Differential photometry -1

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)

Differential photometry -1

A complete example - 1

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)

Differential photometry -1

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:

Differential photometry -1

A complete example - 1

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:
  - Calibration (for *dark* and *flat field*)

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:
  - Calibration (for *dark* and *flat field*)
  - Target selection

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of ”Osservatorio Polifunzionale del Chianti”[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:
  - Calibration (for *dark* and *flat field*)
  - Target selection
  - Image centering

Differential photometry -1

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:
  - Calibration (for *dark* and *flat field*)
  - Target selection
  - Image centering
  - Flux evaluation for the selected targets

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:
  - Calibration (for *dark* and *flat field*)
  - Target selection
  - Image centering
  - Flux evaluation for the selected targets
  - Photometric correction

Differential photometry -1

A complete example - 1

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:
  - Calibration (for *dark* and *flat field*)
  - Target selection
  - Image centering
  - Flux evaluation for the selected targets
  - Photometric correction

[1] Image data kindly provided by L. Naponiello.

Differential photometry -1

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:
  - Calibration (for *dark* and *flat field*)
  - Target selection
  - Image centering
  - Flux evaluation for the selected targets
  - Photometric correction

[1] Image data kindly provided by L. Naponiello.

→

Differential photometry -1

A complete example - 1

The problem:

We have to analyze several CCD images from a star field to perform photometry on an exoplanet transit over its star.
(Observation of HD189733/b on July, 18th, 2017 at the 80 cm telescope of "Osservatorio Polifunzionale del Chianti"[1])

- Observation data:
  - 773 CCD shots (3.1 MB FITS files)
  - 1 *dark* frame (CCD camera offset)
  - 1 *flat field* frame (CCD gain calibration)
- How to proceed:
  - Calibration (for *dark* and *flat field*)
  - Target selection
  - Image centering
  - Flux evaluation for the selected targets
  - Photometric correction

[1] Image data kindly provided by L. Naponiello.

→

Photometric exoplanet detection

- Exoplanet detection is based on a decrease of measured flux of the star due to the occultation of the planet passing in front of the star.

Photometric exoplanet detection

- Exoplanet detection is based on a decrease of measured flux of the star due to the occultation of the planet passing in front of the star.
- Good photometric measures are needed to detect small variations of flux

Photometric exoplanet detection

- Exoplanet detection is based on a decrease of measured flux of the star due to the occultation of the planet passing in front of the star.
- Good photometric measures are needed to detect small variations of flux
- We do not need absolute (calibrated) flux values

Photometric exoplanet detection

- Exoplanet detection is based on a decrease of measured flux of the star due to the occultation of the planet passing in front of the star.
- Good photometric measures are needed to detect small variations of flux
- We do not need absolute (calibrated) flux values
- In order to remove the effects of flux variations due to other causes (e.g.: variation of air transparency) we compute the flux of the target star relative to the flux of other stars in the same image.

Photometric exoplanet detection

- Exoplanet detection is based on a decrease of measured flux of the star due to the occultation of the planet passing in front of the star.
- Good photometric measures are needed to detect small variations of flux
- We do not need absolute (calibrated) flux values
- In order to remove the effects of flux variations due to other causes (e.g.: variation of air transparency) we compute the flux of the target star relative to the flux of other stars in the same image.
- In other terms, given the flux of the target star F_T and the fluxes of some other stars from the same image, F_1, F_2, ..., F_N, the relative flux is computed as:

$$F_R = \frac{F_T}{\sum_{i=1}^{N} F_i}$$

Photometric exoplanet detection

- Exoplanet detection is based on a decrease of measured flux of the star due to the occultation of the planet passing in front of the star.
- Good photometric measures are needed to detect small variations of flux
- We do not need absolute (calibrated) flux values
- In order to remove the effects of flux variations due to other causes (e.g.: variation of air transparency) we compute the flux of the target star relative to the flux of other stars in the same image.
- In other terms, given the flux of the target star F_T and the fluxes of some other stars from the same image, F_1, F_2, ..., F_N, the relative flux is computed as:

$$F_R = \frac{F_T}{\sum_{i=1}^{N} F_i}$$

→

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.filenames[:number]
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must implement _getdata()!")

    def run(self):
        for step, filename in enumerate(self.filenames):
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.filenames[:number]
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must implement _getdata()!")

    def run(self):
        for step, filename in enumerate(self.filenames):
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.filenames[:number]
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must implement _getdata()!")

    def run(self):
        for step, filename in enumerate(self.filenames):
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

get_args(): see below

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.filenames[:number]
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must implement _getdata()!")

    def run(self):
        for step, filename in enumerate(self.filenames):
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

get_args(): see below

See module: **glob**

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.filenames[:number]
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must implement _getdata()!")

    def run(self):
        for step, filename in enumerate(self.filenames):
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

get_args(): see below

See module: **glob**

Manage added arguments

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.filenames[:number]
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must implement _getdata()!")

    def run(self):
        for step, filename in enumerate(self.filenames):
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

get_args(): see below

See module: **glob**

Manage added arguments

Manage options **-f** (*first*) e **-n** (*number*)

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.fi
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must implement _getdata()!")

    def run(self):
        for step, filename in enumerate(self.filenames):
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

get_args(): see below

See module: **glob**

Manage added arguments

Manage options **-f** (*first*) e **-n** (*number*)

Function to apply to each data file

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.fi
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must impl

    def run(self):
        for step, filename in enumerate(self.filenames):
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

get_args(): see below

See module: **glob**

Manage added arguments

Manage options **-f** (*first*) e **-n** (*number*)

Function to apply to each

Method `_getdata` gives an error if not reimplemented

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.fi
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must impl

    def run(self):
        for step, filename in enumerate(sel
            img, args = self._getdata(filename)
            res = self.operate(img, step, args, self.common_args)
            if self.step:
                input("*** Press enter to go on")
```

get_args(): see below

See module: **glob**

Manage added arguments

Manage options **-f** (*first*) e **-n** (*number*)

Function to apply to each

Method `_getdata` gives an error if not reimplemented

Loop on the data file sequence

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.fi
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must impl

    def run(self):
        for step, filename in enumerate(sel
            img, args = self._getdata(filename)
            res = self.operate(img, step,
            if self.step:
                input("*** Press enter to
```

get_args(): see below

See module: **glob**

Manage added arguments

Manage options **-f** (*first*) e **-n** (*number*)

Function to apply to each

Method `_getdata` gives an error if not reimplemented

Loop on the data file sequence

The function is called here

class Pipeline

All the procedures will require to operate on the full sequence of images.

The class **Pipeline** has the purpose to apply a specified function to a selection of (possibly all) the data files in a given directory.

file: pipeline.py

```
import os, json, glob, sys, getopt
from astropy.io import fits
import numpy as np

class Pipeline:
    def __init__(self, operate, common_args=None):
        ddir, glbn, first, number, step = get_args()
        self.filenames = glob.glob(os.path.join(ddir, glbn))
        self.filenames.sort()
        self.step = step
        if common_args is None:
            self.common_args = {}
        else:
            self.common_args = common_args
        if first:
            self.filenames = self.filenames[first:]
        if number:
            self.filenames = self.fi
        self.operate = operate

    def _getdata(self, filename):
        raise Exception("You must impl

    def run(self):
        for step, filename in enumerate(sel
            img, args = self._getdata(filename)
            res = self.operate(img, step,
            if self.step:
                input("*** Press enter to
```

get_args(): see below

See module: **glob**

Manage added arguments

Manage options **-f** (*first*) e **-n** (*number*)

Function to apply to each

Method `_getdata` gives an error if not reimplemented

Loop on the data file sequence

The function is called here

Manage option **-s** (*step*)

function `get_args()`

Has the purpose to manage command line arguments for programs using module `pipeline`.

file: pipeline.py

```
def get_args():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'd:g:f:n:sh')
    except getopt.GetoptError:
        print("Argument error (-h: help)")
        sys.exit()

    ddir = "."
    glbn = "*.*"
    first = None
    number = None
    step = False
    for (o, v) in opts:
        if o == '-h':
            print(__doc__)
            sys.exit()
        elif o == '-d':
            ddir = v
        elif o == "-g":
            glbn = v
        elif o == "-f":
            first = int(v)
        elif o == "-n":
            number = int(v)
        elif o == "-s":
            step = True
    return (ddir, glbn, first, number, step)
```

function `get_args()`

Has the purpose to manage command line arguments for programs using module `pipeline`.

file: pipeline.py

```
def get_args():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'd:g:f:n:sh')
    except getopt.GetoptError:
        print("Argument error (-h: help)")
        sys.exit()

    ddir = "."
    glbn = "*.*"
    first = None
    number = None
    step = False
    for (o, v) in opts:
        if o == '-h':
            print(__doc__)
            sys.exit()
        elif o == '-d':
            ddir = v
        elif o == "-g":
            glbn = v
        elif o == "-f":
            first = int(v)
        elif o == "-n":
            number = int(v)
        elif o == "-s":
            step = True
    return (ddir, glbn, first, number, step)
```

function `get_args()`

Has the purpose to manage command line arguments for programs using module `pipeline`.

file: pipeline.py

```
def get_args():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'd:g:f:n:sh')
    except getopt.GetoptError:
        print("Argument error (-h: help)")
        sys.exit()

    ddir = "."
    glbn = "*.*"
    first = None
    number = None
    step = False
    for (o, v) in opts:
        if o == '-h':
            print(__doc__)
            sys.exit()
        elif o == '-d':
            ddir = v
        elif o == "-g":
            glbn = v
        elif o == "-f":
            first = int(v)
        elif o == "-n":
            number = int(v)
        elif o == "-s":
            step = True
    return (ddir, glbn, first, number, step)
```

⇐ See module **getopt**

Data Analysis *Pipeline* -2

A complete example - 4

function `get_args()`

Has the purpose to manage command line arguments for programs using module `pipeline`.

file: pipeline.py

```
def get_args():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'd:g:f:n:sh')
    except getopt.GetoptError:
        print("Argument error (-h: help)")
        sys.exit()

    ddir = "."
    glbn = "*.*"
    first = None
    number = None
    step = False
    for (o, v) in opts:
        if o == '-h':
            print(__doc__)
            sys.exit()
        elif o == '-d':
            ddir = v
        elif o == "-g":
            glbn = v
        elif o == "-f":
            first = int(v)
        elif o == "-n":
            number = int(v)
        elif o == "-s":
            step = True
    return (ddir, glbn, first, number, step)
```

⇐ See module **getopt**

⇐ Default values

function `get_args()`

Has the purpose to manage command line arguments for programs using module `pipeline`.

file: pipeline.py

```
def get_args():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'd:g:f:n:sh')
    except getopt.GetoptError:
        print("Argument error (-h: help)")
        sys.exit()

    ddir = "."
    glbn = "*.*"
    first = None
    number = None
    step = False
    for (o, v) in opts:
        if o == '-h':
            print(__doc__)
            sys.exit()
        elif o == '-d':
            ddir = v
        elif o == "-g":
            glbn = v
        elif o == "-f":
            first = int(v)
        elif o == "-n":
            number = int(v)
        elif o == "-s":
            step = True
    return (ddir, glbn, first, number, step)
```

See module **getopt**

Default values

Get optional arguments

function `get_args()`

Has the purpose to manage command line arguments for programs using module `pipeline`.

file: pipeline.py

```
def get_args():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'd:g:f:n:sh')
    except getopt.GetoptError:
        print("Argument error (-h: help)")
        sys.exit()

    ddir = "."
    glbn = "*.*"
    first = None
    number = None
    step = False
    for (o, v) in opts:
        if o == '-h':
            print(__doc__)
            sys.exit()
        elif o == '-d':
            ddir = v
        elif o == "-g":
            glbn = v
        elif o == "-f":
            first = int(v)
        elif o == "-n":
            number = int(v)
        elif o == "-s":
            step = True
    return (ddir, glbn, first, number, step)
```

See module **getopt** (for the `getopt.getopt` call and the `except getopt.GetoptError` block)

Default values (for `ddir`, `glbn`, `first`, `number`, `step`)

Get optional arguments (the `for (o, v) in opts:` loop)

Return arguments (the `return` line)

Data Analysis *Pipeline* -2

A complete example - 4

function `get_args()`

Has the purpose to manage command line arguments for programs using module `pipeline`.

file: pipeline.py

```
def get_args():
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'd:g:f:n:sh')
    except getopt.GetoptError:
        print("Argument error (-h: help)")
        sys.exit()

    ddir = "."
    glbn = "*.*"
    first = None
    number = None
    step = False
    for (o, v) in opts:
        if o == '-h':
            print(__doc__)
            sys.exit()
        elif o == '-d':
            ddir = v
        elif o == "-g":
            glbn = v
        elif o == "-f":
            first = int(v)
        elif o == "-n":
            number = int(v)
        elif o == "-s":
            step = True
    return (ddir, glbn, first, number, step)
```

See module **getopt**

Default values

Get optional arguments

Return arguments

The above function allows all programs using function `get_args()` to accept the same command line arguments in a consistent way.

