Working with large FITS files

This tutorial, built on the Create a very large FITS file from scratch guide, works through building a very large (too large to fit in memory) FITS file with multiple HDUs. It covers creating both large Image and Table extensions. It is aimed at users already quite familiar with the FITS format.

Authors

C. E. Brasseur

Learning Goals

- Build a *large* FITS file (*large* means is too large to fit in memory all at once)
- Make a *large* FITS Image extension
- Make a *large* FITS Table extension

Keywords

FITS, file input/output, memory mapping

Companion Content

- FITS Standard
- Astropy FITS Documentation
- Python madvise documentation
- Madvise system call documentation
- Create a very large FITS file from scratch

Summary

This is an advanced tutorial. We will be building a very large multi-extension FITS file from scratch, going through both how to create Images and Arrays too large to fit in memory, and how to fill those structures once created.

If you don't want to know about the inner workings of the FITS format, just stop here. If you don't want to know but nevertheless neeed to, proceed with caution, that's how I started and now here I am writing this tutorial.

Building a large FITS file

But before we begin…

There are a few things we need to know about the FITS format so I will collect them there. And if you are either now, or in the future, wondering "why is it like that" the answer is that the FITS format was originally designed and optimised for magnetic tape. This means that the FITS format was originally designed for sequentual reading rather than random access, so the FITS format has no index (listing at what bytes various parts of the file start) but instead is formatted in 2880 byte chunks so that a tape reader head can simply skip forward by 2880 bytes repeatedly and check if a new section has begun. This is mostly trivia for us as modern users, but there are a few implications. Firstly, when reading FITS files the Astropy FITS module by default reads them "lazily" meaning that it does not tabulate all the extensions until it needs to (i.e. when the user requests a specific extension or calls the `info()` function). Secondly, and most crucially for this tutorial, when creating FITS extension manually, the most critical part of creating a new *valid* FITS extension is making sure the number of bytes is a multiple of 2880. These are of course but a few of the quirks of the FITS format, to read about all of them in their full and eccentric glory, see the FITS standard document.

A short review of terminology:

- The basic block of a FITS file is called a Header Data Unit (HDU).
- Each HDU contains two elements, the header and the data.
- A FITS file consistes of one or more HDUs.
- Astropy represents a FITS file as an HDUList, where each extension is an HDU of a specific type (i.e PrimaryHDU, ImageHDU, etc).
- For more details see the Astropy FITS Documentation.

Imports

We don't need many modules for this. The central one is of course `astropy.io.fits`; the `mmap` import helps with efficiency, but is not available on all systems, and is ultimately not essential. So if yours is a system without it never fear, you can just comment out those lines.

```
import os

from time import time

import numpy as np
import astropy.units as u

from astropy.io import fits
from astropy.table import Table

from mmap import MADV_SEQUENTIAL
```

A little helper function

Because this tutorial is all about building a huge file, we'll write a little function to print the file size in a a variety of units.

```
def print_file_size(path, unit="B"):
    size = os.path.getsize(path) * u.byte

    if unit == "KB":
        print(f"{size.to(u.kB):.1f}")
    elif unit == "MB":
        print(f"{size.to(u.MB):.1f}")
    elif unit == "GB":
        print(f"{size.to(u.GB):.1f}")
    elif unit == "FITS":
        print(f"{size.value / 2880:.1f} FITS block")
    else:
        print(f"{size:.0f}")
```

Primary HDU

In this tutorial we are going to build a properly formated multi-extension FITS file, so before we get into the matter of creating a massive FITS file we will build a basic Primary HDU, and write it to file where it will be the basis for our monstrous FITS file.

```
# Make some header entries for important information
primary_header_cards = [
    ("ORIGIN", "Fancy Archive", "Where the data came from"),
    ("DATE", "2024-03-05", "Creation date"),
    ("MJD", 60374, "Creation date in MJD"),
    ("CREATOR", "Me", "Who created this file"),
]

# Build the Primary HDU object and put it in an HDU list
primary_hdu = fits.PrimaryHDU(header=fits.Header(primary_header_cards))
hdu_list = fits.HDUList([primary_hdu])

# Write the HDU list to file
big_fits_fle = "./patagotitan.fits"
hdu_list.writeto(big_fits_fle, overwrite=True)
```

Before we continue let's verify our (currently tiny) FITS file is valid. We will do this by calling the `info()` function which will hang if the file is not a valid FITS format.

```
with fits.open(big_fits_fle) as hdu_list:
    hdu_list.info()
```

```
Filename: ./patagotitan.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       7   ()
```

Let's also look at the file size in bytes and FITS blocks. Note that it is exactly one FITS block.

```
print_file_size(big_fits_fle)
print_file_size(big_fits_fle, "FITS")
```

```
2880 byte
1.0 FITS block
```

Large Image HDU

Now we get to the meat of the tutorial. We are going to expand out the FITS file to fit a 40,000 x 40,000 pixel image (~13 GB).

