LYMM - a program for photometric analysis of CCD images

John Saxton **20 March 2007**

latest version of software: 2.4 date: February 2010

Introduction

LYMM is a command line program for the photometric analysis of CCD images. It was originally written by JS for images from his MX916 camera, but has now been modified to work with other (and larger) image arrays, although the data must still be 16 bit.

LYMM is especially suitable for **time series photometry** of **faint** and/or **moving** objects.

Since it is a command line program with no graphics whatsoever, you will have to use it in conjunction with an image display program, for example AIP4WIN. You use the image display program to obtain the approximate coordinates of the stars of interest, which are fed to LYMM. Much of the output from LYMM is in comma-separated-variable format files, which can be easily imported into a spreadsheet.

LYMM can be run either by typing commands into the keyboard, or by reading script files which contain lists of commands. The script files can be created with a simple text editor such as Notepad. The actual command syntax is the same in either case. This may seem a complicated way of doing things, but script files do have the advantage of providing a record of how the analysis was carried out.

LYMM enables you to generate an output file which lists the detected signals (counts) for up to 10 stars in each image of interest. Once this file is created, I anticipate you will import it into a spreadsheet for subsequent analysis; my program will not do extinction corrections, nor will it do transformations between magnitude systems.

Installation

You can see the program working by downloading **lymm_demo.zip**, unzipping it and putting the contents in c:\Lymm_demo. Double click the program to run it.

If you want to use the program on your own data, it is more convenient to put the program in one folder and have your data in other folders. You might as well leave **Lymm.exe** in the Lymm_demo folder; if you do move it, note that the folder containing **Lymm.exe** must also contain **path.dat**, **device.ccd** and any other **something.ccd** files. **Path.dat** contains the current path to your data, and also the file naming/numbering convention; it is read on program startup.

Data (image) files, script files and program output can go in another directory, of your choosing.

After doing this, the program may - or may not! - read your fits files. I discovered whilst testing with files from other cameras that the standard fits format isn't completely standard after all! So you should now check the camera setup file. This file is to be found in the same folder as **Lymm.exe** and has a name of the form **something.ccd**; the default file which is loaded on program startup is always **device.ccd**. In the course of developing this software, I found it convenient have different camera setup files. You can tell the program to switch between them with some of the commands described later under 'Commands - General Program Control' (I doubt you will need to do this very often).

Here is an example of the file **device.ccd**:

```
20.0 20.0   X PIXEL SIZE, Y PIXEL SIZE, MICRONS
5           GAIN (ELECTRONS / ADU)
32          INPUT CONTROL VARIABLE
1           DATE FORMAT; 1=YMD; 0=DMY
```

The program must know the correct shape of the pixels otherwise it will not be able to perform operations involving rotation. Strictly speaking, the photometry routines should do, too, otherwise circular apertures will not be exactly circular; they don't presently do this, but this could be added fairly easily if required.

The number of electrons per ADU is required for *ab initio* calculations of the error when doing either aperture or optimal photometry.

The input control variable is intended to take care of the different ways of storing numbers in fits files. The program reads two bytes for each pixel, with the most significant byte first. Some processing is then required to obtain the actual pixel value. Think of the processing as consisting of seven possible steps. The individual bits in the input control variable define which steps are to be carried out, as follows:

Bit value		if bit is set, do (or note) this:
1	bit(1)	MSB is sign
2	bit(2)	MSB is sign *only if* BZERO is non-zero
4	bit(3)	invert the MSB *only if* BZERO is non-zero
8	bit(4)	not specified by user
16	bit(5)	add on BZERO
32	bit(6)	subtract BZERO
64	bit(7)	invert number (e.g. change to 128 to –128)

The processes are carried out in the reverse order when saving an image to file. The value of the input control variable is found by adding up the bit values for all the component bits which are set. For example: suppose the integer is signed (MSB is sign), we have to add on BZERO, and we must delete the MSB before interpreting the byte as a number. In this case, bits 1, 4 and 5 are

set, so the input control variable = 1 + 8 + 16 = 25. For SX MX916 files the input control variable is 0; for Roger Pickard's Hale Camera files it is 32. For Richard Miles' files it is 4. If you want to experiment reading files with different input control variables, you can change the input control variable from within the program with the **icv** command.

If you do find yourself fiddling with the input control variable so that the program will read your files, I strongly suggest you save a file and read it back again to ensure you have the same numbers as you started with — I haven't tested all the possibilities so there may be bugs lurking in this section of the code!

Header Information

LYMM never changes the header information. Therefore when you save an image to file, the header information comes from a previous file. For a series of dark/flat corrected images, the header for each correcte file is copied from the corresponding raw data file. If you stack a series of images and then save them, the header information will come from one of the stacked images (the last one if using the **stack** command; the first useable one if using **rstack**).

Image Time

The frame time is obtained in various ways, depending on what is present in the fits header. If header lines labelled 'UT-START' and 'UT-END' are present, the program takes the average of the times from these two lines (as with Roger's Hale camera files). Failing that, it takes the time from the line labelled 'TIME-OBS'. If none of these lines are present, it assumes the time is to be found after the date on the line lablled 'DATE-OBS'. At present there is no feature to allow for an offset between the time recorded in the file and the time of mid exposure, if one exists; this could be added if required.

When doing photometry of image(s), the time written to the final output file is not derived directly from the header of the processed image, but comes from the image alignment file. This allows time corrections to be made if you want to override the computer's clock; see further comments below.

LYMM was written with time-series photometry in mind, and so tries to keep the time counting at midnight rather than resetting to zero (this is done by a few lines of code in the alignment routines).

A BUG! the time keeps on counting at midnight, but the date rolls over, which is perhaps not very logical!

Image Date

Finally, in trying to read different fits files, I encountered problems with the date being either day/month/year or year/month/day. If it gets it wrong, try changing this line!

Coordinate Systems

The program has three ways of referring to positions on the ccd pixel grid. Since the following notes refer to the program notation, I had better define this before we go any further:

The pixels themselves are referred to by *integer* numbers, which have the indices **(i,j)**.

For star positions, we obviously need to refer to fractions of a pixel; for this we use the coordinates **(u,v)**. ***All positions you feed to the program are (u,v) coordinates,*** although you do not have to specify the decimal place(s) every time. The centres of the pixels are at whole values of u,v so the pixel values are at half values (e.g. centre of pixel i=45, j=33 is at u=45.0,v=33.0).

Unless the pixels are exactly square, one unit in u is not the same size as one unit in v. For some purposes — basically anything involving rotation or Pythagoras — the program has to convert to (x,y) coordinates. This is done as follows:

$$x = u$$

$$y = v \times \text{pixel height} / \text{pixel width}.$$

Where required this is done by the program and you don't need to worry about it. The two main exceptions will be rectified in due course, but are of no significance provided your pixels are roughly square (my pixels on the MX916 are only 10% off square). These concern the 'radius' of the sky patches which are used when measuring the sky background at specified positions, and the outer 'radius' beyond which the PSF is forced to zero in the PSF fitting/optimal extraction calculations.

The Commands - ordered (more or less) as you would do an analysis

The commands are case-insensitive; spaces are ignored. Although the full commands are written here - otherwise they would be rather unintelligible - the program only takes note of the first ten characters (so you can save yourself some typing if you remember this!).

General Program Control

path - set working directory

filetype - set file types

init - initialise image arrays

script - execute script file

log - open/close log file

stop = **exit** = **x** = quit program

verb - set verbosity (default 0)

dv - reload device.ccd, *and* print out how the image bytes will be processed

camera - set new default camera setup file; program copies specified file to device.ccd and then loads the new device.ccd

device - display parameters for the current camera

You should put all files for a given series of observations — e.g. an evening's run on a single star — into one directory, and set the **path** accordingly (it doesn't matter whether or not you type the final \ at the end of the path). When you start the program, it remembers the path from last time.

