) Jacobian Sparsity Pattern Structure. The squares represent the locations of the non-zero elements of the Jacobian matrix for the first 4 revs of the problem.

Fig. 22 NRHO Example Solution. For the 20 rev case, there are 160 trajectory segments, 567 control variables, and 480 equality constraints. The Jacobian is quite sparse, and so the problem is amenable to sparsity pattern partitioning and parallelization. When evaluating specific elements of the function vector during computation of the Jacobian, only the segments that those functions depend on need to be propagated (see Fig. 21).

We propose two possible strategies to implement this proposed task-based parallelism using CAF. Both approaches involve images taking responsibility for one or more propagation segments, as depicted in Fig. 7. The first strategy is the easiest and fastest to implement, but likely will not scale well to very large numbers of segments and requires that the total number of segments be known when the program is launched. This approach can be implemented with Fortran 2008 features that are currently supported by the Intel, GCC, and Cray Fortran compilers. The second proposed strategy is the most flexible, robust and generic, however it relies on features from TS 18508 [63] which will become part of the Fortran 2018 standard that are less widely supported at the present time.

The data dependencies between segments to propagate can be represented by a DAG, including the one depicted in Fig. 7. Currently, topological sorting is used to serialize the DAG; however, it is evident from Fig. 7 certain segments may not have any dependencies at all, or multiple other segments may have one or more common dependencies. As soon as all of the data dependencies of a segment are met, nothing is preventing it from being propagated. In the NRHO problem function source listing shown in Fig. 21, `isegs` represents the topologically sorted order required to propagate the dependencies, and the `do` loop on lines 17–19 is executed serially. Both proposed task-based parallelism strategies would distribute the work in this loop among multiple CAF images, allowing segments with satisfied data dependencies to be propagated concurrently.

The first, and most naive, of these implementation strategies is to simply allocate the number of images equal to the number of segments when the program is launched. Outside of the work done by the `constraint_violations()` function the rest of the program can be wrapped in `if (this_image() == 1)` statements. This prevents IO from being duplicated by each image and prevents interactions with GUI front ends, etc. from being corrupted. Image one can then broadcast the required initialization and setup data to all the other images using the intrinsic `co_broadcast()` function. Before each image starts their propagation, they must examine the segment propagation DAG and compute the parent nodes representing segments on which they depend, if any, and the child nodes that cannot begin work until the current node finishes propagating its segment. Once the broadcast of initial data from the master image is complete, and each node has computed its parents and children in the data dependency graph, segment propagation may begin. To start, each image issues a `sync images()` call if any data dependencies exist, with the list of participating images equal to the list of parent segments on which the current segment depends. After the sync images statement, the segment is propagated on the image, assuming any required data was placed into the image's memory with a "put." Once the image has finished its segment propagation computations, it sends the data required by child dependents to the list of images

corresponding to dependent segments with a "put." After sending the data, a final `sync images()` is called with the list of dependent image segments.

The benefits of this design are the simplicity of the approach, and the natural expression of data dependencies using more commonly available CAF features. Even if the number of mission segments are more numerous than PEs available on a system, the negative performance impact should be limited if the two following conditions are met: First, the dependencies are structured in such a way that as many—or only a few more—segments may be concurrently propagated as there are available PEs. Second, that there is enough available space in memory to accommodate each instance of a coarray variable associated with an image. So long as these two conditions are met, even on grossly oversubscribed systems, each physical PE will only be actively performing computations for one or fewer segments most of the time. The drawbacks of this approach are that the program must be launched knowing the number of segments in advance so that the correct number of images are spawned, and, for large problems where there are many more segments than available PEs, unnecessary replication of coarray variables—one per image—may exhaust the available system memory.