Note: If this is problamatically big for your system, adjust `array_dims` below. All of the steps will still work as expected with smaller data, it's simply an unnessesarily complex methodology when dealing with data sizes that fit in memory.

```
array_dims = [40_000, 40_000]
```

First we build an ImageHDU object with a small data array. The data in the array does not matter because we won't be using it, but the data type needs to be correct, and you need to take note of how many bytes per element goes with that data type. In this example we are building a `float64` array, so each element uses 8 bytes of memory.

```
data = np.zeros((100, 100), dtype=np.float64)
hdu = fits.ImageHDU(data)
```

Now we pull out just the header, and adjust the NAXIS keywords to match our desired giant-array dimensions. This is a critical step, it is telling the FITS file how large the data array is and must match the data array size we are going to add.

We also set a name for the extension which is optional, but helpful, because it allows us to refer to that extension by name as well as index.

```
hdu.name = "BIG_IMG"
header = hdu.header

header["NAXIS2"] = array_dims[0]
header["NAXIS1"] = array_dims[1]
```

The next step is to write *just the header* to the end of our soon to balloon FITS file.

Note: At the end of this step our file is temporarily NOT a valid FITS file.

```
with open(big_fits_fle, "ab") as FITSFLE:  # 'ab' means open to append bytes
    FITSFLE.write(bytearray(header.tostring(), encoding="utf-8"))
```

```
print_file_size(big_fits_fle)
print_file_size(big_fits_fle, "FITS")
```

```
5760 byte
2.0 FITS block
```

Now we calculate the number of bytes we need for our gargantuan array, remembering that the result *must* to be a multiple of 2880 bytes to conform to the FITS standard.

Note: The `astype(np.int64)` is not necessary on all systems, but some still default to int32 and therefore throw an overflow error.

```
elt_size = 8  # Bytes needed for an array element
arraysize_in_bytes = (
    (np.prod(array_dims).astype(np.int64) * elt_size + 2880 - 1) // 2880
) * 2880
```

Now we need to expand the file by that many bytes. To do this we seek to the desired new end of the file and write a null byte.

```
filelen = os.path.getsize(big_fits_fle)

with open(big_fits_fle, "r+b") as FITSFLE:
    FITSFLE.seek(filelen + arraysize_in_bytes - 1)
    FITSFLE.write(b"\0")
```

Now lets see how big our FITS file has become.

```
print_file_size(big_fits_fle, "GB")
print_file_size(big_fits_fle, "FITS")
```

```
12.8 Gbyte
4444447.0 FITS block
```

So just about 13 GB as expected. And a lot more FITS blocks, but still an exact multiple of the FITS block size which is what we want to see.

Filling the big array

Now we have a big ol' empty array that we want to put some stuff in. At this point we are working with a gigantic file, so we need to start being careful we don't ask for it to be loaded wholesale into memory (if you *do* try to do that, it won't break anything you will just get a `Cannot allocate memory` error).

What this means is that the memmap argument must be set to True. This is usually the default (unless you have changed it in your astropy configuration settings). There are also a number of modes the file can be opened with:

- `readonly`: Default behavior. Opens the file in readonly mode, meaning that to save any changes you need to write to a whole new file (or overwrite the existing one). This means that while with `memmap=True` the entire FITS file is not loaded into memory, the system is prepared to load it all in memory if the user changes something, and so will still throw an error if it is not *possible* to allocate memory for the whole file even as it does not allocate it at the minute. For big files where this is not possible it will fall back to `denywrite` mode and produce a warning.
- `denywrite`: This is similar to readonly except that it does not allow the FITS object to be altered and then written to a new file. For our tremendous FITS file we will use this mode when we want to access but not change the file.

