A word about it...

Image Reduction and Analysis Facility, or on short **IRAF**, is a collection of software developed by NOAO geared towards the reduction of astronomical images in pixel array form.

It is the primary tool for Astronomy students who need to reduce their first sets of data and understand how to work with them. If only it would not have two defects:

- it is **unintuitive** to work with;
- despite its complexity, not much is left for us, the students, to grasp from it and make it **our own**.

Why bother?

What you see above, is the interface of a **Jupyter Notebook**. It is so much cleaner compared to the old fashioned one offered by IRAF, giving us the opportunity to work faster with our programs and most importantly, to understand why and how we use them.

While we can all agree that time is a valuable resource and not every tool at our disposal should be built from the ground up every time, for those of us who find a couple more minutes while doing a project, reading a few comments through the code lines unveils so much more of what we are constructing.

Seeing how our data is reduced provides us with a much better understanding of what we are dealing with and what we are really looking for into them.

Also, perhaps a more official reason why one should bother with finding alternatives to IRAF comes right from NOAO's website, where an announcement was made:

IRAF - Image Reduction and Analysis Facility

NOAO is transitioning IRAF to an end-of-support state, and has taken NOAO's IRAF distribution offline pending a final copyright and licensing review of the source code. Users interested in new IRAF installations during this review period may wish to consider the IRAF Community Distribution.

A small tip: don't bother trying the download links listed below it... they don't work anymore.

```
# Because we want to make this code work for any given number of filters, we need to make the program itself be able to
# create lists (or later on titles) where to store the data from each file in each filter, so we will use a small tirck: exec().
lnf = len(filters)
nn = ['new', 'norm']
fl_li = ['flat', 'light']
dntgtdh = ['data', 'name', 'time', 'gain', 'temp', 'dateobs', 'hdr']
for i in range(2): #fl_li
    mycode_0 = str('flat_data_' + nn[i] + '_q = []')
    exec(mycode_0)
    for j in range(lnf):
        mycode_1 = str(fl_li[i] + '_' + filters[j] + '_q = []')
        exec(mycode_1)
        for k in range(len(dntgtdh)):
            mycode_2 = str(fl_li[i] + '_' + filters[j] +'_' + dntgtdh[k] + '= []; ' + fl_li[i] + '_' + filters[j] + '_q.append(' + fl_li[i] + '_' + filters[j] + dntgtdh[k] + ')')
            exec(mycode_2)
        mycode_3 = str('all_q.append(' + fl_li[i] + '_' + filters[j] + '_q)')
        exec(mycode_3)
        mycode_4 = str('flat_' + filters[j] + '_data_' + nn[i] + ' = []; flat_data_' + nn[i] + '_q.append(flat_' + filters[j] + '_data_' + nn[i] + ')')
        exec(mycode_4)


# END RESULT: all_q = [dark_q, bias_q, flat_B_q, flat_V_q, light_B_q, light_V_q]
# Note that this is just an example, where two filters were used: B & V.
```

(Fig.1)

Thanos

Thanos is a program developed as an alternative to IRAF for a university project. Now, thanks to its **simplicity** regarding the user input and its **complexity** in terms of options and features, it stands as a compelling tool for other students too.

In the order one would encounter them throughout its usage, the main features of this mad titan are:

- the ability to use files from different folders and auto-select those that are viable;
- the ability to work with any filters, regardless of their type or number, **by creating individual, specially named lists** containing the respective data sets, thus making easy for the user to search through them *(Fig.1)*;
- **performing data reduction**;
- **sorting them** based on the their type (bias frames, dark frames, flat fields or light frames) and (for the last three ones) also on the filters used when capturing them;
- viewing the obtained frames with the option to cut a percentage of the top/lower values in each, thus making the analyzing process easier by not occupying a range of the colors that can be displayed with unimportant values (counts) *(Fig.2)*;
- **shifting the obtained frames and (after letting the user chose one for reference) compiling them into a single one** *(Fig.3)*;
- repeating the above procedure for the resulted shifted frames and obtaining **a single, final, frame** *(Fig.4)*;
- saving by choice any of the obtained frame with a conclusive name.

Thanos - still... why bother?

As it turns out, making this program was not something to be done over the weekend. There have been numerous times when, despite knowing how the theory works (in terms of data reduction), applying it came up to be a challenge. This, however, proved to a very useful and memorable experience from which there is a lot to be learned.