The **verb** (verbosity) command controls how much information the program puts on the screen as it runs; the larger the number - up to 6 - the more information. This can sometimes be useful if you suspect a problem or bug with the data processing. If you set the verbosity to –1, then the program will print out each line of the fits header (and then wait for you to press return) as it reads each file.

Note that you can put comments in the script files. Comments lines shall begin with the / or ; characters. You can usually put a comment after a command line, too, for example:

```
stack              ; stack the images
img100             ; use images 100 to 107
…
img107
```

In this way you can provide a 'running commentary'. Used in this way, the ; has to be the sixth character or later. I say you can *usually* do this, since although I modified the software to reject the ; and subsequent characters, a few commands might have slipped through the net.

Some commands have default inputs which, in manual mode, you can select by just pressing return. Do not forget these inputs when creating your script files – you can just leave a blank line.

A convenient way to develop script files is to have two windows open: one for the program and the other for notepad. You can try out the commands in manual mode, and as they are found to work, they can be added to the script file in Notepad.

Image File Names

File names (including the file type) have a maximum of 20 characters; the complete name including path must not exceed 80 characters.

You can have different file types for different types of fits images, if you want; *e.g.* raw files can be something.fit, dark whilst dark/flat corrected files are something.fts. You can also change one of the first three characters of the filename for corrected files, too, if you wish. You can set these with the **filetype** command. All my images started life as, for example, img12345.fit and I call the corrected files imc12345.fit. You don't normally need to specify the file type when typing in the file name.

If the program fails dismally on asking to process a series of images, then make sure you have specified the file naming convention correctly.

Very often we will want to process a series of images. This is easy if the filenames are of form letter(s)-numbers, or number(s)-letter(s)-number, and the final number increments by one from one image to the next. To specify all the files from, say, img12340 to img12345, we could type them all in, but the program will also accept the following three lines

```
img12340
…
img12345
```

(note the three dots) which involves less typing! This entry format is understood by all the commands which require a list of input files (e.g. **dfs**, **stack**, **photometry**). If you want to use, say, files from img12340 to img12360, but *not* img12351 and img12352, you would enter

```
img12340
…
img12350
img12353
…
img12360
```

In fact, you can make things even easier: you can use the **files** command to specify a list of files once and subsequent commands will use this list, without having to specify it again. For example:

```
files             ; specify the files once
img12340
…
img12345

/ various commands could go here

stack             ; stack the files we specified earlier
…
```

Note also that the file numbering can be img1, img2 .... img9, img10 etc, or img001, img002, img003... You change this with the filetype command (you essentially specify the number of digits - with leading zeros if necessary - or 0 to have leading zeros suppressed).

WARNING: Be careful not to overwrite your original images when generating dark/flat corrected images! Since the program automatically renames files when doing this, I recommend you keep a backup of original images, just in case something goes wrong (don't say I didn't warn you…) The image save routine will (hopefully!) warn you if one of the following occurs:

The raw file type (set with the **file type** command) matches either of the other two *and*

the first three characters of file name are **img** or **IMG**, *or*
you did not tell it to change any of the first three characters

I would recommend you keep a backup of your raw data before doing any processing – to be on the safe side. It is a fact that software can (i.e. does) have bugs and people can (i.e. do) make mistakes!

Image Processing

load	load image
save	save image
avfiles	average images, no offset or alignment
clear	clear image arrays
avpix	compute average pixel value
dfc	dark and flat correct series of images
dfs	dark and flat correct series of images, and find stars
stack	stack images with offsets (NB you must do image alignment first)
rstack	stack images with offsets and rotation (NB you must do image alignment first)
reject	image rejection parameter

multiply multiply image by factor

dark subtract dark frame

list list pixels in 11x11 grid centred on specified position. This is the closest the program comes to actually letting you 'see' the image!

The first stage of image processing involves generating average dark and flat files, which you do with the commands listed above. (It is also possible to do this with other software packages, of course). The next stage involves dark and flat correcting the images. For this, use the **dfs** command (both **dfs** and **dfc** require dark corrected flat files). This does a simple star search on each frame, at the same time as the dark/flat correction. All the star positions - once batch for each image processed - are written to a file called **stars.lst** - you will need this file for what comes next (**stars.lst** is an example of what I refer to as a 'position list file' - it contains a list of star positions and the program needs to refer to it for various purposes). One thing to watch here is that with the default threshold of 100, the program sometimes finds too many stars - it can only cope with 400 per frame. So I suggest that before you process all the images, you load just the first image, type **findstars** (see below) and see how many stars you get. Too many, and you can change the threshold (**threshold** command - see below). When you're happy, you can let it process all the files. This may take a few minutes. The positions found by the star searching routine, which are written to file, are actually centroids.

The star finding process also returns an estimate of the full-width-half-maximum (fwhm) of the star image, given as an integer number of pixels. Strictly, 'width' refers to the i (or u or x) coordinate (so it will only be 'width' in RA if you have an equatorial mount and the camera is mounted on the telescope appropriately). This parameter is quite useful, since you can use it to reject badly trailed images from further processing. To do this you should set the image rejection parameter (the **reject** command) to the largest fwhm you are prepared to accept: I usually use 3 pixels (this is the default value). Then, when other commands such as **stack** or **photometry** are executed, files with fwhm *greater than* 3 pixels will be ignored, *up to and including* 3 pixels will be processed.

The program can also find positions by PSF fitting if you want it to, but this is quite a bit more complicated, and comes later on. There is evidence that for faint objects, PSF fitting gives more accurate positions.

Star Finding and Image Alignment

setbox defines box in which program will look for stars. Normally, you don't need to worry about this - the default is to within 5 pixels of the edges of the frame. The sky value for star finding is also based on the pixels within this box.

threshold threshold for star finding (above approx median pixel)

proximity if program finds two stars whose separation is less than this value, the brighter one is kept and the fainter one rejected

loadalign load image alignment file (usually done automatically)

align carry out image alignment, no rotation

alignr carry out image alignment, with rotation

fitrot fit a curve to the field rotation data

fiducialstars set fiducial stars

positionlistfile or just **plf** set file which contains list of star positions if default is not OK

findstars findstars in *one* image (the currently loaded image). The star positions are written to **stars1.lst (**if done automatically, as part of the dfs command, the star positions end up in **stars.lst**)

tolerance tolerance (horizontal) pixels, for position matching (e.g. image alignment, refining star positions)

If your telescope tracking was perfect, each star would always fall at the same position on the CCD. This doesn't happen in practice, of course, and we need to sort this problem out before we can get anywhere with doing photometry (or any other process which involves a series of images).

We take one frame (typically the first), which shall be the reference frame. The coordinates of the stars are those on the ccd pixel grid in this frame. Owing to imperfect tracking, the same stars will have different coordinates - on the ccd pixel grid - in the other frames. We need to determine the offsets, and usually rotations, for these other frames.

To do this, we pick several stars to act as *fiducial stars*, in the first frame. Their coordinates are put in the file **stars.fid**. At this point, your image viewing software is going to have to give you the approximate (u,v) coordinates of stars in the image. You can either get the fiducial star coordinates as described in the next section (star positions), or you do this: Copy **stars.lst** to **stars.fid**. Open up **stars.fid** with an editor, then delete everything except the stars in the first image, and then delete everything except the fiducial stars you want to keep. (It won't matter that **stars.fid** has other stuff after the first two columns, as long as *it does not have any header lines*).

