Programming IDL Objects: Why and how to do it

D. M. Zarro (ADNET/GSFC)

CONTENTS

---

1. INTRODUCTION

IDL objects were first introduced in version 5.0. The concept of object oriented programming (OOP) is not new. It is the basis of many modern languages such as C++ and Java. The idea behind OOP is that operations and data go hand in hand. Why? Because if you are handed a piece of data without any way of operating on it, then it would not be very useful to you. Furthermore, even if you were given software to operate on the data, the data may still not be useful without a set of rules to describe how the software interacts with it.

Consequently, the main thrust of OOP techniques is to keep operations and data married together so that they understand each other. This concept is called **encapsulation**. A valid question to ask is: what is wrong with non-OOP programming techniques in which procedures (or functions) are written to operate on data? For example, one can develop a reader to read a file, a plotter to plot it, and a writer to save it, as in the following pseudocode:

```
IDL> read,file,data
IDL> plot,data
IDL> write,file,data
```

The short answer is that there is nothing wrong with using non-OOP techniques. The long answer is that, as data sets become complicated and operations more complex, simple procedures are no longer simple. Procedures often become a series of steps that are called repeatedly or they require enhancements to support different datasets, which were not originally considered. In the OOP world, these procedures are called **methods**. These methods operate on data in exactly the same way as regular non-OOP procedures but they are special in that they function differently depending on the data in question. This concept is called **polymorphism** and will be described in more detail in **section 2.3**

The purpose of this tutorial is not to make the case for using OOP techniques in IDL, but to illustrate situations in which such techniques are more advantageous than non-OOP techniques. We start with demonstrating how to create a very basic IDL data object, followed by how to interact and manipulate it, and finally how to apply it to a real world example of reading and displaying a FITS image. Although not a strong pre-requisite, we assume that the reader is familiar with the use of IDL pointers and

structures.

1.1 Creating an IDL object

An IDL object is simply a container in memory space. This container holds:

- data
- descriptions of the data
- procedures for operating on the data

The descriptions are called **properties**, and the procedures are called **methods**. The interesting twist is that both properties and methods can change as the data changes.

The first step in creating an object is deciding a name. The name is important because it should signify something about the the data. For example, if we are developing software to sell different types of cars, such as Fords, Toyotas, and Hondas, then the logical object name is probably *car*. This generic name is referred to as the object *class*. There is nothing special or magic about the name, although it should be unique. However, the name does become important later when we develop different objects.

Since we are keeping things simple - and are already thinking in terms of data - we name our object class *data*. We define this class in a file named *data__define.pro* that contains the following code:

```
function data::init

 return,1

end

pro data__define

 void={data,ptr:ptr_new()}
 return

end
```

So, what does the above all mean. Let's take it in steps:

1. **CLASS filename:** naming the file *data__define.pro* is an IDL convention for writing a class definition file. The class name has to be part of the file name, followed by double underscore and the word *define*. If we were defining a *car* class, then the car class definition file would be named *car__define.pro*.

2. **INIT method:** the first line in the file defines a method to initialize the object. It is mandatory for all definition files. The naming convention for this (and all methods) is class name, followed by double colon and the method name *(::init)*. In this example, the *init* method doesn't do very much. We will modify it later to actually do something useful. For now, it simply returns unity, which signifies that the object is successfully created.

3. **CLASS definition:** the second line defines an IDL structure in which the data will be contained. An IDL structure is a variable that can hold different data types such as strings, floats, integers, and even other structures. In this example, it contains a pointer field (called *ptr*) that we create with the command: *ptr_new()*. The latter function creates an address in memory where future data will reside. The advantage of using a pointer is that we can dynamically insert data even though we don't know its size ahead of time. In general, the class definition file is where the attributes of the object are defined as structure tags. These attributes are called properties. In our *data* object example, we have a single property namely *ptr*.

Having written and saved the above file into the current IDL working directory, we create a *data* object as follows:

```
IDL> a=obj_new('data')
IDL> help,a
A               OBJREF    = ObjHeapVar37(DATA)
```

Creating the *data* object is called **instantiation**. The IDL function call to *obj_new('data')* looks for the file *data__define.pro*, compiles it, and executes the init and structure definition code that creates the *data* object. Calling help on the output variable *a* shows that it is an object of class type *data*. In the language of OOP, the variable *a* is an **instance** of the *data* class. Currently, this object doesn't do very much other than waiting around for some data to come along and some operations to perform on the data.

