Introduction to SPICE
COSPAR Workshop on Planetary Science
Montevideo, Uruguay
July 23, August 3 — 2007

Jose Luis Vázquez García
European Space Astronomy Centre — European Space Agency

July 20, 2007

This page is left intentionally blank

Contents

The present introduction to SPICE has been written taking as main source of information the *SPICE Required Readings* and the *SPICE Tutorials*, downloadable from NAIF:

http://naif.jpl.nasa.gov/naif

This introduction will focus on some SPICE parts, namely time and reference frames. Also some general examples of SPICE usage will be shown in a final chapter.

Although the C SPICE toolkit with the C++ programming language have been used for all the examples in these notes, it shouldn't be difficult for the reader to translate them to other languages supported by SPICE, such as Fortran or IDL. C++ is used instead of C because of its better I/O capabilities and the possibility of in-place declaration of variables; no other C++ specific features are used.

List of Tables

List of Figures

Chapter 1

What is SPICE?

SPICE is a set of software routines (the SPICE toolkit) and a suite of data formats that help a scientist use ancillary data to plan scientific observations from a space vehicle and to analyze the science data gathered from those observations. It may also help scientists and engineers in planning future missions. In this context "ancillary data" means observation geometry data and time conversion functions.

SPICE was developed and is maintained by the **Navigation and Ancillary Information Facility** (NAIF) team of the **Jet Propulsion Laboratory**, California Institute of Technology, under contract with the U.S. **National Aeronautics and Space Administration** (NASA). The SPICE system is free of U.S. export restrictions and is available at no cost to the international space science community. The system is available in several languages and works (or can be configured to work) on all popular computing environments. It was originally developed in Fortran; versions of SPICE in C (*CSPICE*) and IDL (*Icy*) are available. There also is a Matlab beta version (*Mice*).

This multi-mission capability has been used for more than 20 years now on many NASA missions and more recently on ESA's planetary missions: Mars Express, Venus Express, Rosetta, Huygens and Smart-1.

SPICE helps scientists to perform all kinds of geometrical and time calculations involving the spacecraft, instruments on board the spacecraft, Solar

System bodies, ground stations, etc. Some particular situations where SPICE can help are:

- What was the Rosetta position with respect to the Earth at one particular moment?
- For a given image of Mars taken by MGS, I know the spacecraft clock reading. At what time was the image taken?
- I want to use images taken by Mars Express to search for rovers on the surface of Mars. Is the pixel resolution of the MEX camera good enough for that task?

SPICE stores geometry and time data in files called kernels. These are the core of the system, since they provide position (ephemeris) information for solar system bodies and spacecraft, time information and parameters for the instruments onboard a spacecraft.

Understanding SPICE involves having some knowledge of the several (physical and logical) parts that make up the SPICE system, namely:

- The SPICE kernels.
- The utility programs distributed as part of the SPICE toolkit.
- The different time systems used in SPICE.
- The different reference frames used in SPICE.
- The SPICE toolkit.

The NAIF web page, *http://naif.jpl.nasa.gov*, is the official source of information about SPICE.

Chapter 2

Getting and installing SPICE

Contents

The SPICE toolkit can be found at *http://naif.jpl.nasa.gov/naif/toolkit.html*. You have to choose the specific distribution for your programming language and platform. Download it to the directory where you want it to be installed.

The following instructions apply to the UNIX/Linux operating systems. Refer to the NAIF documentation to install SPICE in Windows and Mac machines.

2.1 Fortran

Open a shell and go to the directory where you've downloaded the toolkit. To uncompress it, run the following command:

```
$ tar xZf toolkit.tar.Z
```

The software will be uncompressed under a directory named *toolkit*. Go to that directory, and run the script that you'll find there:

```
$ cd toolkit
$ ./makeall.csh
```

It will take a while for the toolkit to compile. Once the compilation is done, you are ready to use the SPICE toolkit under Fortran.

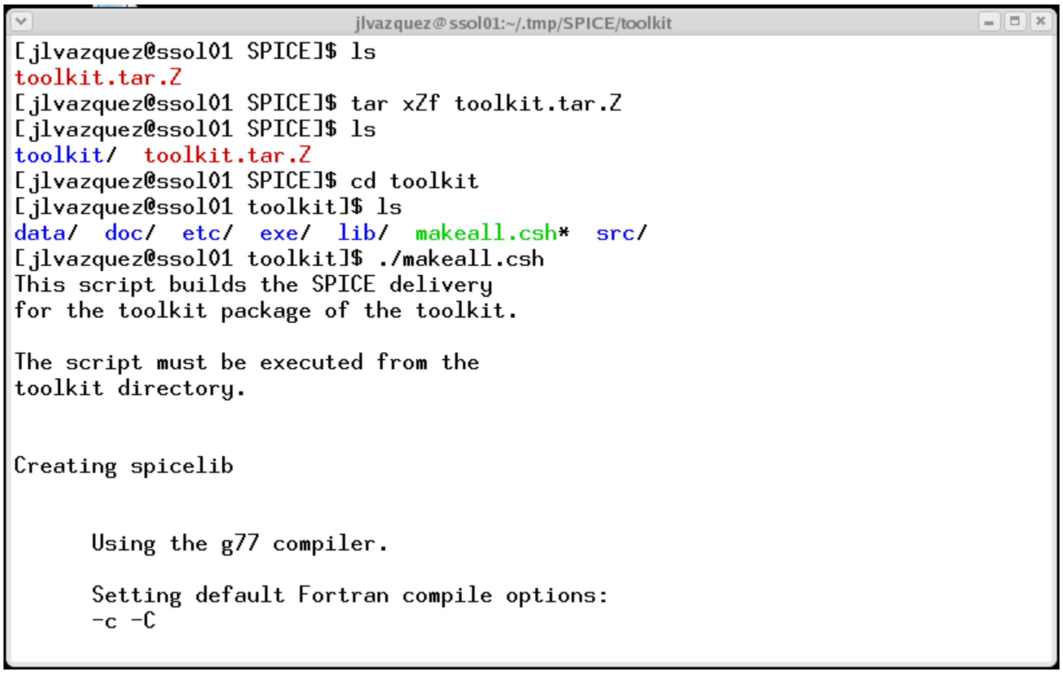

Figure 2.1: Installing the Fortran SPICE toolkit.

To compile SPICE programs in Fortran, you just need to specify the path to the library in the command line when you call your Fortran compiler. For example, suppose you have a source program called *test.f*, and the toolkit installed at /usr/local/SPICE/toolkit. You can compile the program with the following command line:

```
$ f77 -o test test.f /usr/local/SPICE/toolkit/lib/spicelib.a
```

2.2 C

Open a shell and go to the directory where you've downloaded the toolkit. To uncompress it, run the following command:

```
$ tar xZf cspice.tar.Z
```

The software will be uncompressed under a directory named *cspice*. Go to that directory, and run the script that you'll find there:

```
$ cd cspice
$ ./makeall.csh
```

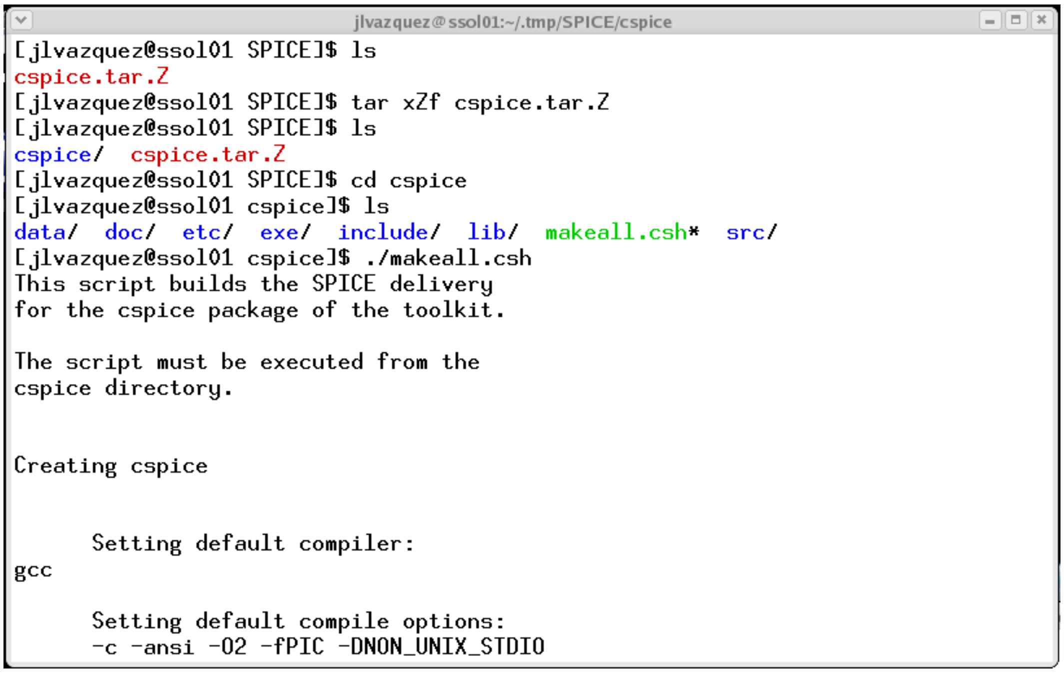

Figure 2.2: Installing the C SPICE toolkit.

It will take a while for the toolkit to compile. Once the compilation is done, you are ready to use the SPICE toolkit under C.

To compile SPICE programs in C, you just need to specify the path to the library and the include directory in the command line when you call your C compiler. For example, suppose you have a source program called *test.c*, and the toolkit installed at /usr/local/SPICE/toolkit. You can compile the program with the following command line:

```
$ gcc -o test test.c /usr/local/SPICE/toolkit/lib/spicelib.a \
      -I/usr/local/SPICE/toolkit/include
```

2.3 IDL

Open a shell and go to the directory where you've downloaded the toolkit. To uncompress it, run the following command:

```
$ tar xZf icy.tar.Z
```

The software will be uncompressed under a directory named icy. Go to that directory, and run the script that you'll find there:

```
$ cd icy
$ ./makeall.csh
```

It will take a while for the toolkit to compile. Once the compilation is done, you have to create a new environment variable called *IDL_DLM_PATH* to the directory where *icy.dlm* and *icy.so* are located (the directory *lib* under the *icy* main directory).

If you work on Linux with the *bash* shell, run on the command line[1]:

```
$ export IDL_DLM_PATH="/usr/local/SPICE/icy/lib"
```

[1] You might want to add the line to your *.bashrc* file to avoid having to run the same command every time.

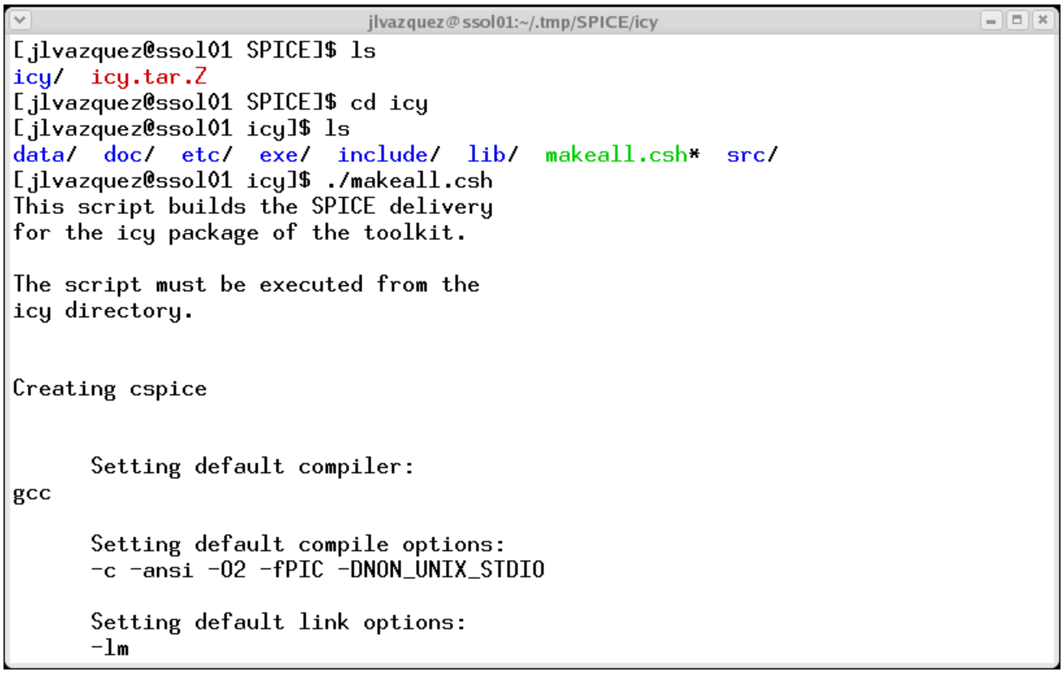

Figure 2.3: Installing the IDL SPICE toolkit.