Having created the file **stars.fid**, we next use either **align** or **alignr** to determine the offsets/rotations for the images. These commands use a pattern recognition algorithm to try to match the fiducial stars to the stars listed in stars.lst. If you want to determine image rotation, the

fiducial stars should preferably be widely spaced over the field of view (not clustered in a small area). All the fiducial stars must be matched for the pattern to be recognised. The tolerance for matching is set by the **tolerance** command (the default — 2 pixels — is probably OK and you don't need to worry about it). Double images - which may be caused, for example, by a glitch in the tracking - will normally be rejected. The results of this image alignment process are written to the file **align.dat**. The data written to this file include image file name, time and approximate fwhm as well as the offset and rotation in degrees. Note that the times written to this file are those which ultimately end up in the photometry output files.

The program also writes a file called **align.txt**, which contains alignment data in comma-separated-variable format, although only for the images which were aligned successfully. You can import this file into a spreadsheet should you wish to investigate your telescope's tracking in more detail.

It is worth using **alignr** and looking at the field rotation versus time, even if you have an equatorial mounting. If the polar alignment is not perfect, you may find that a plot of rotation versus time shows a clear trend, despite a fair amount of scatter between individual frames. The rotation varies smoothly with time, of course, and the scatter arises just because we can only determine the alignment for one frame to a certain degree of accuracy. If this is the case, it is better to fit a curve to describe the (smooth) rotation and re-determine du and dv, rather than trying to use the noisy rotation for image alignment purposes. You can fit a polynomial curve to the rotation data as follows:

After running **alignr**, type **fitrotation** (or just **fitrot**). The program asks you the order of the polynomial (1= straight line, 2=quadratic etc). It will then fit the curve (and write the coefficients of the fit to rotate.mdl). Now type **alignr** again, only this time use second option to take the rotation from a model rather than determining it from scratch. The old **align.txt** will first be copied to **alignold.txt**, and then the program will go through the images contained in stars.lst, and re-determine du and dv, having taken the rotation from the model. New files **align.dat** and **align.txt** will be created, containing du, dv and a smoothly varying rotation. If you import **align.txt** and **alignold.txt** into a spreadsheet, and do a bit of cutting and pasting, you will be able to see the polynomial fit plotted against the original data.

Now we have set up the image alignment array, life gets more interesting…

Stacking Images and Hot Pixel Removal

Once you have created the file **align.dat**, you can stack images. There are two commands to do this:

stack stacks images allowing for image offset but not rotation, and

rstack stacks images allowing for both offset and rotation.

In both cases the image de-shifted/de-rotated back to the original (fiducial frame) orientation. Stacking images is two to three times slower if you rotate the image as well as shift it (this is because the pixel is split in 25 [=5×5] subpixels before resampling onto the rotated pixel grid). However, this option is only likely to be necessary if you are using an altazimuth mounting.

Beware of an artefact which arises from stacking frames: if the telescope tracking is not perfect, then each frame may cover a slightly different area of sky. Therefore, when you stack them so that images of a single star from different frames are coincident with one another, any regions near the edge of a frame, which are not present in all frames, will effectively be 'underexposed' in the combined image (this should be apparent when you view the stacked image). This can cause problems for star finding (see below), since the star finding algorithm needs to know the sky background, which it estimates based on the median sky pixel; if there are large underexposed 'strips' along the edges of image, this value may be misleading. If this is a problem, then you will have to redefine the box in which the program searches for stars (**setbox** command). Stars positions can also be determined with the **psffiles** command; this uses a more restricted area of sky in the vicinity of each star of interest.

stack and **rstack** may also be used for purpose of obtaining the path of a moving object from groups of stacked frames. This special use is described later.

LYMM also provides a means of removing hot pixels from stacked images. The key to this method is that the stars *must* move with respect to the CCD chip during the series of exposures.

	clearhotpixels	clear hot pixel array
	loadhotpixels	setup hot pixel array, requires dark frame and threshold; pixels more than threshold away from median value (above or below) are flagged as hot pixels

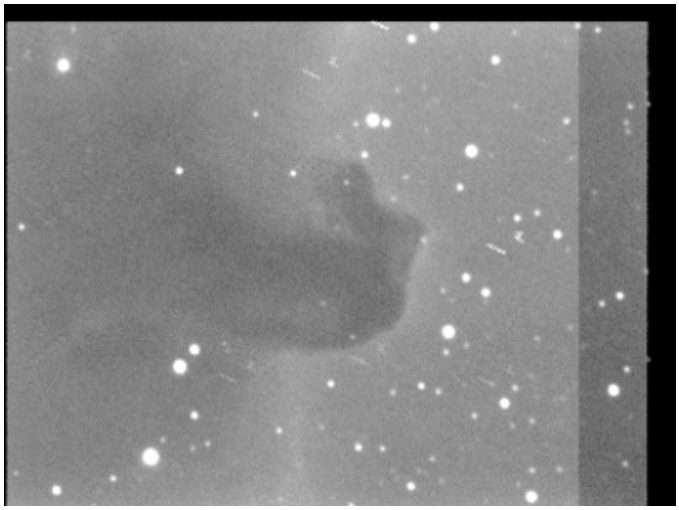
When the images are stacked, the program (1) forms a sum for each pixel, *excluding any bad pixels*, (2) records a number of good pixels used to form the sum and (3) records the number of frames used. You now have two methods of normalising the sum (1):

	meanpixel	
or	**meanpixels**	divide by sum for each pixel [(1) divided by (2)]
	mainframe	
or	**meanframes**	divide by number of frames [(1) divided by (3)]

On starting the program, the default is **meanpixels**.

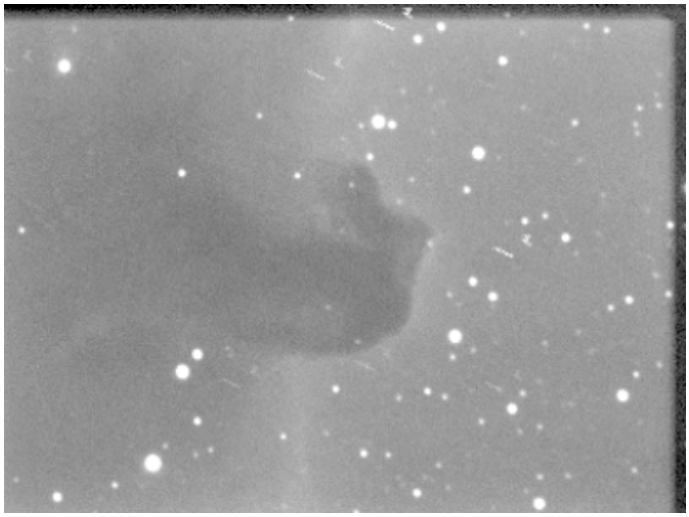
Here are some examples showing how the different options work.

