NimCfitsio Documentation

Release 0.1

Maurizio Tomasi

August 08, 2015

A set of Nim bindings to the CFITSIO library.

CHAPTER 1

Introduction

This manual describes NimCfitsio, a set of bindings to the CFITSIO library for the Nim language.

The purpose of NimCfitsio is to allow the creation/reading/writing of FITS files (either containing images or tables) from Nim programs. The interface matches the underlying C library quite close, but in a number of cases the syntax is nicer, thanks to Nim's richer and more expressive syntax.

So far the library provides an extensive, albeit not complete, coverage of the functions to read/write keywords and ASCII/binary tables. More extensive support for reading/writing images (i.e., 2D matrices of numbers) is yet to come.

The specification of the FITS file format is provided in the article Definition of the Flexible Image Transport System (FITS), version 3.0 (Astronomy & Astrophysics, 524, A42, 2010).

CHAPTER 2

Installation

[I plan to add support for Nimble very soon. At the moment, you’re on your own, sorry...]

CHAPTER 3

Basic access to FITS files

In this section we describe the functions used to access FITS files and get general information about their content.

All the code from now on can be used only if the NimCfitsio module is imported with the following command:

```
import cfitsio
```

Virtually every function in NimCfitsio requires as its first argument a variable of type `FitsFile`.

object**`FitsFile`**

This object contains the following fields:

Name	Type	Meaning
`file`	`InternalFitsStruct` (private)	Used internally by CFITSIO
`fileName`	`string`	Name of the file

In case of error, all the NimCfitsio functions raise an exception of type `EFitsException`:

object**`EFitsException`**

The fields of this object are the following:

Field name	Type	Meaning
`code`	`int`	CFITSIO error code identifier
`message`	`string`	Descriptive error message
`errorStack`	`seq[string]`	List of all the CFITSIO error messages raised

3.1 Opening FITS files for read/write

The CFITSIO library provides several functions to open a file for reading/writing, and NimCfitsio provides a wrapper to each of them. Here is a general overview of their purpose:

Function	Purpose
`openFile()`	Open a generic file. Access through FTP and HTTP is allowed
`openData()`	Open a file and move to the first HDU containing some data
`openTable()`	Like `openData`, but the HDU must contain a table
`openImage()`	Like `openData`, but the HDU must contain an image

All the prototypes of these functions accept the same parameters and return the same result. Here is a short example that shows how to use them:

```
import cfitsio

var f = cfitsio.openFile("test.fits", ReadOnly)
```

```
try:
    # Read data from "f"
finally:
    cfitsio.closeFile(f)
```

If the underlying CFITSIO function fails when opening the file (e.g, because the file does not exist), a `EFitsException` will be raised.

enum**IoMode**= `ReadOnly, ReadWrite`

This enumeration is used by all the procedures that open an existing FITS file.

proc**openFile** (fileName : string, ioMode : IoMode) → *FitsFile*

Open the FITS file whose path is *fileName*. If *ioMode* is `ReadOnly`, the file is opened in read-only mode and any modification is forbidden; if *ioMode* is `ReadWrite`, then write operations are allowed as well as read operations.

If the file cannot be opened, a `EFitsException` is raised.

If the underlying CFITSIO library supports them, protocols like `ftp://` or `http://` can be used for *fileName*. Compressed files (e.g. `.gz`) may be supported as well.

You must call `closeFile()` once the file is no longer needed, in order to close the file and flush any pending write operation.

proc**openData** (fileName : string, ioMode : IoMode) → *FitsFile*

This function can be used instead of `openData()` when the user wants to move to the first HDU containing either an image or a table. Its usage is the same as `openFile()`.

proc**openTable** (fileName : string, ioMode : IoMode) → *FitsFile*

This function is equivalent to `openData()`, but it moves to the first HDU containing either a binary or ASCII table.

If the file cannot be opened, or it does not contain any table, a `EFitsException` is raised.

proc**openImage** (fileName : string, ioMode : IoMode) → *FitsFile*

This function is equivalent to `openData()`, but it moves to the first HDU containing an image.

If the file cannot be opened, or it does not contain any image, a `EFitsException` is raised.

3.2 Creating files

enum**OverwriteMode**= `Overwrite, DoNotOverwrite`

proc**createFile** (fileName : string, overwriteMode : OverwriteMode = Overwrite) → *FitsFile*

Create a new file at the path specified by *fileName*. If a file already exists, the behavior of the function is specified by the *overwriteMode* parameter: if it is equal to `DoNotOverwrite`, a `EFitsException` exception is raised, otherwise the file is silently overwritten.