For *csh* based shells, run:

```
$ setenv IDL_DLM_PATH "/usr/local/SPICE/icy/lib"
```

Chapter 3

SPICE kernels

Contents

SPICE stores geometry and time data in files called kernels. These are the core of the system, since they provide:

- information about the different reference frames used to describe the position and movement of the bodies of the solar system.
- ephemeris information for spacecrafts, solar system bodies and even ground stations.
- attitude (orientation) information, with respect to some reference frame, for spacecrafts.
- information about the spacecraft clock and how to convert from it to ephemeris time and/or UTC, and the other way around.
- mounting alignment and field-of-view geometry for the spacecraft's instruments and antennas.

Each kernel is a separated file whose extension identifies its type, as well as whether it is a binary or text kernel[1]. A kernel stores information which can be read by any application that uses the SPICE toolkit.

3.1 Types of kernels

A SPICE kernel can have one of the following types:

SPK Spacecraft and Planetary ephemeris. SPK kernels store position (ephemeris) information for planets and/or other solar system bodies and spacecrafts. They are binary kernels, and their file names have extension 'bsp'.

PCK Planetary constants. PCK kernels store information about planets and other bodies of the solar system. Information stored in a PCK kernel can be, for instance, the mass of a planet, or how it rotates as a function of time. They can be binary (extension 'bpc') or text (extension 'tpc') files.

IK Instrument. IK kernels contain information about instrument parameters, like field of view of a camera or the number of pixels of a CCD. They are text kernels, with extension 'ti'.

CK Pointing. C-kernels are binary kernels with extension 'bc' which store information about the orientation of the spacecraft or any of its substructures.

EK Events. Events like the science plan or the experimenters notebook are stored in this kernels. This type of kernels are the least developed part of the SPICE system, and are not used within the ESA planetary missions.

FK Reference Frame Specifications. In this type of text kernel (with extension 'tf') can be found information about different reference frames used in a mission, and how to transform vectors from one reference system to another.

[1] For a binary kernel, the extension starts with a 'b' and for a text kernel it starts with a 't'.

SCLK Spacecraft Clock Correlation data. This text kernel (with file extension 'tsc') allow the SPICE system to translate from the spacecraft clock coefficients to UTC or other time systems and vice-versa.

LSK Leapseconds. In a leapseconds kernel (text kernel with extension 'tls') is stored information about the leapseconds that have occurred.

3.2 Generation of the kernels

There basically are three main sources for the SPICE kernels:

1. The SCLK and the SPK and CK kernels that store position and attitude of a spacecraft are made by measuring its state (position, orientation and velocity), or by predicting it. They are mission specific. **For NASA missions, they are produced by NAIF, whereas for ESA they are produced by the mission Science Operations team[2] with information from the ESOC flight dynamics team and the support of NAIF**.

2. FK and IK kernels, with information about the different reference frames attached to the spacecraft and its substructures and about the instruments on board the spacecraft, are made by **the creators of the instrument with the support of NAIF, and, for ESA missions, the mission Science Operations team**.

3. Other kernels with generic information, like some SPK with ephemeris of the Solar System bodies, or the LSK kernel are **provided by NAIF**.

[2]The Science Operations team is a team dedicated to coordinate all the spacecraft operations from a scientific point of view, so that the spacecraft can be be optimized in terms of science production.

A SPICE application will need some kernels to run, and the programmer must know which ones and how to get them[3]. To make a kernel available to an application, it has to be loaded via a call to the function *furnsh_c*[4]. As a first example of SPICE usage, we'll create a program that loads the leapseconds kernel:

```
#include <iostream>
using namespace std;

// The SPICE header file has to be included
#include <SpiceUsr.h>


int main( char argc, char *argv[] ) {

   // Load the leapseconds kernel
   furnsh_c( "naif0008.tls" );
   cout << "Kernel loaded!" << endl;

   return 0;

}
```

Although this is not a very useful program, it can give us some ideas as to how SPICE works. For this program to run, the leapseconds kernel and the executable have to be located in the same directory[5]. If that's the case, we won't see nothing happen when we run the program; it jut loads the kernel, prints the message and then exits.

We can try and see what happens if we change the name of the kernel or delete it. SPICE will complain with an error message because it cannot find the kernel, and then the program will abort. The output message "Kernel loaded!" is used for us to find out where the program finishes: it is not printed in this case, which indicates that the program stops immediately after the failure.

[3]failing to load one of the needed kernels will cause the application to abort.

[4]*FURNSH* in Fortran, *cspice_furnsh* in IDL.

[5]The programmer can alternatively provide *furnsh_c* with the full path to the kernel.

```
jlvazquez@ssol01:~/Workshops/COSPAR_MontevideoJul07/SPICE_Intro/code
add2prj.tpl   load_kernel.cpp      naif0008.tls.off
et_utc.cpp    makefile             sc_clock*
load_kernel*  mex_070605_step.tsc  sc_clock.cpp
[jlvazquez@ssol01 code]$ ./load_kernel

================================================================================

Toolkit version: N0061

SPICE(NOSUCHFILE) --

The attempt to load "naif0008.tls" by the routine FURNSH failed. It could not
be located.

A traceback follows.  The name of the highest level module is first.
furnsh_c --> FURNSH --> ZZLDKER

Oh, by the way:  The SPICELIB error handling actions are USER-TAILORABLE.  You
can choose whether the Toolkit aborts or continues when errors occur, which
error messages to output, and where to send the output.  Please read the ERROR
"Required Reading" file, or see the routines ERRACT, ERRDEV, and ERRPRT.

================================================================================
[jlvazquez@ssol01 code]$
```

Figure 3.1: Error message when failing to load a kernel.

Along with the toolkit (for C, Fortran and IDL), comprehensive hyper-text documentation is provided, readable with a web browser. The reader is encouraged to open it and see the explanation of the functions used in the examples of this introduction.

Chapter 4

Time standards used in SPICE

Contents

There are two widely used types of time standards, those related to the rotation of the earth (TAI), and those related with the frequency of atomic oscillations (ET, UTC,...). The earth rotation is not uniform, and therefore the rate of the clocks based on it exhibit both periodic changes and long term drifts. Atomic standards are the closest approximations we currently have to a uniform time. Well see how this different types of times are handled by SPICE.

SPICE supports several time systems, namely:

- **UTC**, Coordinated Universal Time.
- **ET**, Ephemeris Time.

- **SCLK**, Spacecraft Clock Time.

4.1 UTC, Coordinated Universal Time.

The basis for the UTC is the **International Atomic Time** (TAI). It is based on the atomic second as defined by the "oscillation of the undisturbed cesium atom". Atomic time is a count of the number of atomic seconds that have occurred since the astronomically determined instant of January 1, 1958 00:00:00, at the Royal Observatory in Greenwich.

UTC is a system of time keeping that gives a name to each instant of time of the TAI system. These names are formed from the calendar date and time of day we use in our daily affairs. A date in UTC format has the look we are used to for dates. For example, the next names refer to the same TAI instant:

```
20 JUNE 2007
2007 JUNE 20
2007 20 JUNE
2007 6 20
6 20 2007
```

Ideally, every UTC day at 00:00:00 hours should correspond with midnight at Greenwich, as observed astronomically[1]. However, the rotation of the Earth is not uniform, which means that there is a difference between UTC midnight and UT1. To keep the difference from being too large, UTC is occasionally adjusted so that it never exceeds 0.9 seconds.

UTC can be represented in several ways in SPICE, two of the most used are:

[1]This time is called **UT1**. It is based on the rotation of the earth. It assumes that the time between two consecutive passes of the sun above Greenwich is 24 hours.

YYYY-MM-DDThh:mm:ss.fff The time is represented as year, month, day of month, hours, minutes, seconds and fraction of second. For example, 2005-10-12T12:50:55.103.

YYYY-DDDThh:mm:ss.fff The time is represented as year, day of year, hours, minutes, seconds and fraction of second. For example, 2007-138T00:00:00.000.

Leapseconds.

When Greenwich UT1 midnight lags behind midnight UTC by more than 0.7 seconds, a leap second will be added to the collection of UTC names. This leapsecond has traditionally been added after the last "normal" UTC name of December 31 or June 30. When a leapsecond is added at the end of a year, UTC time progresses in the following way:

```
... DECEMBER 31 23:59:58
... DECEMBER 31 23:59:59
... DECEMBER 31 23:59:60
... JANUARY   1 00:00:00
```

If Greenwich UT1 midnight runs ahead of UTC midnight by more than 0.7 seconds, a negative leapsecond will be added. In this case, the progression will be:

```
... DECEMBER 31 23:59:57
... DECEMBER 31 23:59:58
... JANUARY   1 00:00:00
```

4.2 ET, Ephemeris Time.

Ephemeris time is the uniform time scale represented by the independent variable in the differential equations that describe the motions of the planets, sun and moon. There are two forms of ephemeris time: **Barycentric Dynamical Time** (TDB) and **Terrestrial Dynamical Time** (TDT).

4.2.1 Barycentric Dynamical Time (TDB).

Barycentric dynamical time is used when describing the motion of bodies with respect to the solar system barycenter.

4.2.2 Terrestrial Dynamical Time (TDT).

Terrestrial dynamical time is used when describing motions of objects near the earth. As far as measurements have been able to detect, TDT and TAI change at the same rate. The difference between them is defined to 32.184 seconds.

4.2.3 Relationship between TDT and TDB.

TDB is believed to be in agreement with the time that would be kept by an atomic clock located at the solar system barycenter. A comparison of the times kept by a clock at the solar system barycenter with a TDT clock on earth would reveal that the two clocks are in close agreement but that they run at different rates at different times of the year. This is due to relativistic effects. At some times in the year the TDT clock appears to run fast when compared to the TDB clock, at other times of the year it appears to run slow. In SPICE, the difference between TDT and TDB is computed as follows:

$$TDB - TDT = K sin(E)$$

where K is a constant, and E is the eccentric anomaly of the heliocentric orbit of the Earth−Moon barycenter (see figure 4.1).

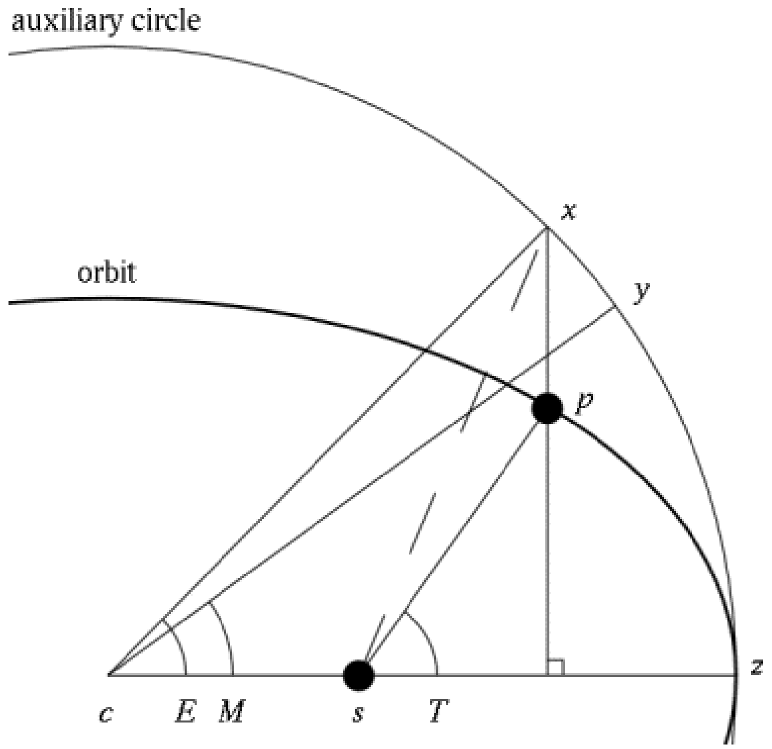

Figure 4.1: Eccentric anomaly of the Earth orbit. The point **p** represents the Earth, and the point **s** the Sun.