1. Understanding the calculations**

- A level of **rigorousness** and **understanding** of the undergoing processes must be achieved in order to make sure they perform accordingly;
- Improvements have to be constantly brought to the code in order to make it run faster, thus allowing for grater sets of data to be compiled in a respectable time interval.

2. Working with headers

- Painful and time consuming tasks like categorizing all the files must be automated and the only way to do it is through the information present in the headers.
- Their importance has to be recognized also after the work is done, because they might still have to be used in places where just the title will not be enough to clarify what they really consist of.

3. Learning from mistakes

- using numeral values instead of matrices (and figuring when you should or shouldn't);
- allowing the user to chose when they want to stop and not drag them through stages they don't care about;
- at every stage, placing the saving procedure of the frames immediately after they were produced (in case of the program crashing);
- giving the user the option to save (or not) the frames from any stage of the process;
- etc.

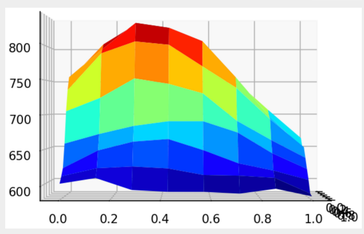
4. Know what you're looking for

- Remember how Gordon always spots the microwaved food? Well, we have to spot the cosmic rays in our data. This job takes a bit of practice and experience, but visualizing the profile of a star *(Fig.5)* and then of the whole frame *(Fig.6)* can be a game changer;
- The same applies to choosing a reference frame for the shifts, especially in the case of compiling all the final frames from all the filters.

```
centoventi = input('(If you will opt to visualize any images later on,) would you want to see them unaltered or with'
                   'a specific percentage of their upper and lower values being cut (this makes the dimmer objects in'
                   'the frame much brighter, but will take a few extra minutes) (u/a)?')
if centoventi == 'a':
    up = int(input('Then by how much do you want to reduce the upper values as a percentage (just the number)?'))
    down = int(input('And by how much do you want to reduce the lower values as a percentage (just the number)?'))
    def cento(x):
        shape = x.shape
        data_5cent = x.flatten()
        data_50cent = data_5cent
        data_50cent = list(set(data_50cent))
        data_50cent.sort()
        no_up = int((up/100)*len(data_50cent))
        no_down = int((down/100)*len(data_50cent))
        if no_up == 0:
            no_up = 1
        if no_down ==0:
            no_down = 1
        data_50cent_min = data_50cent[:no]
        data_50cent_max = data_50cent[-no:]
        data_50cent_min_max = np.max(data_50cent_min)
        data_50cent_max_min = np.min(data_50cent_max)
        for k in range(len(data_5cent)):
            if data_5cent[k] <= data_50cent_min_max:
                data_5cent[k] = (data_50cent_min_max + 1)
            elif data_5cent[k] >= data_50cent_max_min:
                data_5cent[k] = (data_50cent_max_min - 1)
        data_mega_fab = data_5cent.reshape(shape)
        return data_mega_fab
if centoventi == 'u':
    def cento(x):
        data_mega_fab = x
        return data_mega_fab
```

(Fig.2)

(Fig.5)




```
# There is one problem: as time passes, the telescope may
# or may not move perfectly with respect to the patch of sky we were observing, so the images might be ever so slightly shifted.
# Here we will fix this problem by chosing one of them from each filter as the 'zero-shift' one.
for i in range(lnf):
    print('From 1 to', len(data_ggwp[i]), 'what image in the ', filters[i],' filter do you think is the best one as a zero-shift one... or else type 1.')
    the_chosen_one[i] = int(input()) - 1
    zero_shift_image[i] = data_ggwp[i][the_chosen_one[i]]


# We now make lists where we will keep the shifts that we will find shortly.
imshifts_q = []
shifted_imggwp_q = []
for i in range(lnf):
    mycode_9 = str('imshifts_' + filters[i] + ' = []; imshifts_q.append(imshifts_' + filters[i] + ')')
    exec(mycode_9) # dictionary to hold the x and y shift pairs for each image
    mycode_10 = str('shifted_imggwp_' + filters[i] + ' = []; shifted_imggwp_q.append(shifted_imggwp_' + filters[i] + ')')
    exec(mycode_10)


# Now we will find all the shifts for the other images.
for i in range(lnf):
    for j in range(len(data_ggwp[i])):
        # Note: register_translation() is a function that calculates shifts by comparing 2-D arrays
        result, error, diffphase = register_translation(zero_shift_image[i], data_ggwp[i][j], 1000)
        imshifts_q[i].append(result)
    for j in range(len(data_ggwp[i])):
        shifted_imggwp_q[i].append(interp.shift(data_ggwp[i][j],imshifts_q[i][j]))
```