Deriving useful classes from Pipeline

Class `Pipeline` is only a framework: in order to do something useful its `_getdata()` method must be implemented.

Actually, the base class method, if used as it is, generates an error: this is wanted, so that the user is reminded that he must derive from `Pipeline` and provide a proper implementation.

file: pipeline.py

```
class RawPipeline(Pipeline):
    "Read raw data files"
    def _getdata(self, filename):
        data, hdr = fits.getdata(filename, 0, header=True)
        return data, {"hdr": hdr, "filename": filename}
```

Deriving useful classes from Pipeline

Class `Pipeline` is only a framework: in order to do something useful its `_getdata()` method must be implemented.

Actually, the base class method, if used as it is, generates an error: this is wanted, so that the user is reminded that he must derive from `Pipeline` and provide a proper implementation.

file: pipeline.py

```
class RawPipeline(Pipeline):
    "Read raw data files"
    def _getdata(self, filename):
        data, hdr = fits.getdata(filename, 0, header=True)
        return data, {"hdr": hdr, "filename": filename}
```

file: pipeline.py

```
class CalPipeline(Pipeline):
    "Read calibration files"
    def _getdata(self, filename):
        data = utils.load(filename)
        return data, {"filename": filename}
```

Deriving useful classes from Pipeline

Class `Pipeline` is only a framework: in order to do something useful its `_getdata()` method must be implemented.

Actually, the base class method, if used as it is, generates an error: this is wanted, so that the user is reminded that he must derive from `Pipeline` and provide a proper implementation.

file: pipeline.py

```
class RawPipeline(Pipeline):
    "Read raw data files"
    def _getdata(self, filename):
        data, hdr = fits.getdata(filename, 0, header=True)
        return data, {"hdr": hdr, "filename": filename}
```

file: pipeline.py

```
class CalPipeline(Pipeline):
    "Read calibration files"
    def _getdata(self, filename):
        data = utils.load(filename)
        return data, {"filename": filename}
```

In both the above classes the `_getdata()` method is called with the filename as argument and returns data from the file and some auxiliary information, e.g.: the file name.

Deriving useful classes from Pipeline

Class `Pipeline` is only a framework: in order to do something useful its `_getdata()` method must be implemented.

Actually, the base class method, if used as it is, generates an error: this is wanted, so that the user is reminded that he must derive from `Pipeline` and provide a proper implementation.

file: pipeline.py

```
class RawPipeline(Pipeline):
    "Read raw data files"
    def _getdata(self, filename):
        data, hdr = fits.getdata(filename, 0, header=True)
        return data, {"hdr": hdr, "filename": filename}
```

file: pipeline.py

```
class CalPipeline(Pipeline):
    "Read calibration files"
    def _getdata(self, filename):
        data = utils.load(filename)
        return data, {"filename": filename}
```

In both the above classes the `_getdata()` method is called with the filename as argument and returns data from the file and some auxiliary information, e.g.: the file name.

Properly showing the image

The following utility function will be used several times in our procedures

file: show.py

```
import matplotlib.pyplot as plt
from astropy.visualization import SqrtStretch
from astropy.visualization.mpl_normalize import ImageNormalize

def show(img):
    norm = ImageNormalize(stretch=SqrtStretch())
    plt.imshow(img, cmap="tab10", norm=norm)
```

Properly showing the image

The following utility function will be used several times in our procedures

file: show.py

```
import matplotlib.pyplot as plt
from astropy.visualization import SqrtStretch   ⟸
from astropy.visualization.mpl_normalize import ImageNormalize

def show(img):
    norm = ImageNormalize(stretch=SqrtStretch())
    plt.imshow(img, cmap="tab10", norm=norm)
```

The module `astropy.visualization` provides useful tools for proper visualization of astronomical images

Properly showing the image

The following utility function will be used several times in our procedures

file: show.py

```
import matplotlib.pyplot as plt
from astropy.visualization import SqrtStretch   ⟸
from astropy.visualization.mpl_normalize import ImageNormalize

def show(img):
    norm = ImageNormalize(stretch=SqrtStretch())
    plt.imshow(img, cmap="tab10", norm=norm)
```

The module `astropy.visualization` provides useful tools for proper visualization of astronomical images

Using show()

```
In [1]: from show import show
In [2]: from astropy.io import fits
In [3]: img = fits.getdata("img-459.fit")
In [4]: show(img)
```

Properly showing the image

The following utility function will be used several times in our procedures

file: show.py

```
import matplotlib.pyplot as plt
from astropy.visualization import SqrtStretch   ⟸
from astropy.visualization.mpl_normalize import ImageNormalize

def show(img):
    norm = ImageNormalize(stretch=SqrtStretch())
    plt.imshow(img, cmap="tab10", norm=norm)
```

The module `astropy.visualization` provides useful tools for proper visualization of astronomical images

Using show()

```
In [1]: from show import show
In [2]: from astropy.io import fits
In [3]: img = fits.getdata("img-459.fit")
In [4]: show(img)
```

⟶

To perform data calibration we use files: `masterdark.fits` and `masterflat.fits`. The files are already preprocessed so that the calibrated image (C) is generated as:

$$C = \frac{(R - D)}{F}$$

where *R* is the raw image *D* is the *dark* and *F* is the *flat*.

file: calibrate.py

```python
import os
from pipeline import RawPipeline, fits
from numpy import ma

DATADIR = "/dati-18.7.2017"

MIN_DARK = -15.0
MAX_DARK = 15.0
MIN_FLAT = 0.85
MAX_FLAT = 1.10

def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)
    cal = (img-common_args["dark"])/common_args["flat"]
    calname = os.path.join("work", shortname.replace("fit", "cal"))
    cal.dump(calname)
    print("OK")

if __name__ == "__main__":
    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")
    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)
    pipe = RawPipeline(calibrate, {"dark":dark, "flat":flat})
    pipe.run()
```

Let’s start a description of the procedure from the bottom of file `calibrate.py`

```
if __name__ == "__main__":

    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")









    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)

    pipe = RawPipeline(calibrate, {"dark":dark, "flat":flat})

    pipe.run()
```

Data calibration -2

Let's start a description of the procedure from the bottom of file `calibrate.py`

Direct execution

```
if __name__ == "__main__":

    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")

    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)

    pipe = RawPipeline(calibrate, {"dark":dark, "flat":flat})

    pipe.run()
```

Let's start a description of the procedure from the bottom of file `calibrate.py`

```
if __name__ == "__main__":

    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")
```

Direct execution

Set up file names for *dark* e *flat*

```
    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)

    pipe = RawPipeline(calibrate, {"dark":dark, "flat":flat})

    pipe.run()
```

Data calibration -2

Let's start a description of the procedure from the bottom of file `calibrate.py`

Direct execution

Set up file names for *dark* e *flat*

Read *dark* and *flat* images.

Function **masked_outside()** converts the image into a *masked array*, masking out "unreasonable" pixels (i.e.: pixels with value outside the specified interval).

```
if __name__ == "__main__":

    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")

    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)

    pipe = RawPipeline(calibrate, {"dark":dark, "flat":flat})

    pipe.run()
```

Let's start a description of the procedure from the bottom of file `calibrate.py`

Direct execution

```
if __name__ == "__main__":

    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")
```

Set up file names for *dark* e *flat*

Read *dark* and *flat* images.

Function **masked_outside()** converts the image into a *masked array*, masking out "unreasonable" pixels (i.e.: pixels with value outside the specified interval).

```
    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)

    pipe = RawPipeline(calibrate, {"dark":dark, "fla

    pipe.run()
```

Create an object `RawPipeline`

Let's start a description of the procedure from the bottom of file `calibrate.py`

```
if __name__ == "__main__":

    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")

    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)

    pipe = RawPipeline(calibrate, {"dark":dark, "fla

    pipe.run()
```

Direct execution

Set up file names for *dark* e *flat*

Read *dark* and *flat* images.

Function **masked_outside()** converts the image into a *masked array*, masking out "unreasonable" pixels (i.e.: pixels with value outside the specified interval).

Create an object `RawPipeline`

Fire the loop on raw data files

Data calibration -2

A complete example - 8

Let’s start a description of the procedure from the bottom of file `calibrate.py`

```
if __name__ == "__main__":

    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")

    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)

    pipe = RawPipeline(calibrate, {"dark":dark, "fla

    pipe.run()
```

Direct execution

Set up file names for *dark* e *flat*

Read *dark* and *flat* images.

Function **masked_outside()** converts the image into a *masked array*, masking out “unreasonable” pixels (i.e.: pixels with value outside the specified interval).

Create an object `RawPipeline`

Fire the loop on raw data files

Being `dark` and `flat` masked arrays, any operations involving them will be performed ignoring non valid pixels.

Let's start a description of the procedure from the bottom of file `calibrate.py`

```
if __name__ == "__main__":

    dfile = os.path.join(DATADIR, "masterdark.fits")
    ffile = os.path.join(DATADIR, "masterflat.fits")

    dark = ma.masked_outside(fits.getdata(dfile), MIN_DARK, MAX_DARK)
    flat = ma.masked_outside(fits.getdata(ffile), MIN_FLAT, MAX_FLAT)

    pipe = RawPipeline(calibrate, {"dark":dark, "fla

    pipe.run()
```

Direct execution

Set up file names for *dark* e *flat*

Read *dark* and *flat* images.

Function **masked_outside()** converts the image into a *masked array*, masking out "unreasonable" pixels (i.e.: pixels with value outside the specified interval).

Create an object `RawPipeline`

Fire the loop on raw data files

Being `dark` and `flat` masked arrays, any operations involving them will be performed ignoring non valid pixels.