When ephemeris time is called for by Toolkit routines, TDB is the implied time system. We cal this time **Ephemeris Time** (ET). Ephemeris Time is given in terms of seconds past a reference epoch. The reference epoch used throughout the Toolkit is the epoch $J2000$ (roughly noon on January 1, 2000).

SPICE provides functions to convert from UTC to ET and vice-versa. The following example shows it:

```
#include <iostream>
using namespace std;
```

```
#include <SpiceUsr.h>


int main( char argc, char *argv[] ) {

   // Load the leapseconds kernel
   furnsh_c( "naif0008.tls" );

   SpiceChar utc[ 50 ] = "20007-07-23T00:00:00";
   SpiceDouble et;

   // Convert from UTC to ET
   utc2et_c( utc, &et );
   cout << utc << " UTC => " << et << " ET" << endl;

   // Convert from ET to UTC
   et2utc_c( et, "ISOC", 3, 50, utc );
   cout << et << " ET => " << utc << " UTC" << endl;

   return 0;

}
```

Here a few more characteristics of SPICE are presented.

- CSPICE defines its own set of data types, and the user is encouraged to use them instead of the C standard types; this will guarantee that the software will still compile if the definition of the SPICE data types is changed in the future. For example, the type *SpiceChar* is defined to be a normal C *char*. However, that definition can be changed in future, and since functions like *et2utc_c* expect a string of *SpiceChar*, backwards compatibility is assured by sticking to *SpiceChar* instead of *char*.

- As we said before, the conversion from ET to UTC and vice-versa needs a leapseconds kernel to be performed; therefore, the kernel has to be loaded via a call to *furnsh_c*. If the line where the function is called is deleted, the program will abort when it tries to make the first conversion (*utc2et_c*) (see figure 4.2).

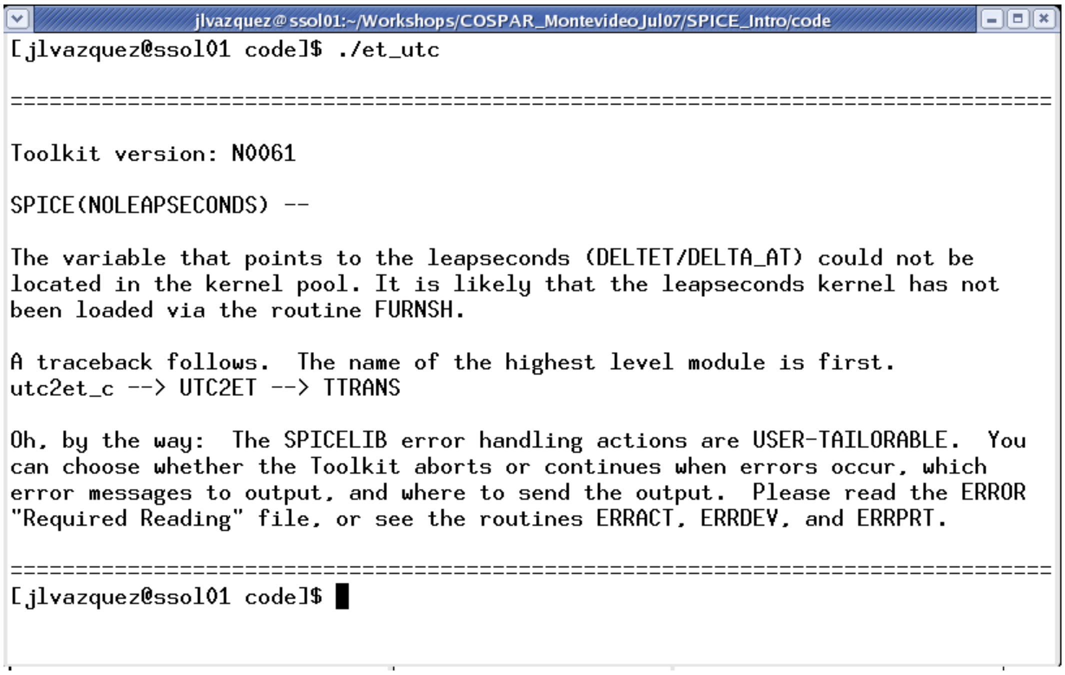

Figure 4.2: Doing time conversions without a leapseconds kernel.

- The prototypes of almost all the CSPICE functions follow the same convention: all the input arguments are at the beginning of the list of parameters, followed by pointers to the output arguments. This is why in the call to *utc2et_c* the address of *et* is specified.

The call to *et2utc_c* might seem a bit dark. The list of parameters is as follows:

1. *et* the ephemeris time to be converted to UTC.
2. *"ISOC"* a string specifying the format of the output UTC string. In this case, it will be YYYY-MM-DDThh:mm:ss.fff
3. *3* an integer specifying the number of digits for the fraction of second.
4. *50* an integer specifying the length of the UTC string.

5. *utc* an array of *SpiceChar* that holds the UTC string. Since an array in C is actually a pointer, the address of the string is implicitly passed to the routine.

The program converts an UTC time to ET, and then back to UTC. Obviously, the result will give us the original UTC time.

4.3 SCLK, Spacecraft Clock Time.

Most spacecrafts have an onboard clock, that controls the times at which various actions are performed by the spacecraft and its science instrument. This clock is not a normal one, but rather a counter, which counts instants of time, called ticks. The duration of each instant of time and how it is represented by the clock depends on the particular spacecraft. A spacecraft clock can be made up of several counters, each one increasing its value in steps of one when the previous one reaches its maximum value[2]. For example, the Galileo clock has the following format:

```
rrrrrrrr:mm:t:e
```

where

Field	Time unit	Modulus
rrrrrrrr	$60\frac{2}{3}s.$	16777215
mm	$\frac{2}{3}s.$	91
t	$\frac{1}{15}s.$	10
e	$\frac{1}{120}s.$	8

Table 4.1: Components of the Galileo spacecraft clock.

In this context, modulus means the maximum value the field can reach, minus one. For example, one run of the counter could look like this:

```
...
00001234:15:8:6
00001234:15:8:7
```

[2]This is also how a wall clock behaves.

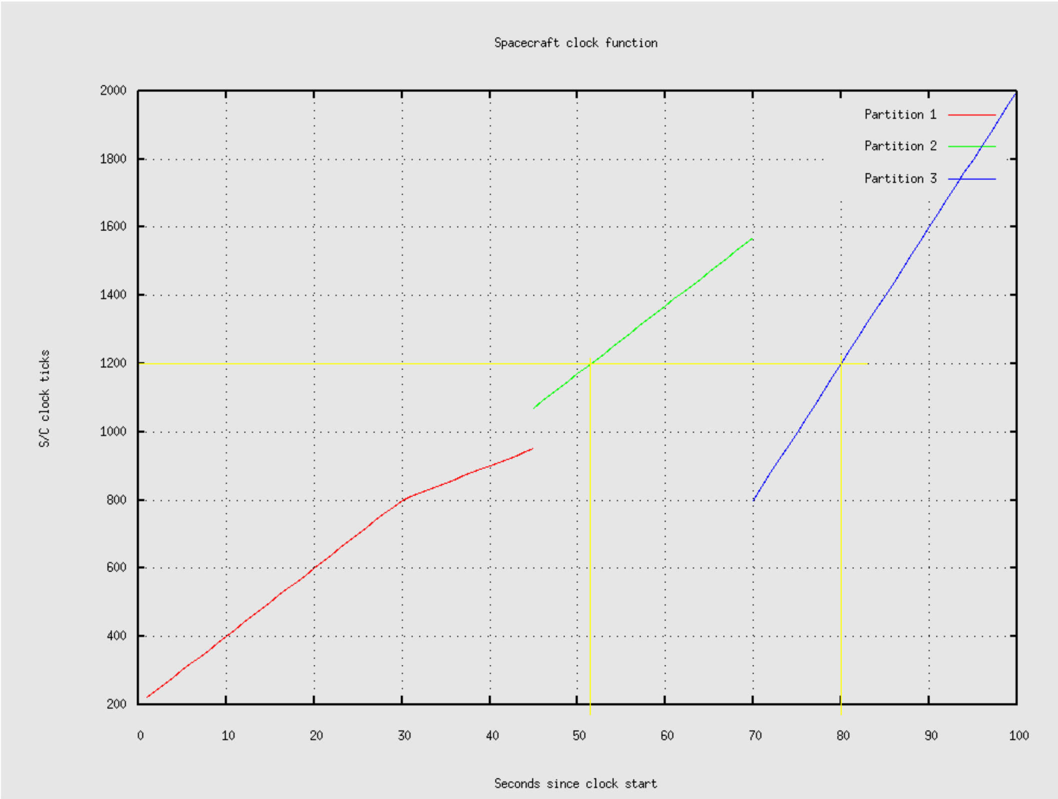

Figure 4.3: Example of the evolution of a S/C clock with time.

```
00001234:15:9:0
...
00001234:15:9:6
00001234:15:9:7
00001234:16:0:0
00001234:16:0:1
...
```

To make things more complicated, the spacecraft clock is not guaranteed to have a constant rate, it can start going faster or slower than before, and it even can jump or suffer a reset. In the figure 4.1 an example of how a S/C clock can behave is plot. The continuous counting of ticks is represented, like we would do, for example, if we represented the time in seconds. We can see that the clock starts counting at $20\frac{ticks}{second}$, and 30 seconds after the start the clock starts counting more slowly. Then, at second number 46, the clock jumps forward, and at second 70 there's a jump back. The consequence of this jumps is that it might be that the same reading of the clock corresponds to several time instants; in the figure, the tick 1200 of the counter corresponds roughly to the seconds 52 and 80 since the clock start.

Therefore, more information is needed in order to be able to use the spacecraft clock information without ambiguity. To this purpose, a new component, called *partition*, is prepended to the clock string. A partition starts at the beginning of the count, and any time there is a discontinuity in the clock. For example, the next string

```
2/00001234:15:8:7
```

means that the reading $00001234 : 15 : 8 : 7$ belongs to the second partition of the clock, that is, after the first abrupt jump, and before the second one. Note that a change in the speed of the clock does not necessarily mean the start of a new partition, as it is stated in the figure 4.1.

When accessing the data from a spacecraft, the only information as to when that data was generated or captured is the S/C clock. Therefore, a means to turn the clock string into a readable time format is needed. SPICE provides several functions to deal with clock strings, some of which are shown in the next example. Note that direct conversion from a clock string to UTC is not possible; it has to be done in two steps: first convert the string to ET, and then convert the resulting ET time to UTC.