The second, more sophisticated approach is to use a traditional master/worker paradigm. This is not completely dissimilar to the first approach but it is more complicated to implement and can most naturally be expressed using events, a feature introduced in TS 18508 and included in the forthcoming release of the Fortran 2018 standard. Rather than mapping each mission segment to a unique CAF image, the first image is designated the "master" and controls a pool of the remaining "worker" images. As in the naive example, image one is responsible for all of the IO, setup, initialization and cleanup. If the Fortran 2018 teams feature is available, a master team containing only image one and a worker team with the remaining images can be created. This is mostly a syntactic convenience for the current case and the same behavior may be easily implemented using the technique outlined in the previous approach by wrapping the appropriate code sections in `if` statements to detect if the current image is the master image or a worker image. The master image is then responsible for sending individual mission segments to the worker images, and ensuring that those workers have the data required from previous segments and execute in the correct order. Events are used to ensure proper execution segment ordering between the master and worker images and to communicate when a segment is ready to be propagated and when it is finished being propagated.

This approach is much more generic and robust, since any number of images may be specified when the program starts, and a problem of any size will be able to be loaded and run after the program has begun execution. Furthermore, this approach allows incremental optimizations to be made to the scheduling algorithm responsible for analyzing the mission segment DAG and dispatching work to the workers. For example, the simplest implementation could naively loop through ready mission segments on the master image, dispatching them to the next worker that finishes its previously assigned segment. Once a basic implementation has been implemented, further optimizations can be made. For example, linear sections of segments with only one parent and one child can all be sent at once to the same worker image. Once the worker image has received the data associated with the first segment to be propagated it may begin computations, even if data associated with subsequent segments is still in transit, and intermediate coordination between the master and worker can be eliminated. If some measure of the cost of each segment can be estimated, a more sophisticated graph partitioning algorithm or library, such as METIS[¶¶¶] [64], could be used to dispatch multiple interdependent subgraph clusters to each worker image.

VIII. Conclusion

Various algorithms and their implementation in modern Fortran are shown. The modern programming concepts (such as dynamic data structures, polymorphism, and parallelization) available in Fortran can be very useful in the creation of spacecraft trajectory design and optimization tools. It is also shown how refactoring can breathe new life into legacy Fortran 77 code without a total rewrite in another programming language. Copernicus, originally designed in the early 2000s, has made great use of the new capabilities of modern Fortran as they have become available. A stand-alone NRHO solver has also been developed using many of the basic algorithms described in this paper, demonstrating how the codes can be used in a very flexible and modular way. Finally a task-based parallelization method using Coarray Fortran (CAF) is proposed that could be used to speed up convergence for complex missions.

Funding Sources

This work was partially funded by NASA JSC under contract NNJ13HA01C.

[¶¶¶] METIS – Serial Graph Partitioning and Fill-reducing Matrix Ordering. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

Acknowledgments

The authors wish to thank the members of the small but dedicated Fortran internet-based user community (especially the *comp.lang.fortran* user group, the Intel Fortran forum, GitHub and Gitter) who have been such a great resource for learning the language.

References

[1] Backus, J., "The History of FORTRAN I, II and III," *Annals of the History of Computing*, Vol. 1, No. 1, 1979, pp. 21–37.

[2] Metcalf, M., Reid, J., and Cohen, M., *Modern Fortran Explained*, 4th ed., Oxford University Press, Inc., New York, NY, USA, 2011.

[3] Metcalf, M., "The Seven Ages of Fortran," *Journal of Computer Science & Technology*, Vol. 11, No. 1, 2011.

[4] Reid, J., "The New Features of Fortran 2003," *SIGPLAN Fortran Forum*, Vol. 26, No. 1, 2007, pp. 10–33.

[5] Reid, J., "The New Features of Fortran 2008," *SIGPLAN Fortran Forum*, Vol. 33, No. 2, 2014, pp. 21–37.

[6] Reid, J., "The New Features of Fortran 2015," *SIGPLAN Fortran Forum*, Vol. 36, No. 2, 2017, pp. 3–28.

[7] Rouson, D., Xia, J., and Xu, X., *Scientific Software Design: The Object-Oriented Way*, 1st ed., Cambridge University Press, New York, NY, USA, 2011.

[8] Markus, A., *Modern Fortran in Practice*, Cambridge University Press, New York, NY, USA, 2012.

[9] Hanson, R. J., and Hopkins, T., *Numerical Computing With Modern Fortran*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2013.

