Invoking Cspice Library Functions Via Swift Code

1.0 Introduction

The Spacecraft Planet Instrument C-matrix Events (SPICE) information system consists of software, together with supporting data kernels, "to assist NASA scientists in planning and interpreting scientific observations from space-borne instruments, and to assist NASA engineers involved in modeling, planning and executing activities needed to conduct planetary exploration missions."[1] This system is developed, maintained, and supported by Navigation and Ancillary Information Facility (NAIF) personnel at the Jet Propulsion Laboratory (JPL) under direction of NASA's Planetary Science Division. A SPICE toolkit (including source code, user/programmer documentation, a linkable library, and executable utilities) is publicly available in a variety of programming languages for download to multiple compatible hardware platforms.[2]

For astrodynamicists, among the most useful of SPICE functions is access to and time-driven interpolation of ephemerides for bodies and spacecraft throughout the Solar System. These ephemerides are stored in binary Spacecraft and Planet Kernel (SPK) files.[3] This paper documents how the 64-bit macOS C-based SPICE toolkit can be incorporated into an application written with Apple Computer's new Swift programming language to obtain interpolated state vectors from a binary SPK file. Since NAIF is unaware of any other Swift-based use of SPICE as of early 2018, this paper is submitted for use by any programmers interested in this capability.

The C-based library for SPICE is ideally suited for use with Swift because Apple's Xcode integrated development environment (IDE) currently supports both Swift and Objective-C source code to facilitate transition from the latter to the former. Objective-C is a superset of the C programming language, and Xcode is capable of compiling C source along with linking C libraries. In accord with Swift convention, libraries are written with upper camel case (also known as PascalCase) characters. From loose adaptation of this convention ("SPICE" is an acronym; not a word), the C-based SPICE library is referred to as "Cspice" in this paper.

2.0 Adding Cspice to an Xcode Project

Although the author has been programming on Apple computers since 1985, his language of choice was exclusively FORTRAN until it became virtually impossible to utilize Apple's application programming interfaces with Mac OS X circa 2010[4]. Only in October 2017 was programming in Swift under Xcode initiated, so there may be better actions to invoke Cspice functions than those documented in this paper. Furthermore, Swift was announced in 2014[5] and became a rapidly evolving open source language in December 2015.[6] Regular updates to Xcode

---

[1] Reference https://naif.jpl.nasa.gov/naif/spiceconcept.html (accessed 16 January 2018).

[2] Reference https://naif.jpl.nasa.gov/naif/toolkit.html (accessed 16 January 2018).

[3] Reference the Introduction section of the NAIF Toolkit's `spk.req` required reading document.

[4] Around 2010, then-current Macintosh personal computers ran on the Mac OS X ("X" being pronounced "ten") operating system. The last OS X release under which the author could program productively with FORTRAN was Version 10.4 "Tiger". At present, the Macintosh operating system is known as macOS. Reference http://www.operating-system.org/betriebssystem/_english/bs-macos.htm (accessed 29 January 2018).

[5] Reference https://swift.org (accessed 19 January 2018).

[6] Reference https://swift.org/about/ (accessed 19 January 2018).

are also made by Apple. The following configuration applies to procedures and illustrations that follow, and it renders them perishable, likely on a timescale of a few years.

1) Operating system = macOS Version 10.12.6 "Sierra"
2) Platform = 13-inch MacBook Pro, vintage 2017
3) Processor = 3.1 GHz Intel Core i5
4) Programming language = Swift 4.0
5) IDE = Xcode Version 9.2

There is currently no plan to update this paper as the foregoing configuration becomes dated. Although the target application associated with this paper's examples is only intended to run on its MacBook Pro development platform, no known obstacles would prevent other desktop and mobile Apple devices from hosting Cspice-enabled software.