1.2 Putting data into an object

As currently defined, we cannot interact with the *data* object because we have not defined any methods to communicate with it. The next step is to write a method that allows us to insert data into the object. We re-edit the file *data__define.pro* as follows:

```
function data::init

;-- allocate memory to pointer when initializing object

 self.ptr=ptr_new(/allocate)
 return,1

end

;---------------------------------------------------------------

pro data::set,value

;-- if data value exists, then insert into pointer location

 if n_elements(value) ne 0 then *(self.ptr)=value
 return

end

;---------------------------------------------------------------

pro data__define

 void={data,ptr:ptr_new()}
 return

end
```

Several things are going on that require explanation:

1. **SELF variable:** we have added the following line to *data::init*,

   self.ptr=ptr_new(/allocate)

   What is *self*? When we are working inside an object, we reference the object using the variable name *self*. The above line says:

   "Take the structure field named ptr, which we have defined to be a pointer, and allocate new memory to it using the IDL pointer function ptr_new(/allocate)."

   We only need to do this once when the *data* object is first created.

2. **SET method**: we name the method for inserting data *set*. Any name will do, but following IDL convention we precede it with the class name *data::*. This method is a procedure that takes the variable *data* as a command line argument. As is always good practice, we check that the variable *value* exists before proceeding by calling *n_elements(value)*. If the latter returns a non-zero value, then we insert the value into the pointer value *ptr*.

3. **Inserting data**: if you are confused by the syntax **(self.ptr)=value*, don't worry. Here is the simple breakdown. Recall that *self* is an internal reference to the *data* object, which is defined to be a structure with a tag (or field) named *ptr*. We reference this pointer as *self.ptr*. The latter is not an actual data value but an address to where data is located (or stored). The asterisk symbol references the value of the data stored at the pointer location. When the *data* object is first initialized, there is no data at this location. The IDL statement **(self.ptr)=value* says:

   "Take my input data value and insert (or copy) it to this pointer location."

   In the language of OOP, this action is called *setting the object property*.

1.3 Getting data out of an object

In addition to inserting data, we need a method for extracting data from the object. We shall call this method *data::get*, and include it in the file *data__define.pro* as follows:

```
function data::init

;-- allocate memory to pointer when initializing object

 self.ptr=ptr_new(/allocate)
 return,1

end

;---------------------------------------------------------------

pro data::set,value

;-- if data value exists, then insert into pointer location

 if n_elements(value) ne 0 then *(self.ptr)=value
 return

end

;---------------------------------------------------------------

function data::get,value

;-- if data value is stored in object pointer, then copy it out

 if n_elements(*(self.ptr)) ne 0 then value=*(self.ptr)

 return,value

end

;---------------------------------------------------------------

pro data__define

 void={data,ptr:ptr_new()}
 return

end
```

The *get* method differs from *set* in that we have defined it to be a function. The choice of function versus procedure is more of a matter of convenience than convention. The function first checks for the existence of a data value at the pointer location *self.ptr*. If data is present, then the value **(self.ptr)* is copied to the output variable *value*. If there is no data value, then an undefined value is returned.

1.4 Destroying an object

When finished with using an object, it is recommended that the memory allocated to the object be released. All objects should therefore have a method that will take care of cleaning up after themselves. The IDL naming convention for this method is *::cleanup*. In the case of the *data* object, this cleanup method would involve freeing the pointer property of any allocated data. To implement a cleanup method, we include the following lines in the file *data__define.pro*:

```
function data::cleanup

;-- free memory allocated to pointer when destroying object

 ptr_free,self.ptr
 return

end
```

The IDL procedure *ptr_free* flushes the pointer variable *self.ptr* of any saved data and re-initializes it. This method is not called directly. Instead it is called automatically when the object is destroyed using the IDL *obj_destroy* procedure as follows:

```
IDL> obj_destroy,a                              ;-- destroy object
```

```
IDL> help,a
A               OBJREF    = ObjHeapVar4        ;-- object is now null
```

2. WORKING WITH OBJECTS

Having defined a *data* class, we next demonstrate how it can be used in common applications, and how it can be extended to perform different functions.

2.1 Modifying object properties

Because we have used a pointer as its property, the *data* object can accomodate any data type. For example, let's create a 2-dimensional float array and insert it and extract it as follows:

```
IDL> image=findgen(512,512)
IDL> a=obj_new('data')        ;-- create object variable a
IDL> a->set,image             ;-- insert image
IDL> image2=a->get()          ;-- extract image
IDL> help,image2
IMAGE2            FLOAT     = Array[512, 512]
```

This example introduces the **arrow** syntax for calling methods. The statement *a->set,image* says:

"Call the method named set on the object variable named a, and pass the argument variable named image."

We apply the same syntax when calling the *get* method except that we invoke it as a function and return the value into the output variable *image2*.

Next, let's try inserting and extracting a data structure such as the system variable *!d*:

```
IDL> a->set,!d
IDL> var=a->get()
IDL> help,var,/st
** Structure !DEVICE, 17 tags, length=88:
   NAME            STRING    'X'
   X_SIZE          LONG                640
   Y_SIZE          LONG                512
   X_VSIZE         LONG                640
   Y_VSIZE         LONG                512
   X_CH_SIZE       LONG                  6
   Y_CH_SIZE       LONG                 12
   X_PX_CM         FLOAT           40.0000
   Y_PX_CM         FLOAT           40.0000
   N_COLORS        LONG          16777216
   TABLE_SIZE      LONG               256
   FILL_DIST       LONG                 1
   WINDOW          LONG                 0
   UNIT            LONG                 0
   FLAGS           LONG            328124
   ORIGIN          LONG      Array[2]
   ZOOM            LONG      Array[2]
```

In this example, we insert the variable *!d* into the object and then retrieve it into the variable *var*. Note that we only create the object variable once, and recycle it as necessary. We can of course create as many data objects as we like and store different data types accordingly. Just remember to give them different variable names.