Case 1: **clearhotpixels** (no hot pixels defined) and **meanframes**



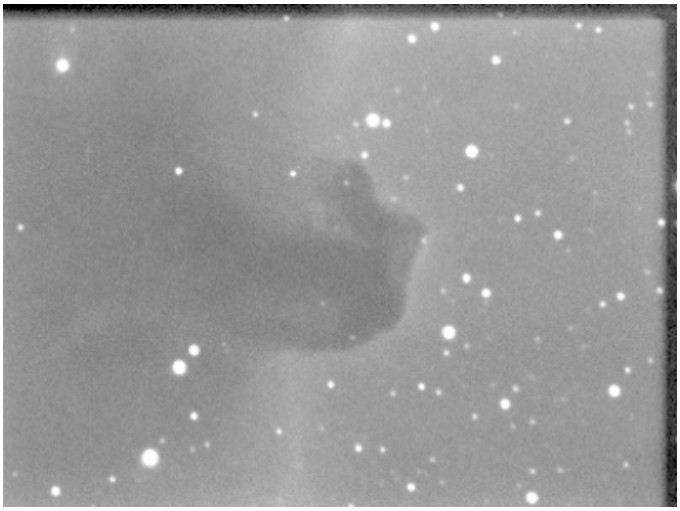
Hot pixels are visible as short trails. Also, since we normalised the image stack with **meanframes**, areas of sky which did not always fall on the CCD chip, owing to image drift, come out fainter after normalisation (right and top edges of image).

Case 2: **clearhotpixels** (no hot pixels defined) and **meanpixels**



The hot pixels are still visible, but use of **meanframes** has removed much of the problem associated with areas of sky which drift off the CCD chip during the exposure. Some residual effects are still visible, for two reasons. First, the noise in these areas of sky will be greater, since they are the average of fewer pixels. Second, if the sky brightness changes during the series of exposures, the mean level formed from an incomplete series of frames will be incorrect (*i.e.* different to the main body of the image).

Case 3: **loadhotpixels** (defined from dark frame and suitable threshold) and **meanpixels**



The hot pixels have now disappeared, and the image looks much cleaner.

Star Positions

OK, I admit it, this is where it would be nice if my program had a graphical interface, so you could select stars of interest just by pointing and clicking with a mouse. But it doesn't, so you'll have to follow the instructions given here. I have, however, tried to keep things as simple as possible.

First of all, the program can work with up to 100 'stars of interest' - these may be fiducial stars, comparison stars, target stars or anything else. Each star has a full name (20 characters), a short name (up to 5 characters, although this will usually be just a, b, c, v etc) and two sets of (u,v) coordinates – one set for the approximate positions, one set for the refined positions:

	full name	**short name**	**u (initial, or approximate)**	**v (initial or approximate)**	**u (refined)**	**v (refined)**
1	up-to-20-chars	a	105	77	104.78	76.34
2	GSC-whatever-1	b	44	144	43.77	143.44
3	GSC-whatever-2	upto5	322	210	322.04	210.43
4	XYZ	chars	211.5	125.2	211.23	124.92
…						
…						
100						

All this information is held in a star data table, and whenever this is updated, it is written a file called **stardata.txt** - which is a simple ASCII file, so you can also edit it with a text editor such as Notepad, if necessary. (In stardata.txt, for each object: the text variables are on successive lines, and all the numerical quantities on the following line). The idea is that you start by specifying the full and short names, and approximate coordinates (nearest whole pixel). The positions then go through several stages of refinement before we are ready to do the photometry. (There is a lot more to getting accurate positions than point and click with the mouse, anyway, so the lack of this feature is perhaps not such a great drawback after all!).

Commands concerned with star positions are:

clear positions	clear (reset) star data table
positions	add basic star data
loadpositions	load star data from file
update positions or just **up**	refine positions; see below
updatemulti or just **um**	refine positions; see below
reset positions	clear refined, but not initial positions
copy positions	copy refined positions to initial positions
start positions	copy initial positions to refined positions

Let's suppose you want to get star positions for photometry, starting from scratch. First clear the star data array, and then type **positions**. The program asks for the long and short names, and approximate positions, for the stars. I envisage that you will have both my program and the viewing program open simultaneously, and you find the approximate star coordinates from the viewing program, and then type the number straight into my program (without the need for a pencil and paper - the ability to have several programs open in Windows at the same time is useful!). When you've finished doing this, the star data is saved to file automatically. If you want to load the star data — for example after restarting the program — type **loadpositions**.

Now we have to 'refine' the positions. There are different ways of doing this - in any case, you will usually have to go through more than one stage of refinement in order to obtain good positions for photometry. Basically, the program looks up a position list file (*e.g.* **stars.lst**), finds one or more occurrences of each star of interest, and uses these to obtain updated positions. Here are some examples:

use centroids derived from a single image

This is the simplest way to improve the positions after you've typed in the approximate values. Load the image which defines the fiducial frame. Type **findstars**; the program will search for stars in this one image and write centroid positions to **stars1.lst**. The baseline used for star finding is the approximately the median pixel.

Now type **update positions**. The program will hunt through **stars1.lst**; when it finds a star close to one of the initial (or approximate) positions which we typed in, it uses its coordinates to update the position. 'Close to' means within the distance specified by the **tolerance** command; the default is 2 horizontal pixels. After this, file **stardata.txt** is updated; this includes short and long names, and approximate and refined positions.

If you wanted, you could now use the **fiducialstars** command to define the fiducial stars and write them to **stars.fid**. You could then generate the image alignment table with **align** or **alignr**.

use centroids from a series of images

Better positions will be obtained if we use the information contained in a series of images. Before we can do this, the following conditions must be met::

- we must have generated the image alignment table (with **align** or **alignr**), otherwise the program will not be able to recognise a particular star in a series of images
- the program must know the approximate positions, either through typing them in, or loading them from stardata.txt with **loadpositions**
- the program must know the file containing the list of star positions — **stars.lst** in this case; this file is generated during the dark/flat correction procedure and is also used to generate the image alignment table. The program will remember the name of the most recent position list file to be generated (since the program was started); if you are not sure, or want to change it, use **positionlistfile** (or **plf**) **command**. The default upon startup is **stars.lst**. The files **stars1.lst**, **stars1.psf** and **stars.psf** may also be used — but more about these later.

Now type **updatemulti** or just **um**. The program will read through stars.lst, and de-shift/de-rotate each star; it then attempts to match it with the approximate positions and update the positions, in the same way as before.

use PSFs from a series of images

You can use the **psffiles** command to generate a file **stars.psf**, which contains positions in the same format as stars.lst. You can also this to refine the positions, in just the same way as with stars.lst (described above). The **psffiles** command is described more fully later.

use centroids from a stacked image

First you should generate the stacked image. Then type **findstars**; the centroids of stars found will be written to **stars1.lst**. You can use this to refine to the positions as described under 'centroids from a single image'. Be aware that the findstars routine uses a value based on the median pixel to get the sky background level and this may be inaccurate if there is poor overlap between the stacked images (have a look at the stacked image; the background value used is written to **stars1.lst**). If this is a problem you may wish to reduce the size of the box in which the program searches for stars (**setbox** command). Alternatively, use PSFs:

use PSFs from a stacked image

First generate the stacked image; then obtain the PSF positions with **psffiles**. These will be written to **stars1.psf**. It is convenient to use **psffiles** with the automatic sky subtraction option; this uses a square patch of sky centred on each target star. The default sky box size is 50 pixels on a side; you should make sure this is not clipped by the edge of any of the frames (have a look at the stacked image).

For faint stars it may be preferable to use one good stacked image than a large number of noisy single images. You should always try ensure that the fiducial stars are fairly bright. For very faint stars, PSF positions may be preferable to centroid ones, partly because the centroid calculations may have been done with an inaccurate sky background value. The centroid sky background is actually the median of about 400 pixels scattered uniformly over the whole image (why about 400? Because in the MS-DOS version of this program I couldn't define a much larger array for the median calculations owing to memory limitations). A sky background derived in this way may be inaccurate due to either poor overlap between stacked images and/or gradients across the image (although you can reduce the region over both which stars are sought and the sky estimated with the **setbox** command). You can think of **findstars** as quick and dirty but doing all the stars in the image, whilst **psffiles** is more sophisticated but only works on the stars you specify.