→

Let's see the actual calibration function in file calibrate.py

```
def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)




    cal = (img-common_args["dark"])/common_args["flat"]











    calname = os.path.join("work", shortname.replace("fit", "cal"))


    cal.dump(calname)
    print("OK")
```

Data calibration -3

A complete example - 9

Let's see the actual calibration function in file calibrate.py

```
def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)




    cal = (img-common_args["dark"])/common_args["flat"]











    calname = os.path.join("work", shortname.replace("fit", "cal"))


    cal.dump(calname)
    print("OK")
```

Some *output* to show that the procedure is running

Let's see the actual calibration function in file calibrate.py

```
def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)
```

Some *output* to show that the procedure is running

```
    cal = (img-common_args["dark"])/common_args["flat"]
```

Computing calibrated image. Expression's arguments are *arrays*, thus the operations are performed element by element.

```
    calname = os.path.join("work", shortname.replace("fit", "cal"))

    cal.dump(calname)
    print("OK")
```

Data calibration -3

A complete example - 9

Let's see the actual calibration function in file calibrate.py

```
def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)
```

> Some *output* to show that the procedure is running

```
    cal = (img-common_args["dark"])/common_args["flat"]
```

> Computing calibrated image. Expression's arguments are arrays, thus the operations are p

> `dark` and `flat` are *masked arrays*, thus non valid element are not considered and the result is again a *masked array*.

```
    calname = os.path.join("work", shortname.replace("fit", "cal"))

    cal.dump(calname)
    print("OK")
```

Let's see the actual calibration function in file calibrate.py

```
def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)
```

Some *output* to show that the procedure is running

```
    cal = (img-common_args["dark"])/common_args["flat"]
```

Computing calibrated image. Expression's ar-
guments are arrays, thus the operations are

`dark` and `flat` are *masked arrays*, thus non valid element are not considered and the result is again a *masked array*.

Generate a proper file name for the calibrated image

```
    calname = os.path.join("work", shortname.replace("fit", "cal"))

    cal.dump(calname)
    print("OK")
```

Data calibration -3

A complete example - 9

Let's see the actual calibration function in file calibrate.py

```
def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)

    cal = (img-common_args["dark"])/common_args["flat"]

    calname = os.path.join("work", shortname.replace("fit", "cal"))

    cal.dump(calname)
    print("OK")
```

Some *output* to show that the procedure is running

Computing calibrated image. Expression's ar-

`dark` and `flat` are *masked arrays*, thus non valid element are not considered and the result is again a *masked array*.

Generate a proper file name for the calibrated image

Write calibrated image onto file

Data calibration -3

Let's see the actual calibration function in file calibrate.py

```
def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)
```

Some *output* to show that the procedure is running

```
    cal = (img-common_args["dark"])/common_args["flat"]
```

Computing calibrated image. Expression's ar-

`dark` and `flat` are *masked arrays*, thus non valid element are not considered and the result is again a *masked array*.

Generate a proper file name for the calibrated image

```
    calname = os.path.join("work", shortname.replace("fit", "cal"))

    cal.dump(calname)
    print("OK")
```

Write calibrated image onto file

Reminder: function **calibrate()** is called once for each data file in the list.

Let's see the actual calibration function in file calibrate.py

```python
def calibrate(img, step, args, common_args):
    shortname = os.path.basename(args["filename"])
    print("--- Calibrating:", shortname, end=" ", flush=True)
```

Some *output* to show that the procedure is running

```python
    cal = (img-common_args["dark"])/common_args["flat"]
```

Computing calibrated image. Expression's ar-

dark and flat are *masked arrays*, thus non valid element are not considered and the result is again a *masked array*.

Generate a proper file name for the calibrated image

```python
    calname = os.path.join("work", shortname.replace("fit", "cal"))

    cal.dump(calname)
    print("OK")
```

Write calibrated image onto file

Reminder: function **calibrate()** is called once for each data file in the list.

Running calibrate.py

```
$ python calibrate.py -d /dati-18.7.2017 -g img-\*
```

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, image):
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADIUS])
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, image):
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADIUS])
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Stars selection -1

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADIUS])
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

file: select.py

```python
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADIUS])
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

file: select.py

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to corresponding functions

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADIUS])
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADIUS])
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to ... nc-

Execution loop.
Note: **plt.pause()**

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADIUS])
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to ... nc-

Execution loop.
Note: **plt.pause()**

Function called on button press

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADIUS])
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to corresponding func-

Execution loop.
Note: **plt.pause()**

Function called on button press

Function called on button release

file: select.py

Find candidate stars

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADI
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Set interactive mode

Show the image for star selection

Connect events to ... nc-

Execution loop.
Note: **plt.pause()**

Function called on button press

Function called on button release

- Generate a circle
- Save position
- Draw it

Stars selection -1

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADI
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to ... nc-

Execution loop.
Note: **plt.pause()**

Function called on button press

Function called on button release

- Generate a circle
- Save position
- Draw it

Program execution

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADI
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to ... func-

Execution loop.
Note: **plt.pause()**

Function called on button press

Function called on button release

- Generate a circle
- Save position
- Draw it

F

Get image name from command line

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADI
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(img)
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to ... func-

Execution loop.
Note: **plt.pause()**

Function called on button press

Function called on button release

- Generate a circle
- Save position
- Draw it

F

Get image name from command line

Instantiate `ObjectSelector`

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADI
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(im
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to corresponding func-

Execution loop.
Note: **plt.pause()**

Function called on button press

Function called on button release

- Generate a circle
- Save position
- Draw it

F

Get image name from command line

elector

Start the display *loop*

Stars selection -1

file: select.py

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from show import show

RADIUS = 50
class ObjectSelector:
    def __init__(self, imag
        plt.ion()
        self.fig = plt.figure()
        show(image)
        plt.title("Left click to select object. End with right click")
        self.ax = self.fig.gca()
        cid1 = self.fig.canvas.mpl_connect("button_press_event", self._press)
        cid2 = self.fig.canvas.mpl_connect("button_release_event", self._release)
        self.goon = True
        self.circles = []
        plt.show()

    def start(self):
        while self.goon: plt.pause(0.10)

    def _press(self, event):
        if event.button == 1: self.xc = event.xdata; self.yc = event.ydata

    def _release(self, event):
        if event.button == 1:
            circle = plt.Circle((self.xc, self.yc), RADIUS, color="cyan", fill=Fal
            self.ax.add_artist(circle)
            self.circles.append([self.xc, self.yc, RADI
            plt.draw()
        elif event.button == 3:
            self.goon = False

if __name__ == "__main__":
    img = np.load(sys.argv[1])
    sel = ObjectSelector(im
    sel.start()
    with open("selected.dat", "w") as fpt:
        print(sel.circles, file=fpt)
```

Find candidate stars

Set interactive mode

Show the image for star selection

Connect events to ... nc-

Execution loop.
Note: **plt.pause()**

Function called on button press

Function called on button release

- Generate a circle
- Save position
- Draw it

F

Get image name from command line

elector

Start the display *loop*

Eventually, write circle positions into a file

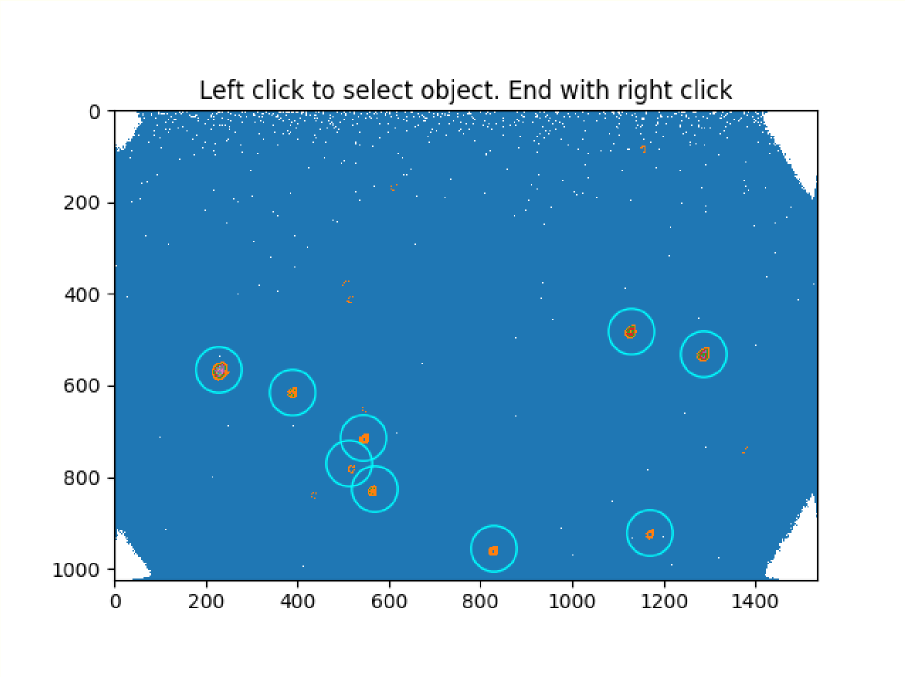
How does `select.py` work

```
$:  python select.py work/img-460.cal
```

How does `select.py` work

```
$:  python select.py work/img-460.cal
```

- Left *Click* on stars to add to selection

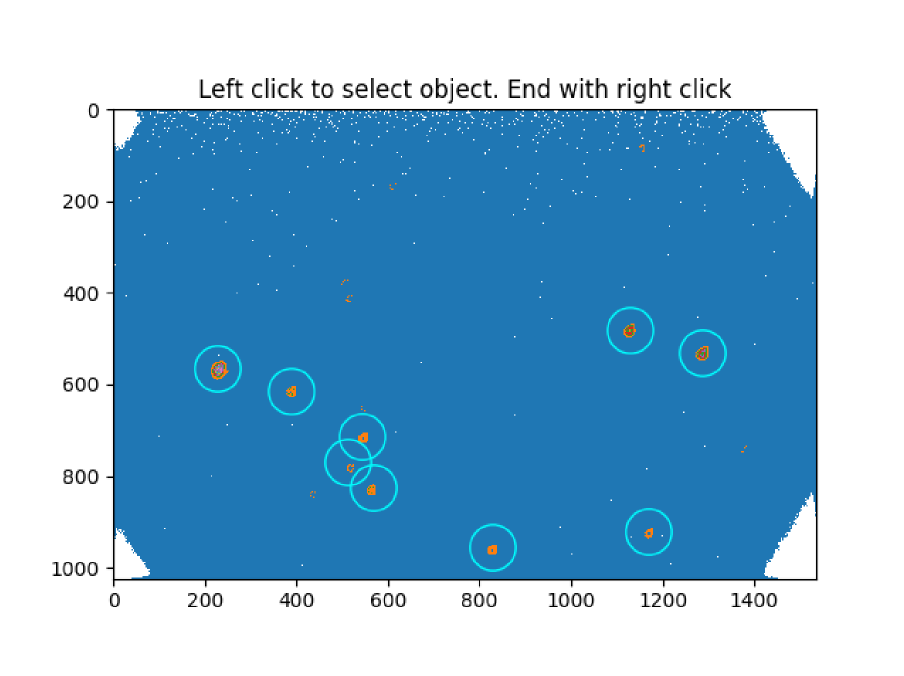


Stars selection -2

How does `select.py` work

```
$:  python select.py work/img-460.cal
```

- Left *Click* on stars to add to selection

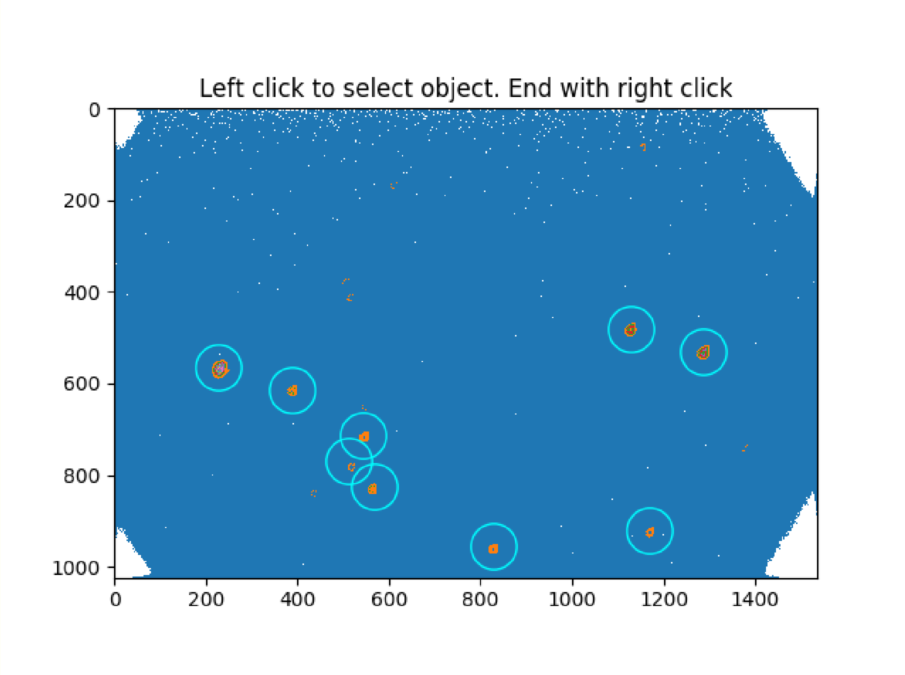


- To terminate: right *click*

How does `select.py` work

$: python select.py work/img-460.cal

- Left *Click* on stars to add to selection



- To terminate: right *click*
- Positions are written into file `selected.dat` as:

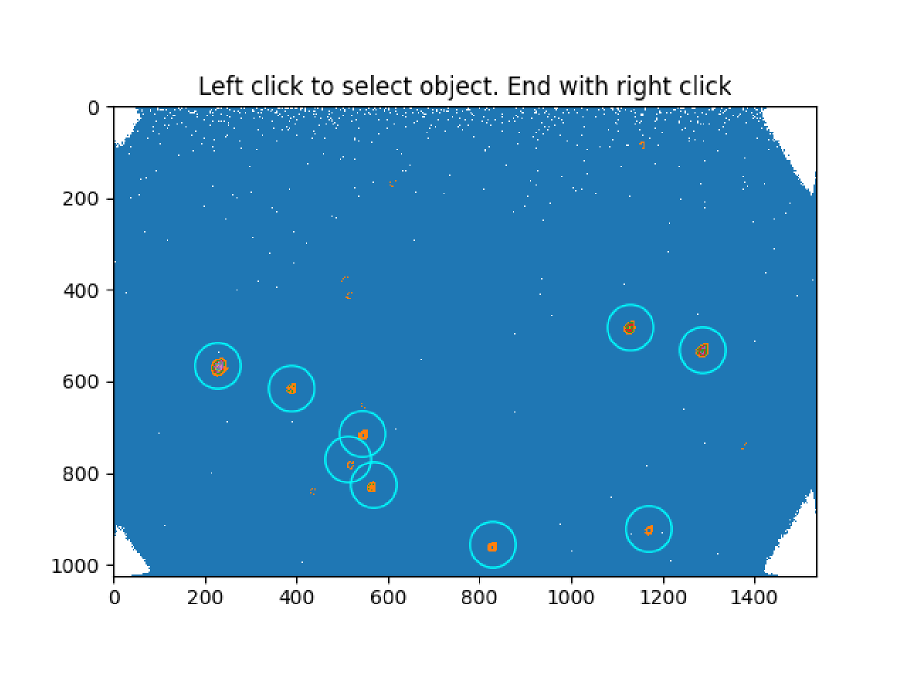
```
[[228.66129032258061, 566.00322580645161, 50], [1129.8225806451612,
482.39032258 064515, 50], [1287.758064516129, 531.9387096774193, 50],
[389.69354838709671, 615.55161290322576, 50], [544.5322580645161,
714.64838709677417, 50], [513.5645161290322, 770.39032258064515, 50],
[569.30645161290317, 826.13225806451612, 50], [ 829.4354838709678,
956.19677419354844, 50], [1170.0806451612902, 922.13225806451612, 50]]
```

Defining position of selected circular areas (nine, in the example)

How does `select.py` work

`$:  python select.py work/img-460.cal`

- Left *Click* on stars to add to selection



- To terminate: right *click*
- Positions are written into file `selected.dat` as:

```
[[228.66129032258061, 566.00322580645161, 50], [1129.8225806451612,
482.39032258 064515, 50], [1287.758064516129, 531.9387096774193, 50],
[389.69354838709671, 615.55161290322576, 50], [544.5322580645161,
714.64838709677417, 50], [513.5645161290322, 770.39032258064515, 50],
[569.30645161290317, 826.13225806451612, 50], [ 829.4354838709678,
956.19677419354844, 50], [1170.0806451612902, 922.13225806451612, 50]]
```

Defining position of selected circular areas (nine, in the example)

→

Stars selection -3

Program `select.py` was maintained as simple as possible. In “real life” some enhancements would be necessary:

- Make it clear that the exoplanet’s star must be selected as first.

Stars selection -3

Program `select.py` was maintained as simple as possible. In “real life” some enhancements would be necessary:

- Make it clear that the exoplanet’s star must be selected as first.
- Make cross size proportional to star brighness

Stars selection -3

Program `select.py` was maintained as simple as possible. In “real life” some enhancements would be necessary:

- Make it clear that the exoplanet’s star must be selected as first.
- Make cross size proportional to star brighness
- Make error messages more explanatory

Stars selection -3

Program `select.py` was maintained as simple as possible. In "real life" some enhancements would be necessary:

- Make it clear that the exoplanet's star must be selected as first.
- Make cross size proportional to star brighness
- Make error messages more explanatory
- Allow interactive selection of data image

Stars selection -3

Program `select.py` was maintained as simple as possible. In “real life” some enhancements would be necessary:

- Make it clear that the exoplanet’s star must be selected as first.
- Make cross size proportional to star brighness
- Make error messages more explanatory
- Allow interactive selection of data image
- Display improvements (e.g.: dynamic *colormap*)

Stars selection -3

Program `select.py` was maintained as simple as possible. In “real life” some enhancements would be necessary:

- Make it clear that the exoplanet’s star must be selected as first.
- Make cross size proportional to star brighness
- Make error messages more explanatory
- Allow interactive selection of data image
- Display improvements (e.g.: dynamic *colormap*)
- Allow interactive deselection of stars

Stars selection -3

Program `select.py` was maintained as simple as possible. In “real life” some enhancements would be necessary:

- Make it clear that the exoplanet’s star must be selected as first.
- Make cross size proportional to star brighness
- Make error messages more explanatory
- Allow interactive selection of data image
- Display improvements (e.g.: dynamic *colormap*)
- Allow interactive deselection of stars
- Allow interactive modification of circle diameter

Stars selection -3

Program `select.py` was maintained as simple as possible. In “real life” some enhancements would be necessary:

- Make it clear that the exoplanet’s star must be selected as first.
- Make cross size proportional to star brighness
- Make error messages more explanatory
- Allow interactive selection of data image
- Display improvements (e.g.: dynamic *colormap*)
- Allow interactive deselection of stars
- Allow interactive modification of circle diameter
- Use JSON as position recording format

Stars selection -3

Program `select.py` was maintained as simple as possible. In “real life” some enhancements would be necessary:

- Make it clear that the exoplanet’s star must be selected as first.
- Make cross size proportional to star brighness
- Make error messages more explanatory
- Allow interactive selection of data image
- Display improvements (e.g.: dynamic *colormap*)
- Allow interactive deselection of stars
- Allow interactive modification of circle diameter
- Use JSON as position recording format

Stars selection -3

Program `select.py` was maintained as simple as possible. In "real life" some enhancements would be necessary:

- Make it clear that the exoplanet's star must be selected as first.
- Make cross size proportional to star brighness
- Make error messages more explanatory
- Allow interactive selection of data image
- Display improvements (e.g.: dynamic *colormap*)
- Allow interactive deselection of stars
- Allow interactive modification of circle diameter
- Use JSON as position recording format

→

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    for cc in common_args["circles"]:
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                        cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    for cc in common_args["circles"]:
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                     cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    for cc in common_args["circles"]:
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                            cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Stars centering -1

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes
    axes.cla()
    show(img)
    for cc in common_args["circles"]:
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                                  cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes
    axes.cla()
    show(img)
    for cc in common_args["circles"]:
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                          cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

Show the new image

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes"
    axes.cla()
    show(img)
    for cc in common_args["circ
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                                  cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

Show the new image

Draw circles

Stars centering -1

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes"
    axes.cla()
    show(img)
    for cc in common_args["circ
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                                  cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

Show the new image

Draw circles

Add title

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar[illegible]
    axes = common_args["axes[illegible]
    axes.cla()
    show(img)
    for cc in common_args["circ[illegible]
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                       cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

Show the new image

Draw circles

Add title

Visualization pause

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes"
    axes.cla()
    show(img)
    for cc in common_args["circ
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                       cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

Show the new image

Draw circles

Add title

Visualization pause

Set matplotlib interactive mode

Stars centering -1

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes"
    axes.cla()
    show(img)
    for cc in common_args["circ
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                        cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Show the new image

Add title

Visualization pause

Set matplotlib interactive mode

Generate circles based on data from file

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes"
    axes.cla()
    show(img)
    for cc in common_args["circ
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                                  cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

Show the new image

Draw circles

Add title

Visualization pause

Set matplotlib interactive mode

Generate circles based on data from file

Create the `CalPipeline` object

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes"
    axes.cla()
    show(img)
    for cc in common_args["circ
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                          cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

Show the new image

Draw circles

Add title

Visualization pause

Set matplotlib interactive mode

Generate circles based on data from file

Create the `CalPipeline` object

Start loop

The following simple procedure gives an idea of the centering problem.