2.2 Adding object methods

As we have already seen, we can add new methods to an object by editing its class definition file. We can make the *data* object more useful by giving it the ability to read data from a file and plot the data. In the following example, we open the file *data__define.pro* and add two methods that we conveniently call *data::read* and *data::plot*:

```
pro data::read,file

 if n_elements(file) ne 1 then return ;-- at least one file name entered
 check=findfile(file,count=count)     ;-- check if file exists
```

```
 if count ne 1 then return           ;-- bail if not there
 image=fltarr(512,512)               ;-- assume a 512x512 image file
 openr,lun,file,/get_lun             ;-- open file
 readf,lun,image                     ;-- read image
 free_lun,lun                        ;-- close file
 self->set,image                     ;-- insert image into object
 return

end

;---------------------------------------------------------------------

pro data::plot

 value=self->get()                                    ;-- extract data value from object
 dsize=size(value)                                    ;-- determine data dimensions
 if dsize[0] eq 2 then tvscl,congrid(value,512,512)   ;-- if 2-dimensional, CONGRID and TVSCL it
 return

end
```

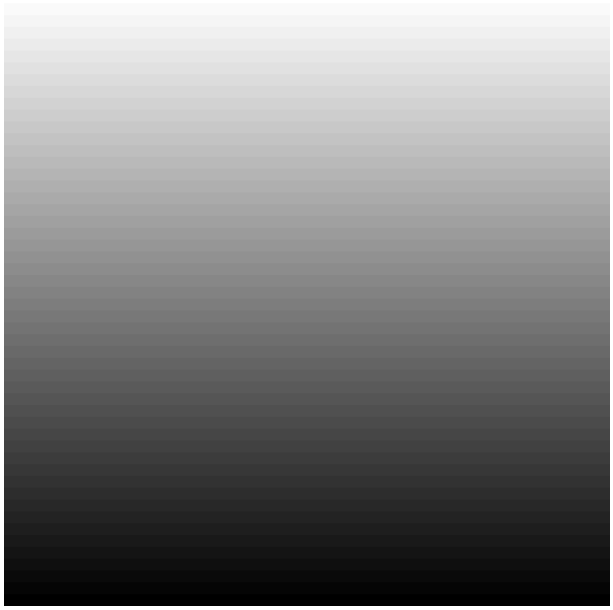
The *read* method accepts a file name as its argument. As is good practice, we check if the filename argument is entered and use IDL's *findfile* to test if the file actually exists. We subsequently open the file, read the data, and insert into the object using the *set* method call: *self->set,value*. Note that because we are referencing the object itself, we use the *self* variable name for the object.

The *plot* method extracts the data value from itself via the *get* method call: *self->get()*. It checks that the data is a 2-dimensional image using the IDL *size* function, and plots it with a call to IDL's *tvscl* command and *congrid*, which expands the image to a 512x512 array size. To demonstrate this sequence of steps, let's create an image file *image.dat* in the current directory and deploy the *data* object as follows:

```
IDL> image=findgen(512,512)        ;-- create 512x512 image array
IDL> file='image.dat'
IDL> openw,lun,file,/get_lun       ;-- write image to file
IDL> printf,lun,image
IDL> free_lun,lun


IDL> .run data__define             ;-- recompile class definition file
IDL> a->read,file                  ;-- read image and display it
IDL> a->plot
```

The call to the *plot* method produces the following simple image:



Sample data plot

Note that it is not necessary to reinitialize the *data* object since we are using the same object variable to store the image data. However, it is necessary to recompile the class definition file since we have added new methods to it.

2.3. How does inheritance work?

Looking at the last two lines in the previous example, it doesn't appear from our *data* object example that we have advanced very far using OOP techniques. In particular, compared to the opening example of reading and plotting data using non-OOP procedures, it seems that a lot of effort was invested in writing object methods that essentially reproduce the same functionality as conventional procedure calls.

The real power of OOP techniques comes not in what we have just done, but in what we are about to do. The *data* object that we have created is a building block that can be used to develop objects that perform more complicated functions. Consider the following problem:

"I have an image in a FITS file that I would like to read and display. I would also like to remember the name of the FITS file so that I can track it."

The *data* object that we have created provides a convenient storage facility for FITS image data, but it lacks the required FITS file reader and it doesn't have a way of remembering the file name. We could of course re-edit the *data* class definition file to add this functionality, but that would involve much more work. The OOP solution to the problem is to define a new class that somehow inherits the functionality of the *data* class. The following is how to do it.