- `update`: This mode allows a file to be updated in place.
- `append`: This allows more extensions to be added to an existing FITS file, but doe *not* allow changing data already in the file when it is opened.

We will use the `update` mode to fill our immense array in place.

```
hdu_list = fits.open(big_fits_fle, mode="update", memmap=True)
```

Before we comence we will call the `info()` function to verify that our FITS file is valid and the enormous array is the size we expect.

```
hdu_list.info()
```

```
Filename: ./patagotitan.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       8   ()
  1  BIG_IMG       1 ImageHDU         8   (40000, 40000)   float64
```

Pulling out the majestic array for convenience.

```
data_array = hdu_list[1].data
```

If you are on a system with the `madvise` call (you're on your own figuring that out), you can set madvise to `MADV_SEQUENTIAL` for the `data_array`. This tells the memory mapping that you are going to be accessing the array in a sequential manner and allows it to be more efficient in how it handles memory allocation based on that. How much this actually affects the time it takes to perform the filling operation will depend on your specific system and the array sizes you are working with.

Note: If you change how you fill your array to something not sequential, don't set this.

```
mm = fits.util._get_array_mmap(data_array)
mm.madvise(MADV_SEQUENTIAL)
```

Now we fill the jumbo array block by block. We want the block size to comfortably fit in memory. The `block_size` I am using yields an ~1.3 GB array, adjust as your system requires.

We also print the time it take for every block. If you have the `MADV_SEQUENTIAL` flag set, the individual block fill operation will generally take longer, and the close operation quite fast, while if the `MADV_SEQUENTIAL` flag is not set the reverse is generally true. This is because in the first case, the data is being flushed to disk at once, while in the second it builds up untill the system needs more memory or the file is closed and it writes it all at once. Which is more efficent on your setup will vary with block and file size.

```
block_size = 4000

tt = 0
for i, j in enumerate(range(0, array_dims[0], block_size)):
    it = time()

    sub_arr = np.ones((block_size, array_dims[1])) * i
    data_array[j : j + block_size, :] = sub_arr

    tm = time() - it
    print(f"{i}: {tm:.0f} sec")
    tt += tm

it = time()
hdu_list.close()
tm = time() - it
print(f"Closing: {tm:.0f} sec")
tt += tm

print(f"Total fill time: {tt / 60:.1f} min")
```

```
0: 2 sec

1: 2 sec

2: 2 sec

3: 3 sec

4: 3 sec

5: 4 sec

6: 3 sec

7: 3 sec

8: 3 sec

9: 2 sec


Closing: 6 sec
Total fill time: 0.5 min
```

Checking the file contents

We've filled our outsize array, but we want to make sure that it is correct. So we'll open the elephantine file in `denywrite` mode and check.

```
hdu_list = fits.open(big_fits_fle, mode="denywrite", memmap=True)
data_array = hdu_list[1].data
```

```
it = time()
for i, j in enumerate(range(0, array_dims[0], block_size)):
    print(
        f"{i}: Data match is {(data_array[j : j + block_size, :] == i).all()}: {time
    )
    it = time()

hdu_list.close()
```

```
0: Data match is True: 0 sec


1: Data match is True: 0 sec


2: Data match is True: 0 sec


3: Data match is True: 0 sec


4: Data match is True: 0 sec


5: Data match is True: 0 sec


6: Data match is True: 0 sec


7: Data match is True: 0 sec


8: Data match is True: 0 sec


9: Data match is True: 0 sec
```

Large Table HDU

In the last section we expanded our FITS file to add a colossal image extension, in this section we will do the same for a table extension. The method is similar, but with a few key differences.

As with the mighty array, we start by making a small table where the specific data is not important but the data types are. In particular, the maximum string length for columns cannot be changed on the fly, so any string columns must be given the maximum number of characters needed.

```
small_tbl = Table(
    names=["Name", "Population", "Prince", "Years since fall", "Imports", "Exports"],
    dtype=["U128", int, "U128", np.float64, "U2048", "U2048"],
    rows=[
        [
            "Vangaveyave",
            1297382,
            "Oriana",
            34.6,
            "wine, cheese",
            "ahalo cloth, pearls, foamwork",
        ],
        ["Tkinele", 50000, "n/a", 92.3, "none", "none"],
        [
            "Amboloyo",
            50937253,
            "Rufus",
            1504.2,
            "pears, textiles, spices",
            "wine, timber",
        ],
        [
            "Xiputl",
            3627373,
            "Anastasiya",
            346.8,
            "silk, perfumes, pigments",
            "stone, cotton",
        ],
        [
            "Old Damara",
            437226732,
            "Melissa Damara",
            25.3,
            "wool, timber",
            "spices, silk",
        ],
        [
            "Western Dair",
            8045728302,
            "Belu",
            876.3,
            "shellfish, salt",
            "cured meat, wool",
        ],
    ],
)

table_hdu = fits.BinTableHDU(data=small_tbl)
table_hdu.header["EXTNAME"] = "BIG_TABLE"
```