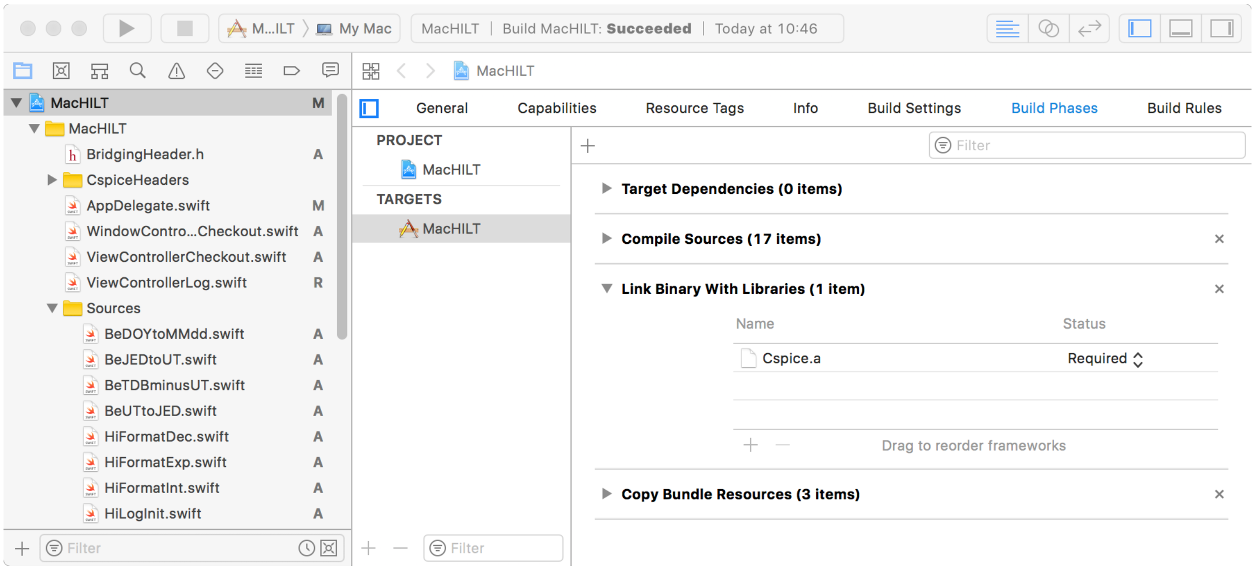
2.1 Linking Cspice to an Xcode Target Application

The Cspice library is located at `cspice/lib/cspice.a` in NAIF's current 64-bit Mac C toolkit download (named "N0066" and dated 10 April 2017). The following steps bring it into your Xcode project for linking to Swift code.

1) In the interest of traceability and conformity with Swift convention, move a copy of `cspice.a` into your Xcode project's directory tree and rename it `Cspice.a`.

2) Use Xcode's Project Navigator to select your project (typically at the top of the list) and then select the application with which to link Cspice from the resulting **TARGETS** list.

3) Select `Build Phases` in Xcode's Editor pane and open the item list for the `Link Binary With Libraries` phase. Click the + in this phase and the `Add Other…` button in the resulting panel. Using the resulting file selection panel, navigate to the `Cspace.a` file moved and renamed in your Xcode project, and click the Open button. Your Xcode display should resemble Figure 1.



Name	Status
Cspice.a	Required

Figure 1: Xcode's display after the `Cspice.a` library has been designated for linking to the MacHILT target application.

2.2 Making the Target Application's Swift Code Cspice-Aware

Even with Cspice linked into the target application, Swift code accessing SPICE functions will not compile without properly configured `.h` (C header) files in your Xcode project. This is called "exposing" the code to Cspice. Start this process by adding the `cspice/include/` directory (or a copy of it) from the NAIF toolkit download to your Xcode project. This directory contains all Cspice header files and is renamed `CspiceHeaders` in Figure 1's Project Navigator list.

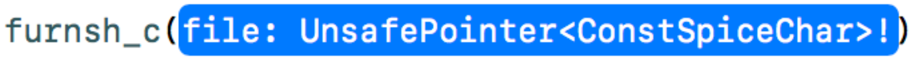
If your Xcode project has no bridging header file (that file in Figure 1 appears as the third item in the Project Navigator list), one can be created by navigating the Xcode menu bar path `File` > `New` > `File`... and selecting the target application's platform from the resulting panel (`iOS`, `watchOS`, `tvOS`, *or* `macOS`). In the `Source` pane of the panel, select `Header File`, click the `Next` button, and specify a `BridgingHeader` name.[7] Edit the `BridgingHeader.h` file to ensure the following statements are present (if other `#import` statements are present, add the one for `SpiceUsr.h` appearing below).

```
#ifndef BridgingHeader_h
#define BridgingHeader_h

#import "SpiceUsr.h"

#endif /* BridgingHeader_h */
```

---

[7] Reference https://developer.apple.com/library/content/documentation/Swift/Conceptual/BuildingCocoaApps/MixandMatch.html (accessed 17 January 2017).