We define a new class called *fits* by creating a file named *fits__define.pro*, which contains the lines:

```
pro fits::read,file

 if n_elements(file) ne 1 then return   ;-- at least one file name entered
 check=findfile(file,count=count)       ;-- check that file exists
 if count eq 0 then return
 image=mrdfits(file)                    ;-- call Astronomy library FITS reader
 self->set,image                        ;-- insert image data into property
 self.filename=file                     ;-- save filename in property
 return

end

;---------------------------------------------------------------------------

pro fits__define

 void={fits,filename:'', inherits data} ;-- inherit from data class
 return

end
```

Several new concepts are happening here. Let's start from the bottom up.

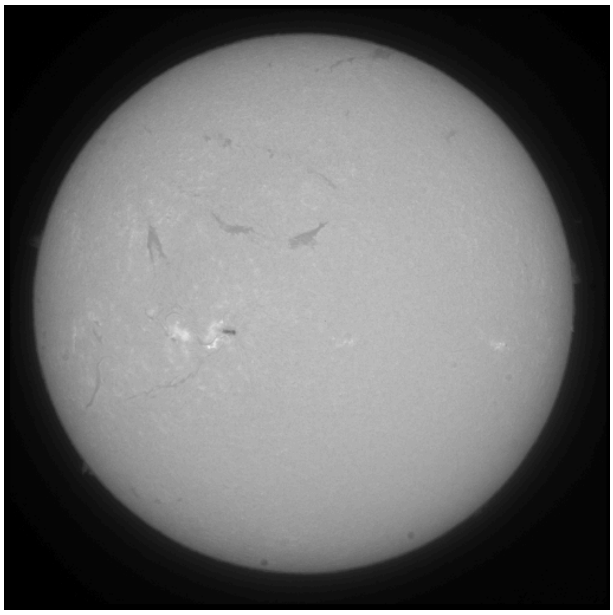
1. **FITS__DEFINE**: the procedure that defines our fits data structure looks very different from the original data structure in *data__define.pro*. It appears to be missing the data pointer property. Actually, the latter is is not missing. In OOP language, it is **inherited** from the *data* class via the statement *inherits data*. In addition to inheriting data's property, the *fits* class inherits all of its methods: *init, cleanup, set, get, read,* and *plot.* Hence, inheritance has saved us from writing a lot of extra code. Note that the fits data structure contains a new property called *filename*, which is a string data type. This property will be used to store the FITS file name.

2. **FITS::READ**: object inheritance imports data's simple *read* method *data::read*, which clearly will not work on a FITS file. To overcome this problem, we write a new method that uses the Astronomy library routine ***mrdfits.pro*** to read the FITS file and insert the resulting image data into the object pointer by using the inherited *set* method. The fits *read* method thus **overrides** data's *read* method. This new method also saves the FITS filename as a property of the *fits* object once the data is saved.

In OOP language, the *fits* class is a **derived** class of *data*, and a *fits* object is referred to as **child** of the **parent** *data* object. The following example demonstrates how to use the *fits* class to create an object to read a Big Bear H-alpha image contained in the file **bbso_halph_fl_20040310_173531.fts**:

```
IDL> file='bbso_halph_fl_20040310_173531.fts'
IDL> f=obj_new('fits')                 ;-- create object
IDL> help,f
F             OBJREF    = ObjHeapVar37(FITS)
```

```
IDL> f->read,file                    ;-- read file
IDL> f->plot                         ;-- plot data
IDL> image=f->get()                  ;-- extract image
IDL> help,image
IMAGE           INT       = Array[512, 512]
```



BBSO H-alpha

We create a *fits* object using the *obj_new* function, and feed it the FITS file name. After reading, we plot the image, and extract the image data using the *get* method. As defined, the *get* method inherited from the *data* class only allows us to extract the data value from the property pointer. However, we would also like to extract the filename that is associated with the data, which is also a property. To include this functionality, we override the *get* method in *data__define.pro* with a new *get* method in *fits__define.pro* as follows:

```
function fits::get,filename

 filename=self.filename              ;-- copy filename in variable
 image=self->data::get()             ;-- call DATA's GET method to return data
 return,image

end
```

The above lines illustrate the simplicity and elegance of inheritance. We have added a new output argument *filename* in which we return the string value of the filename, which is saved in the property *self.filename*. Since we also wish to return the data value, we include a call to the *get* method that we have already defined for the *data* class. There is no need to rewrite the latter. Hence, in the last example, we can execute the following:

```
IDL> data=f->get(filename)
IDL> help,data,filename
DATA            INT       = Array[512, 512]
FILENAME        STRING    = 'bbso_halph_fl_20040310_173531.fts'
```

Note also that the *fits* object methods (*read* and *get*) retain the same names as their parent *data* method names. The difference is in their behaviors, which depends upon which data type is being operated on. This ability to behave differently depending upon class (or data type) is called **polymorphism**.