The header for this table HDU gives us the information to determine how many bytes we need for our mammoth table.

```
table_hdu.header
```

```
XTENSION= 'BINTABLE'           / binary table extension
BITPIX  =                    8 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                 4368 / length of dimension 1
NAXIS2  =                    6 / length of dimension 2
PCOUNT  =                    0 / number of group parameters
GCOUNT  =                    1 / number of groups
TFIELDS =                    6 / number of table fields
TTYPE1  = 'Name    '
TFORM1  = '128A    '
TTYPE2  = 'Population'
TFORM2  = 'K       '
TTYPE3  = 'Prince  '
TFORM3  = '128A    '
TTYPE4  = 'Years since fall'
TFORM4  = 'D       '
TTYPE5  = 'Imports '
TFORM5  = '2048A   '
TTYPE6  = 'Exports '
TFORM6  = '2048A   '
EXTNAME = 'BIG_TABLE'
```

The `NAXIS1` keyword gives the length of a single table row in bytes, and the `NAXIS2` keyword holds the number of rows in the table. So to get the total size of the humongous table in bytes we simply multiply `NAXIS1` by the number of rows desired (adjusting for FITS block size). Here I choose a million rows which is about 4GB, adjust as necessary for your system.

```
num_rows = 1_000_000
tablesize_in_bytes = ((table_hdu.header["NAXIS1"] * num_rows + 2880 - 1) // 2880) *
```

Now we adjust the `NAXIS2` keyword to match our new table length and write just the header to the end of our towering FITS file, as we did for the oversize array extension.

```
table_hdu.header["NAXIS2"] = num_rows

with open(big_fits_fle, "ab") as FITSFLE:
    FITSFLE.write(bytearray(table_hdu.header.tostring(), encoding="utf-8"))
```

Before we expand the file, we'll look at the current file size.

```
print_file_size(big_fits_fle, "GB")
```

```
12.8 Gbyte
```

Now, just as for the vast array, we seek `tablesize_in_bytes` beyond the current end of the file and write a null byte.

```
filelen = os.path.getsize(big_fits_fle)

with open(big_fits_fle, "r+b") as FITSFLE:
    FITSFLE.seek(filelen + tablesize_in_bytes - 1)
    FITSFLE.write(b"\0")
```

And we can see that the filesize has indeed increased by about 4GB.

```
print_file_size(big_fits_fle, "GB")
```

```
17.2 Gbyte
```

Adding data to the titanic table

We can now open the prodigious FITS file in `update` mode and fill in our table. Here we'll run a little comparison. FITS files store table data row by row, so it should be faster to fill the table by row rather than column (and doing so allows us to again advise the memory mapper with `MADV_SEQUENTIAL`), but memory handling is complex and system dependent so when it really matters it's best to do testing for your individual setup.

We'll start by printing the file info to ensure we have the valid FITS file we expect.

```
hdu_list = fits.open(big_fits_fle, mode="update", memmap=True)
hdu_list.info()
```

```
Filename: ./patagotitan.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       8   ()
  1  BIG_IMG       1 ImageHDU         8   (40000, 40000)   float64
  2  BIG_TABLE     1 BinTableHDU     21   1000000R x 6C   [128A, K, 128A, D, 2048A,
```

```
table_data = hdu_list["BIG_TABLE"].data

tt = 0

for col in small_tbl.colnames:
    it = time()

    if col == "Name":
        table_data[col] = ["Zunidth", "Astandalas"] * (num_rows // 2)
    elif col == "Population":
        table_data[col] = np.arange(num_rows)
    elif col == "Prince":
        table_data[col] = ["Cliopher Lord Mdang", "His Radiancy Artorin Damara"] *
            num_rows // 2
        )
    elif col == "Years since fall":
        table_data[col] = np.linspace(5, 1000, 1000000)
    elif col == "Imports":
        table_data[col] = ["tea", "roses"] * (num_rows // 2)
    elif col == "Exports":
        table_data[col] = ["magic", "empire"] * (num_rows // 2)

    tm = time() - it
    print(f"{col}: {tm:.0f} sec")
    tt += tm

it = time()
hdu_list.close()
tm = time() - it
print(f"Closing: {tm:.0f} sec")
tt += tm

print(f"Total fill time: {tt / 60:.1f} min")
```