Ideally, you should include the fiducial stars in the stars of interest, obtain the best possible positions for them, update **stars.fid**, and then repeat the whole alignment and position determination processes. The best possible positions require some iteration.

It is worth mentioning that good positions are important for precise photometry and are vital if the star is faint — since in the latter case you must use a small aperture to minimise sky noise. Furthermore, for time series photometry of faint objects, the positions *must* be obtained from the information in *all* the images and *not* determined individually for each frame. Since the noise is non-negligible compared to the star, the positions determined in each frame will be biased by noise in the vicinity of the star, so that the photometry based on these individually determined positions will systematically overestimate the brightness of the object.

summary of position list files and update position commands

command	*reads positionlist file*	*number of images in file*	*number which are searched for positions*	*positions are from*
up	`stars.lst`	one or more	the first one	centroids
up	`stars1.lst`	one	one (the only one)	centroids
um	`stars.lst`	one or more	all	centroids
up	`stars.psf`	one or more	the first one	PSFs
up	`stars1.psf`	one	one (the only one)	PSFs
um	`stars.psf`	one or more	all	PSFs

So you see - there is a lot more to position determination than just clicking with the mouse!

Handy hint: If the updateposition commands don't seem to work properly, have a look at **stars.lst** with a text editor and check that the entries for the first file look sensible. I have sometimes noticed that the first file doesn't contain any stars – no doubt an error on my part, in defining the file names (maybe the first was a dark frame by mistake!).

Sky Subtraction

An estimate of the sky level is carried out automatically using a box centred on the target position. Commands concerned with sky subtraction are:

skybox size of sky box used in automatic sky subtraction (default is 50 pixels on a side, which should usually be satisfactory). You can have any value between 21 and 99.

skymedian sky value will be median of pixels in box

skyclipped sky value will be mean of clipped pixel distribution (the method used by AIP4WIN). The pixels are sorted, the bottom and top 20% rejected, and the mean taken of the remaining 60% of pixels.

Photometry and Point Spread Functions (PSFs)

At last — the interesting bit! Commands concerned with photometry are:

log	open/close log file
soft aperture or **soft**	select aperture photometry with soft aperture
hard aperture or **hard**	select aperture photometry with hard aperture
hardaverage or **hardmean**	return mean value of pixels in aperture (see below)
optimal extraction or **optimal**	select optimal photometry
romax	set outer radius for PSF fitting (PSF forced to zero beyond this; should be <5.0)
rphot	radius of photometry aperture (pixels, default 2.0)
output sky	sky backgrounds as the second batch of 10 columns in output file (default)
output error	errors as the second batch of 10 columns in output file
psffiles	determine PSFs and positions (shape for first star, positions for all stars); must do this before doing optimal photometry
psffilesall	determine PSFs and positions (shape and position for all stars)
psfsearchbox	defines size of box (centred on expected object position) which will be searched when fitting PSF
photometry	do the photometry (aperture or optimal)

Remember to open a log file before doing the photometry or else the results will not be saved to file! This output file will always contain intensities, not delta-mag. Note that you have the choice of having *either* 10 star values and 10 sky values, *or* 10 star values and 10 error values, in the output file.

LYMM can, as a subsequent step, compute delta-mag (the **dmag** command) and do statistical analysis of the results, but this section of the program is not very user friendly. I expect most users will import the import the intensities into a spreadsheet and proceed from there.
It will be convenient to deal with doing photometry in three sections:

Aperture Photometry

You can three types of aperture:

Soft aperture This is the most useful type of aperture. Imagine a circle drawn on the CCD, centred upon the expected position of the object. The unit of measurement is the horizontal (*x* or *u*) pixel dimension and the aperture will be circular even if the pixels are not square.
The pixels are weighted according to the fraction of their area within this circle: pixels wholly within the circle are given a weight 1, pixels wholly outside are given a weight of zero (i.e. ignored) and pixels which straddle the boundary are given a weight between 0 and 1. The program reports the sum of the pixels.

It is found that a soft aperture works well in practice, although for faint objects you have to choose the aperture radius carefully.

Hard aperture With a hard aperture, the pixels are either inside or outside the aperture, depending on whether the centre of the pixel lies within the circle. There are no fractional pixels. As a result the total number of pixels used depends upon the exact position of the object relative to the pixel grid, and this reduces the quality of the photometry. For this reason I suggest a hard aperture is not of much use in practice; it is provided here as an option solely for comparison with other photometry methods. As with the soft aperture the program reports the sum of the pixels.

Hard mean this is like the hard aperture but returns the mean of the pixels. I provided it only as a means of testing how well the sky background subtraction is working.

So, before you can use the **photometry** command, you must do all of the following:

- Use **rphot** to specify the aperture radius
- Set the sky background subtraction parameters (or use the default values)
- Declare the aperture type

You must also, of course,

- have accurate star positions (if necessary load these with **load positions**)
- have carried out the image alignment procedures.
- specify the maximum permissible integer fwhm with the **reject** command (or use the default of 3).

Now you use the **photometry** command. The program will then prompt you to enter the short names for the stars for which you want to do photometry — just press <return> when you've finished the list — and then go off and actually do the photometry.

Point Spread Function (PSF) Determination

LYMM can estimate PSFs and thereby obtain object positions. The facility is useful for two reasons:

- PSF positions are often more accurate the centroid ones – especially for faint objects
- The optimal extraction algorithm (see below) requires an estimate of the PSF

PSFs are estimated with the **psf files** command, and you should use this before you attempt optimal photometry. **psf files** works in a rather similar way to the **photometry** command, in as much as it requires decent star positions, good sky subtraction and an alignment table (**align.dat**) to be set up. Note that for all PSF/optimal extraction operations, an 11x11 pixel section of the image is copied to the relevant subroutines, so the star image and PSF must fit comfortably on an 11x11 pixel grid. The PSF is estimated by fitting a 2-D Gaussian to the object; it needs to know the radius beyond which the PSF can be forced to zero; this is set with the **romax** command and the default value is 3.5 pixels.

The **psf files** command will prompt you for the short names of the stars of interest (up to 10 of them) in just the same way as the **photometry** command. The program will then proceed to estimate the PSF (both shape and position) for the first star in the list, and determine the positions — from fitting this PSF — for the remainder. At the end of this procedure, the program will have created a file **stars.psf** (if you are processing more than one image) or **stars1.psf** (for one image), which contains both star positions and PSF fit parameters. This file is of similar format to **stars.lst** (but it also contains psf data, of course) and can also be used for updating positions and aligning images. The PSF shape is described by two parameters, which correspond to a sort of average radius and a parameter describing the elongation of the image. Any elongation must be aligned east-west or (north- south); the program was written with images from an equatorially mounted telescope in mind, images from which may easily be slightly elongated east-west due to imperfect tracking.

Alternatively, you can use **psffilesall** command, which fits both shape and position for all the stars in the list, but is, of course, slower.

psfsearchbox sets the size of the box to be searched when fitting the PSF. The program goes to the expected postion of the object and starts by looking for the brightest pixel within the box (careful with definition here!!!).

Note that both optimal extraction and aperture photometry require that all stars in one image should have the same PSF. Due to field rotation, this condition will not be met with a telescope on an altazimuth mounting, unless the exposure is very short. Ideally the PSF star should be quite bright — so you can see the PSF clearly without its being contaminated by noise — but don't pick a double star!