Once the edited `BridgingHeader.h` file is saved in Xcode, SPICE functions appearing in Swift source code will be recognized in the editor, including its auto-completion and help functions. For example, consider the call to SPICE routine `furnsh_c` in Swift code. A single call to this function is a prerequisite to accessing an SPK file and its ephemerides. The Xcode editor appearance after typing "fur" in a new line of Swift code and pressing the return key is illustrated in Figure 2.



```
furnsh_c(file: UnsafePointer<ConstSpiceChar>!)
```

Figure 2: the Xcode editor's auto-completion prompt for a call to SPICE function `furnish_c` after `BridgingHeader.h` configuration.

The **blue** reverse-field auto-completion prompt in Figure 2 refers to the single input parameter for `furnsh_c` (there are no output parameters from this function) consisting of the SPICE kernel filename to be accessed. The "file" parameter argument's type is `ConstSpiceChar`, and the Swift compiler supports this type with a String constant such that an `UnsafePointer` is not required. For the `de430.bsp` SPK file in the current NAIF toolkit at `cspice/exe/de430.bsp`, the Swift call is as follows.

```
furnsh_c("de430.bsp")
```

To ensure an SPK file never becomes separated from an application, a good practice is to store it in the application's main bundle directory tree using the Resources subdirectory. The following Swift statements would enable this practice if placed before the call to `furnsh_c` loading the SPK file.

```
pathToResources = Bundle.main.bundlePath + "/Contents/Resources"
FileManager.default.changeCurrentDirectoryPath(pathToResources)
```

3.0 Geometric Geocentric Lunar State Vector Test Case and Results

This test case illustrates a relatively complex call to SPICE function `spkezr_c` accessing multiple ephemerides in SPK file `de431_part-2.bsp`[8] to produce geometric geocentric lunar position and velocity with components in the Earth mean equator and equinox of epoch J2000.0 (TDB Julian date = 2451545.0 days). Accuracy of this lunar state vector is assessed with respect to JPL's Horizons ephemeris server,[9] currently supported by de431 data. At the epoch associated with this test, geocentric position vector difference magnitude between de430 and de431 lunar ephemerides only amounts to 9.710E-04 km, while the corresponding velocity vector difference magnitude is 2.587E-09 km/s.[10]

---

[8] This file is downloadable from https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/de431_part-2.bsp (accessed 31 January 2018).

[9] Reference https://ssd.jpl.nasa.gov/?horizons (accessed 28 January 2018). The Horizons telnet user interface is used for this test.

[10] Differences between de430 and de431, together with associated implications to users, are documented at https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/aareadme_de430-de431.txt (accessed 31 January 2018).

The Swift editor's help function displays the following information for a call to `spkezr_c`.

```
func spkezr_c(_ target: UnsafePointer<ConstSpiceChar>!,
              _ epoch: SpiceDouble,
              _ frame: UnsafePointer<ConstSpiceChar>!,
              _ abcorr: UnsafePointer<ConstSpiceChar>!,
              _ observer: UnsafePointer<ConstSpiceChar>!,
              _ state: UnsafeMutablePointer<SpiceDouble>!,
              _ lt: UnsafeMutablePointer<SpiceDouble>!)
```

As with the `furnsh_c` call, all parameters of type `ConstSpiceChar` can be specified with Swift constants of String type. Likewise, the `epoch` parameter can be specified with a Swift constant of Double type. The `state` and `lt` parameters are also input, but their `UnsafeMutablePointer` arguments must be allocated with the appropriate number of contiguous bytes in memory to which SPICE output will be written. The `SpiceDouble` type is 8 bytes in size. Consequently, the state pointer argument must allocate 48 bytes for the 6 components of position and velocity SPICE is to store. Since only one `SpiceDouble` is associated with the `lt` pointer argument, it must allocate 8 bytes. The following Swift code sets input arguments prior to making its `spkezr_c` call.

```
let msSinceJ2K = ((epochTDBjd - epochJ2000)*msPerDay).rounded()
let secSinceJ2K = msSinceJ2K*0.001
let ptrToState = UnsafeMutablePointer<SpiceDouble>.allocate(capacity: 48)
let ptrToLtTime = UnsafeMutablePointer<SpiceDouble>.allocate(capacity: 8)
spkezr_c("Moon", secSinceJ2K, "J2000", "None", "Earth", ptrToState, ptrToLtTime)
```