3. APPLYING OBJECTS TO SOLAR IMAGES

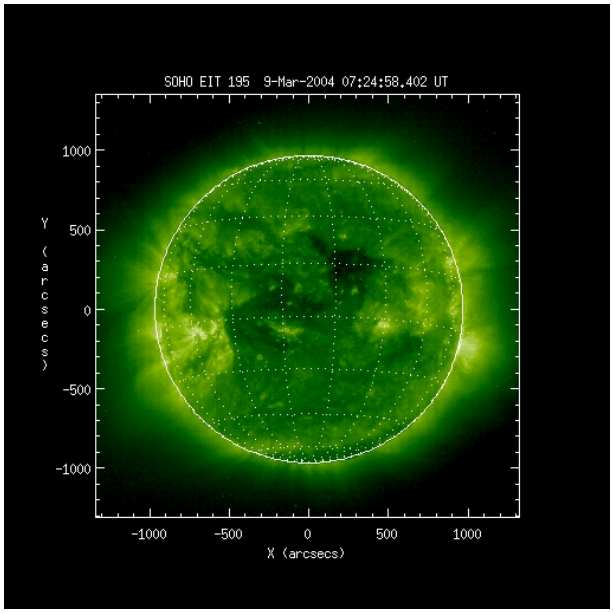
The example classes in this tutorial demonstrate how objects can be designed, created, and used. If interested in experimenting with these classes, you can download complete definition files via the following links: **data__define.pro** and **fits__define.pro**. Although useful for illustrative purposes, these classes are too simplistic for handling more complicated solar datasets. For example, not all solar datasets conform to the FITS format standard. Variations in the use of header keyword names and values often require the use of special readers. Moreover, different datasets usually require the application of instrument-specific

processing algorithms inorder to be useful. This dependence of operations upon the properties of the dataset naturally lends itself to the use of objects as a tool for analyzing solar data.

3.1 Using *MAP* objects

IDL *maps* are described in **maps.html**. An IDL *map* is a structure data type that stores data and associated identifying information. A *map* structure is not a true object since it does not contain methods. Nevertheless, it is useful for displaying and, in particular, overlaying solar images from different instruments. The following example illustrates how to to read, process, and display the quicklook *SOHO/EIT* file **efr20040309.072550** using a map and procedures available in the IDL ***SolarSoft*** library:

```
IDL> files='efr20040309.072550'
IDL> read_eit,files,index,data                  ;-- read EIT QL file
IDL> eit_prep,index,data=data,outindex,outdata  ;-- prep data
IDL> index2map,outindex,outdata,emap            ;-- create map
IDL> eit_colors,195                             ;-- load 195 color table
IDL> plot_map,emap,/log,grid=20,/limb           ;-- plot map
```



SOHO/EIT 195 A

This example highlights several points. Even though we are primarily interested in reading and plotting an *EIT* image, we have to know and perform several steps:

1. read the *EIT* FITS file using the special reader *read_eit*.
2. process the *EIT* image using the procedure *eit_prep*, which performs dark current subtraction and flatfielding.
3. convert the processed image to a map structure *emap* via the procedure *index2map*.
4. plot the map structure *emap* using the procedure *plot_map*, which is called with the keywords */limb, /log*, and *grid=20* inorder to display a log-scaled image with a heliographic grid and limb.

In addition, we have to remember to pass the variables: *index, data* as arguments to these procedures, and know that our *EIT* dataset is a 195 A image that has a custom color table (which we load with the procedure *eit_colors*). That's a lot to remember each time. It would be nice if we could somehow **encapsulate** the above procedures and data into an object and only have to remember as little as necessary to get the job done.

What we need is a *map* class that will allow us to define *map* objects to store data such as *EIT* images and provide methods to manipulate them. Such a class already exists. It is defined in the file ***map__define.pro***. The *map* class is analogous to our example *data* class except that it uses a *map* structure to store data and its corresponding properties (such as pointing).

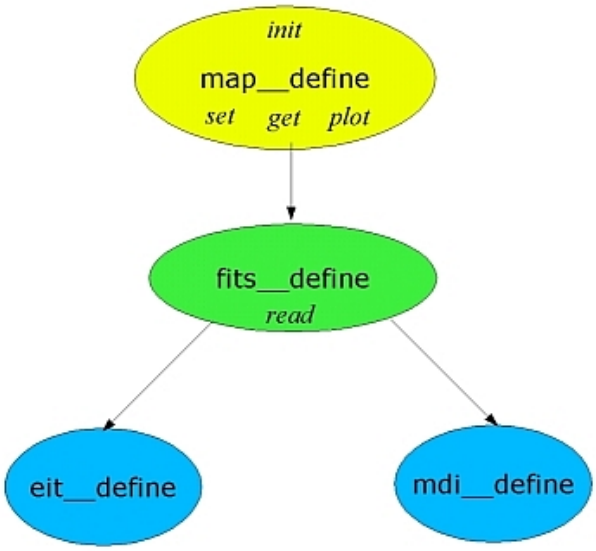
Now consider the following OOP approach to the same example:

```
IDL> file='efr20040309.072550'
```

```
IDL> eobj=obj_new('eit')                    ;-- create an EIT object
IDL> eobj->read,file                        ;-- read EIT file
IDL> eobj->plot                             ;-- plot EIT image
```