[10] Haveraaen, M., Morris, K., Rouson, D., Radhakrishnan, H., and Carson, C., "High-Performance Design Patterns for Modern Fortran," *Sci. Program.*, Vol. 2015, 2015, pp. 3:3–3:3.

[11] Battin, R., *An Introduction to the Mathematics and Methods of Astrodynamics*, AIAA Education Series, American Institute of Aeronautics & Astronautics, 1999.

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

[13] Wahbah, M. M., Berning, M. J., and Choy, T. S., "Simulation of Airplane and Rocket Trajectories," NASA Tech Brief MSC-20933, Jul. 1987.

[14] Riehl, J. P., Sjauw, W. K., Falck, R. D., and Paris, S. W., "Trajectory Optimization: OTIS 4," NASA Tech Brief LEW-18319-1, Apr. 2010.

[15] Sims, J., Finlayson, P., Rinderle, E., Vavrina, M., and Kowalkowski, T., "Implementation of a Low-Thrust Trajectory Optimization Algorithm for Preliminary Design," AIAA/AAS Astrodynamics Specialist Conference, Aug. 2006. AIAA 2006-6746.

[16] Whiffen, G. J., "Mystic: Implementation of the Static Dynamic Optimal Control Algorithm for High-Fidelity, Low-Thrust Trajectory Design," AIAA/AAS Astrodynamics Specialist Conference, Aug. 2006. AIAA 2006-6741.

[17] Lugo, R. A., Shidner, J. D., Powell, R. W., Marsh, S. M., Hoffman, J. A., Litton, D. K., and Schmitt, T. L., "Launch Vehicle Ascent Trajectory Simulation Using the Program to Optimize Simulated Trajectories II (POST2)," AAS/AIAA Space Flight Mechanics Meeting, Feb. 2017. AAS 17-274.

[18] Evans, S., Taber, W., Drain, T., Smith, J., Wu, H.-C., Guevara, M., Sunseri, R., and Evans, J., "MONTE: The Next Generation of Mission Design & Navigation Software," *The 6th International Conference on Astrodynamics Tools and Techniques (ICATT)*, 2016.

[19] Hatfield, J. N., "CATO (Computer Algorithm for Trajectory Optimization): An Implementation of Fortran 95 Object-based Programming," *SIGPLAN Fortran Forum*, Vol. 22, No. 1, 2003, pp. 2–7.

[20] Williams, J., Senent, J. S., and Lee., D. E., "Recent Improvements to the Copernicus Trajectory Design and Optimization System," *Advances in the Astronautical Sciences*, Vol. 143, 2012. AAS 12-236.

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

[22] Blevins, J. R., "A Generic Linked List Implementation in Fortran 95," *SIGPLAN Fortran Forum*, Vol. 28, No. 3, 2009, pp. 2–7.

[23] Williams, J., "A New Architecture for Extending the Capabilities of the Copernicus Trajectory Optimization Program," *Advances in the Astronautical Sciences: Astrodynamics 2015*, Vol. 156, 2016. AAS 15-606.

[24] Dawn, T. F., Gutkowski, J. P., Batcha, A. L., Williams, J., and Pedrotty, S. M., "Trajectory Design Considerations for Exploration Mission 1," AIAA/AAS Space Flight Mechanics Meeting, Jan. 2017.

[25] Pines, S., "Uniform Representation of the Gravitational Potential and its Derivatives," *AIAA Journal*, Vol. 11, 1973, pp. 1508–1511.

[26] Eckman, R. A., Brown, A. J., and Adamo, D. R., "Normalization and Implementation of Three Gravitational Acceleration Models," Tech. Rep. TP-2016-218604, NASA, Jun. 2016.

[27] Spencer, J., "Pines' Nonsingular Gravitational Potential Derivation, Description, and Implementation," Tech. Rep. NASA-CR-147478, NASA, Feb. 1976.

[28] Werner, R. A., and Scheeres, D. J., "Exterior Gravitation of a Polyhedron Derived and Compared with Harmonic and Mascon Gravitation Representations of Asteroid 4769 Castalia," *Celestial Mechanics and Dynamical Astronomy*, Vol. 65, No. 3, 1996, pp. 313–344.