Optimal extraction does not require an *exact* PSF, which is just as well, since the real PSFs are unlikely to match any mathematically simple shape. My program tries to fit a circularly

symmetric 2-D Gaussian, and then adjusts the ellipticity (so it's no longer symmetric) to try and improve the fit. The code is rather slow and crude, but – more importantly – seems to be adequate and works! With the sort of tracking you get with an amateur telescope, the PSF is unlikely to be a perfect Gaussian anyway, so there seems no point getting too bogged down with this issue. As regards object positions, experience has shown that PSF positions are often more accurate than centroid ones, especially for faint objects.

Optimal Photometry

Having got this far, optimal photometry is now easy! We just declare **optimal extraction** (as opposed to aperture photometry) and then use the **photometry** command. This command prompts you for the short names of the stars, and creates an output file, in just the same way as before — the one extra thing you have to remember is that the weight mask will be optimised for the *first* star in the list.

Note also that although the PSF is the same for all stars in the image, the weight mask depends on the brightness of the star, so it has to be optimised for stars of a particular brightness; you cannot have one weight mask for all the stars in the image. If the total error in the final flux ratio (*i.e.* delta-magnitude) is dominated by one star, we simply set the weight mask for that star. In practice, one should set the weight mask for the faintest star of interest (usually the target).

Moving Objects

LYMM works well with moving objects; it will even work with objects so faint that several frames have to be stacked in order to obtain a good position. As for fixed objects, we do the photometry using positions obtained from information in all the frames. For stationary objects this means an average position from all the frames (or positions from a stacked image); for moving objects we fit a curve to the object's position versus time. This gives a set of smoothly varying positions, unaffected by noise on individual frames.

To start with, we shall consider the case of an object bright enough that reliable positions can be obtained from individual frames. Objects so faint as to require stacking of images are more tricky and are dealt with in the next section **Very Faint Moving Objects**.

Commands concerned with moving objects are:

motion on	switches motion on (affects operation of some previously defined commands)
motion off	switches motion off (affects operation of some previously defined commands)
estimate path or **ep**	estimate a path model from start and end positions
fit path or **fit track**	fits model to object positions in path.dat
testpath	allows you to check path model; you enter t, it tells you position
loadpath	loads path model from file
showpath	lists the path model
resetresid	reset the residuals table
residuals or **res**	list the residuals
rejectresiduals or **rr**	reject images which have large residuals
multisky	carry out photometry at asteroid positions, but when the asteroid is somewhere else

Suppose we have an asteroid on which we wish to carry out photometry. We must first define a model which describes its path. One method would be to use the **updatemulti** or **um** command, as follows.

Type in the approximate (initial) positions for the stars and the asteroid in the *first* (fiducial) frame. Use **updatepositions** to refine the positions based on this first frame. Next, with object motion switched **off**, type **updatemulti** (or **um**). This command gives you various options: you want to write the positions for the asteroid to file and you should enable free tracking for this object (the asteroid). In the normal case (without free tracking), the program looks for objects in **stars.lst** near the expected position of the object; with free tracking, it looks for objects near the previous position – and so keeps up with the object as it moves. These positions are written to the file objectpath.txt. At the end of this procedure, the refined position for the asteroid will be the mean position along the path, which is of no practical use, but this does not matter. We can then fit a model path to the positions in objectpath.txt and use this. (The program converts all coordinates back to the fiducial frame prior to trying to identify any objects, of course).

The problem with this approach is that the program may loose of the asteroid if it passes near a star, or if it becomes very faint.

A more robust method is to set up an approximate linear model for the motion and use this to obtain positions. We do this with the **estimate path** or **ep** command. This prompts you for the asteroid positions in specified frames (usually the first and last) – you enter the *observed* positions in the *actual* frames; the program does the coordinate conversion to the fiducial reference frame for you.

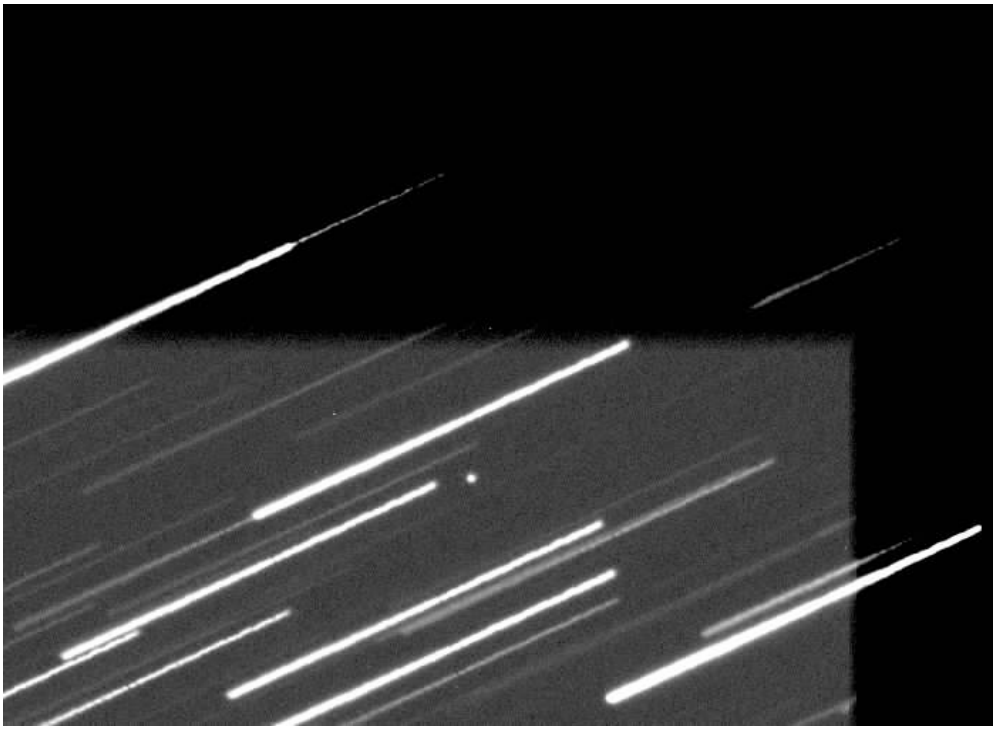
Now go back to **updatemulti**, but this time with the motion switched **on,** as well as having **free tracking enabled.** Since object motion is switched on, the program will now use the model to calculate the asteroid positions (models will be described in more detail shortly). The asteroid positions are again written to file, and we can fit a model to them.
We still have the problem of what happens if the asteroid passes near a star, when the positions will be unreliable. LYMM provides a means of rejecting these unreliable positions. At the same time as it writes the position for the asteroid to objectpath.txt, it also keeps a list of the residuals (deviation of the observed position from the expected position). **resetresid** will reset the residuals array; whilst **residuals** (or just **res**) will list the residuals so you can examine them. The command reject **resid** (or just **rr**) will 'reject' images for which the residual exceeds a specified value. The frame is actually 'rejected' by setting its u,v offsets to 9999,9999 in the image alignment table, which effectively flags it as unusable. The only way to reinstate it is to recreate the alignment table with the **align** or **alignr** commands.

At this point we have reliable set of positions and want a mathematical model to describe them. To fit a model, type **fit path** (or **fit track**). This fits two polynomials to describe both u and v as functions of time. Unless the object is a slow mover, you should not assume that the motion is linear. Try both linear and quadratic (or higher) models and examine the residuals.

Having got a model, and possibly rejected some frames in which the asteroid passes near stars, you can of course iterate by re-determining the positions and re-fitting the model, should you wish.

