Access JSOC data from Matlab environments

JSOC contains a large collection of data files. While the data files are store SUMS, the information about the data files are stored in keywords, in DRMS. These keywords are used to describe the content, the hierarchical relationship among these files as well as their current locations. Users can use these keywords to make queries searching for the data they need. JSOC makes these information accessible via a web interface. Where, users can search, check for storage status, make request and download the data files.

JSOC currently provides these functions:
show_series: for listing series name available in JSOC
jsoc_info : for displaying the series structure, record summary,
keywords, values and pseudo keywords.
jsoc_fetch : for requesting for a record set.

These functions can be called directly when we are JSOC network, or over a web interface when we query remotely. These functions can return data inform of JSON objects, which can parsed into Matlab data structures.

Here is some Matlab scripts (.m files) that users can use to query DRMS interactively from Matlab environments. These scripts are also serve as examples, showing how to extract information returned by these functions.

series_list.m : returns series names existing in JSOC
series_struct.m : examines the structure of a data series.
rs_summary.m : returns number of records in a series
rs_list.m : for selectively displaying keywords values, storage status
of a record set.
exp_request.m : for requesting and exporting data files from JSOC to user.

Note:
It is more efficient to call these functions directly when we are on JSOC network (default).
When access remotely, please append 'web_access' to these commands, which will pass the queries through a web interface (http://jsoc.stanford.edu/cgi-bin/ajax/)
These scripts use a JSON parser written by F Glineur, download from Matlab Central (parse_json.m)

To run, please copy these .m files (from /home/timh/matlab_jsoc/*.m) to your working directory.
Run matlab, then these commands.
(Perhaps, also setenv MATLAB_SHELL /bin/tcsh before running matlab)

---

series_list.m
Usage: series_list 'filter_string'

```
>> series_list hmi.lev
   hmi.lev0
   hmi.lev0TEST_0445_0015
   hmi.lev0TEST_0451_0007
   ...
   hmi.lev0b
   hmi.lev0c
   hmi.lev0d
   hmi.lev0e
   hmi.lev0e_test
   hmi.lev0f
   hmi.lev1
   lm_jps.hmi_lev0
   lm_jps.hmi_lev0_60d
   su_production.hmi_lev1e

Number of series matched [hmi.lev] = 59
```