The first two statements in the foregoing code segment set the argument for `epoch` and are worthy of elaboration. When the difference `epochTDBjd - epochJ2000` is computed from present-day Julian dates with values exceeding 2 million days, resulting precision is on the order of 1.E-04 s. Time skews of this magnitude in `epoch` will produce lunar geocentric position errors on the order of 1.E-04 km and velocity errors on the order of 1.E-10 km/s. Consequently, `secSinceJ2K` reflects an `epochTDBjd - epochJ2000` difference rounded to the nearest ms. This computation works well for a TDB epoch specified as a Julian date such as `epochTDBjd`. If a calendric date and time are specified instead, consider using the SPICE function `str2et_c` to generate a value for the `epoch` parameter.

Following the `spkezr_c` call, components of the lunar state may be accessed using array subscript syntax with the pertinent pointer. For example, the x-component of lunar position would be accessed as `ptrToState[0]`, and the z-component of lunar velocity would be accessed as `ptrToState[5]`. The Swift compiler is pointer-savvy enough to automatically deallocate memory assigned to primitive types like `SpiceDouble` when a pointer's context is departed for other processing. To ensure no memory leaks occur, however, the best practice is to perform these actions explicitly in Swift code. When the foregoing pointers are no longer needed, this is accomplished as follows.

```
ptrToState.deinitialize(count: 48)
ptrToState.deallocate(capacity: 48)
ptrToLtTime.deinitialize(count: 8)
ptrToLtTime.deallocate(capacity: 8)
```

When the `de431_part-2.bsp` SPK file is polled for a geometric (`abcorr` = `"None"`) geocentric lunar state at TDB Julian date = 2457935.5 days or 1.0 July 2017 TDB calendar date (`epoch` = +552139200.0 s), the following Cartesian position and velocity components are obtained. These components are followed by corresponding values from Horizons.

```
Swift Pos:     -3.872301192591970E+05 -6.408440572213630E+04 -1.202519889189460E+03 km
Horizons Pos:  -3.872301192591972E+05 -6.408440572213631E+04 -1.202519889189456E+03 km

Swift Vel:     +9.508308384876750E-02 -9.357442329133700E-01 -3.312170730265120E-01 km/s
Horizons Vel:  +9.508308384876749E-02 -9.357442329133704E-01 -3.312170730265119E-01 km/s
```

Magnitude of the vector difference between the foregoing pair of position vectors is 1.000E-11 km (equivalent to 16 decimal digits of precision), and that between the pair of velocity vectors is 9.992E-16 km/s (equivalent to 15 decimal digits of precision).

4.0 Additional SPICE Configuration Via Swift Programming

The `DEFAULT` error handling option initially invoked for SPICE processing will terminate the target application once a SPICE processing error is detected and reported.[11] In the context of MacHILT, this behavior is undesirable. Overriding `DEFAULT` error handling with the `RETURN` option (report the error and return control to the target application without further SPICE processing) is preferred for MacHILT, and Swift code required to invoke this option is a good supplemental illustration of interaction with SPICE functions.

The key SPICE function used to regulate error handling behavior is `erract_c`. A call to `erract_c` is documented as follows by the Swift editor's help function.

```
func erract_c(_ operation: UnsafePointer<ConstSpiceChar>!,
              _ lenout: SpiceInt,
              _ action: UnsafeMutablePointer<SpiceChar>!)
```

The `operation` parameter for `erract_c` can be `"Get"` or `"Set"`, depending on whether data referenced by the `action` parameter is to be polled or updated, respectively. It is the `action` parameter that points to a character string of `lenout` bytes containing the SPICE error handling option. As noted previously, Swift supports `operation`'s `ConstSpiceChar` type with String constants. The `lenout` parameter receives an integer constant if it is previously declared to be of type `SpiceInt`. More problematic is the `action` parameter because it points to character string data of type `SpiceChar`, an alias for `Int8`. For the desired `"Set"`