Name: 3 sec

Population: 5 sec

Prince: 1 sec

Years since fall: 9 sec

Imports: 11 sec

Exports: 16 sec

Closing: 233 sec
Total fill time: 4.6 min

```
hdu_list = fits.open(big_fits_fle, mode="update", memmap=True)
table_data = hdu_list["BIG_TABLE"].data

# Comment out these lines if your system does not have madvise
mm = fits.util._get_array_mmap(table_data)
mm.madvise(MADV_SEQUENTIAL)

block_len = 200_000
data_list = (list(small_tbl.as_array()) * (block_len // len(small_tbl) + 1))[:block_

tt = 0
for j, i in enumerate(range(0, num_rows, block_len)):
    it = time()

    table_data[i : i + block_len] = data_list

    tm = time() - it
    print(f"{j}: {tm:.0f} sec")
    tt += tm

it = time()
hdu_list.close()
tm = time() - it
print(f"Closing: {tm:.0f} sec")
tt += tm

print(f"Total fill time: {tt / 60:.1f} min")
```

```
0: 58 sec


1: 39 sec


2: 42 sec


3: 41 sec


4: 42 sec


Closing: 227 sec
Total fill time: 7.5 min
```

For my system as I write this tutorial the times are incredibly close, but your mileage may vary.

Checking our data

Now let's again open up our behemothic FITS file and check that the data was loaded correctly.

```
hdu_list = fits.open(big_fits_fle, mode="denywrite", memmap=True)
hdu_list.info()
```

```
Filename: ./patagotitan.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       8   ()
  1  BIG_IMG       1 ImageHDU         8   (40000, 40000)   float64
  2  BIG_TABLE     1 BinTableHDU     21   1000000R x 6C   [128A, K, 128A, D, 2048A,
```

```
table_data = hdu_list["BIG_TABLE"].data
```

Checking the first few rows of each fill block.

```
for j, i in enumerate(range(0, num_rows, block_len)):
    val = (small_tbl == Table(table_data[i : i + 6])).all()
    print(f"{j}: {val}")
```

```
0: True
1: True
2: True
3: True
4: True
```

Closing the file.

```
hdu_list.close()
```

Adding an Extra Small HDU

The last thing we will do is add another small HDU to the monumental FITS file. We can do this in the usual way because the extension we are adding is of a normal size.

```
small_hdu = fits.ImageHDU(data=np.random.random((10, 10)))
small_hdu.header["EXTNAME"] = "MINI_IMG"
```

Before we add the additional HDU we'll remind ourself of the current whopping filesize.

```
print_file_size(big_fits_fle, "KB")
```

```
17168011.2 kbyte
```

Because we don't have to do anything funky with the file size we can just open the magnificent FITS file in `append` mode and write the whole HDU, and it is a very fast operation.

```
with fits.open(big_fits_fle, mode="append", memmap=True) as hdu_list:
    hdu_list.append(small_hdu)
```

And we can see that adding this tiny additional extension barely changes the monster file size.

```
print_file_size(big_fits_fle, "KB")
```

```
17168017.0 kbyte
```

And now if we open the mondo FITS file we can see that additional extension.

```
with fits.open(big_fits_fle, mode="denywrite", memmap=True) as hdu_list:
    hdu_list.info()
```

```
Filename: ./patagotitan.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       8   ()
  1  BIG_IMG       1 ImageHDU         8   (40000, 40000)   float64
  2  BIG_TABLE     1 BinTableHDU     21   1000000R x 6C   [128A, K, 128A, D, 2048A,
  3  MINI_IMG      1 ImageHDU         8   (10, 10)   float64
```

Cleanup

Lastly, we'll remove the leviathan file we created.

```
os.remove(big_fits_fle)
```