[29] Williams, J., "POLYGRAV: Polyhedral Gravity Model Fortran Code (v1.0)," Tech. Rep. JETS-JE23-15-AFGNC-DOC-0034, JSC Engineering, Technology and Science (JETS) Contract, Apr. 2015.

[30] Radhakrishnan, K., and Hindmarsh, A. C., "Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations," Report UCRL-ID-113855, Lawrence Livermore National Laboratory, 1993.

[31] Brown, P. N., Byrne, G. D., and Hindmarsh, A. C., "VODE: A Variable-Coefficient ODE Solver," *SIAM J. Sci. Stat. Comput.*, Vol. 10, No. 5, 1989, pp. 1038–1051.

[32] Shampine, L. F., and Watts, H. A., "DEPAC – Design of a User Oriented Package of ODE Solvers," Tech. Rep. SAND-79-2374, Sandia National Labs, Sep. 1980.

[33] Brent, R. P., "An Algorithm with Guaranteed Convergence for Finding a Zero of a Function," *The Computer Journal*, Vol. 14, No. 4, 1971, pp. 422–425.

[34] "The SPICE Toolkit," `https://naif.jpl.nasa.gov/naif/toolkit.html`, Apr. 2017.

[35] Arora, N., and Russell, R. P., "A Fast, Accurate, and Smooth Planetary Ephemeris Retrieval System," *Celestial Mechanics and Dynamical Astronomy*, Vol. 108, No. 2, 2010, pp. 107–124.

[36] JPL, "Instructions for Ephemeris File Access Through Fortran Programs," `ftp://ssd.jpl.nasa.gov/pub/eph/planets/fortran/userguide.txt`, Mar. 2013.

[37] "HSL: A Collection of Fortran Codes for Large Scale Scientific Computation," `http://www.hsl.rl.ac.uk/`, Feb. 2011.

[38] Kraft, D., "Algorithm 733: TOMP–Fortran Modules for Optimal Control Calculations," *ACM Trans. Math. Softw.*, Vol. 20, No. 3, 1994, pp. 262–281.

[39] Gill, P. E., Murray, W., and Saunders, M. A., "SNOPT: An SQP Algorithm For Large-Scale Constrained Optimization," *SIAM Journal on Optimization*, Vol. 12, 1997, pp. 979–1006.

[40] Wächter, A., and Biegler, L. T., "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming," *Mathematical Programming*, Vol. 106, No. 1, 2006, pp. 25–57.

[41] Parker, J., and Anderson, R., *Low-Energy Lunar Trajectory Design*, JPL Deep-Space Communications and Navigation Series, Wiley, 2014.

[42] Moré, J. J., Garbow, B. S., and Hillstrom, K. E., "User Guide for MINPACK-1," Tech. Rep. ANL-80-74, Argonne National Laboratory, Aug. 1980.

[43] Engeln-Müllges, G., and Uhlig, F., *Numerical Algorithms with Fortran*, Springer-Verlag, 1996.

[44] Stepleman, R. S., and Winarsky, N. D., "Adaptive Numerical Differentiation," *Mathematics of Computation*, Vol. 33, 1976, pp. 1257–1264.

[45] Oliver, J., "An Algorithm for Numerical Differentiation of a Function of One Real Variable," *Journal of Computational and Applied Mathematics*, Vol. 6, No. 2, 1980, pp. 145 – 160.

[46] Coleman, T. F., Garbow, B. S., and Moré, J. J., "Algorithm 618: FORTRAN Subroutines for Estimating Sparse Jacobian Matrices," *ACM Trans. Math. Softw.*, Vol. 10, No. 3, 1984, pp. 346–347.

[47] Hadjidoukas, P., Angelikopoulos, P., Voglis, C., Papageorgiou, D., and Lagaris, I., "NDL-v2.0: A New Version of the Numerical Differentiation Library for Parallel Architectures," *Computer Physics Communications*, Vol. 185, No. 7, 2014, pp. 2217 – 2219. URL `http://www.sciencedirect.com/science/article/pii/S0010465514001258`.