---

[11] Reference the section "Choosing the Error Response Action" of `cspice/doc/error.req` from the NAIF toolkit download,

in a call to `erract_c`, this `Int8` data format requires byte-by-byte specification of `RETURN` using ASCII codes beforehand. Consequently, Swift code associated with the `erract_c` call is as follows (values for ASCII codes are decimal in this implementation).

```
let outputBytes: SpiceInt = 6
let ptrToAction = UnsafeMutablePointer<SpiceChar>.allocate(capacity: 6)
ptrToAction.pointee = 82                     // R
ptrToAction.advanced(by: 1).pointee = 69    // E
ptrToAction.advanced(by: 2).pointee = 84    // T
ptrToAction.advanced(by: 3).pointee = 85    // U
ptrToAction.advanced(by: 4).pointee = 82    // R
ptrToAction.advanced(by: 5).pointee = 78    // N
erract_c("Set", outputBytes, ptrToAction)
ptrToAction.deinitialize(count: 6)
ptrToAction.deallocate(capacity: 6)
```

5.0 Handling C Macros

A necessary condition for successfully accessing a state vector in a specified SPK-resident ephemeris is the associated epoch falling within the time span for that ephemeris. This condition can be tested with two Cspice functions as follows.

1) The time domain pertaining to a particular ephemeris can be mapped into a "window" data structure with `spkcov_c`. Given adequate memory allocation, any number of finite continuous time intervals can reside in a window.

2) Whether or not a specified epoch resides in some interval of a window can be assessed with `wnelmd_c`.

Before spkcov_c is invoked, the window data structure to receive its output must be properly initialized. Although Cspice provides the `SPICEDOUBLE_CELL` macro for this purpose, C macros cannot be invoked from Swift code. Many programmers address this problem by implementing the macro as a Swift structure or function. But such an approach in this Cspice context tends to produce Swift code whose pedigree is ill-defined and whose syntax is full of unsafe pointer types.

As noted in Section 1.0's Introduction, Xcode can compile C code, and this capability enables the recommended solution for handling C macros in Cspice. The following C "wrapper" function[12] provides Swift code calling it exactly what is required: a boolean result regarding whether or not an epoch resides in an SPK-resident ephemeris.

---

[12] Much of this example function's syntax is suggested by Example 1 in comments from the `spkcov_c` source file.

```
#include "SpiceUsr.h"

SpiceBoolean BeEpochInSPK(ConstSpiceChar *spkFilename,
                          SpiceInt        bodyID,
                          SpiceDouble     epochPoint)

{

    // Local parameters
    #define MAXIV       1000
    #define WINSIZ      (2*MAXIV)

    // Local variables
    SpiceBoolean epochWithinSPK;
    SPICEDOUBLE_CELL(cover, WINSIZ);

    // Obtain window segments for the specified bodyID in the specified spkFilename
    spkcov_c(spkFilename, bodyID, &cover);

    // Determine if the specified epochPoint is within any bodyID window segment
    epochWithinSPK = wnelmd_c(epochPoint, &cover);

    return (epochWithinSPK);

}
```

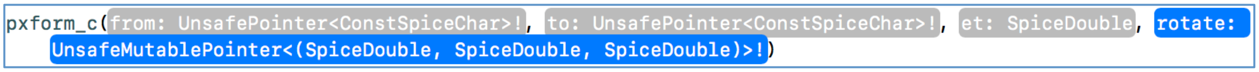
Exposing your Xcode project to a C function would typically be accomplished using a C header file (named `BeEpochInSPK.h` for the preceding example function). But this file would also need to be referenced in the `BridgingHeader.h` file created in Section 2.2 because it is not part of Cspice. Adding a dedicated function-specific header file can be avoided for this example function by adding the following statement to your project's `BridgingHeader.h` file.

```
SpiceBoolean BeEpochInSPK(ConstSpiceChar *spkFilename,
                          SpiceInt        bodyID,
                          SpiceDouble     epochPoint);
```

Many variations of this technique are possible. For example, a specified time interval (as opposed to a discrete epoch) can be assessed against the coverage of an SPK-resident ephemeris by replacing `wnelmd_c` in the foregoing example wrapper function with `wnincd_c`.