---

series_struct.m
Usage: series_struct 'series_name'

```
>> series_struct 'hmi.lev0e'

Keywords:
  ORIGIN                          ORIGIN Location where file made
```

```
  DATE                                Date_time of processing; ISO 8601
  DATE__OBS                           [DATE-OBS] Date when observation started; ISO 8601
  T_OBS                               Observation time
  EXPTIME                             Exposure duration: mean shutter open time
  TIME                                Time of observation: seconds within a day
  MJD                                 Date of observation: modified julian day
  TELESCOP                            For HMI: SDO/HMI
  INSTRUME                            For HMI: HMI_SIDE1 or HMI_FRONT2
  CAMERA                              For HMI: 1 or 2
  WAVELNTH                            For HMI: 617.33 nm
  IMGAPID                             Image Application ID
  IMGFPT                              FIRST_PACKET_TIME
  BITSELID                            Bit select ID, r
  COMPID                              Compression ID; n,k
  CROPID                              Crop table ID
  DATAVALS                            Actual number of data values in image
  FID                                 FID Filtergram ID
  ...
  image_bscale
  cparms_sg000
  cparms_sg001
  image_bzero
  image_sm_bzero
  image_sm_bscale

Segments:
  image
        name: 'image'
       units: 'dn'
    protocol: 'fits'
        dims: 'VARxVAR'
        note: 'lev0 data fits file'

  image_sm

        name: 'image_sm'
       units: 'dn'
    protocol: 'fits'
        dims: 'VARxVAR'
        note: 'lev0 data small fits file with headers'

  image_png

        name: 'image_png'
       units: 'dn'
    protocol: 'generic'
        dims: 'VARxVAR'
        note: 'lev0 data sm image'


Links:
DB Index:
    'FSN'    'T_OBS'


Interval:
    FirstRecord: 'hmi.lev0e[45853]'
    FirstRecnum: 404796
    LastRecord: 'hmi.lev0e[1879935]'
    LastRecnum: 426947
    MaxRecnum: 435652

ans =
         note: 'HMI'
    retention: 60
     unitsize: 1
      archive: 1
    tapegroup: 2
    primekeys: {'FSN'}
      dbindex: {'FSN'  'T_OBS'}
     keywords: {1x86 cell}
     segments: {[1x1 struct]  [1x1 struct]  [1x1 struct]}
        links: {0x1 cell}
     Interval: [1x1 struct]
       status: 0
```

```
rs_summary.m
Usage: rs_summary 'series_name'

>> rs_summary 'hmi.lev0e'
     count: 110591
    status: 0
```

```
rs_list.m
Usage: rs_list query_string

>> rs_list 'hmi.lev0e[1800000-1800010] key=T_OBS,FSN'

Keywords:
  T_OBS
  FSN

2008.09.07_06:22:22.21_UTC    1800000
2008.09.07_06:22:24.21_UTC    1800001
2008.09.07_06:22:26.22_UTC    1800002
2008.09.07_06:22:28.21_UTC    1800003
2008.09.07_06:22:30.23_UTC    1800004
2008.09.07_06:22:32.22_UTC    1800005
2008.09.07_06:22:34.23_UTC    1800006
2008.09.07_06:22:36.23_UTC    1800007
2008.09.07_06:22:38.23_UTC    1800008
2008.09.07_06:22:40.16_UTC    1800009
2008.09.07_06:22:42.16_UTC    1800010

Records found 11

ans =

    keywords: {[1x1 struct]  [1x1 struct]}
    segments: {0x1 cell}
       links: {0x1 cell}
       count: 11
      status: 0
```

```
>>a=rs_list('hmi.lev0e[1800000-1800001] key=T_OBS,FSN,*online*,*sunum*,*recnum*,*size*,*retain*,
*logdir*');

Keywords:
  T_OBS
  FSN
  *online*
  *sunum*
  *recnum*
  *size*
  *retain*
  *logdir*

2008.09.07_06:22:22.21_UTC    1800000 N       10104055.000000       299574.000000  12786184       N/A
       No log avaliable
2008.09.07_06:22:24.21_UTC    1800001 N       10103997.000000       299557.000000  12751824       N/A
       No log avaliable

Records found 2
```

```
To get the segment names (file paths) of a record set, we can do this

>> a = rs_list('hmi.lev0e[1800000-1800010] key=**NONE** seg=**ALL**');
Keywords:
Records found 11
Segments:
  image
      name: 'image'
    values: {1x11 cell}
      dims: {1x11 cell}
  image_sm
```

```
      name: 'image_sm'
    values: {1x11 cell}
      dims: {1x11 cell}

  image_png
      name: 'image_png'
    values: {1x11 cell}
      dims: {1x11 cell}

Links:

>> a.segments{3}.values{1}
ans =
/SUM0/D21848888/D10104055/S00000/image.png
```

```
rs_online_check.m
Usage: rs_online_check 'querry_string'

>> b=rs_online_check('su_production.lev0f_hmi[706300-706500]');
Found 15 records, (all) online = Yes, size = 2136654490
```

```
exp_request.m
Usage: exp_request 'query_string'

>> a=exp_request('su_production.lev0f_hmi[706315]{image}');
        count: 1
         size: 14002560
          dir: [1x0 char]
         data: {[1x1 struct]}
    requestid: [1x0 char]
       method: 'url_quick'
     protocol: 'as-is'
         wait: 0
       status: 0

http://jsoc.stanford.edu//SUM1/D21527735/S00009/image.fits

image.fits downloaded!
```

```
>> a=exp_request('su_timh.supersid_test_data_3[][NAA][]');
        count: 1
         size: 620477
          dir: [1x0 char]
         data: {[1x1 struct]}
    requestid: [1x0 char]
       method: 'url_quick'
     protocol: 'as-is'
         wait: 0
       status: 0

http://jsoc.stanford.edu//SUM3/D19153270/S00000/WSO_NAA_2009-03-02.csv

WSO_NAA_2009-03-02.csv downloaded!
```