With the motion model defined, switching **motion on** will affect the operation of other commands is useful ways:

stack and **rstack** will now stack the frames so that the asteroid is a dot and the stars are trailed (I think **rstack** works but I have not had an opportunity to properly test it). Here's an example of a set of images – courtesy of Dr Richard Miles – stacked so that the asteroid 2005WC1 appears as a point:



photometry will apply the motion model to the *first* object (note that optimal photometry is not suitable for moving objects).

psffiles will apply the motion model to the *second* object (the first is still used to obtain the PSF shape).

The PSF positions for the asteroid may, of course, be used to try to define a better model for the motion.

advanced sky subtraction methods for asteroids

Asteroid photometry suffers from the problem that sometimes the asteroid passes near a field star, which then contaminates the lightcurve. LYMM provides a powerful method of identifying – and, sometimes, correcting – this problem, by using the **multisky** command.

Before using the **multisky** command, you should set up a normal soft aperture photometry routine for the asteroid – i.e. specify image alignment table, path model, sky box size, aperture type (soft) and aperture size. The **multisky** algorithm is based on the usual photometry routines, but with an important difference: it carries out photometry at the asteroid positions when the asteroid is somewhere else! 'Somewhere else' means more than a certain number of pixels away – LYMM asks you how many. You can also request it to process only images which are close to the asteroid in time; if you want to use all the images, just enter a large number (of hours) here. LYMM then asks you which images to process, which you specify in just the same way as for the **photometry** command.

multisky then proceeds as follows:

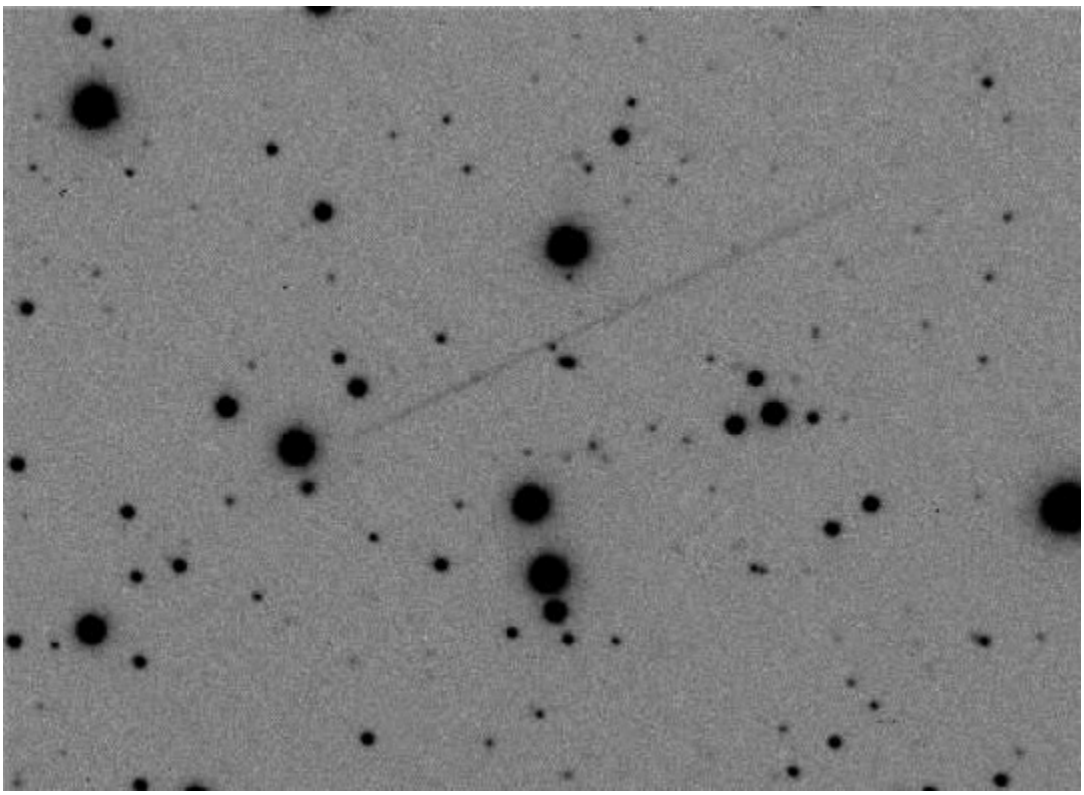
- load image i
- check that this image is suitable for photometry – alignment data available and integer fwhm satisfactory. If it is not, proceed to image i+1
- note the time, t_i, and work out the position (u_i,v_i) for the asteroid in this frame
- with this frame loaded, go through all the entries in the alignment array (j=1,2,3…). For each image j, note the time t_j and work out the asteroid position u_j, v_j. If (u_j,v_j) is closer to (u_i,v_i) than a certain limit, ignore this time t_j and go to entry j+1 in the alignment table
- otherwise, perform photometry at position (u_j,v_j) in frame i. The photometry should be carried out in just the same way as for the asteroid. This gives a sky_j which, ideally, should be zero, if the sky subtraction procedures work perfectly. The results are stored in a table:

j	**t_j**	**Σ sky_j, sum over i**	**number of values contributing to Σ sky_j**
1	23.455	large number	*155*
2	23.555	large number	*155*
3	23.655	large number	*155*
4	23.755	large number	*increment by 1 each time*
	etc	etc	*we increment sum sky_j*
			*should be **nearly** the total*
up to 1000	*taken from alignment array*		*number of frames in the series*

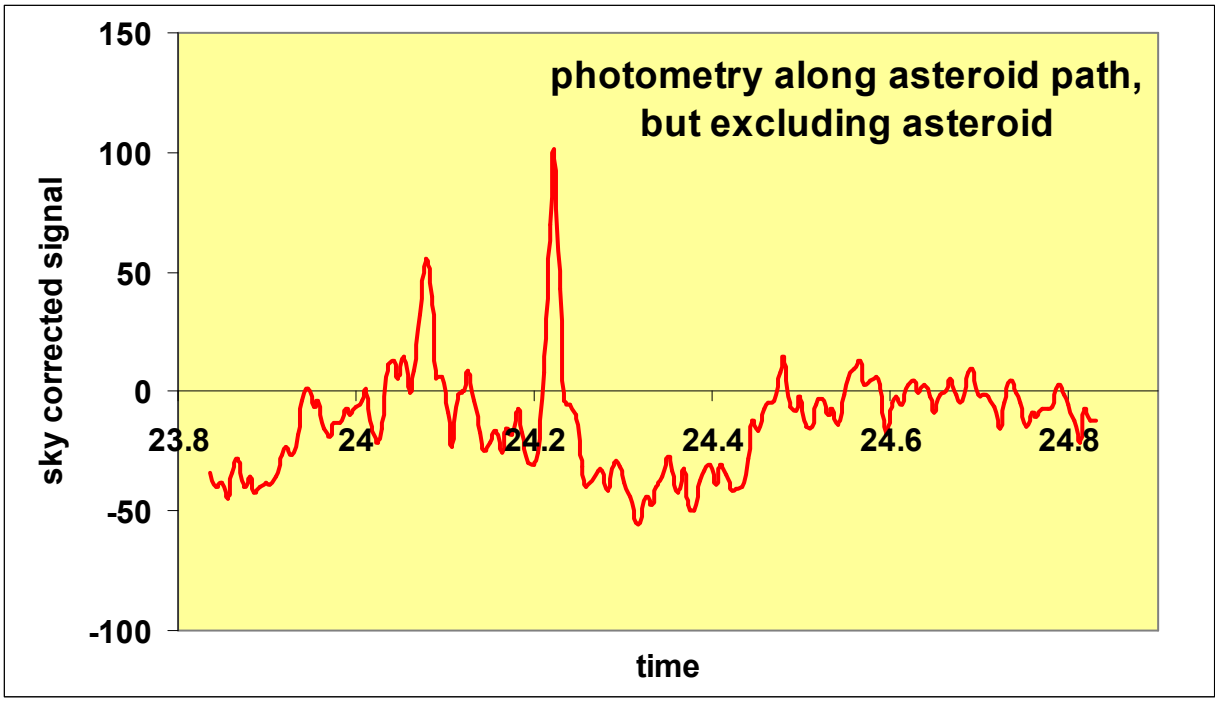
- proceed to next image i+1

At the end we just divide the large number in column 3 by that in column 4 to get an estimate of the average 'sky' value at position j. The results of this are written to file `sky-spots.txt` which contains three columns: time, average sky, and number of values averaged to get each sky value.