6.0 Accessing Coordinate System Transformation Matrices

As documented in the NAIF Toolkit's `frames.req` required reading file, SPICE supports a number of "built-in" coordinate systems, together with others stored in text and binary kernels. A state vector transformation matrix between any pair of defined coordinate systems can be supplied by Cspice, but difficulties arise in trying to access these data from Swift code. For example, consider the Cspice function `pxform_c`. The Xcode auto-completion prompt for `pxform_c` appears in Figure 3.



```
pxform_c(from: UnsafePointer<ConstSpiceChar>!, to: UnsafePointer<ConstSpiceChar>!, et: SpiceDouble, rotate:
    UnsafeMutablePointer<(SpiceDouble, SpiceDouble, SpiceDouble)>!)
```

Figure 3. This Xcode snapshot is the Swift source code editor's auto-completion appearance for a direct call to Cspice function `pxform_c`. The highlighted `rotate` parameter is problematic because of its `(SpiceDouble, SpiceDouble, SpiceDouble)` type.

Unfortunately, the 3x3 transformation matrix being returned by `pxform_c` in its `rotate` parameter cannot be associated with an `UnsafeMutablePointer` in Swift code because the inferred `(SpiceDouble, SpiceDouble, SpiceDouble)` type is not supported. As in Section 5.0, the solution to this difficulty (again with minimal impact to Cspice pedigree) is a C wrapper function in this instance acting as an intermediary between Swift code and calls to `pxform_c`. Code for this customized `BeCspiceXmat` C function appears below.

```
#include "SpiceUsr.h"

SpiceDouble* BeCspiceXmat ( ConstSpiceChar    *from,
                            ConstSpiceChar    *to,
                            SpiceDouble        et )

{

    // Without the "static" attribute, rotate data are lost outside the
       scope of this function
    SpiceDouble static rotate[3][3];

    pxform_c(from, to, et, rotate);

    SpiceDouble* Xmat = &rotate[0][0];

    return Xmat;

}
```

The foregoing wrapper function does not attempt to directly return a 3x3 matrix to Swift code invoking it. Rather, a pointer to the base address of the matrix is returned using the `SpiceDouble` type corresponding to that of the 9 matrix elements. As with any C function not part of Cspice, `BeCspiceXmat` must be defined in the `BridgingHeader.h` file as follows to be recognized by Swift code.

```
SpiceDouble* BeCspiceXmat ( ConstSpiceChar    *from,
                            ConstSpiceChar    *to,
                            SpiceDouble        et );
```

Calls to the C wrapper function in Swift code are performed as follows.

```
var ptrToXmat = UnsafeMutablePointer<SpiceDouble>.allocate(capacity: 0)
ptrToXmat = BeCspiceXmat("B1950", "J2000", secSinceJ2K)
```

Unlike previous `UnsafeMutablePointer` examples, `ptrToXmat` allocates no memory because that task is performed in `BeCspiceXmat`. In the foregoing B1950-to-J2000 example, the `secSinceJ2K` argument's value is irrelevant because the two inertial coordinate systems are each associated with an implicit epoch. Transformations involving non-inertial coordinate systems, such as the many object-fixed systems supported by SPICE kernels, critically depend on `secSinceJ2K`. For testing purposes, the B1950-to-J2000 transformation matrix returned by `BeSpiceXmat` is as follows.

```
+9.99925707952363E-01 -1.11789381377701E-02 -4.85900381535927E-03
+1.11789381264277E-02 +9.99937513349989E-01 -2.71625947142470E-05
+4.85900384145443E-03 -2.71579262585108E-05 +9.99988194602374E-01
```

The foregoing values agree to 15 decimal digits with Murray's Equation (25) on p. 328 of *Astronomy and Astrophysics* Volume 218 (1989).[13]

After a call to `BeCspiceXmat`, each element of the returned matrix is accessed from Swift code using array subscript syntax with the pertinent pointer. For example, elements in the matrix top row would be accessed as `ptrToXmat[0]`, `ptrToXmat[1]`, and `ptrToXmat[2]` in left-to-right order.

7.0 Summary

As documented in this paper, an initial foray into accessing Cspice functions with Swift code has produced satisfying results. The author is grateful to multiple astrodynamics colleagues, together with NAIF personnel, for their counsel and encouragement throughout this experiment. It is hoped this paper will prove useful to other Swift software developers electing to use Cspice.

---

[13] This paper is available for download at athttp://adsabs.harvard.edu/full/1989A%26A...218..325M (accessed 5 March 2018).