In the program below, a clock string read out of the Mars Express clock is converted to UTC, and an UTC time is converted to the corresponding reading in the Mars Express clock.

```
#include <iostream>
using namespace std;

#include <SpiceUsr.h>


int main( char argc, char *argv[] ) {

   // Load the leapseconds kernel
   furnsh_c( "naif0008.tls" );

   // Load the Spacecraft clock kernel for Mars Express
   furnsh_c( "mex_070605_step.tsc" );

   // Get the NAIF ID for Mars Express
   SpiceInt mex_id;
   SpiceBoolean found;
   bodn2c_c( "Mars Express", &mex_id, &found );
   cout << "NAIF ID for Mars Express: " <<
            mex_id << endl << endl;

   // Convert one S/C clock string to UTC
   SpiceChar clock_str[ 50 ] = "1/0029401660.57967";
   SpiceChar utc_str[ 50 ];
   SpiceDouble et;

   // Clock string to ET
   scs2e_c( mex_id, clock_str, &et );
   // ET to UTC
   et2utc_c( et, "ISOC", 0, 50, utc_str );
   cout << "UTC time for the clock string \"" <<
           clock_str << "\": " << utc_str << endl;


   // Convert from UTC to S/C clock string
   strcpy( utc_str, "2007-07-23T00:00:00" );
   // UTC to ET
   utc2et_c( utc_str, &et );
   // ET to Clock string
   sce2s_c( mex_id, et, 50, clock_str );
```

```
   cout << "Clock string for the UTC time \"" <<
            utc_str << "\": " << clock_str << endl << endl;

   return 0;

}
```

The first step in this example is to load the two kernels we need for this calculation, namely the leapseconds kernel and the spacecraft clock kernel for Mars Express.

Then another feature of SPICE is introduced. SPICE assigns an integer, called *NAIF ID* to any body of the Solar System, including spacecrafts and their structures, and tracking ground stations. Many of them are "built in" SPICE, while others have to be defined by the programmer[3]. The NAIF ID for Mars Express is hardcoded in SPICE, so the programmer just has to call the proper function (*bodn2c_c*) to get it. ID's for Solar System bodies are positive, while those for spacecraft and their structures are negative.

The last lines convert a clock string to UTC, and an UTC time to a clock string. As stated above, there has to be an intermediate step to convert either format to ET, since SPICE does not provide a function to do the conversions directly.

[3]When doing that, the programmer is responsible for choosing a NAIF ID which does not clash with the predefined ones. NAIF provides a set of rules for choosing IDs.

Chapter 5

Reference Frames and SPICE

Contents

5.1 Coordinate Systems and Reference Frames.

A Coordinate System is a system for assigning a vector of numbers in R^3 to each and every point in the space. This application has to be unique in the sense that any vector can be assigned to no more than one point.

Typical Coordinate Systems used in Physics are the *cylindrical*, *spherical* and *cartesian systems*.

A Reference Frame, however, is a particular realization of a Coordinate System. For instance, a Reference Frame could be defined as a cartesian system with the origin at the center of the Earth, its $+Z$ axis in the direction of the North pole of the Earth, and its $+X$ axis lying in the equatorial plane and intersecting with the Greenwich meridian[1].

The US Naval Observatory defines a *Reference System* as the complete specification of how a celestial coordinate system is to be formed. This encompasses the definition of its origin and planes, as well as all of the constants, models and algorithms used to transform between observable quantities and reference data that conform to the system. A *Reference Frame* is defined as a set of identifiable points on the sky along with their coordinates, which serves as the practical realization of a Reference System. In this document, however, we'll define a Reference Frame as a realization of a cartesian Coordinate System, specified by its origin and orientation of its axes.

The J2000 Reference Frame.

The main reference frame used in calculations for the Solar System bodies is the J2000 reference frame. This is a cartesian frame with origin in the center of the Earth, whose $+Z$ axis is perpendicular to the mean equatorial plane of the J2000 epoch[2], and the $+X$ axis contains the point where the Sun crosses the equatorial plane from South to North, also at the J2000 epoch.

Reference Frames for planets and other Solar System bodies.

In mathematical and geodetic terminology, the terms *latitude* and *longitude* refer to a right-hand spherical coordinate system in which latitude is defined as the angle between a vector passing through the origin of the spherical coordinate system and the equator, and longitude is the angle between the vector and the plane of the prime meridian measured in an eastern direction. This coordinate system, together with cartesian coordinates, is used in most planetary computations, and is called the *Planetocentric Coordinate Reference Frame.*

Planetographic coordinates, however, are defined such that northern latitudes are designated as positive, and the planetographic longitude of the central

[1]Note that the $+Y$ axis is implicitly defined, because if the reference system is right-handed, $\hat{y} = \hat{z} \times \hat{x}$, where $\hat{n}$ is the unit vector along the positive direction of axis N.

[2]January 1st, 2000 at noon.

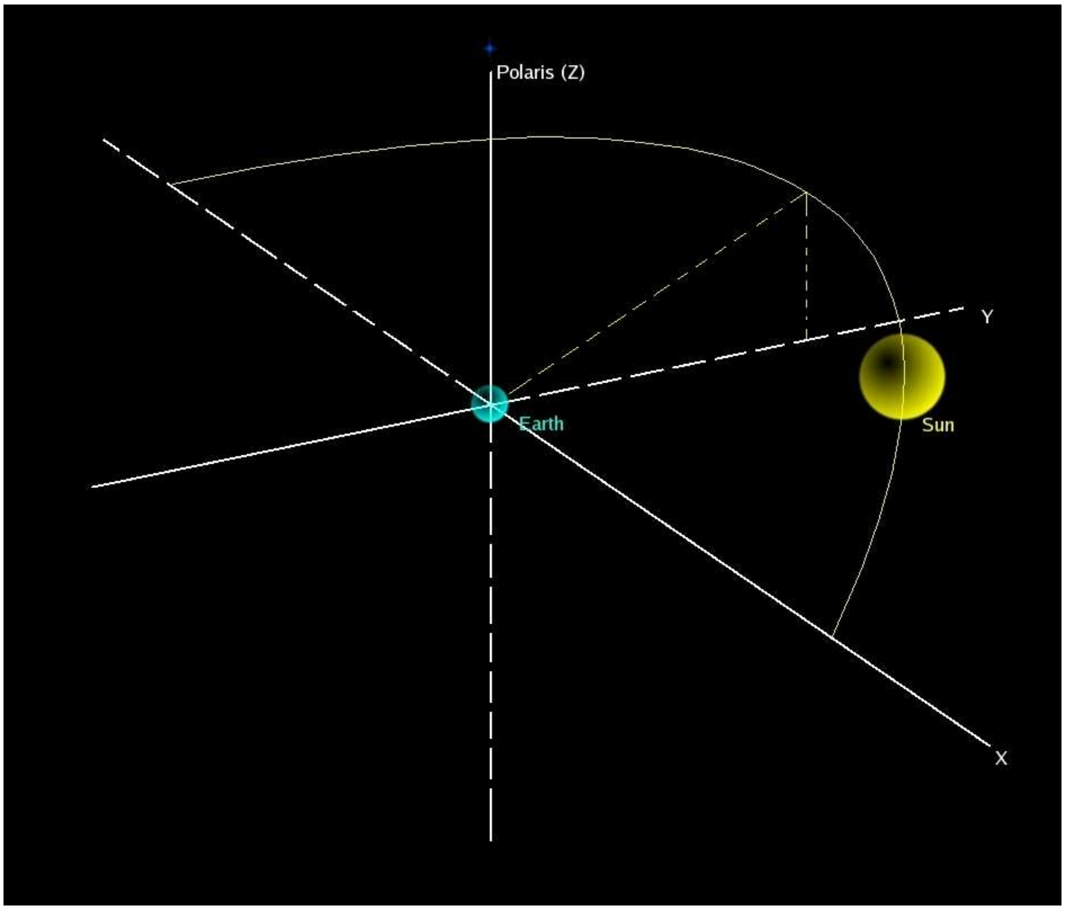

Figure 5.1: The J2000 Reference Frame.

meridian[3], as observed from a direction fixed with respect to an inertial reference frame, will increase with time, from 0^o to 360^o. This means that west longitudes (measured positively to the west) will be used when the rotation is prograde, and east longitudes (measured positively to the east) when the rotation is retrograde[4]. Because of tradition, the Earth, Sun and Moon are defined to have longitudes both east and west from 0^o to 180^o, or east from 0^o to 360^o.

[3]For example, the meridian of Greenwich for the Earth.

[4]A body has a prograde rotation if, seen from above its north hemisphere, it rotates counterclock-wise. Otherwise, it has a retrograde rotation.

Inertial Reference Frames.

An important classification of Reference Frames attend to whether they are inertial or not. In an *inertial reference frame* the Laws of Newton can be applied to the movement of the bodies. Otherwise, the reference frame is called non inertial. For instance, any reference frame attached to the Earth is non inertial, since the Earth rotates; the J2000 reference frame is inertial.

5.2 Rotating frames.

Among the most important operations that are done when dealing with data from a mission is the transformation between reference frames. For the same result can be easily interpreted or not, depending on the reference frame chosen to display it.

We'll talk here about transformations called *rotations*. Intuitively, the notion of rotation is wel understood. The Earth, for example, rotates about its axis. A body that rotates doesn't change its shape or properties, only its orientation, and the points of the body along the rotation axis maintaint their position.

Mathematically, a rotation is a linear transformation R, which maps the coordinates of a vector in a reference frame A to its coordinates in another frame reference B. It also must have the following properties:

- A rotation preserves norms. Given a vector $\vec{v}$, then

$$\|R(\vec{v})\| = \|\vec{v}\|$$

- A rotation preserve cross products. If $\vec{a}$ and $\vec{b}$ are vectors, then

$$R(\vec{a} \times \vec{b}) = R(\vec{a}) \times R(\vec{b})$$

The usual way to rotate a reference frame A, so it becomes other frame B, is by specifying the *rotation matrix*. This is the matrix T that, given the components of a vector v in the frame A, produces the components of the vector in the frame B, via matrix-vector multiplication:

$$v_A = v_x\hat{x} + v_y\hat{y} + v_z\hat{z}$$
$$T = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix}$$
$$v_B = T \times v_A$$

For example, in the figure 5.2, we can see two reference frames, A and B, and a vector, $\vec{r}$, which in the frame A can be described as $\vec{r} = r \times \hat{x}_a$. It is easy to prove that the matrix

$$T = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

transforms the coordinates of the vector $\vec{r}$ in the frame A into the coordinates of $\vec{r}$ in the frame B, where $\vec{r} = r \times \hat{z}_b$.

Other ways of specifying transformations between reference frames (namely *Euler Angles* and *quaternions*) will be introduced later in this chapter.

5.3 Reference Frames in SPICE.

A number of Reference Frames are “built into” the SPICE frame subsystem, so the user doesn’t have to do any special operation in order to use them. Among these frames are:

1. inertial frames such as J2000, Galactic System I frame, etc.
2. body-fixed frames based on IAU rotation models provided in text PCK files, such as the Earth body-fixed rotating frame, and body-fixed frames based on high precision Earth rotation models provided in binary PCK files.

All other frames are not "built into" SPICE. Instead, these frames have to be specified via a set of parameters in a frames kernel file. The types of frames defined in FK text kernels include:

1. CK-based frames, i.e. frames that change their orientation with time, orientation being provided in CK kernels.
2. fixed offset frames, i.e. frames for which orientation is constant with respect to another frame and is specified as part of the frame definition stored in a text kernel.
3. Dynamic frames, i.e. frames for which orientation is based on dynamic directions computed based on SPICE kernel data (SPKs, CKs, PCKs), on mathematical models implemented in CSPICE functions or on formulae defined in frame kernels.