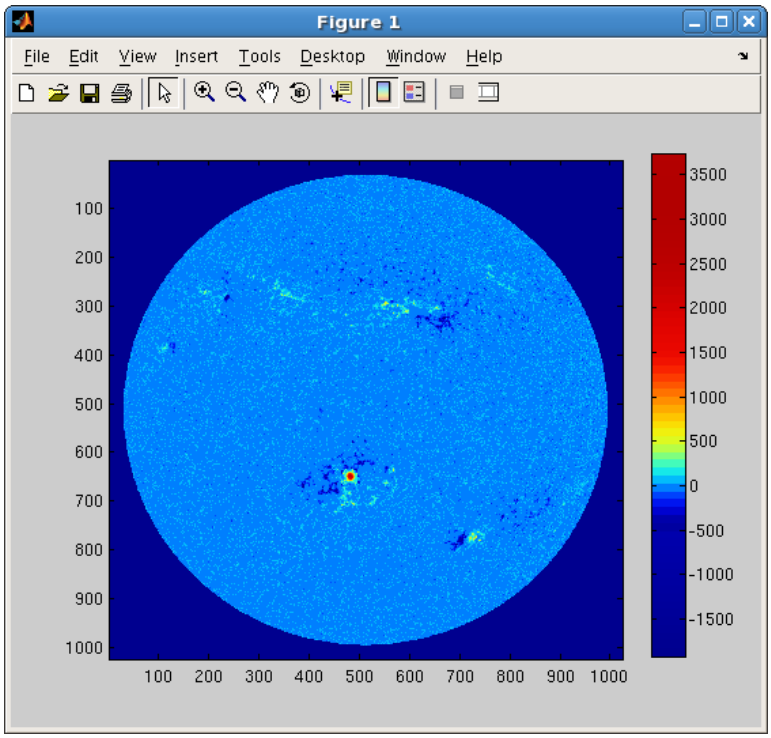
To query JSOC remotely

When we are not on JSOC network, we still can query similarly by appending 'web_access' to above commands. They will be passed through the web interface http://jsoc.stanford.edu/cgi-bin/ajax/

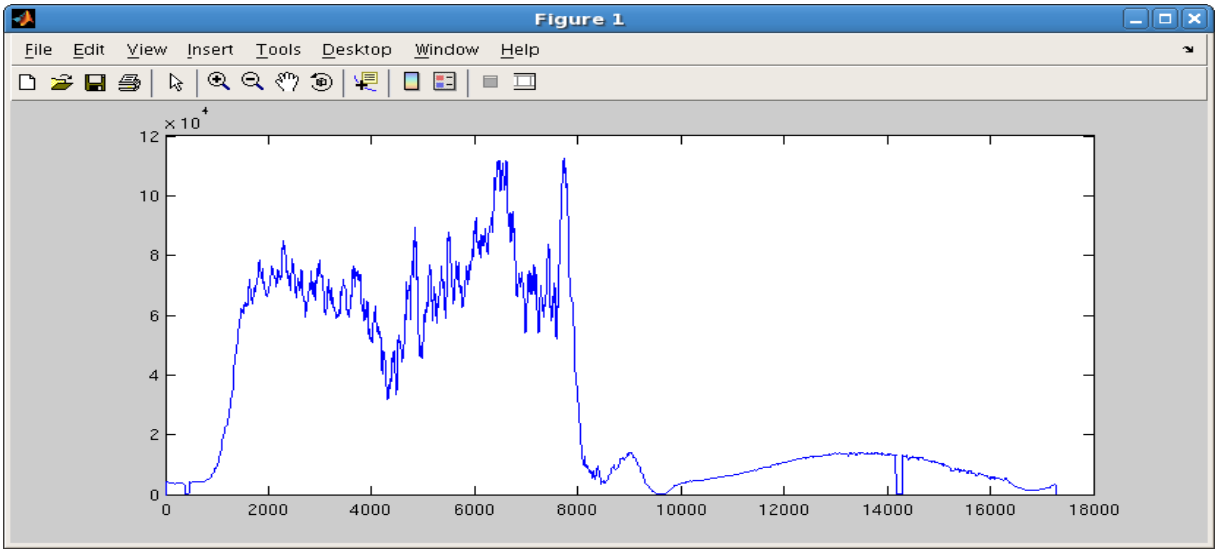
```
>> series_list('hmi.lev','web_access')
>> series_struct('hmi.lev0e','web_access')
>> a=rs_list('hmi.lev0e[1800000-1800001] key=T_OBS,FSN,*online*,*sunum*,
*recnum*,*size*,*retain*,*logdir*','web_access');
```

Here are some useful commands to display a FITS file or plot a SID file.

```
>> fitsinfo('image.fits');
>> fitsinfo('coffee_cup_sunspot.fits');
>> handle = fitsread('coffee_cup_sunspot.fits');
>> imagesc(handle);
```




```
>> s = csvread('WSO_NAA_2009-03-02.csv',11,1);
>> plot(s);
```



To display the FITS header:

For instance, we would like to check whether the file is compressed or not, here are 2 examples, examining the FITS file headers.