The above example produces exactly the same plot output as the conventional example, but what is different? Let's take it in steps:

- We have created an object variable *eobj* by calling the object creation function *obj_new('eit')* with the class name *eit*. But where did this class come from? The answer is that we have already created a class definition file ***eit__define.pro*** in which we have incorporated the *EIT* data as a property. The *eit* class definition file is slightly more complicated than the *fits* and *data* classes defined earlier. The *eit* class inherits from a slightly more complicated *fits* class that is defined in ***fits__define.pro***, which in turn inherits from the *map* class. The *map* class works as a storage device in much the same way as the simple *data* class that we defined in our earlier example. This inheritance relationship is illustrated in the diagram below.



MAP->FITS->EIT Class Inheritance

- After creating the *eit* object, we read the *EIT* file using the *read* method. This method is nothing other than the *read_eit* procedure, which we have incorporated into *eit__define.pro*. This is **polymorphism** at work. Because our object is *aware* that it belongs to the *eit* class, its *read* method correctly calls the corresponding *eit* reader, followed by *eit_prep* to process the image. Hence, the *EIT* image is automatically read, processed, and saved into the *eit* object.

- With the *EIT* image in memory, we display it using the *plot* method. This method is the *plot_map* procedure that we have already incorporated into *map__define.pro* and, hence, is inherited by the *eit* object. We have also modified the *plot* method slightly to include a call to *eit_colors* in order to load the corresponding image color table.

In summary, the object variable *eobj* object is a *map* object since the *eit* class is derived from a *map* class. The details of the relationship between *eit* and *map* classes are not overly important to the average user who is interested in performing basic data analysis. The main point is that, by using an *eit* map object, the overhead of remembering several instrument-specific procedure names is significantly reduced.

3.2 Image analysis with objects

The *eit* class has a *get* method that provides access to the *EIT* image data and map structure as follows:

```
IDL> edata=eobj->get(/data)
IDL> help,edata
EDATA           INT       = Array[1024, 1024]

IDL> emap=eobj->get(/map)
IDL> help,/st,emap
** Structure <40e0fb08>, 13 tags, length=2097256, refs=2:
   DATA            INT       Array[1024, 1024]
   XC              FLOAT          -8.15294
   YC              FLOAT           21.0663
   DX              FLOAT           2.63500
```

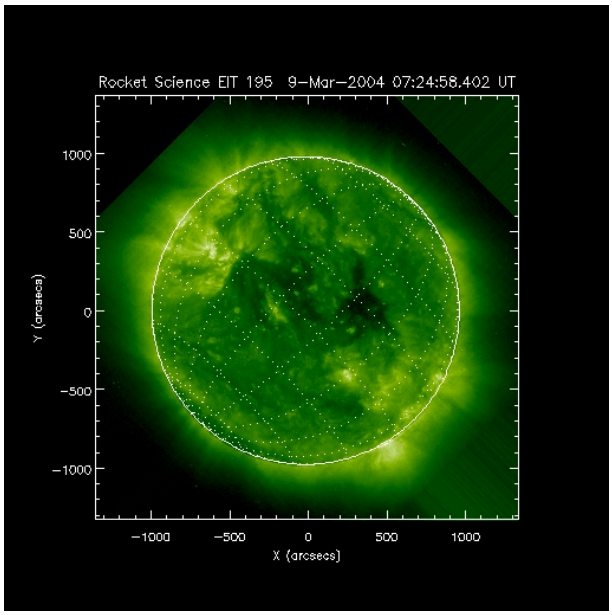
```
DY              FLOAT           2.63500
TIME            STRING    ' 9-Mar-2004 07:24:58.402'
ID              STRING    ' Rocket Science EIT 195'
ROLL_ANGLE      FLOAT           0.00000
ROLL_CENTER     FLOAT     Array[2]
DUR             FLOAT           12.5970
XUNITS          STRING    'arcsecs'
YUNITS          STRING    'arcsecs'
SOHO            BYTE         1
```

The pointing information contained within a map structure allows us to analyze different images regardless of the image source. By making a *map* structure a property of a *map* object, we simplify the steps involved in performing typical image processing operations. We will conclude this tutorial by demonstrating three such operations: rotating an image; correcting for differential solar rotation; and overlaying images.