Transformations between reference frames will often be needed. It is important to understand that **the only transformations that can be done without loading any kernels are those involving only inertial reference frames**. In the following example we'll see the usual way to calculate the transformation matrix, and how to transform vectors from one frame into another:

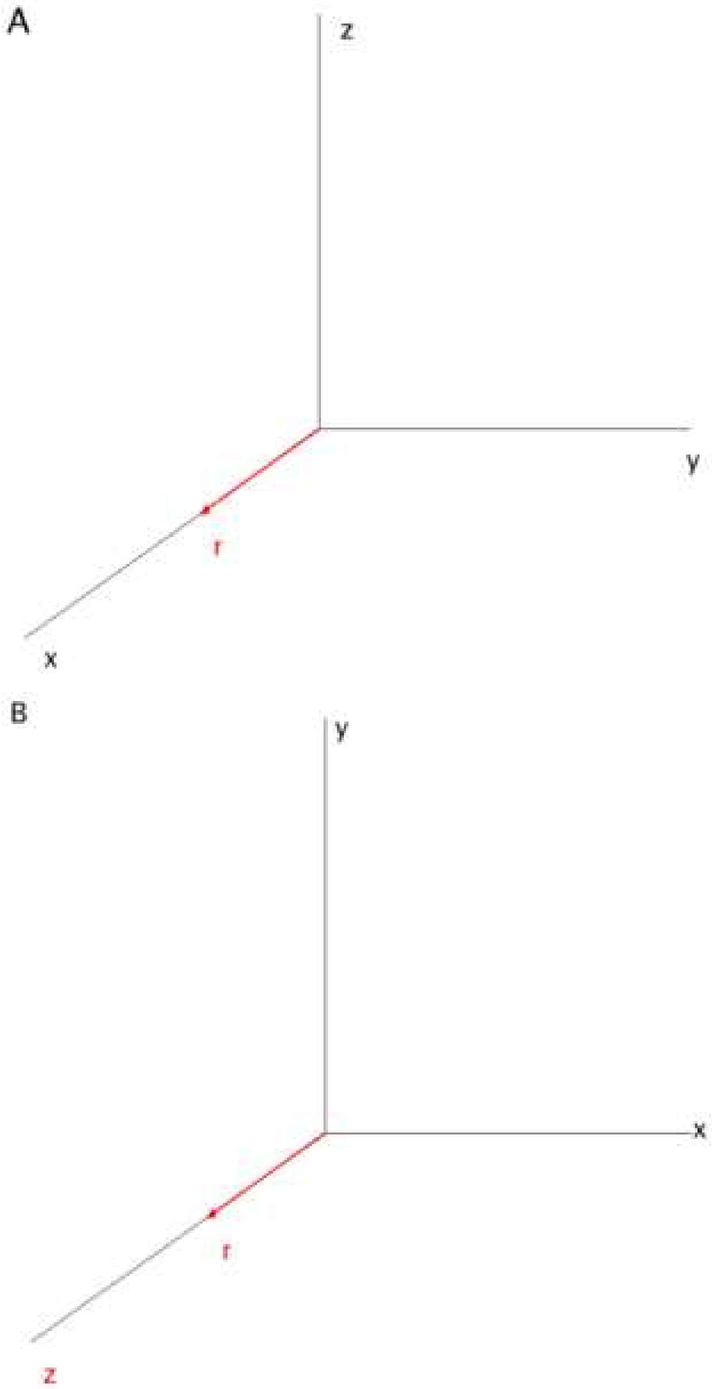

Figure 5.2: The same vector in two different frames, A and B.

```
#include <iostream>
using namespace std;

#include <SpiceUsr.h>


int main( char argc, char *argv[] ) {

   // Load the leapseconds kernel
   furnsh_c( "naif0008.tls" );

   // Convert to ET
   SpiceDouble et;
   utc2et_c( "2007-07-21T00:00:00", &et );

   // Calculate the transformation matrix to go
   // from 'J2000' to 'Galactic System II'
   SpiceDouble matrix[ 3 ][ 3 ];
   pxform_c( "J2000", "GALACTIC", et, matrix );

   // Print the transformation matrix
   cout << "Transformation Matrix:" << endl;
   for ( int i = 0 ; i < 3 ; i++ ) {

      for ( int j = 0 ; j < 3 ; j++ )
         cout << matrix[ i ][ j ] << "\t";

      cout << endl;

   }

   cout << endl;

   // Convert the vector ( 10, 10, 10 ) in 'J2000'
   // into the 'Galactic System II'
   SpiceDouble rj[ 3 ] = { 10, 10, 10 };
   SpiceDouble rg[ 3 ];
   mxv_c( matrix, rj, rg );

   cout << "Vector in the Galactic System II frame:"
```

```
  << endl;
   cout << "( " << rg[ 0 ] << ", " << rg[ 1 ] << ", "
        << rg[ 2 ] << " )" << endl;

   return 0;

}
```

The function *pxform_c* computes the transformation matrix, *matrix*, to transform vectors given in the origin frame, in this example *J2000*, into vectors in the destination frame, *Galactic System II*, for a given instant of time, *et*. Once the matrix is calculated, the vectors can be transformed via matrix multiplication, by calling to the function *mxv_c*.

Transforming between inertial frames is easy, since they are "hard-coded" into the SPICE system. But most of times the programmer will have to deal with other kinds of frames. For example, for every mission, a full set of reference frames have to be defined, in order to specify the orientation of the spacecraft with respect to some other reference frame (for example J2000), and also the orientation of any of the spacecraft subsystems with respect to the spacecraft reference frame[5].

5.3.1 Creating frames kernels.

Creating a frames kernel is a difficult task which involves several steps, different depending on the type of kernel we create. We'll see in this chapter how to create fixed offset kernels, which will help us to understand the frames kernels provided for the planetary missions[6]. Our reference frame will be a frame with origin on the center of the ESA ground station located at New Norcia (Australia)[7], and whose x axis points north along the local meridian, the y axis points west along the local latitude and the z axis points up from the surface of the Earth at that point.

[5]The first would be a CK-based frame, whereas the second would be a fixed offset one.

[6]For information about how to create another type of kernels, refer to the SPICE Frames Required Reading.

[7]We'll need the coordinates later on, which are 116.191502^o East longitude, 31.048223^o South latitude.

To create the fixed offset frames kernel, the first step is to specify the name of the kernel, the ID code for the frame, the class number of the frame, the ID for the frame center, and the internal ID code (*CLASS_ID*) to refer to the frame.

The name chosen for a frame must no exceed 26 characters taken from the set including uppercase letters, numbers, underscore, and plus and minus signs. We'll call our frame *COSPAR*.

The class number for fixed offset frames is 4.

The frame ID is the NAIF ID[8] whereby the frame will be identified. NAIF reserves the codes from 1400000 to 4000000 for private uses; we'll choose the code 1400010 for the *COSPAR* reference frame.

The center of the frame is the New Norcia station, for which the NAIF ID is 398990.

The class ID is an integer used internally by the SPICE software. For fixed offset frames, the ID must match the frame ID.

With all this information, we can create the first part of the kernel.

```
\begindata

   FRAME_COSPAR = 1400010
   FRAME_1400010_NAME = 'COSPAR'
   FRAME_1400010_CLASS = 4
   FRAME_1400010_CENTER = 398990
   FRAME_1400010_CLASS_ID =  1400010

\begintext
```

Here we can see the general structure of any SPICE text kernel, which is a set of pairs *keyword* = *value*. Everything in the kernel, until the first

```
\begindata
```

[8]more on NAIF ID's in the next chapter.

which appears alone in a line is taken as a comment. All the pairs *keyword* = *value* have to be placed after a

```
\begindata
```

and before a

```
\begintext
```

to be considered. There can be any number of comment blocks and data blocks in the kernel, for example:

```
This is a comment...

\begindata

   KEYWORD1 = VALUE1
   KEYWORD2 = VALUE2
   ...

\begintext

This is another comment...

\begindata

   KEYWORD3 = VALUE3
   KEYWORD4 = VALUE4
   KEYWORD5 = VALUE5
   ...

\begintext
```

The last *begintext* is optional.

Since it's a fixed offset frame, now the relation between the frame we are defining and another known frame has to be specified in the kernel. We'll choose the Earth body-fixed rotating frame (*IAU_EARTH*):

```
TK_FRAME_1400010_RELATIVE = 'IAU_EARTH'
```

The most difficult part is specifying the way our frame is orientated with respect to the *relative* frame. It can be done via a rotation matrix, *Euler Angles* or *quaternions*.

Euler Angles.

In the figure 5.3, we can see that frame A of figure 5.2 can be turned into frame B by two rotations of 90^o, first about the Z axis, and then about the X axis. Thus, rotating the frame A in the described way produces the frame B. Each of the rotations can be described via a *rotation matrix*, which transforms vectors from the original frame to the rotated frame.

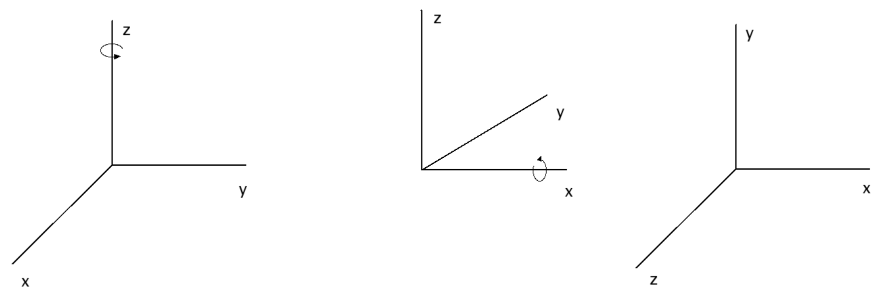

Figure 5.3: Rotating frames.

According to *Euler's theorem*, any rotation may be decomposed into three rotations about the axis of the reference frame. If the rotations are written in terms of rotation matrices B, C, and D, then a general rotation A can be written as:

$A = BCD$

The matrix $[\alpha]_x$ rotating α^o about the x axis, is defined as follows:

$$[\alpha]_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{pmatrix}$$

In the same way,

$$[\beta]_y = \begin{pmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{pmatrix}$$

$$[\gamma]_z = \begin{pmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

So, any rotation can be performed by choosing the proper rotation axis and angles. For example, a rotation could be specified as rotating first $\frac{\pi}{2}$ about the x axis, then $\frac{\pi}{4}$ about the z axis, and again π about the z axis:

$$R = [\pi]_x[\frac{\pi}{4}]_y[\frac{\pi}{2}]_x$$

The three angles giving the three rotations are called *Euler Angles*. The Euler Angles are not unique, that is, the same rotation can be decomposed in different ways.

Quaternions.

Quaternions are four dimensional vectors, on which a particular kind of arithmetic is defined. The quaternions that have norm equal to 1 are called *unit quaternions*.

Unit quaternions may be associated with rotations in the following way: if a rotation R has unit vector $\vec{n} = (n_1, n_2, n_3)$ as an axis and ω as a rotation angle, then we represent R by:

$$\vec{Q} = (\cos\frac{\omega}{2}, n_1 \sin\frac{\omega}{2}, n_2 \sin\frac{\omega}{2}, n_3 \sin\frac{\omega}{2})$$

This association is not unique: substituting $\omega + 2\pi$ for ω, we see that $-\vec{Q}$ is also a representation for R.

We'll use Euler Angles to specify the relation between the COSPAR reference frame and IAU_EARTH. Given the longitude α and latitude β, we can easily find that the rotation matrix R from IAU_EARTH to COSPAR is given by:

$$R = [180]_z[90 - \beta]_y[\alpha]_z$$

, where all the angles are specified in degrees. It means that the IAU_EARTH can be converted in the COSPAR frame by first rotating α^o about the z axis, then $90^o - \beta^o$[9] about the y axis, and 180^o again about z.

In spice, it has to be specified the transformation from the frame we are defining to the *RELATIVE* one, in this case IAU_EARTH. Thus, Euler Angles to be specified in the kernel are those given by

$$R^{-1} = [-\alpha]_z[\beta - 90]_y[180]_z$$

[9]Note that the latitude angle for New Norcia has to be negative, since it is the latitude of a point in the south hemisphere.

The complete frame kernel is shown below:

```
\begindata

    FRAME_COSPAR               = 1400010
    FRAME_1400010_NAME         = 'COSPAR'
    FRAME_1400010_CLASS        = 4
    FRAME_1400010_CENTER       = 398990
    FRAME_1400010_CLASS_ID     = 1400010

    TKFRAME_1400010_RELATIVE = 'IAU_EARTH'
    TKFRAME_1400010_SPEC     = 'ANGLES'
    TKFRAME_1400010_ANGLES   = ( -116.192, -121.05, 180 )
    TKFRAME_1400010_AXES     = (        3,       2,   3 )
    TKFRAME_1400010_UNITS    = 'DEGREES'

\begintext
```

We tell SPICE that we are going to use Euler Angles to specify the frame via the keyword *SPEC*. The keywords *ANGLES* specify the angles to be rotated, while *AXES* define the axes, where 1 means x, 2 means y and 3 means z. Each value in the *ANGLES* set refers to the corresponding value in the *AXES* one.