Note: it uses again a Pipeline object.

file: timelapse.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
def time_lapse(img, step, ar
    axes = common_args["axes"
    axes.cla()
    show(img)
    for cc in common_args["circ
        axes.add_artist(cc)
    plt.title("img: %d"%step)
    plt.pause(0.05)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure
    ax = fig.gca()
    circles = []
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    for cd in cdata:
        circles.append(plt.Circle((cd[0], cd[1]),
                      cd[2], color="cyan", fill=False))
    pipe = CalPipeline(time_lapse, {"axes":ax, "circles":circles})
    pipe.run()
```

Function called on each data file

Clean image display

Show the new image

Draw circles

Add title

Visualization pause

Set matplotlib interactive mode

Generate circles based on data from file

Create the CalPipeline object

Start loop

→

Stars centering -2

Resuming:

- Each image must be re-centered so that it covers the same area on sky.

Stars centering -2

Resuming:

- Each image must be re-centered so that it covers the same area on sky.
- On each image an area around selected stars must be defined .

Stars centering -2

Resuming:

- Each image must be re-centered so that it covers the same area on sky.
- On each image an area around selected stars must be defined .
- We start from circle data in file `selected.dat`

Stars centering -2

Resuming:

- Each image must be re-centered so that it covers the same area on sky.
- On each image an area around selected stars must be defined .
- We start from circle data in file `selected.dat`
- As a double-check you might also verify that all selected objects have the same shift.

Resuming:

- Each image must be re-centered so that it covers the same area on sky.
- On each image an area around selected stars must be defined .
- We start from circle data in file `selected.dat`
- As a double-check you might also verify that all selected objects have the same shift.

... in general:

Stars centering -2

Resuming:

- Each image must be re-centered so that it covers the same area on sky.
- On each image an area around selected stars must be defined .
- We start from circle data in file `selected.dat`
- As a double-check you might also verify that all selected objects have the same shift.

  ... in general:
- It could be necessary to take field rotation into account

Stars centering -2

Resuming:

- Each image must be re-centered so that it covers the same area on sky.
- On each image an area around selected stars must be defined .
- We start from circle data in file `selected.dat`
- As a double-check you might also verify that all selected objects have the same shift.

  ... in general:
- It could be necessary to take field rotation into account

file: photometry1.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

Doing photometry -1

A complete example - 15

file: photometry1.py

> Import required tools

```
import json
from photutils import centro                          , aperture_photometry
from pipeline import CalPipe

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

file: photometry1.py

```
import json
from photutils import centro                 , aperture_photometry
from pipeline import CalPipe

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

Import required

Selected star positions are here. The first one is our target

file: photometry1.py

```
import json
from photutils import centro                                  , aperture_photometry
from pipeline import CalPipe

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2],
    flx = aperture_photometry(img, apertu
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

Import required

Selected star positions are here. The first one is our target

Function **flux()** computes the total flux in a circular area around the given point **cc**

file: photometry1.py

```
import json
from photutils import centro                , aperture_photometry
from pipeline import CalPipe

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2],
    flx = aperture_photometry(img, apertu
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

Import required

Selected star positions are here. The first one is our target

Function **flux()** computes the total flux in a circular area around the given point **cc**

Function **calc_flux()** computes the differential flux of target star for each input image

file: photometry1.py

```
import json
from photutils import centro
from pipeline import CalPipe

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2], S
    flx = aperture_photometry(img, apertu
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles
    pipe.run()
```

, aperture_photometry

Import required

Selected star positions are here. The first one is our target

Function **flux()** computes the total flux in a circular area around the given point **cc**

Function **calc_flux()** computes the differential flux of target star for each input image

We use the `Pipeline` class to iterate on the calibrated images

file: photometry1.py

```
import json
from photutils import centro                              , aperture_photometry
from pipeline import CalPipe

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2],
    flx = aperture_photometry(img, apertu
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles
    pipe.run()
```

Import required

Selected star positions are here. The first one is our target

Function **flux()** computes the total flux in a circular area around the given point **cc**

Function **calc_flux()** computes the differential flux of target star for each input image

We use the `Pipeline` class to iterate on the calibrated images

Note Function flux, at each iteration, updates the position of the `CircularAperture()` object for the following step (because `aperture_photometry()` returns the centroid position together with the computed flux).

file: photometry1.py

```
import json
from photutils import centro                    , aperture_photometry
from pipeline import CalPipe

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2],
    flx = aperture_photometry(img, apertu
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles
    pipe.run()
```

Import required

Selected star positions are here. The first one is our target

Function **flux()** computes the total flux in a circular area around the given point **cc**

Function **calc_flux()** computes the differential flux of target star for each input image

We use the `Pipeline` class to iterate on the calibrated images

Note Function flux, at each iteration, updates the position of the `CircularAperture()` object for the following step (because `aperture_photometry()` returns the centroid position together with the computed flux).

→

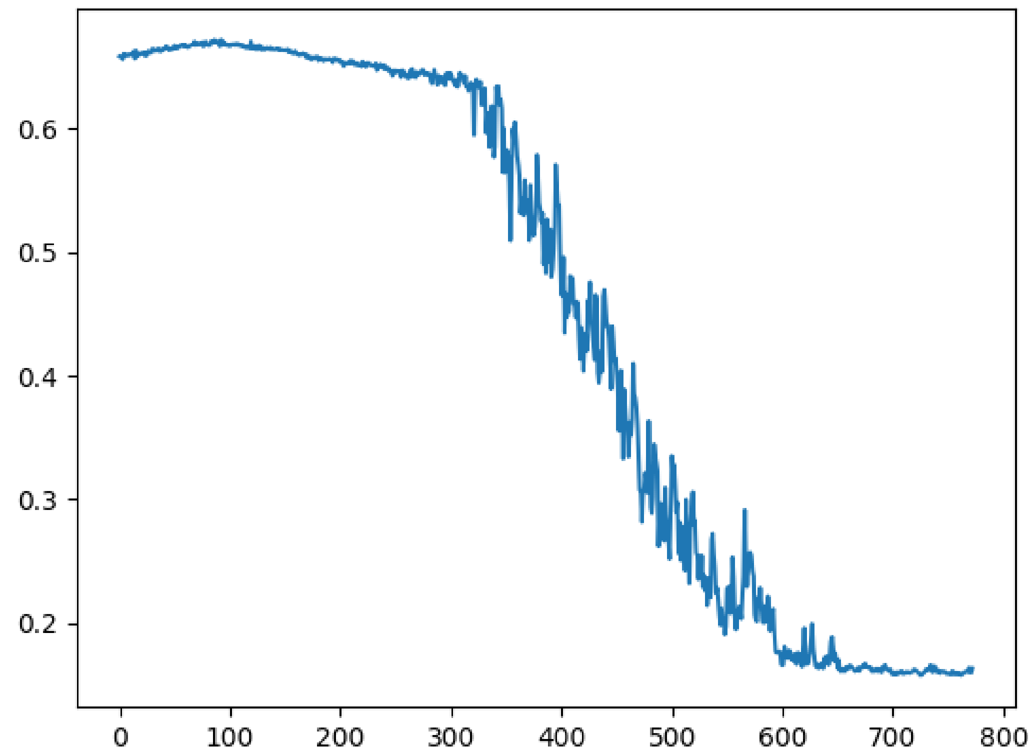
Running photometry.py from `ipython`

```
In [1]: %run photometry.py -d work
Step 0 done
Step 1 done
....
Step 771 done
Step 772 done

In [2]: plt.plot(DIFF_FLUX)
```

Running photometry.py from `ipython`

```
In [1]: %run photometry.py -d work
Step 0 done
Step 1 done
....
Step 771 done
Step 772 done

In [2]: plt.plot(DIFF_FLUX)
```

... and here's the result:

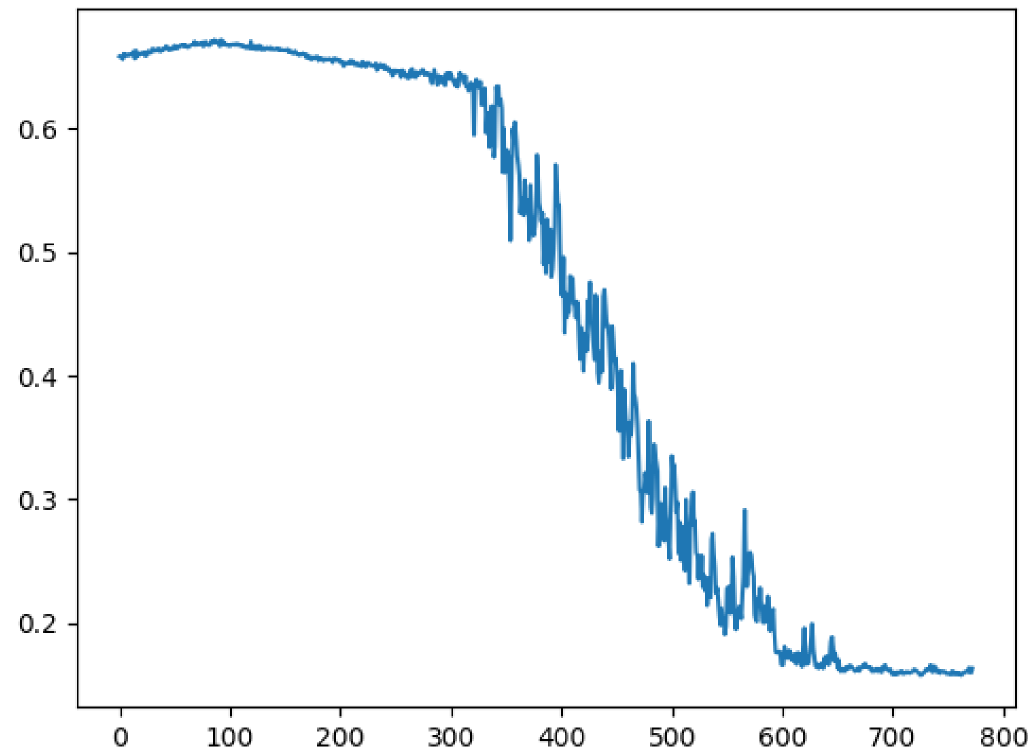


Clearly that's not what we would expect!

Running photometry.py from `ipython`

```
In [1]: %run photometry.py -d work
Step 0 done
Step 1 done
....
Step 771 done
Step 772 done

In [2]: plt.plot(DIFF_FLUX)
```

... and here's the result:



Clearly that's not what we would expect!

→

Let's look at single stars.

Let's look at single stars.

file: photometry1.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
STAR_FLUXES=[]
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    STAR_FLUXES.append(fluxes)
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

Let's look at single stars.

file: photometry1.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
STAR_FLUXES=[]
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    STAR_FLUXES.append(fluxes)
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

photometry1.py is very similar to the previous function

Let's look at single stars.

file: photometry1.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
STAR_FLUXES=[]
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    STAR_FLUXES.append(fluxes)
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

`photometry1.py` is very similar to the previous function

⇐ but it also stores fluxes for each star

Let's look at single stars.

file: photometry1.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = flx["xcenter"][0].value
    cc[1] = flx["ycenter"][0].value
    return flx["aperture_sum"][0]

DIFF_FLUX = []
STAR_FLUXES=[]
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    STAR_FLUXES.append(fluxes)
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

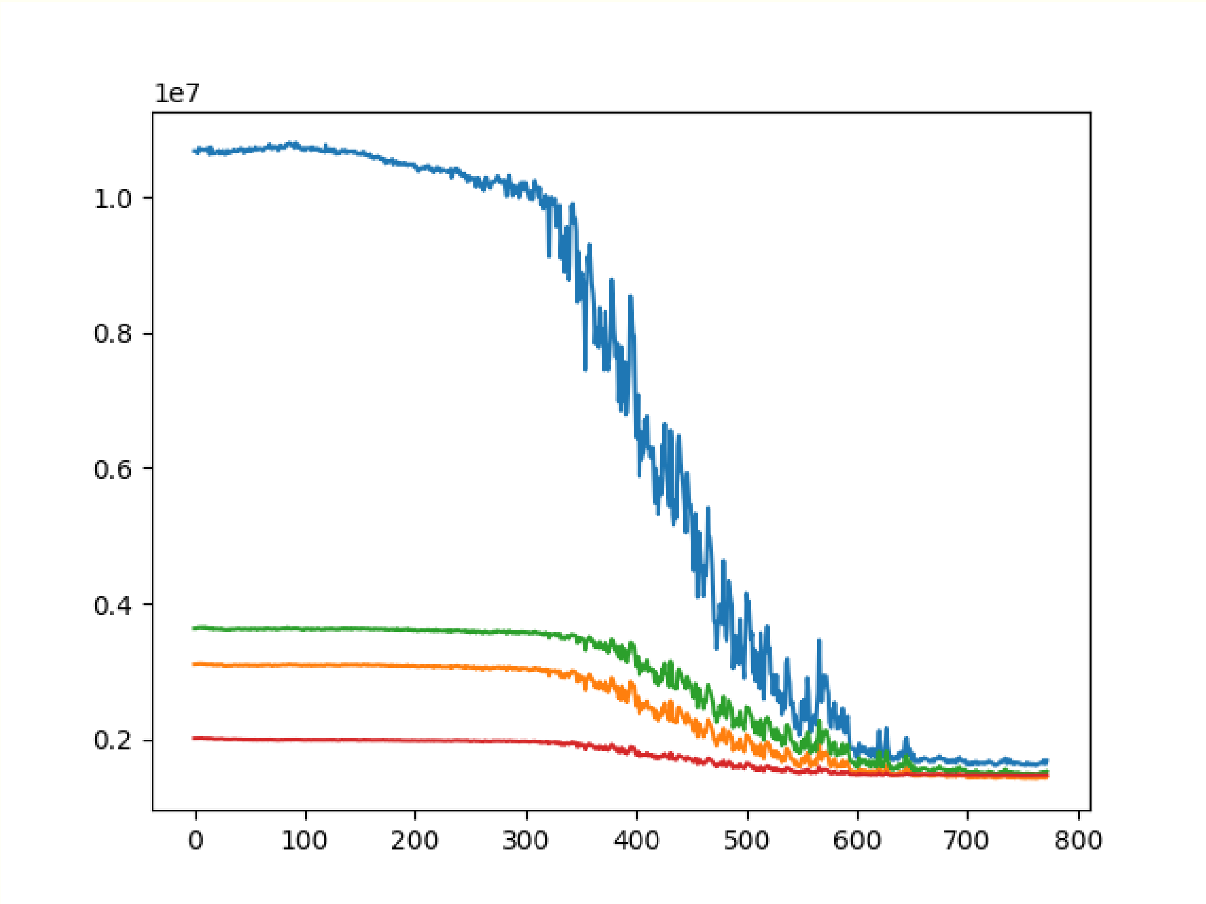
photometry1.py is very similar to the previous function

but it also stores fluxes for each star

→