The return value is a `FitsFile` object that should be closed using either `closeFile()` or `deleteFile()`.

Here is an example about how to use this procedure:

```
import cfitsio

var f = cfitsio.createFile("test.fits")
try:
    # Write data into "f"
finally:
    cfitsio.closeFile(f)
```

proc**createDiskFile***(fileName : string, overwriteMode : OverwriteMode = Overwrite) → *FitsFile*
: This function is equivalent to :nim:proc::*createFile*, but it does not attempt to interpret *fileName* according to CFITSIO's extended syntax rules.

3.3 Closing files

proc**closeFile**(fileObj : var FitsFile)
: Close the file and flush any pending write operation on it. The variable *fileObj* can no longer be used after a call to `closeFile`.

  See also *deleteFile()*.

proc**deleteFile**(fileObj : var FitsFile)
: This procedure is similar to *closeFile()*, but the file is deleted after having been closed. It is mainly useful for testing purposes.

3.4 Other file-related functions

In this section we list all the other functions that work on the file as a whole, but do not fit in any of the previous sections.

proc**getFileName**(fileObj : var FitsFile) → string
: Return the name of the file associated with the FITS file variable *fileObj*. Since this variable calls CFITSIO instead of simply returning the *file* field of *FitsFile*, it could fail. In the latter case, it will throw a *EFitsException* exception.

proc**getFileMode**(fileObj : var FitsFile) → *IoMode*
: Return the I/O mode of the file.

proc**getUrlType**(fileObj : var FitsFile) → string
: Return the kind of URL of the file. Possible values are e.g. `file://`, `ftp://`, `http://`.

CHAPTER 4

HDU functions

4.1 Moving through the HDUs

A FITS files is composed by one or more HDUs. NimCfitsio provides a number of functions to know how many HDUs are present in a FITS file and what is their content. (To create a new HDU you have first to decide which kind of HDU you want. Depending on the answer, you should read *Table functions* or *Image functions*.)

enum**HduType**= `Any = -1, Image = 0, AsciiTable = 1, BinaryTable = 2`

HDU types recognized by NimCfitsio. The `Any` type is used by functions which perform searches on the available HDUs in a file. See the FITS specification documents for further information about the other types.

NimCfitsio (and CFITSIO itself) uses the concept of "current HDU". Each `FitsFile` variable is a stateful object. Instead of specifying on which HDU a NimCfitsio procedure should operate, the user must first select the HDU and then call the desired procedure.

proc**moveToAbsHdu**(fileObj : var FitsFile, num : int) → *HduType*

Select the HDU at position *idx* as the HDU to be used for any following operation on the FITS file. The value of *num* must be between 1 and the value returned by `getNumberOfHdus()`.

proc**moveToRelHdu**(fileObj : var FitsFile, num : int) → *HduType*

Move the current HDU by *num* positions. If *num* is 0, this is a no-op. Positive as well as negative values are allowed.

proc**moveToNamedHdu**(fileObj : var FitsFile, hduType : HduType, name : string, ver : int = 0)

Move to the HDU whose name is *name*. If *ver* is not zero, then the HDU must match the version number as well as the name.

If no matching HDU are found, a `EFitsException` is raised.

proc**getNumberOfHdus**(fileObj : var FitsFile) → int

Return the number of HDUs in the FITS file.

Table functions

5.1 Creating tables

enum**TableType**= `AsciiTable, BinaryTable`

This enumeration lists the two types of tables that can be found in a FITS file. Binary tables have the advantage of allowing any datatype supported by CFITSIO; moreover, they are more efficient in terms of required storage.

enum**DataType**= `dtBit, dtInt8, dtUint8, dtInt16, dtUint16, dtInt32, dtInt64, dtFloat32, dtFlo`

Data types recognized by NimCfitsio.

object**TableColumn**

This type describes one column in a table HDU. It is used by *`createTable()`*. Its fields are listed in the following table:

Field	Type	Description
`name`	`string`	Name of the column (not longer than 8 chars)
`dataType`	*`DataType`*	Data type
`width`	`int`	For strings, this gives the maximum number of chars
`repeatCount`	`int`	Number of items per row
`unit`	`string`	Measure unit

proc**createTable**(fileObj : var FitsFile, tableType : TableType, numOfElements : int64, fields : openArray[TableColumn], extname : string)

Create a new table HDU after the current HDU. The file must have been opened in `ReadWrite` mode (this is automatically the case if *f* has been returned by a call to *`createFile()`*).

The value of *numOfElements* is used to allocate some space, but it can be set to zero: calls to functions like *`writeColumn()`* will make room if needed.