Finally, the units the angles are expressed in is specified.

In the next example we use the frame just created, to calculate the coordinates of the center of the Earth in the COSPAR fame.

```
#include <iostream>
using namespace std;

#include <SpiceUsr.h>


int main( char argc, char *argv[] ) {

   // Load the necessary kernels
   furnsh_c( "naif0008.tls" );
   furnsh_c( "cospar.tf" );
   furnsh_c( "new_norcia.bsp" );
   furnsh_c( "earth_fixed.tf" );
   furnsh_c( "de414.bsp" );
   furnsh_c( "pck00008.tpc" );

   // Convert to ET
   SpiceDouble et;
   utc2et_c( "2007-07-21T00:00:00", &et );

   // Calculate the position of the center of the
   // Earth in the COSPAR frame
   SpiceDouble center[ 3 ];
   SpiceDouble lt;
   spkpos_c( "399", et, "COSPAR", "LT+S", "398990",
             center, &lt );

   cout << center[ 0 ] << ", " << center[ 1 ]
        << ", " << center[ 2 ] << endl;

   return 0;

}
```

As expected, the resulting vector points along the $-z$ axis. Note that two SPK kernels have to be loaded.

- *new_norcia.bsp* contains information about the position of the New Norcia station with respect to the center of the earth.

- *de414.bsp* contains ephemeris for all the planets of the Solar System and for the Moon, relative to the Solar System barycenter[10].

Also, in order to make available the IAU_EARTH reference frame to the application, the kernels *earth_fixed.tf*, *pck00008.tpc* and *earth_fixed.tf* must be loaded.

The function that gives the position of a body with respect to another is *spkpos_c*. In the example, we want to calculate the position of the center of the Earth (NAIF ID 399) with respect to New Norcia (NAIF ID 398990), in the COSPAR frame. We want the result to be corrected for light-time and stellar aberration[11]. The function also returns the light-time between the target and the observer.

[10]The barycenter of a system is the point about all its bodies rotate; we can talk about the Solar System barycenter, the Earth-Moon barycenter, etc.

[11]See the NAIF tutorials for an explanation about these terms.

Chapter 6

Using SPICE.

Contents

So far, we've seen a few examples of using SPICE that can be helpful to get the feeling about how SPICE works and what can be done with it. This chapter will introduce a few more examples of the SPICE usage.

6.1 More on NAIF ID's.

SPICE system kernel files and subroutines refer to ephemeris objects, reference frames, and instruments by integer codes.

An ephemeris object is any object that may have ephemeris or trajectory data such as a planet, natural satellite, tracking station, spacecraft, barycenter, asteroid or comet. Each body in the Solar System is associated with an integer code for use with SPICE.

In theory, a unique integer can be assigned to each body in the Solar System, including interplanetary spacecraft. SPICE uses integer codes instead of names to refer to ephemeris bodies for three reasons:

1. Space. Integer codes are smaller than alphanumeric names.
2. Uniqueness. The names of some satellites conflict with the names of some asteroids and comets. Also, some satellites are commonly referred to by names other than those approved by IAU.
3. Context. The type of a body (barycenter, planet, satellite, comet, asteroid or spacecraft) and the system to which it belongs (Eart, Mars...) can be recovered algorithmically from the integer code assigned to a body. This is not generally true for names.

NAIF ID's for natural bodies are positive, whereas for spacecraft are negative. Instrument mounted on spacecraft also have ID codes, determined multiplying the spacecraft ID by 1000 and substracting the ordinal number of the instrument from the resulting product. For example, the NAIF ID for the Rosetta spacecraft is -226, and for the Osiris instrument onboard Rosetta, -226110.

The Barycenters of the systems of the Solar System get the integers from 0 to 9, and 10 is the ID for the Sun (see table 6.1).

For each system, the main body ID is obtained by multiplying the barycenter ID by 100 and adding 99, and the ID's for the satellites by multipling the

System	NAIF ID
Solar System barycenter	0
Mercury barycenter	1
Venus barycenter	2
Earth barycenter	3
Mars barycenter	4
Jupiter barycenter	5
Saturn barycenter	6
Uranus barycenter	7
Neptune barycenter	8
Pluto barycenter	9

Table 6.1: NAIF ID's for the Solar System barycenters.

barycenter ID by 100 and adding 1, 2, and so on. For example, the NAIF ID for the Earth is 399 and for the Moon 301. The NAIF ID's for the Jupiter system are shown in table 6.2.

System	NAIF ID
Jupiter	599
Io	501
Europa	502
Ganymede	503
Callisto	504

Table 6.2: NAIF ID's for Jupiter and its satellites.

Two functions allow the programmer to convert NAIF ID's to names and vice-versa, as shown in the next example.

```
#include <iostream>
using namespace std;

#include <SpiceUsr.h>


int main( char argc, char *argv[] ) {

   ConstSpiceInt SIZE = 100;
   SpiceChar bodyname[ SIZE ];
   SpiceInt id;
   SpiceBoolean found;

   cout << "Enter a Name: ";
   cin >> bodyname;

   // Get the NAIF ID for the given name
   bodn2c_c( bodyname, &id, &found );
   if ( found )
      cout << "ID for " << bodyname << ": " << id << endl;
   else
   cout << "ID not found for " << bodyname << endl;

   cout << "Enter an ID: ";
   cin >> id;

   // Get the name for the given ID
   bodc2n_c( id, SIZE, bodyname, &found );
   if ( found )
      cout << "Name for " << id << ": " << bodyname << endl;
   else
   cout << "Name not found for " << id << endl;

   return 0;

}
```

6.2 More on SPICE kernels.

The reader might wonder how he can find out which kernels he needs for his appliations. There is no straight answer for the question. Common sense plays a key role, as well as experience.

Usually, SPK and CK kernels for spacecrafts don't cover the whole mission[1], but only an interval of time that can be of days or weeks. So, if we need to get an SPK kernel for an application, how to find out which one?

SPICE provides a comprehensive set of applications that help users and programmers to read comments and time coverage from the kernels, create kernels and more. We'll see in this section a few of them that will help us to get information from binary kernels[2].

6.2.1 brief.

brief is an utility program that shows the body and time coverage a SPK or binary PCK kernel stores. To run brief, you just have to provide it with the name of the kernel you are interested in:

```
$ brief [kernel_name]
```

In the figure 6.1, we can see the kind of information that we can obtain by running brief: a list of the bodies for which there is information in the kernel, and the time interval covered.

For a full list of parameters, run brief without arguments.

[1]This is particularly true for the ESA missions.
[2]For text kernels, information can be obtained by just opening them with a text editor.

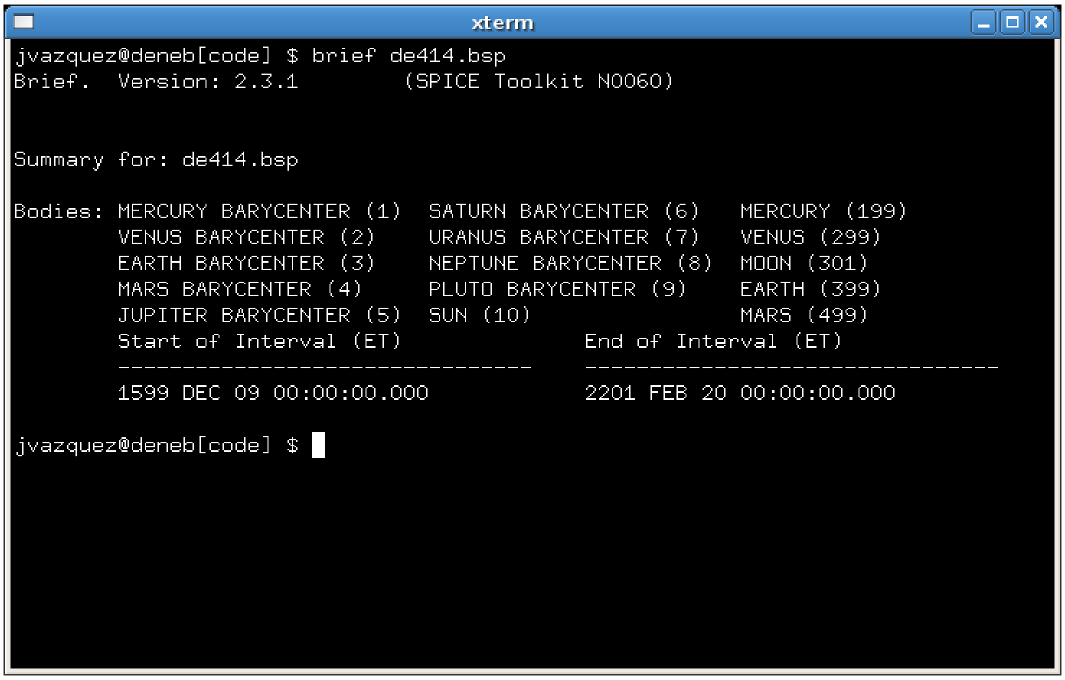

Figure 6.1: Running brief.

6.2.2 ckbrief.

ckbrief is the equivalent to brief for CK kernels. Since information in CK kernels is stored as clock ticks, we have to provide ckbrief also with a valid leapseconds kernel and with a valid clock for the spacecraft for which information is stored in the file:

```
$ ckbrief [kernel_name] [sclk_kernel] [lsk_kernel]
```

For a full list of parameters, run ckbrief without arguments.

6.2.3 commnt

Usually, the creator of a binary kernel embeds in it a lot of information about the kernel content and how it was created, that can be useful for the kernel user.

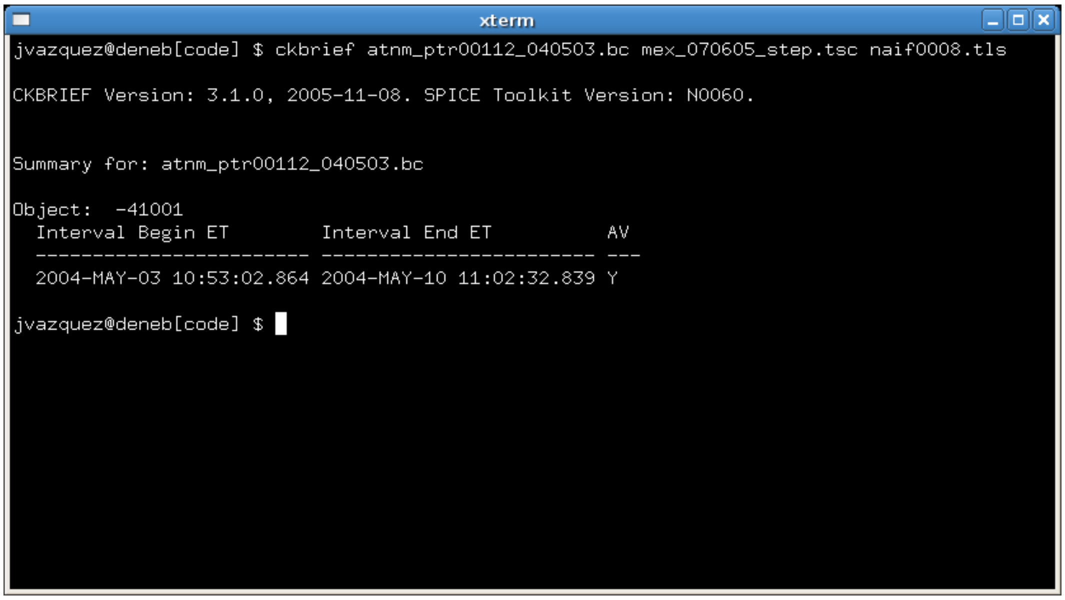

Figure 6.2: Running ckbrief.

commnt is an utility that allows to read and insert comments in a binary kernel. If you run commnt without arguments, you'll get a menu where you can specify the action to perform. If you want to read directly the comments from a file, run

```
$ commnt -r [kernel_name]
```

6.2.4 Getting SPICE kernels.

The location where to get the SPICE kernels for NASA missions is

ftp://naif.jpl.nasa.gov/pub/naif

For ESA missions, the kernels can be downloaded from

ftp://gorilla.estec.esa.int/pub/projects

Refer to the *aareadme* files in each directory to find out how they are organized and the conventions used to name and archive the kernels.