[48] Martins, J. R. R. A., Kroo, I. M., and Alonso, J. J., "An Automated Method for Sensitivity Analysis Using Complex Variables," *Proceedings of the 38th AIAA Aerospace Sciences Meeting*, Reno, NV, 2000. AIAA 2000-0689.

[49] Yu, W., and Blair, M., "DNAD, a Simple Tool for Automatic Differentiation of Fortran Codes Using Dual Numbers," *Computer Physics Communications*, Vol. 184, No. 5, 2013, pp. 1446 – 1452.

[50] Bailey, D. H., "A Thread-Safe Arbitrary Precision Computation Package (Full Documentation)," Tech. rep., Mar. 2017. `http://www.davidhbailey.com/dhbsoftware/`.

[51] Numrich, R. W., and Reid, J., "Co-array Fortran for Parallel Programming," *SIGPLAN Fortran Forum*, Vol. 17, No. 2, 1998, pp. 1–31.

[52] Shterenlikht, A., Margetts, L., Cebamanos, L., and Henty, D., "Fortran 2008 Coarrays," *SIGPLAN Fortran Forum*, Vol. 34, No. 1, 2015, pp. 10–30.

[53] Shan, H., Wright, N. J., Shalf, J., Yelick, K., Wagner, M., and Wichmann, N., "A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI," *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40, No. 2, 2012, pp. 92–98.

[54] Chamberlain, B. L., Callahan, D., and Zima, H. P., "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, Vol. 21, No. 3, 2007, pp. 291–312.

[55] Nakao, M., Lee, J., Boku, T., and Sato, M., "Productivity and Performance of Global-View Programming with XcalableMP PGAS Language," *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, IEEE Computer Society, 2012, pp. 402–409.

[56] Fanfarillo, A., Burnus, T., Cardellini, V., Filippone, S., Nagle, D., and Rouson, D., "OpenCoarrays: Open-Source Transport Layers Supporting Coarray Fortran Compilers," *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ACM, 2014, p. 4.

[57] Rouson, D., Gutmann, E., Friesen, B., and Fanfarillo, A., "Performance Portability of an Intermediate-Complexity Atmospheric Research Model in Coarray Fortran," *PGAS Applications Workshop (PAW)*, IEEE, 2016, pp. 25–32.

[58] Overbey, J., Xanthos, S., Johnson, R., and Foote, B., "Refactorings for Fortran and High-performance Computing," *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, ACM, New York, NY, USA, 2005, pp. 37–39.

[59] Radhakrishnan, H., Rouson, D. W. I., Morris, K., Shende, S., and Kassinos, S. C., "Using Coarrays to Parallelize Legacy Fortran Applications: Strategy and Case Study," *Sci. Program.*, Vol. 2015, 2015, pp. 2:2–2:2.

[60] "NASTRAN User Guide," Tech. Rep. NASA-CR-2504, NASA, Apr. 1975.

[61] Williams, J., Lee, D. E., Whitley, R. L., Bokelmann, K. A., Davis, D. C., and Berry, C. F., "Targeting Cislunar Near Rectilinear Halo Orbits for Human Space Exploration," AAS/AIAA Space Flight Mechanics Meeting, Feb. 2017. AAS 17-267.

[62] Lemoine, F. G., Goossens, S., Sabaka, T. J., Nicholas, J. B., Mazarico, E., Rowlands, D. D., Loomis, B. D., Chinn, D. S., Caprette, D. S., Neumann, G. A., Smith, D. E., and Zuber, M. T., "High-Degree Gravity Models from GRAIL Primary Mission Data," *Journal of Geophysical Research: Planets*, Vol. 118, No. 8, 2013, pp. 1676–1698.

[63] ISO, *ISO/IEC TS 18508:2015: Information Technology – Additional Parallel Features in Fortran*, International Organization for Standardization, Geneva, Switzerland, 2015.

[64] Karypis, G., and Kumar, V., "Multilevel k-way Partitioning Scheme for Irregular Graphs," *Journal of Parallel and Distributed Computing*, Vol. 48, No. 1, 1998, pp. 96–129.