```
In [30]: %run photometry1.py -d work

In [31]: stars = np.array(STAR_FLUXES).transpose()

In [32]: plt.plot(stars[0])

In [33]: plt.plot(stars[1])

In [34]: plt.plot(stars[2])

In [35]: plt.plot(stars[3])
```

```
In [30]: %run photometry1.py -d work

In [31]: stars = np.array(STAR_FLUXES).transpose()

In [32]: plt.plot(stars[0])

In [33]: plt.plot(stars[1])

In [34]: plt.plot(stars[2])

In [35]: plt.plot(stars[3])
```



My first interpretation of the above plot: the recentering algorithm is not working.

I.e.: starting aroud the 300th image the stars drop out of the aperture

Let's see it better:

file: timelapse1.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    ccx = flx["xcenter"][0].value
    ccy = flx["ycenter"][0].value
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(time_lapse, {"axes":ax, "center":cdata[0]})
    pipe.run()
```

Doing photometry -5

Let's see it better:

file: timelapse1.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    ccx = flx["xcenter"][0].value
    ccy = flx["ycenter"][0].value
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(time_lapse, {"axes":ax, "center":cdata[0]})
    pipe.run()
```

`timlapse1.py` merges code from `timelapse.py` and `photometry.py` to plot the centroid position at each step

Let's see it better:

file: timelapse1.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    ccx = flx["xcenter"][0].value
    ccy = flx["ycenter"][0].value
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(time_lapse, {"axes":ax, "center":cdata[0]})
    pipe.run()
```

> `timlapse1.py` merges code from `timelapse.py` and `photometry.py` to plot the centroid position at each step

> Notably it shows the aperture center and the computed centroid

Let’s see it better:

file: timelapse1.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    ccx = flx["xcenter"][0].value
    ccy = flx["ycenter"][0].value
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(time_lapse, {"axes":ax, "center":cdata[0]})
    pipe.run()
```

`timlapse1.py` merges code from `timelapse.py` and `photometry.py` to plot the centroid position at each step

Notably it shows the aperture center and the computed centroid

```
python timelapse1.py -d work
```

Let’s see it better:

file: timelapse1.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    ccx = flx["xcenter"][0].value
    ccy = flx["ycenter"][0].value
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(time_lapse, {"axes":ax, "center":cdata[0]})
    pipe.run()
```

> `timlapse1.py` merges code from `timelapse.py` and `photometry.py` to plot the centroid position at each step

> Notably it shows the aperture center and the computed centroid

python

> In conclusion: RTFM (Read the f... manual)

Let’s see it better:

file: timelapse1.py

```
import json, os
import matplotlib.pyplot as plt
from show import show
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    aperture = CircularAperture(cc[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    ccx = flx["xcenter"][0].value
    ccy = flx["ycenter"][0].value
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)

if __name__ == "__main__":
    plt.ion()
    fig = plt.figure()
    ax = fig.gca()
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(time_lapse, {"axes":ax, "center":cdata[0]})
    pipe.run()
```

`timlapse1.py` merges code from `timelapse.py` and `photometry.py` to plot the centroid position at each step

Notably it shows the aperture center and the computed centroid

`python`

In conclusion: RTFM (Read the f... manual)

Doing photometry -6

A complete example - 20

Let’s compute the centroid

file: centroid.py

```
from photutils import centroid_com

def centroid(img, ctr):
   x0 = int(max(ctr[0]-ctr[2], 0))
   x1 = int(min(ctr[0]+ctr[2], img.shape[1]-1))
   y0 = int(max(ctr[1]-ctr[2], 0))
   y1 = int(min(ctr[1]+ctr[2], img.shape[0]-1))
   subimg = img[y0:y1, x0:x1]
   ctd = centroid_com(subimg)
   return (x0+ctd[0], y0+ctd[0])
```

... and use it again in timelapse

file: timelapse2.py

```
...
from centroid import centroid

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    ccx, ccy = centroid(img, cc)
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)
...
```

Let's compute the centroid

file: centroid.py

```
from photutils import centroid_com

def centroid(img, ctr):
   x0 = int(max(ctr[0]-ctr[2], 0))
   x1 = int(min(ctr[0]+ctr[2], img.shape[1]-1))
   y0 = int(max(ctr[1]-ctr[2], 0))
   y1 = int(min(ctr[1]+ctr[2], img.shape[0]-1))
   subimg = img[y0:y1, x0:x1]
   ctd = centroid_com(subimg)
   return (x0+ctd[0], y0+ctd[0])
```

... and use it again in timelapse

file: timelapse2.py

```
...
from centroid import centroid

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    ccx, ccy = centroid(img, cc)
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)
...
```

```
python timelapse2.py -d work
```

Let's compute the centroid

file: centroid.py

```
from photutils import centroid_com

def centroid(img, ctr):
   x0 = int(max(ctr[0]-ctr[2], 0))
   x1 = int(min(ctr[0]+ctr[2], img.shape[1]-1))
   y0 = int(max(ctr[1]-ctr[2], 0))
   y1 = int(min(ctr[1]+ctr[2], img.shape[0]-1))
   subimg = img[y0:y1, x0:x1]
   ctd = centroid_com(subimg)
   return (x0+ctd[0], y0+ctd[0])
```

... and use it again in timelapse

file: timelapse2.py

```
...
from centroid import centroid

CIRCFILE = "selected.dat"
BRACE = 30
STAR_RADIUS = 30
def time_lapse(img, step, args, common_args):
    axes = common_args["axes"]
    axes.cla()
    show(img)
    cc = common_args["center"]
    plt.plot((cc[0]-BRACE, cc[0]+BRACE), (cc[1], cc[1]), color="cyan")
    plt.plot((cc[0], cc[0]), (cc[1]-BRACE, cc[1]+BRACE), color="cyan")
    ccx, ccy = centroid(img, cc)
    plt.title("img: %d - Center: %.2f, %.2f / %.2f, %.2f"%(step, cc[0], cc[1], cc
    cc[0] = ccx
    cc[1] = ccy
    plt.pause(0.10)
...
```

```
python timelapse2.py -d work
```

Here's the new version:

file: photometry2.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

from centroid import centroid

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    ctd = centroid(img, cc)
    aperture = CircularAperture(ctd[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = ctd[0]
    cc[1] = ctd[1]
    return flx["aperture_sum"][0]

DIFF_FLUX = []
STAR_FLUXES=[]
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    STAR_FLUXES.append(fluxes)
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

Here's the new version:

file: photometry2.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

from centroid import centroid

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    ctd = centroid(img, cc)
    aperture = CircularAperture(ctd[0:2], STAR_RADIUS)
    flx = aperture_photometry(img, aperture)
    cc[0] = ctd[0]
    cc[1] = ctd[1]
    return flx["aperture_sum"][0]

DIFF_FLUX = []
STAR_FLUXES=[]
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    STAR_FLUXES.append(fluxes)
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

We compute the centroid before doing photometry ...

Here's the new version:

file: photometry2.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

from centroid import centroid

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    ctd = centroid(img, cc)
    aperture = CircularAperture(ctd[0:2], STAR_RADIUS)
    flx = aperture_photometry(
    cc[0] = ctd[0]
    cc[1] = ctd[1]
    return flx["aperture_sum"][0]

DIFF_FLUX = []
STAR_FLUXES=[]
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    STAR_FLUXES.append(fluxes)
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

We compute the centroid before doing photometry ...

Then we update the center position for the next step

Here’s the new version:

file: photometry2.py

```
import json
from photutils import centroid_com, CircularAperture, aperture_photometry
from pipeline import CalPipeline

from centroid import centroid

CIRCFILE = "selected.dat"
STAR_RADIUS = 30

def flux(img, cc):
    ctd = centroid(img, cc)
    aperture = CircularAperture(ctd[0:2], STAR_RADIUS)
    flx = aperture_photometry(
    cc[0] = ctd[0]
    cc[1] = ctd[1]
    return flx["aperture_sum"][0]

DIFF_FLUX = []
STAR_FLUXES=[]
def calc_flux(img, step, args, common_args):
    global DIFF_FLUX
    fluxes = []
    for cc in common_args["circles"]:
        fluxes.append(flux(img, cc))
    STAR_FLUXES.append(fluxes)
    diff_flux = fluxes[0]/sum(fluxes[1:])
    DIFF_FLUX.append(diff_flux)
    print("Step %d done"%step)

if __name__ == "__main__":
    with open(CIRCFILE) as cfile:
        cdata = json.load(cfile)
    pipe = CalPipeline(calc_flux, {"circles": cdata})
    pipe.run()
```

We compute the centroid before doing photometry ...