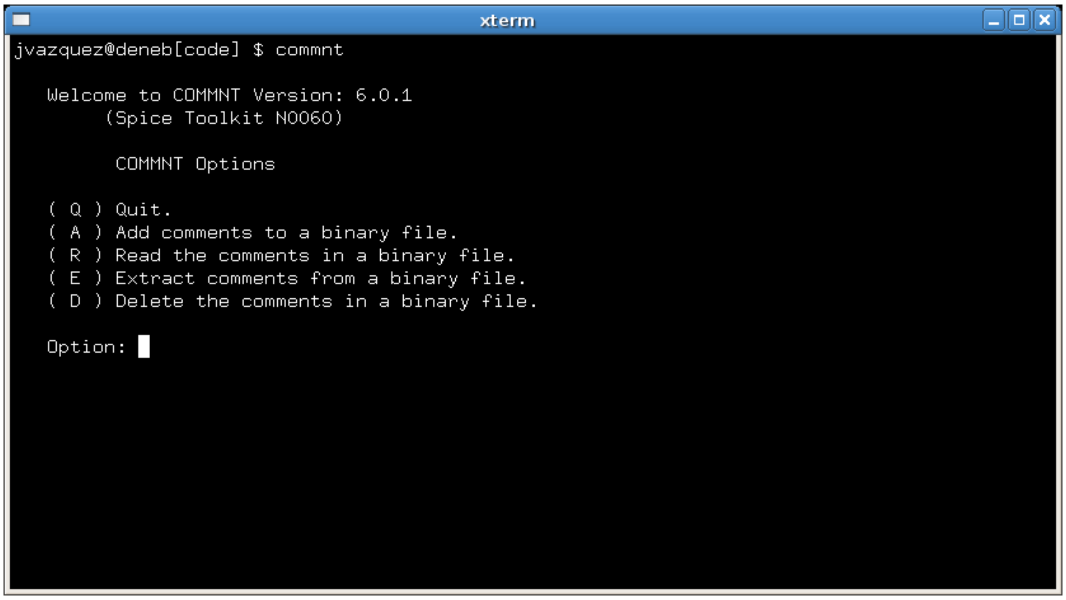

Figure 6.3: Running commnt.

6.2.5 The generic kernel.

In all the examples of this introduction, the needed kernels are loaded one by one. This has the drawback that if we find out that we need one more kernel, for example, we have to recompile the application.

SPICE allows the user to define a generic text kernel where we specify all the kernels to be loaded by the application. This way, should any change need to be done to the list of kernels, the generic kernel is modified, without needing to recompile the application.

A simple generic kernel would be:

```
\begindata
KERNELS_TO_LOAD = ( 'naif0008.tls',
                    'mex_070605_step.tsc',
                    'mex_v08.tf' )
\begintext
```

The generic kernel can be loaded by an application by a call to *furnsh_c*.

When all the kernels are located in several directories, the kernel variables *PATH_VALUES* and *PATH_SYMBOLS* help the user to save a lot of typing. Imagine that, for an application, we need CK kernels located in

```
/usr/local/mex/kernels/ck
```

, and generic kernels in

```
/usr/local/SPICE/kernels/generic
```

We can construct the following generic kernel:

```
\begindata

PATH_VALUES = ( '/usr/local/mex/kernels/ck',
                '/usr/local/SPICE/kernels/generic' )

PATH_SYMBOLS = ( 'CK',
                 'GENERIC' )

KERNELS_TO_LOAD = ( '$CK/atnm_p060401000000_00403.bc',
                    '$CK/atnm_p030602191822_00135.bc',
                    '$GENERIC/naif0008.tls'
                    '$GENERIC/earth_000101_070828_070607.bpc' )

\begintext
```

Here, every *PATH_SYMBOL* acts as an alias for the corresponding *PATH_VALUE*.

6.3 Using SPICE.

Calculating the position of a body.

SPICE provides two functions to calculate the position and velocity of a body, *spkpos_c* and *spkezr_c*. In the next examples we show how to use them.

```
#include <iostream>
using namespace std;

#include <SpiceUsr.h>

// Prints a vector on the console
void printVector( SpiceDouble *, int );

int main( char argc, char *argv[] ) {

   // Load the needed kernels
   furnsh_c( "naif0008.tls" );
   furnsh_c( "de414.bsp" );
   furnsh_c( "pck00008.tpc" );

   // Calculate the position of the Moon
   // with respect to the Earth in the IAU_EARTH
   // frame for a given date
   SpiceDouble et;
   utc2et_c( "2007-08-01T00:00:00", &et );

   SpiceDouble position[ 3 ];
   SpiceDouble lt;
   spkpos_c( "MOON", et, "IAU_EARTH", "LT+S",
             "EARTH", position, &lt );

   cout << "r = ";
   printVector( position, 3 );
   cout << endl;

   // Calculate the state of the Moon with
   //respect to the Earth in the IAU_EARTH frame
   // for a given date
   SpiceDouble state[ 6 ];
```

```
   spkezr_c( "MOON", et, "IAU_EARTH", "LT+S",
             "EARTH", state, &lt );

   cout << "s = ";
   printVector( state, 6 );
   cout << endl;

   return 0;

}

void printVector( SpiceDouble *v, int n ) {

   cout << "[ ";

   for ( int i = 0 ; i < n - 1 ; i++ )
   cout << v[ i ] << ", ";

   cout << v[ n - 1 ] << " ]";

}
```

This example calculates the position and state of the Moon with respect to the Earth for at a given time.

The *printVector* function just prints a vector on the console.

The program first loads the needed kernels, and then calculates the ET corresponding to 1st August, 2007 ad midnight. The function *spkpos_c* receives as parameters:

1. the target for which we want to calculate the position (the Moon),
2. the instant of time for which we are interested in the position of the target,
3. the frame reference in which we want the position to be expressed (IAU_EARTH)[3],

[3]Note that, should we use another frame, the coordinates of the vector position would

4. the applied correction (LT+S),
5. the observer (Earth).

It gives back the postion and the one way light time between the observer and the target.

spkezr_c, instead, returns a 6-coordinate vector with the position (coordinates 1 to 3) and velocity of the body (coordinates 4 to 6). In the example above, the first three coordinates of the state vector are indeed the position vector.

The units of the results are Km and Km/s.

The next example calculates the position of Mars with respect to Mars Express in the Mars Express reference frame. Note that for the application to be able to use that frame, the corresponding kernel (*mex_v08.tf*) has to be loaded. Also, since the location of the frame depends on the orientation and position of the spacecraft, SPK and CK kernels with information about Mars Express for the given date have to be loaded.

```
#include <iostream>
using namespace std;

#include <SpiceUsr.h>

// Prints a vector on the console
void printVector( SpiceDouble *, int );

int main( char argc, char *argv[] ) {

   // Load the needed kernels
   furnsh_c( "naif0008.tls" );
   furnsh_c( "de414.bsp" );
   furnsh_c( "ormm__070701000000_00403.bsp" );
   furnsh_c( "atnm_p060401000000_00403.bc" );
   furnsh_c( "mex_v08.tf" );
```

change, but its modulus (the distance from the Moon to the Earth) would remain the same.

```
   furnsh_c( "mex_070605_step.tsc" );

   SpiceDouble et;
   utc2et_c( "2007-07-01T00:00:00", &et );

   SpiceDouble position[ 3 ];
   SpiceDouble lt;

   // Calculate the position of the MARS with respect
   // to MEX in the MEX frame for a given date
   spkpos_c( "MARS", et, "MEX_SPACECRAFT",
             "LT+S", "MEX", position, &lt );

   cout << "r = ";
   printVector( position, 3 );
   cout << endl;

   return 0;

}

void printVector( SpiceDouble *v, int n ) {

   cout << "[ ";

   for ( int i = 0 ; i < n - 1 ; i++ )
      cout << v[ i ] << ", ";

   cout << v[ n - 1 ] << " ]";

}
```

Calculating the field of view of a camera.

We'll use for this example the superresolusion sensor (SRC) of HRSC (the camera on board Mars Express), and the instrument kernels will be introduced. This is a very simple example that gives the angular field of view of the camera.

```
#include <iostream>
using namespace std;

#include <SpiceUsr.h>

int main( char argc, char *argv[] ) {

   // Load the MEX frame kernel
   furnsh_c( "mex_v08.tf" );
   // Load the HRSC instrument kernel
   furnsh_c( "mex_hrsc_v03.ti" );

   // Get the ID for the HRSC superresolution
   // filter. It also can be found in the MEX
   // frames kernel
   SpiceInt hrsc_id;
   SpiceBoolean found;
   bodn2c_c( "MEX_HRSC_SRC", &hrsc_id, &found );

   ConstSpiceInt SIZE = 100;
   ConstSpiceInt MAX_VECTORS = 4;
   SpiceChar shape[ SIZE ];
   SpiceChar frame[ SIZE ];
   SpiceDouble bsight[ 3 ];
   SpiceDouble bounds[ MAX_VECTORS ][ 3 ];
   SpiceInt n;

   // Get the field fo view for HRSC
   getfov_c( hrsc_id, MAX_VECTORS, SIZE, SIZE,
             shape, frame, bsight, &n, bounds );

   cout << "Shape of the field of view: " <<
            shape << endl;
   cout << "Reference frame:            " <<
            frame << endl;

   SpiceDouble angles[ 4 ];

   // Compute the angles between one of the
   // corner vectors and the boresight
```

```
   angles[ 0 ] = vsep_c( bsight, bounds[ 0 ] )
                 * dpr_c();

   cout << endl << "Angle with the boresight: "
        << angles[ 0 ] << " degrees" << endl;

   cout << endl << "Angle with the other corners: "
        << endl;
   // Compute the angles between one of the corner
   // vectors and the others
   for ( int i = 1 ; i < n ; i++ ) {

      angles[ i ] = vsep_c( bounds[ i ], bounds[ 0 ] )
                    * dpr_c();
      cout << "\t" << angles[ i ] << " degrees" << endl;

   }

   return 0;

}
```

The shape of the field of view can be rectangular, polygonal, circular or elliptical. The field of view for the SRC sensor is a rectangle, as indicated in the figure 6.4, where the boresight and the angles beween one of the corners and the other three are represented.

The function that calculates the field of view is *getfov_c*. It returns the shape of the field of view, the frame in where the f.o.v. vectors and the boresight are represented, the boresight, and as many vectors as needed to represent the f.o.v.[4]

Then the angles between the first returned vector and the boresight and the other 3 (a, b, and c in the figure 6.4) are calculated by calling to the function *vsep*, which returns angles in radians. *dpr_c* returns the amounts of degrees per radian.

[4] 4 in this case, since the f.o.v. for SRC is rectangle.