Example1:
coffee_cup_sunspot.fits is an uncompressed FITS file, which contains 1 HDU 'PrimaryData'. This HDU has 183 keywords, DataSize = 4194304.

```
>> h= fitsinfo('coffee_cup_sunspot.fits')
h =
       Filename: 'coffee_cup_sunspot.fits'
    FileModDate: '18-Nov-2009 10:05:52'
       FileSize: 4213440
       Contents: {'Primary'}
    PrimaryData: [1x1 struct]

>> h.PrimaryData
ans =
            DataType: 'single'
                Size: [1024 1024]
            DataSize: 4194304
    MissingDataValue: []
           Intercept: 0
               Slope: 1
              Offset: 17280
            Keywords: {183x3 cell}
>> h.PrimaryData.Keywords
ans =
    'SIMPLE'      'T'                      [1x48 char]
    'BITPIX'      [               -32]              ''
    'NAXIS'       [                 2]              ''
    'NAXIS1'      [              1024]              ''
    'NAXIS2'      [              1024]              ''
    ...
    'DSNAME'                [1x47 char]              ''
    'PROTOCOL'    'RDB.FITS'                         ''
    'DSDS_UID'    [                 0]              ''
    ...
```

Example 2:
mdi.fd_M_96m_lev18.141166.data.fits is a compressed FITS file, which has 2 HDU:
- PrimaryData contain only a few standard keywords without data.
- BinarayTable has the first keyword 'XTENSION' = 'BINTABLE', with data.

```
>> h=fitsinfo('mdi.fd_M_96m_lev18.141166.data.fits')
h =
       Filename: 'mdi.fd_M_96m_lev18.141166.data.fits'
    FileModDate: '18-Nov-2009 10:14:45'
       FileSize: 1451520
       Contents: {'Primary'  'Binary Table'}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]

>> h.PrimaryData
ans =
            DataType: 'int16'
                Size: []
            DataSize: 0
    MissingDataValue: []
           Intercept: 0
               Slope: 1
              Offset: 2880
            Keywords: {7x3 cell}
>> h.BinaryTable
ans =
                Rows: 1024
             RowSize: 8
             NFields: 1
         FieldFormat: {'1PB(4134)'}
      FieldPrecision: {'int32'}
           FieldSize: 2
            DataSize: 8192
    MissingDataValue: {[]}
           Intercept: 0
               Slope: 1
              Offset: 5760
```

```
       ExtensionSize: 1435085
     ExtensionOffset: 13952
            Keywords: {26x3 cell}


>> h.PrimaryData.Keywords
ans =
    'SIMPLE'     'T'     [1x48 char]
    'BITPIX'     [16]    [1x48 char]
    'NAXIS'      [ 0]    [1x48 char]
    'EXTEND'     'T'     [1x48 char]
    'COMMENT'      ''    [1x72 char]
    'COMMENT'      ''    [1x72 char]
    'END'          ''            ''

>> h.BinaryTable.Keywords
ans =
    'XTENSION'    'BINTABLE'           [1x48 char]
    'BITPIX'      [              8]    [1x48 char]
    'NAXIS'       [              2]    [1x48 char]
    'NAXIS1'      [              8]    [1x48 char]
    'NAXIS2'      [           1024]    [1x48 char]
    'PCOUNT'      [        1435085]    [1x48 char]
    'GCOUNT'      [              1]    [1x48 char]
    'TFIELDS'     [              1]    [1x48 char]
    'TTYPE1'      'COMPRESSED_DATA'    [1x48 char]
    'TFORM1'      '1PB(4134)'          [1x48 char]
    'ZIMAGE'      'T'                  [1x48 char]
    'ZBITPIX'     [             32]    [1x48 char]
    'ZNAXIS'      [              2]    [1x48 char]
    'ZNAXIS1'     [           1024]    [1x48 char]
    'ZNAXIS2'     [           1024]    [1x48 char]
    'ZTILE1'      [           1024]    [1x48 char]
    'ZTILE2'      [              1]    [1x48 char]
    'ZCMPTYPE'    'RICE_1'             [1x48 char]
    'ZNAME1'      'BLOCKSIZE'          [1x48 char]
    'ZVAL1'       [             32]    [1x48 char]
    'ZNAME2'      'BYTEPIX'            [1x48 char]
    'ZVAL2'       [              4]    [1x48 char]
    'BLANK'       [    -2.1475e+09]            ''
    'BZERO'       [              0]            ''
    'BSCALE'      [         0.0100]            ''
    'END'                        ''            ''
```