(Fig.3)

```
# Now we stack the images one filter at a time.
stacked = []
for i in range(lnf):
    stacked.append(np.average(shifted_imggwp_q[i], axis=0))
lns = len(stacked)
```

Photutils

After the first frames were obtained using Thanos, the next step in the original project was to perform photometry on them. Here, we observed a recurring problem: despite the theory being pretty straight forward, going through all the steps one by one proved to be more challenging.

There is a level of rigorousness that can only be achieved while applying the theory and once again, coding it is the perfect example.

This time around though, a very useful piece of code was used - **Photutils**. It is a package of Astropy and it contains all that is needed for this process. One of its main additions was **DOAStar Finder**. It is an algorithm that searches in an image for points that, due to their profile, resemble stars *(Fig.5)* and creates a list with their ID's and their X&Y positions on the "grid" *(Fig.7)*. This was our first encounter with **the problem of differentiating stars from other astronomical objects**, which we had to fix in our code.

Thanos - the show must go on

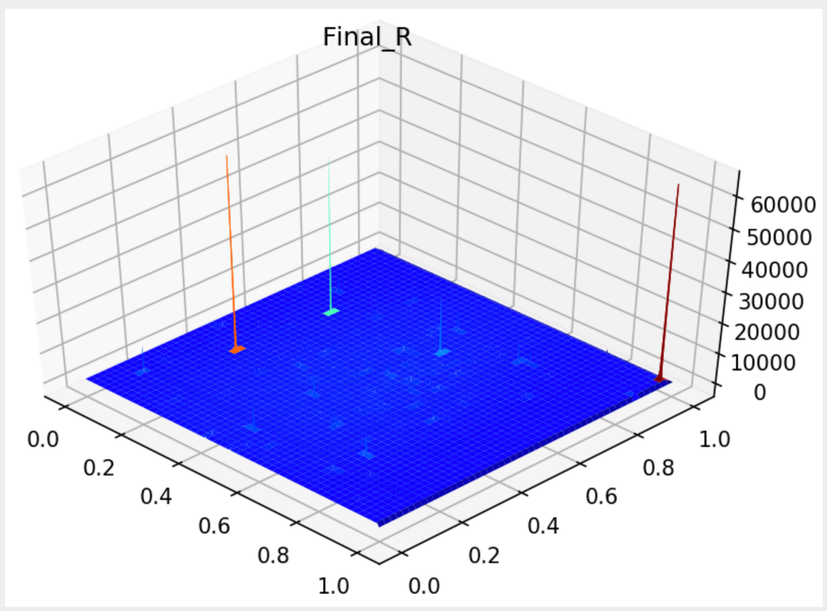
As time goes on, this piece of software will be improved. At the moment, the key features to be implemented are:

- a more complete description of the individual files in their headers;
- the ability to perform photometry;
- the addition of other correcting techniques.

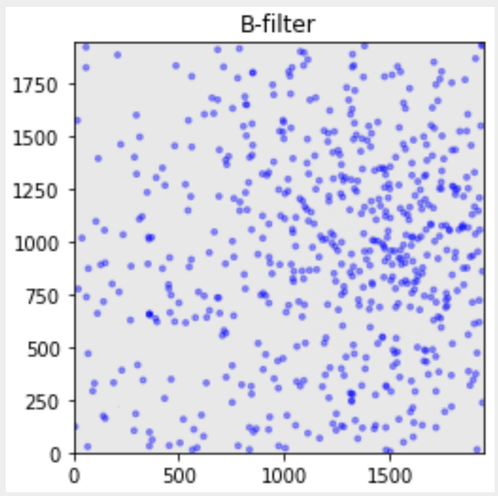
But our biggest hope for Thanos is to make it a tool for **others** too. Through the code lines, descriptions have been added that serve not only as short reminders for future developing, but as actual **indications of what processes are undergoing and how they can be modified in order to achieve different results**.

```
In [1]: # Thanos stands as an alternative to IRAF in terms of data reduction, but with the small distinction that...
        # it also has descriptions. ^^
```

(Fig.6)



(Fig.7)



(Fig.4)