Then we update the center position for the next step

→

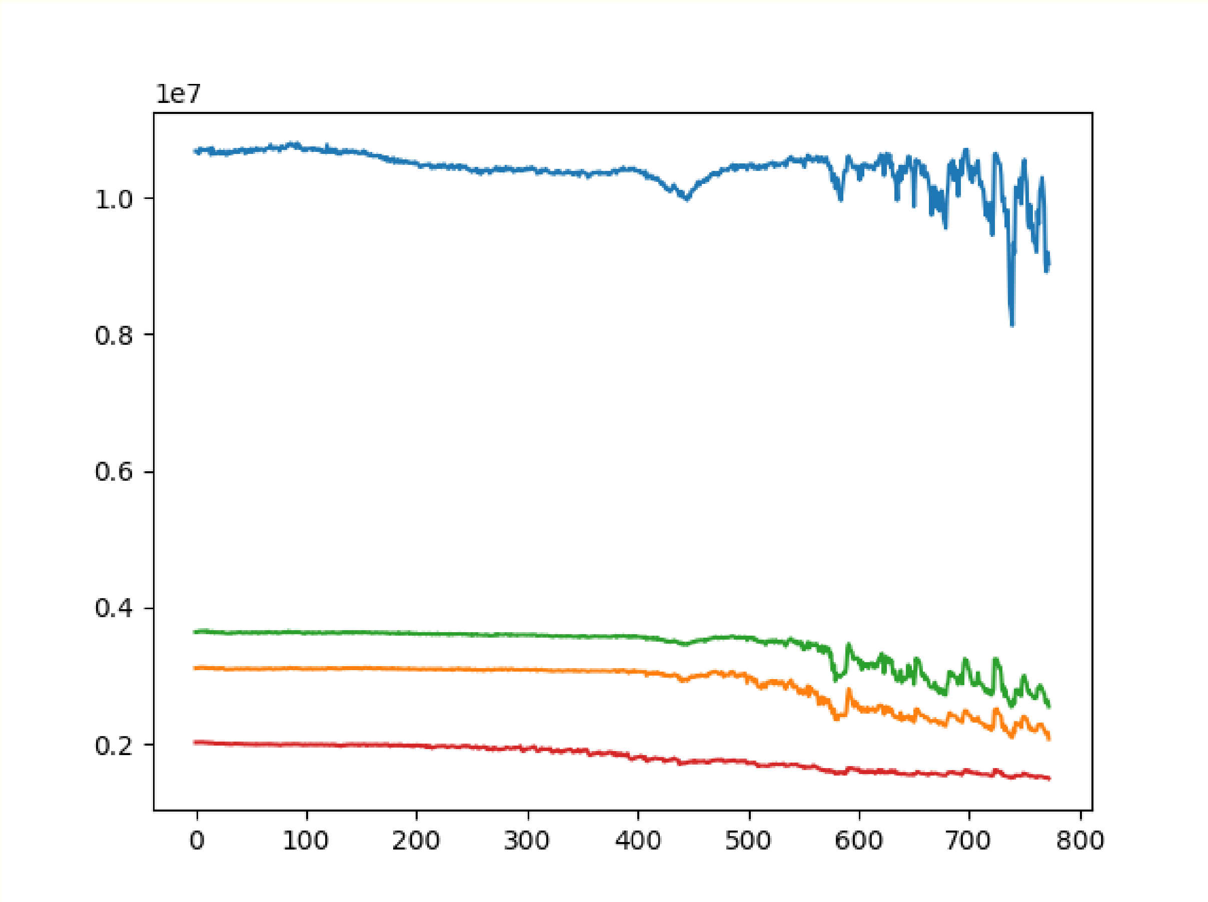
Let's try again:

Let's try again:

```
In [1]: %run photometry2.py -d work
Step 0 done
Step 1 done
...
Step 771 done
Step 772 done

In [2]: stars=np.array(STAR_FLUXES).transpose()

In [3]: plt.plot(stars[0])

In [4]: plt.plot(stars[1])

In [5]: plt.plot(stars[2])

In [6]: plt.plot(stars[3])
```

Let's try again:

```
In [1]: %run photometry2.py -d work
Step 0 done
Step 1 done
...
Step 771 done
Step 772 done

In [2]: stars=np.array(STAR_FLUXES).transpose()

In [3]: plt.plot(stars[0])

In [4]: plt.plot(stars[1])

In [5]: plt.plot(stars[2])

In [6]: plt.plot(stars[3])
```



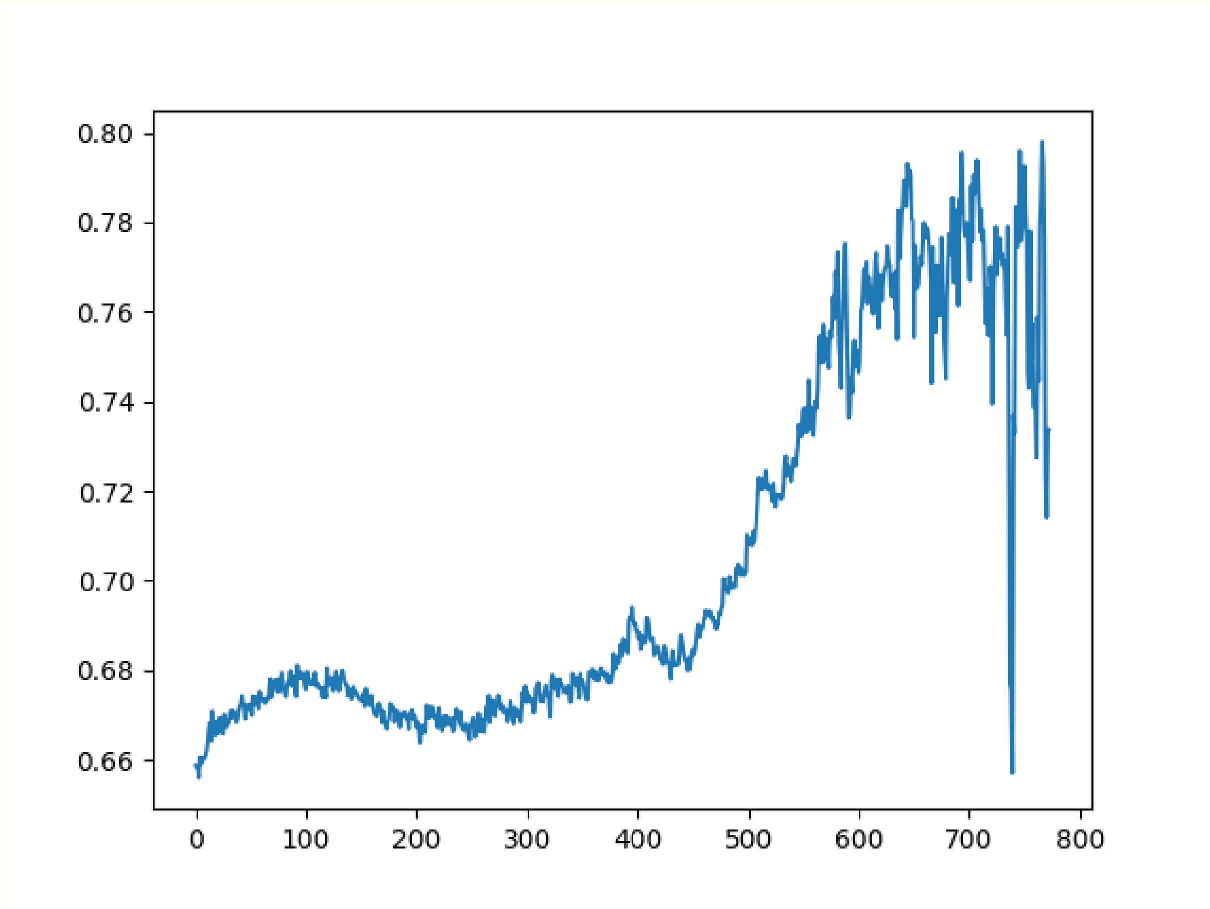
And the new photometry curve:

And the new photometry curve:

```
In [7]: plt.plot(DIFF_FLUX)
```

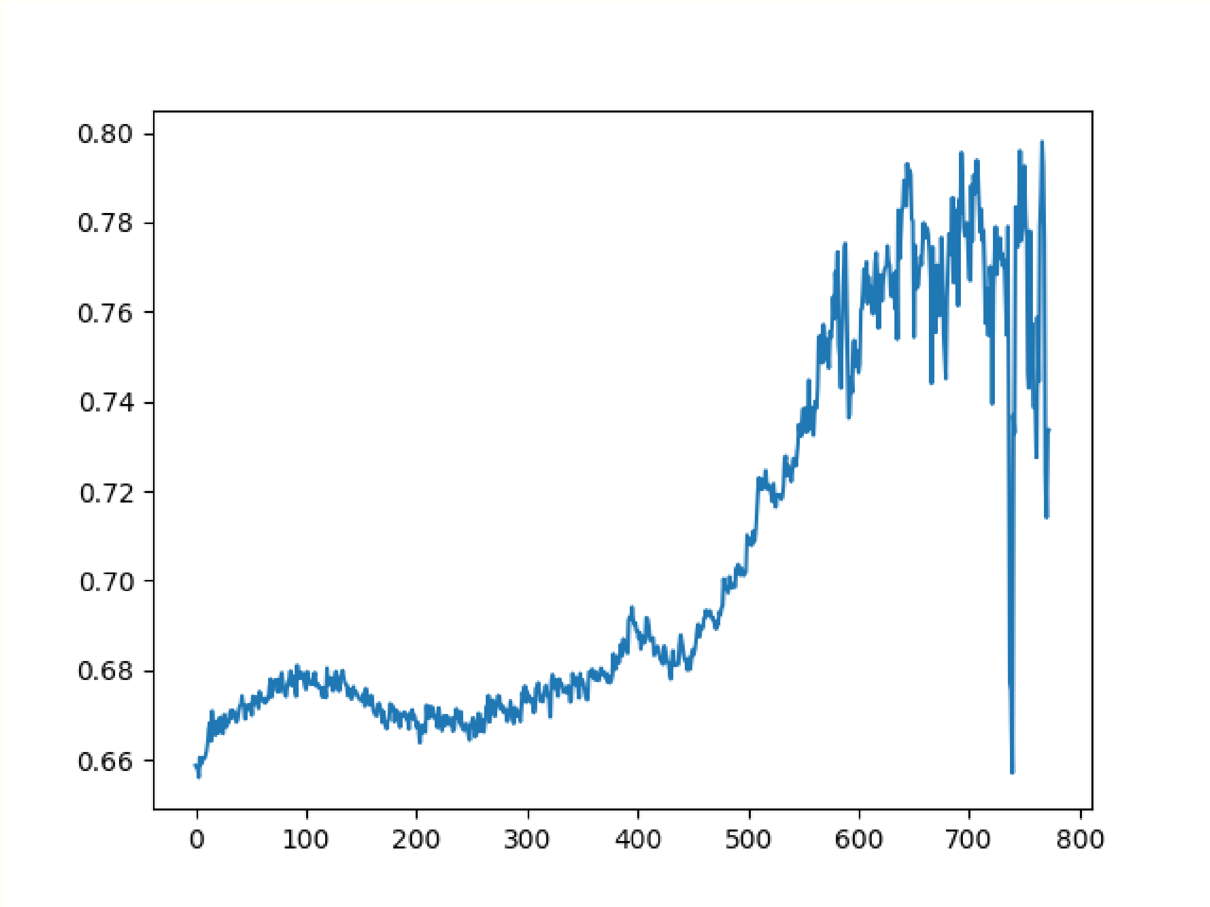
And the new photometry curve:

```
In [7]: plt.plot(DIFF_FLUX)
```



And the new photometry curve:

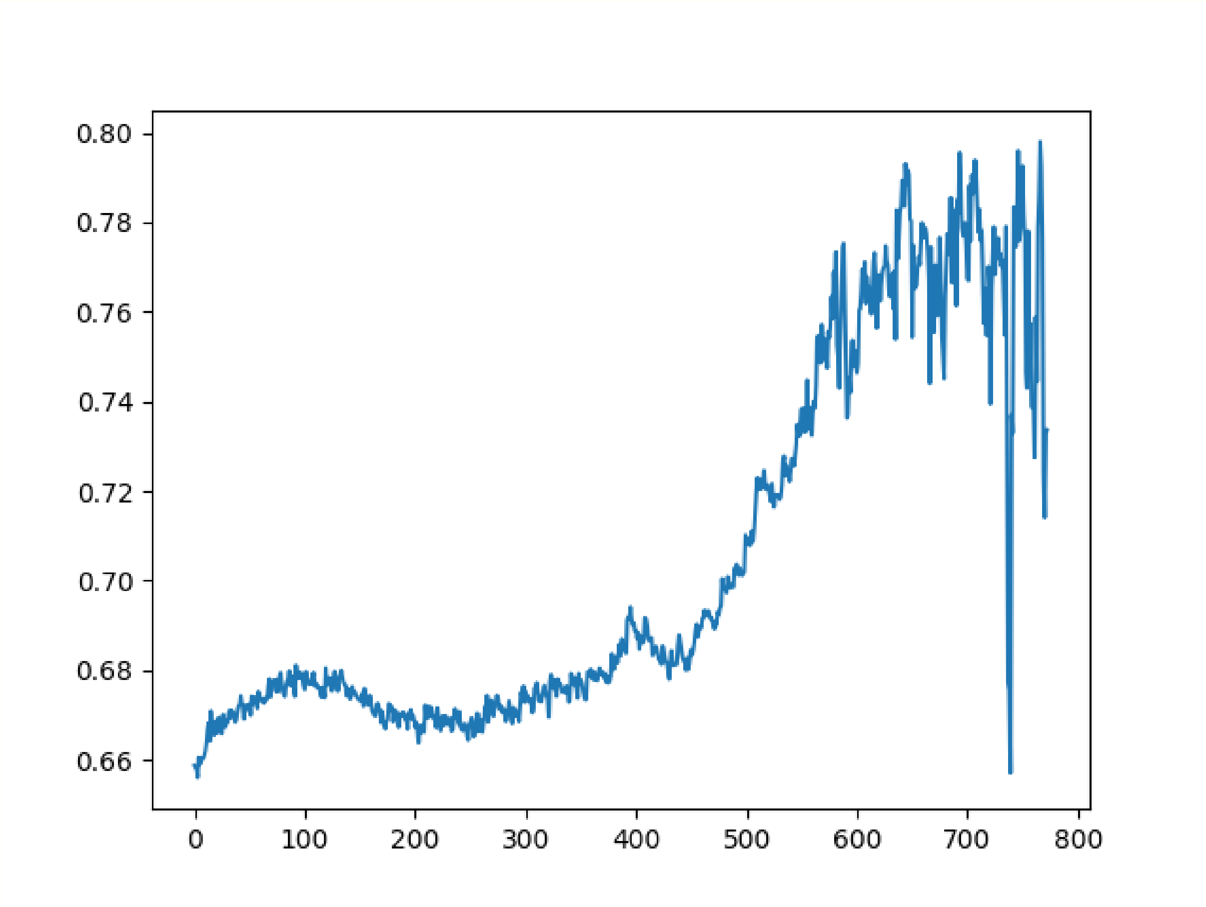
```
In [7]: plt.plot(DIFF_FLUX)
```



And clearly we re not there, yet ...

And the new photometry curve:

```
In [7]: plt.plot(DIFF_FLUX)
```



And clearly we re not there, yet ...

→

What is next?

- Fix the recenter algorithm

What is next?

- Fix the recenter algorithm
- Improve photometry:

What is next?

- Fix the recenter algorithm
- Improve photometry:
  - Evaluate and subtract local background

What is next?

- Fix the recenter algorithm
- Improve photometry:
  - Evaluate and subtract local background
  - Use adaptive area size to compute flux

What is next?

- Fix the recenter algorithm
- Improve photometry:
  - Evaluate and subtract local background
  - Use adaptive area size to compute flux
  - Increase the number of reference stars

What is next?

- Fix the recenter algorithm
- Improve photometry:
  - Evaluate and subtract local background
  - Use adaptive area size to compute flux
  - Increase the number of reference stars
  - Remove variable stars (from catalogs ...)

What is next?

- Fix the recenter algorithm
- Improve photometry:
  - Evaluate and subtract local background
  - Use adaptive area size to compute flux
  - Increase the number of reference stars
  - Remove variable stars (from catalogs ...)
- Find occultation start and end times by fitting proper curves

Doing photometry -9

What is next?

- Fix the recenter algorithm
- Improve photometry:
  - Evaluate and subtract local background
  - Use adaptive area size to compute flux
  - Increase the number of reference stars
  - Remove variable stars (from catalogs ...)
- Find occultation start and end times by fitting proper curves
- Study documentation of package **photutils** to know how complex can be star photometry in real work

What is next?

- Fix the recenter algorithm
- Improve photometry:
  - Evaluate and subtract local background
  - Use adaptive area size to compute flux
  - Increase the number of reference stars
  - Remove variable stars (from catalogs ...)
- Find occultation start and end times by fitting proper curves
- Study documentation of package **photutils** to know how complex can be star photometry in real work

Another approach to recentering

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center of the target star in the first and last images.

```
In [1]: from myselect import ObjectSelector

In [2]: img1=np.load("work/img-001.cal")

In [3]: img2=np.load("work/img-773.cal")

In [4]: sel = ObjectSelector(img1)

In [5]: sel.start()

In [6]: CIRC0=sel.circles[0]

In [7]: sel = ObjectSelector(img2)

In [8]: sel.start()

In [9]: CIRC1=sel.circles[0]

In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0], 773)

In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1], 773)

In [12]: SHX.dump("shx.dat")

In [13]: SHY.dump("shy.dat")
```

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center first and last images.

Note: I had to rename file `select.py` to `myselect.py` due to a name clash with a function named "select" from the package NumPy

```
In [1]: from myselect import ObjectSel

In [2]: img1=np.load("work/img-001.cal")

In [3]: img2=np.load("work/img-773.cal")

In [4]: sel = ObjectSelector(img1)

In [5]: sel.start()

In [6]: CIRC0=sel.circles[0]

In [7]: sel = ObjectSelector(img2)

In [8]: sel.start()

In [9]: CIRC1=sel.circles[0]

In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0], 773)

In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1], 773)

In [12]: SHX.dump("shx.dat")

In [13]: SHY.dump("shy.dat")
```

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center first and last images.

Note: I had to rename file `select.py` to `myselect.py` due to a name clash with a function named "select" from the package NumPy

```
In [1]: from myselect import ObjectSel

In [2]: img1=np.load("work/img-001.cal")

In [3]: img2=np.load("work/img-773.cal")

In [4]: sel = ObjectSelector(img1)

In [5]: sel.start()

In [6]: CIRC0=sel.circles[0]

In [7]: sel = ObjectSelector(img2)

In [8]: sel.start()

In [9]: CIRC1=sel.circles[0]

In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0], 773)

In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1], 773)

In [12]: SHX.dump("shx.dat")

In [13]: SHY.dump("shy.dat")
```

Load first and last images

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center first and last images.

Note: I had to rename file `select.py` to `myselect.py` due to a name clash with a function named "select" from the package NumPy

Use `ObjectSelector` to find the center of the target star from the first image

```
In [1]: from myselect import ObjectSel

In [2]: img1=np.load("work/img-001.cal")

In [3]: img2=np.load("work/img-773.cal")

In [4]: sel = ObjectSelector(img1)

In [5]: sel.start()

In [6]: CIRC0=sel.circles[0]

In [7]: sel = ObjectSelector(img2)

In [8]: sel.start()

In [9]: CIRC1=sel.circles[0]

In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0], 773)

In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1], 773)

In [12]: SHX.dump("shx.dat")

In [13]: SHY.dump("shy.dat")
```

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center [...] first and last images.

```
In [1]: from myselect import ObjectSel

In [2]: img1=np.load("work/img-001.cal")

In [3]: img2=np.load("work/img-773.cal")

In [4]: sel = ObjectSelector(img1)

In [5]: sel.start()

In [6]: CIRC0=sel.circles[0]

In [7]: sel = ObjectSelector(img2)

In [8]: sel.start()

In [9]: CIRC1=sel.circles[0]

In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0], 773)

In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1], 773)

In [12]: SHX.dump("shx.dat")

In [13]: SHY.dump("shy.dat")
```

Note: I had to rename file `select.py` to `myselect.py` due to a name clash with a function named "select" from the package NumPy

Use `ObjectSelector` to find the center of the target star from the first image

On exit from the interactive loop, `sel.circles` holds the coordinates of the selected center

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center first and last images.

Note: I had to rename file `select.py` to `myselect.py` due to a name clash with a function named "select" from the package NumPy

```
In [1]: from myselect import ObjectSel

In [2]: img1=np.load("work/img-001.cal")

In [3]: img2=np.load("work/img-773.cal")

In [4]: sel = ObjectSelector(img1)

In [5]: sel.start()

In [6]: CIRC0=sel.circles[0]

In [7]: sel = ObjectSelector(img2)

In [8]: sel.start()

In [9]: CIRC1=sel.circles[0]

In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0], 773)

In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1], 773)

In [12]: SHX.dump("shx.dat")

In [13]: SHY.dump("shy.dat")
```

Use `ObjectSelector` to find the center of the target star from the first image

On exit from the interactive loop, `sel.circles` holds the coordinates of the selected center

... then do the same for the last image of the set

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center first and last images.

```
In [1]: from myselect import ObjectSel

In [2]: img1=np.load("work/img-001.cal")

In [3]: img2=np.load("work/img-773.cal")

In [4]: sel = ObjectSelector(img1)

In [5]: sel.start()

In [6]: CIRC0=sel.circles[0]

In [7]: sel = ObjectSelector(img2)

In [8]: sel.start()

In [9]: CIRC1=sel.circles[0]

In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0]

In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1]

In [12]: SHX.dump("shx.dat")

In [13]: SHY.dump("shy.dat")
```

Note: I had to rename file `select.py` to `myselect.py` due to a name clash with a function named "select" from the package NumPy

Use `ObjectSelector` to find the center of the target star from the first image

On exit from the interactive loop, `sel.circles` holds the coordinates of the selected center

... then do the same for the last image of the set

Generate a sequence of positions interpolating a straight line between the two points

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center first and last images.

```
In [1]: from myselect import ObjectSel

In [2]: img1=np.load("work/img-001.cal")

In [3]: img2=np.load("work/img-773.cal")

In [4]: sel = ObjectSelector(img1)

In [5]: sel.start()

In [6]: CIRC0=sel.circles[0]

In [7]: sel = ObjectSelector(img2)

In [8]: sel.start()

In [9]: CIRC1=sel.circles[0]

In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0]

In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1]

In [12]: SHX.dump("shx.dat")

In [13]: SHY.dump("shy.dat")
```

Note: I had to rename file `select.py` to `myselect.py` due to a name clash with a function named "select" from the package NumPy

Use `ObjectSelector` to find the center of the target star from the first image

On exit from the interactive loop, `sel.circles` holds the coordinates of the selected center

... then do the same for the last image of the set

Generate a sequence of positions interpolating a straight line between the two points

And save the x, y sequences

Another approach to recentering

I use the `ObjectSelector` defined earlier to find the best possible center first and last images.

```
In [1]: from myselect import ObjectSel
In [2]: img1=np.load("work/img-001.cal")
In [3]: img2=np.load("work/img-773.cal")
In [4]: sel = ObjectSelector(img1)
In [5]: sel.start()
In [6]: CIRC0=sel.circles[0]
In [7]: sel = ObjectSelector(img2)
In [8]: sel.start()
In [9]: CIRC1=sel.circles[0]
In [10]: SHX=np.linspace(CIRC0[0],CIRC1[0]
In [11]: SHY=np.linspace(CIRC0[1],CIRC1[1]
In [12]: SHX.dump("shx.dat")
In [13]: SHY.dump("shy.dat")
```

Note: I had to rename file `select.py` to `myselect.py` due to a name clash with a function named "select" from the package NumPy

Use `ObjectSelector` to find the center of the target star from the first image

On exit from the interactive loop, `sel.circles` holds the coordinates of the selected center

... then do the same for the last image of the set

Generate a sequence of positions interpolating a straight line between the two points

And save the x, y sequences