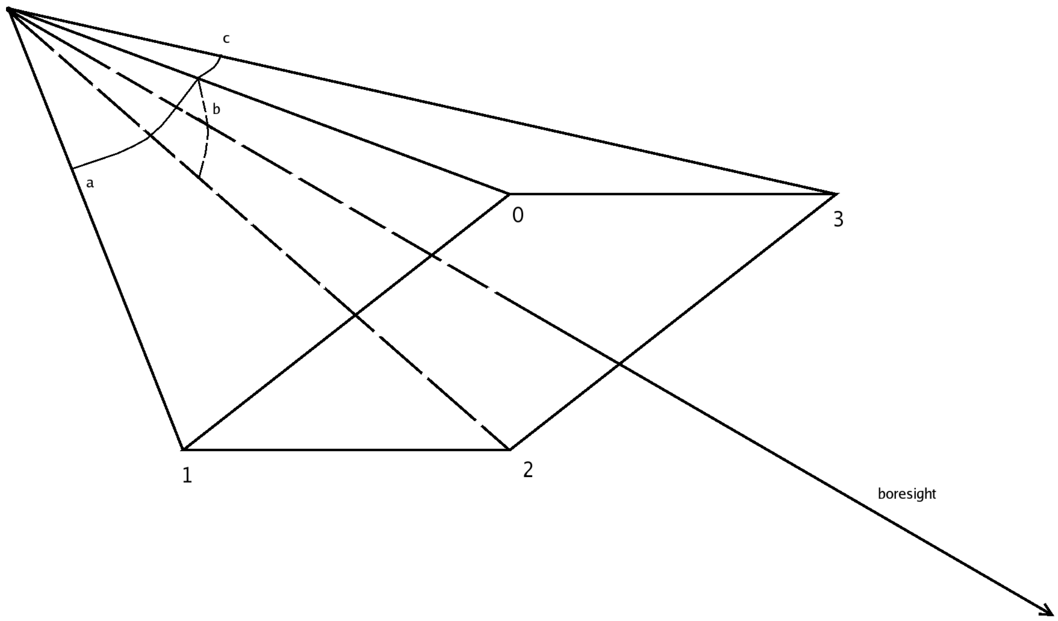

Figure 6.4: Rectangular field of view.

In the figure 6.5 can be seen that two of the angles are equal, whereas the other is twice as big as the angle between the first vector and the boresight, as it is expected.

Getting the coordinates of an image.

Let's imagine we have an image of Mars, and we want to calculate its coordinates. We'll see how to do it in the next example.

```
#include <iostream>
using namespace std;

#include <SpiceUsr.h>


int main( char argc, char *argv[] ) {

   // Load the needed kernels
   furnsh_c( "naif0008.tls" );
```

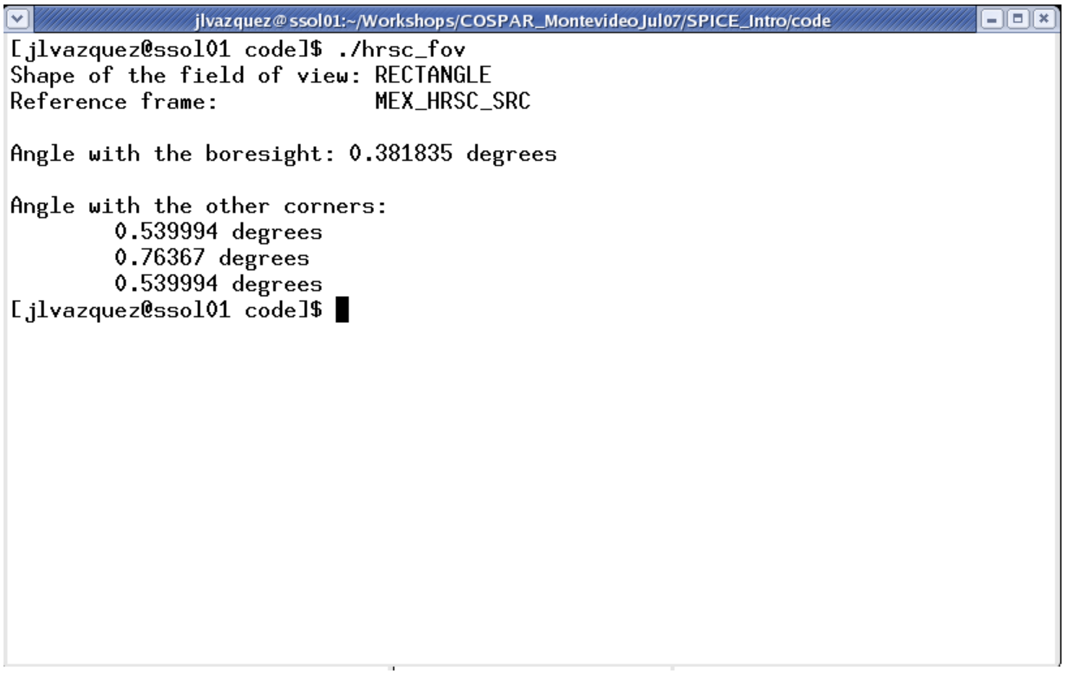

Figure 6.5: Running the field of view example.

```
furnsh_c( "de414.bsp" );
furnsh_c( "ormm__041101000000_00100.bsp" );
furnsh_c( "atnm_p030602191822_00135.bc" );
furnsh_c( "mex_v08.tf" );
furnsh_c( "mex_070605_step.tsc" );
furnsh_c( "pck00008.tpc" );
furnsh_c( "mex_v08.tf" );
furnsh_c( "mex_hrsc_v03.ti" );

SpiceDouble et;
// Calculate the ET when the image was taken
utc2et_c( "2004-11-12T04:43:11.594", &et );

// Get the ID for the HRSC GREEN filter. It also can
// be found in the MEX frames kernel
SpiceInt hrsc_id;
SpiceBoolean found;
bodn2c_c( "MEX_HRSC_GREEN", &hrsc_id, &found );

SpiceDouble mex_pos[ 3 ];
```

```
SpiceDouble lt;

// Get the position of MEX with respect to Mars in the
// the IAU_MARS frame
spkpos_c( "MEX", et, "IAU_MARS", "LT+S", "MARS",
           mex_pos, &lt );

ConstSpiceInt SIZE = 100;
ConstSpiceInt MAX_VECTORS = 4;
SpiceChar shape[ SIZE ];
SpiceChar frame[ SIZE ];
SpiceDouble bsight[ 3 ];
SpiceDouble bounds[ MAX_VECTORS ][ 3 ];
SpiceInt n;

// Get the field fo view for HRSC
getfov_c( hrsc_id, MAX_VECTORS, SIZE, SIZE, shape,
          frame, bsight, &n, bounds );

SpiceDouble rotation[ 3 ][ 3 ];
// Calculate a transformation matrix from the frame
// returned by getfov_c to IAU_MARS
pxform_c( frame, "IAU_MARS", et, rotation );

SpiceInt mars_id;

// Calculate the NAIF ID for Mars
bodn2c_c( "MARS", &mars_id, &found );

SpiceDouble mars_r[ 3 ];
SpiceInt dim;
// Calculate the radii of Mars
bodvcd_c( mars_id, "RADII", 3, &dim, mars_r );

SpiceDouble bs_mars[ 3 ];
// Rotate the boresight to the IAU_MARS frame
mxv_c( rotation, bsight, bs_mars );

SpiceDouble surface_point[ 3 ];

// Calculate the intersection of the boresight with
```

```
   // the Mars surface
   surfpt_c( mex_pos, bs_mars, mars_r[ 0 ], mars_r[ 1 ],
             mars_r[ 2 ], surface_point, &found );
   if ( found == SPICEFALSE ) {
      cout << "The vector doesn't intersect the Mars surface"
           << endl;
      return 0;
   }

   SpiceDouble latitude;
   SpiceDouble longitude;
   SpiceDouble r;

   // Convert the rectangular coordinates to latitude and
   // longitude
   reclat_c( surface_point, &r, &longitude, &latitude );

   // Translate to degrees
   longitude *= dpr_c();
   latitude *= dpr_c();

   cout << "Longitude: " << longitude << " degrees" << endl;
   cout << "Latitude:  " << latitude << " degrees" << endl;

   return 0;

}
```

In the figure 6.6 the position of MEX with respect to Mars and the boresight of SRC are represented.

We'll assume that we have an image, and we found out, by inspecting its PDS label, that it was taken at 2004-11-12T04:43:11.594 UTC. For the sake of simplicity, we'll calculate the coordinates of the boresight, although a more exhaustive work would calculate the coordinates of the four corner points. The boresight is returned by a call to *getfov_c*. In order to get its coordinates in the MARS_IAU frame, we first have to calculate a transformation matrix for the given instant of time, and then multiply the matrix by the boresight vector. The call to *surfpt_c* returns the intersection of the boresight (in the MARS_IAU) frame with the Mars surface. Previously, the radii of Mars has

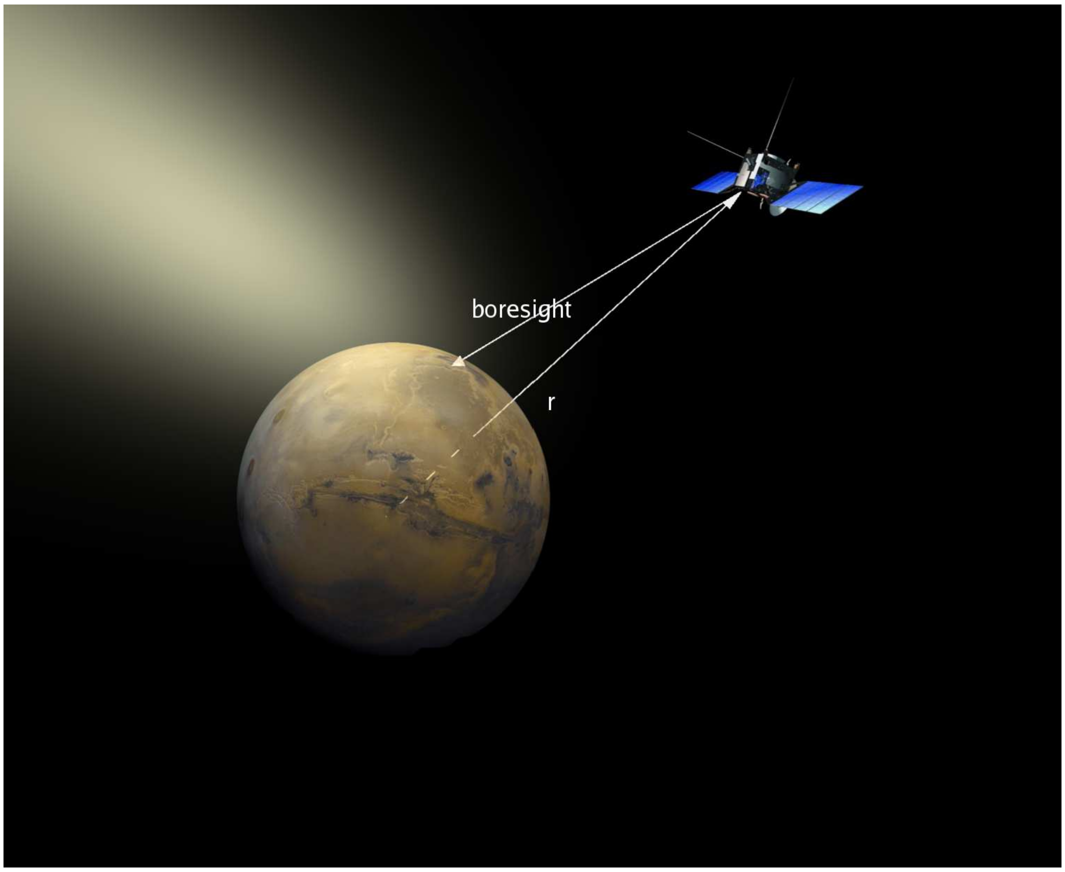

Figure 6.6: The coordinates of an image in Mars.

to be obtained by a call to *bodvcd_c*[5]. *surfpt_c* takes as parameters a vector (r in the figure 6.6), that defines the origin of the intersecting vector, the vector (the boresight), and the 3 values for the radii of the body; it returns the coordinates of the surface point where the given vector and the body surface intersect.

The call to *reclat_c* converts the coordinates of the surface point to longitude and latitude.

[5]Note that Mars is modelled as an ellipsoid. Therefore, the call to the function will return 3 values for the radius of Mars.

This page is left intentionally blank

Index