MFITSIO

From the above examples, we saw that, when the FITS file is compressed, the BITPIX, NAXIS, NAXIS1, NAXIS2 values do not reflect the actual image 's data type and dimensions. And, data array returned is not readily usable.

```
    'BITPIX'      [              8]    [1x48 char]
    'NAXIS'       [              2]    [1x48 char]
    'NAXIS1'      [              8]    [1x48 char]
    'NAXIS2'      [           1024]    [1x48 char]
```

Currently out of the box, Matlab provides only 2 functions to read FITS uncompressed files into Malab:

```
    fitsinfo() to return HDU header
    fitsread() to return HDU data
```

From CFITSIO main page (www.heasarc.gsfc.gov/fitsio/), it is recommended to use MFITSIO developed by Damina Eads at Los Alamos National Laboratory.

MFITSIO (http://public.lanl.gov/eads/mfitsio/) provides a few more functions:

```
    fits_read_header()
    fits_write_header()
    fits_delete_keyword()
    fits_read_image()
    fits_write_image()
    fits_read_image_subset()
    fits_write_image_subset()
```

These allows users to read/write FITS header and data.

However, most FITS images stored SUMS are as Rice compressed format. For efficiency, we added these 2 functions allow users to directly read compressed file in SUMS.

fits_read_header_compress() which returns the correct (uncompressed) image's type and dimensions
fits_read_image_compress() which returns the uncompress image in mxArray ready for use.

To use these functions in matalb, we can add path to (/home/timh/matlab_jsoc/mex/) then call these functions like this.

```
>> path(path,'/home/timh/matlab_jsoc/mex')
>> a=fits_read_header_compress('coffee_cup_sunspot.fits');
>> a
a =
      BITPIX: -32
       NAXIS: 2
      NAXIS1: 1024
      NAXIS2: 1024
        .....
      DSNAME: 'prog:mdi,level:lev1.8,series:fd_M_96m_01d[1994]'
    PROTOCOL: 'RDB.FITS'
    DSDS_UID: 0
    CONFORMS: 'TS_EQ'
       .....

>> b=fits_read_image_compress('coffee_cup_sunspot.fits');
>> imagesc(b)
```

For example: image.fits is a compressed image.

Here, we use
fits_read_header() to inspect the HDU keywords and data.
fits_read_header_compress() to acquire header and data for use.

Note:
HDU 0 does not contain any data
HDU 1 contains keywords which does not reflect the actual (uncompressed) dimensions of the image.
fits_read_header_compress() filters out ("Z" structure) keywords and return only the user's keywords along with BITPIX, NAXIS, NAXIS1, NAXIS2 as of actual, uncompress image.

```
>> a=fits_read_header('image.fits[0]')
a =
    BITPIX: 16
     NAXIS: 0

>> a=fits_read_header('image.fits[1]')
a =
    XTENSION: 'BINTABLE'
      BITPIX: 8
       NAXIS: 2
      NAXIS1: 8
      NAXIS2: 4096
      PCOUNT: 8001556
      GCOUNT: 1
     TFIELDS: 1
      TTYPE1: 'COMPRESSED_DATA'
      TFORM1: '1PB(2438)'
      ZIMAGE: 1
     ZBITPIX: 16
      ZNAXIS: 2
     ZNAXIS1: 4096
     ZNAXIS2: 4096
      ZTILE1: 4096
      ZTILE2: 1
    ZCMPTYPE: 'RICE_1'
      ZNAME1: 'BLOCKSIZE'
       ZVAL1: 32
      ZNAME2: 'BYTEPIX'
       ZVAL2: 2
       BLANK: -32768

>> a=fits_read_header_compress('image.fits[1]')
a =
    BITPIX: 16
```

```
     NAXIS: 2
    NAXIS1: 4096
    NAXIS2: 4096
     BLANK: -32768

>> d=fits_read_image_compress('image.fits[1]');
>> imagesc(d)
```