5.2 Reading columns

The NimCfitsio library provides an extensive set of functions to read data from FITS table HDUs. Each of them initializes an “open array” type that is passed as a `var` argument: this allows to initialize arrays as well as `seq` types.

The functions implemented by NimCfitsio to read columns of data are the following:

Function name	Type
`readColumnOfInt8()`	`int8`
`readColumnOfInt16()`	`int16`
`readColumnOfInt32()`	`int32`
`readColumnOfInt64()`	`int64`
`readColumnOfFloat32()`	`float32`
`readColumnOfFloat64()`	`float64`
`readColumnOfString()`	`string`

We describe here the many incarnations of a function `readColumn()` which operates on a generic type `T`. Such function however does not exist: such description should be applied to any of the procedures listed in the table above.

proc**`readColumn`** (fileObj : var FitsFile, colNum : int, firstRow : int, firstElem : int, numOfElements : int, dest : var openArray[T], destNull : var openArray[bool], destFirstIdx : int)

Read a number of elements equal to *numOfElements* from the column at position *colNum* (the position of the first column is 1), starting from the row number *firstRow* (starting from 1) and the element *firstElem* (within the row; this also starts from 1). The destination is saved in the *dest* array, starting from the index *destFirstIdx*. The array *destNull* must be defined on the same indexes as the array *dest*; `readColumn()` initializes it with either *true* or *false*, according to the nullity of the corresponding element in *dest*.

As an example, the following call reads 3 elements from the first column of file *f*. The values read from the file are saved in `dest[2]`, `dest[3]`, and `dest[4]`, because *destFirstIdx* is 2. Note that *nullFlag* is not as long as *dest* (4 elements instead of 10): this is ok, as the upper limit of the indexes used by the procedure is 4.

```
var dest : array[int32, 10]
var nullFlag : array[int32, 4]
f.readColumnOfInt32(1, 4, 1, 3, dest, destNull, 2)
```

proc**`readColumn`** (fileObj : var FitsFile, colNum : int, firstRow : int, firstElem : int, numOfElements : int, dest : var openArray[T], destFirstIdx : int, nullValue : T)

This second version of the procedure allows for quickly substitute null values with the value *nullValue*.

proc**`readColumn`** (fileObj : var FitsFile, colNum : int, firstRow : int, firstElem : int, dest : var openArray[T], nullValue : T)

In many cases it is not needed to save data in the middle of the *dest* array. This version of `readColumn` uses the length of *dest* as the value to be used for *numOfElements*. The implicit value of *firstElem* is `low(dest)`.

proc**`readColumn`** (fileObj : var FitsFile, colNum : int, dest : var openArray[T], nullValue : T)

This is the simplest possible version of `readColumn`. It reads as many values as they fit in *dest*, starting from the first one (i.e., *firstRow* and *firstElem* are implicitly set to 1).

5.3 Writing columns

The functions implemented by NimCfitsio to write columns of data are the following:

Function name	Type
`writeColumnOfInt8()`	`int8`
`writeColumnOfInt16()`	`int16`
`writeColumnOfInt32()`	`int32`
`writeColumnOfInt64()`	`int64`
`writeColumnOfFloat32()`	`float32`
`writeColumnOfFloat64()`	`float64`
`writeColumnOfString()`	`string`

proc**`writeColumn`** (fileObj : var FitsFile, colNum : int, firstRow : int, firstElem : int, numOfElements : int, values : var openArray[T], valueFirstIdx : int, nullPtr : ptr T = nil)

Write *numOfElements* values taken from *values* into the column at position *colNum* in the current HDU of the

FITS file *f*. The elements will be written starting from the row with number *firstRow* (the first row is 1) and from the element in the row at position *firstElem* (the first element is 1). The values that are saved in the file start from the index *valueFirstIdx*, i.e., they are `values[valueFirstIdx]`, `values[valueFirstIdx+1]` and so on.

The *nullPtr* argument is a **pointer** to a variable that contains the "null" value: any value in *values* that is going to be written is compared with `nullPtr[]` and, if it is equal, it is set to NULL.

proc**`writeColumn`**(fileObj : var FitsFile, colNum : int, firstRow : int, firstElem : int, values : var openArray[T], nullPtr : ptr T = nil)

This is a wrapper around the previous definition of *`writeColumn()`*. It assumes that `valueFirstIdx = low(values)`.

proc**`writeColumn`**(fileObj : var FitsFile, colNum : int, values : var openArray[T], nullPtr : ptr T = nil)

This function is a wrapper around the previous definition of *`writeColumn()`*. It writes all the elements of the *values* array into the column *colNum*.

CHAPTER 6

Image functions

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index