- **Rotation:** to rotate (or roll) an image clockwise about its center, we pass the angle of rotation as an argument to the method *rotate*.

```
IDL> robj=eobj->rotate(45)
IDL> robj->plot
```

In this example, we rotate the *EIT* image contained in the *eit* map object *eobj* by 45 degrees to create a new map object *robj* which produces the image:

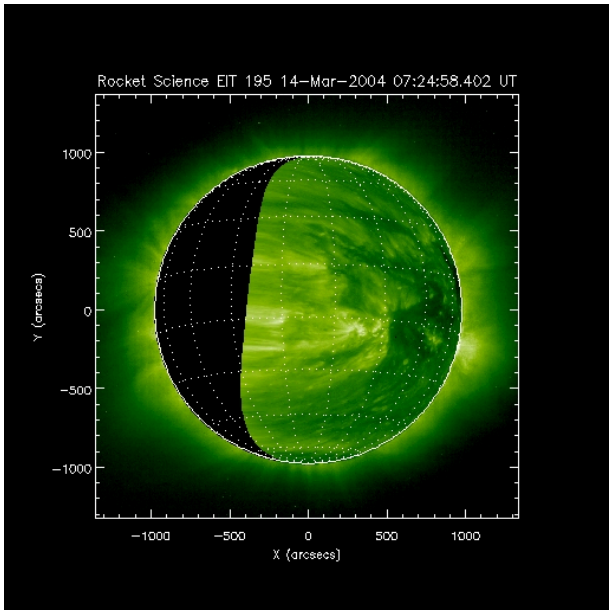


Rolled *EIT 195 A*

- **Solar differential rotation:** to differentially rotate an image we pass the time interval over which to rotate as an argument to the method *drotate*.

```
IDL> dobj=eobj->drotate(5,/days)
IDL> dobj->plot
```

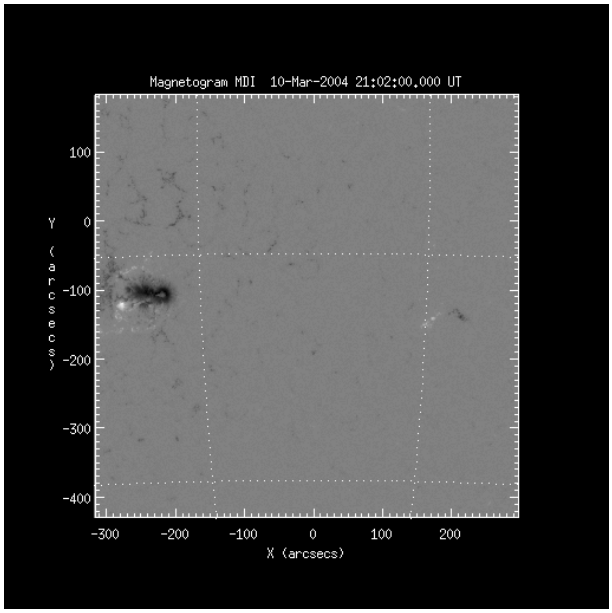
In this example, we solar rotate the *EIT* image forward in time by 5 days to produce the image:



Differentially rotated *EIT 195 A*

- **Overlaying images:** to overlay two images, we first plot a base image using the *plot* method and overplot a second image as a contour using *plot* with the keyword */over*. We demonstrate this operation by overlaying a *SOHO/MDI* image on an *EIT* image. But first we need to create an *mdi* map object. We have already defined an *mdi* class by writing a definition file ***mdi__define.pro***. As with the *eit* class, the *mdi* class inherits from a *fits* class (which inherits from a *map* class). Because of this common inheritance, an *mdi* object has the same *plot* method as an *eit* object. This class allows us to create an *mdi* object to read and display a high-resolution *MDI* image in the file **mdi_maglc_re_20040310_2102.fts**:

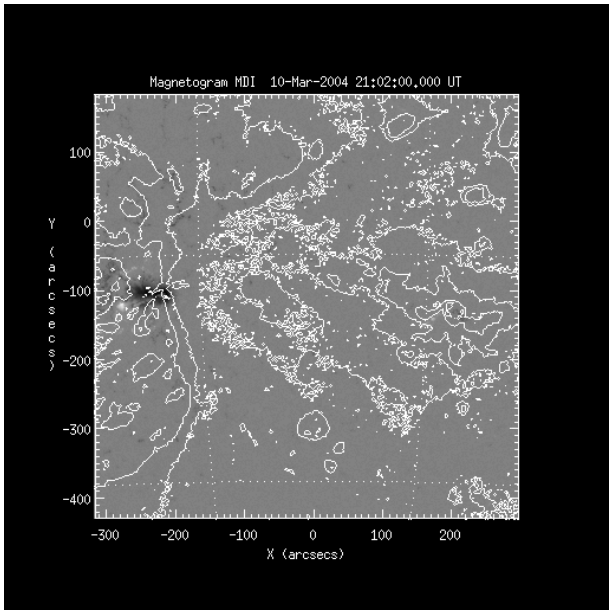
```
IDL> file='mdi_maglc_re_20040310_2102.fts'
IDL> mobj=obj_new('mdi')                        ;-- create an MDI object
IDL> mobj->read,file                            ;-- read MDI file
IDL> mobj->plot                                 ;-- plot MDI image
```



SOHO/MDI

Note how the steps for reading and plotting the *MDI* image are identical to those for *EIT*. The internal details of how these steps are performed are handled by the corresponding methods. Having created *eit* and *mdi* objects, the final step of overlaying their corresponding images is reduced to the following single line:

```
IDL> eobj->plot,/over,/drotate    ;-- overlay EIT image on previous MDI image
                                  ;   (correcting for solar differential rotation)
```



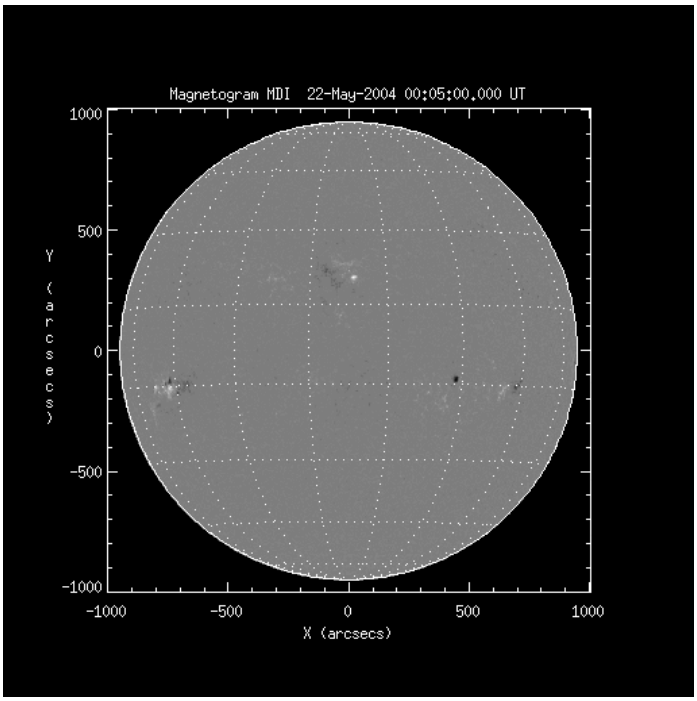
***EIT/MDI* overlay**

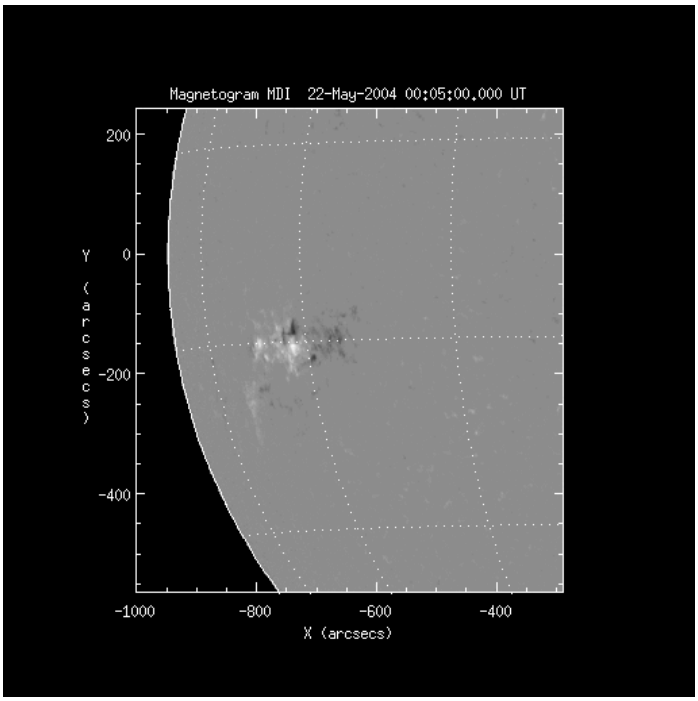
In this example, we used the */drotate* keyword to differentially rotate the *EIT* image contours to the same time as the base *MDI* image.

- **Extracting a sub-field:** the *extract* method extracts a sub-field graphically or via keywords.

```
IDL> mobj=obj_new('mdi')
IDL> mobj->read,'mdi_maglc_fd_20040522_0005.fts.gz'
IDL> cobj=mobj->extract()
IDL> cobj=mobj->extract(xrange=[-500,500],yrange=[-500,500])
```

In this example, we read an *MDI* full-disk image in the file **mdi_maglc_fd_20040522_0005.fts.gz**, and call *extract* without any keywords. A box-cursor is used to select a sub-field. Alternatively, the keywords: *xrange* and *yrange* can be used to specify the sub-field.





***MDI* Full Disk** ***MDI* Partial Disk**

Last Revised: *04/20/2025 20:11:23 by* ***Dominic Zarro***