Here is an example dataset, showing a trailed image of asteroid 2005WC1:



And here is the result of carrying out the **multisky** procedure on the same data:



The length of the red trace corresponds to the length of the asteroid's path in the image. This x-axis is given as 'time', but only as an indicator of *position along the track*, since in this example each point is derived from photometry of almost all the frames in the series.

Contaminating field stars show up very clearly in this plot! For example, the positive spike at an x-value of 24.22 shows that at 24.22 hours the asteroid passes near a faint field star. It should be emphasised that this method of examining field star contamination is very sensitive; remember that, in the above image, the asteroid is image is trailed, whilst the stars images have been stacked on top of each other about 150 times. The photometry – both of the asteroid and with the **multisky** command –was done with a soft aperture of 4 pixels diameter. With this aperture, the asteroid signal was 2000 to 3000 units, so even the largest spike in the above figure represents only a 3 to 5 % correction.

In addition to the positive spikes, there are some broader negative dips. These are due to field stars contaminating the box used for sky determination.

multisky works on the just same images used for photometry, and in the same order, so you should be able to import sky-spots.txt into a spreadsheet and compare it directly with the asteroid lightcurve.

In principle, we could decontaminate the asteroid photometry simply by subtracting from it the red trace shown above, but this will only be an exact correction if certain conditions are met:

- The PSFs are the same for all the images. One could envisage the following – rather pessimistic – scenario: The seeing is usually very good and **multisky** tells you that there is no field star contamination, but just as the asteroid passes near a field star, the seeing becomes very poor. **multisky** would not see this image, and the corresponding contamination.

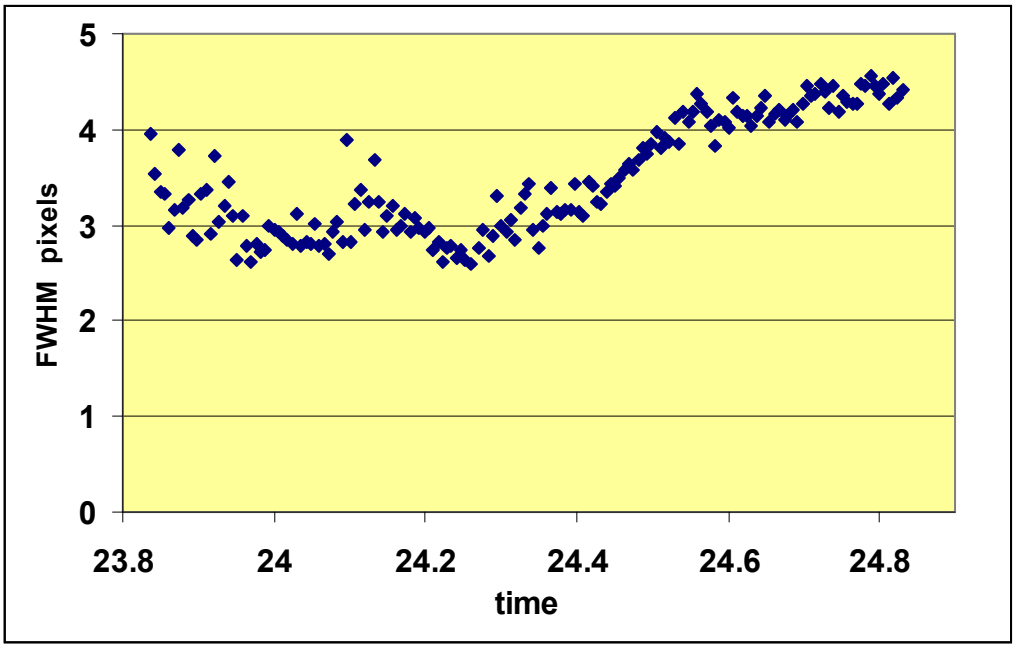
- The sky background does not change

- The transparency does not change

It is worth examining these in slightly more detail.

PSF changes with time:

Inspection of the asteroid trail in the above example shows that the image size does indeed change with time, and this is confirmed estimates of the PSF width, which may be obtained on suitable field stars with the **psffiles** command.



Possible reasons for the variations in image size include

- Seeing variations
- Drift in telescope focus
- Tracking errors (including wind)

On this particular night the image size tended to vary slowly with time, apart from some larger fluctuations near the beginning of the run. It is not really possible to distinguish between the different possible causes, other than remarking that a drift in focus might well be expected to be slow, whilst variations due to wind and tracking errors might be more erratic.

If the PSF is found to be varying slowly with time, then it would be appropriate to restrict the time span in the **multisky** command: photometry of the asteroid positions will then be carried out fairly close in time, and thus with a similar PSF, to the corresponding asteroid frames (we cannot use frames too close in time, of course, or else we would encounter the asteroid!).

Variations in sky background:

This is quite likely in practice – for example due to the Moon rising or setting during an observing run.

In itself, varying sky background is not a problem, since it should cancel out! Consider the ideal case where the sky box does not suffer contamination from field stars, and the target aperture is blank: the *sky corrected* signal from the target should be zero, regardless of the sky brightness.

Positive spikes are due to field stars, which are affected by transparency, not sky brightness.

The broader negative dips are also due to stars – stars which contaminate the box used for sky determination. The magnitude of these dips might be dependent on the sky brightness. The reason for this is rather subtle: the sky determination is usually done using a clipped mean, and as the sky brightness changes, the contaminating star-bearing pixels may move between the clipped and un-clipped parts of the pixel intensity distribution. More investigation is required here.

Nevertheless, if the sky background is changing slowly with time, the correction could be improved by reducing the time window in the **multisky** command; the correction would then be derived from images of similar sky brightness to asteroid frames.

Very faint moving objects – using positions from stacked frames

Principle

LYMM allows you to study objects so faint that several frames must be stacked in order to obtain a reliable position. The principle – if not the practice – of this method is straightforward.

Since we will only be stacking frames covering a short time interval, we do not need a precise description of the motion to stack the frames. So, set up a linear model which describes the motion. Now stack the frames in groups. The frames are stacked so as to remove both telescope tracking errors and also motion of the asteroid away from its starting position (the position in the first frame). Therefore, in each stacked frame, the asteroid will appear as a dot near the intial position. The average time and asteroid position for each group of stacked frames is written to a file.

If our initial model is in fact correct, the asteroid will stay exactly at the initial position. In practice, it will probably move slightly. Fit another model, to describe the motion of the asteroid in the stacked frames. Now add the coefficients for the two models together, *except that* we ignore the constant terms in the intial, linear, model. For example, if the u coordinate in the initial model is

$$u = 250 + 100\,(t - t_0),$$

where $(t - t_0)$ is the time (relative to the first position in `objectpath.txt`), and the positions from the stacked frames yield

$$u = 252 + 2\,(t - t_0) + 0.5\,(t - t_0)^2,$$

then the best description of the u coordinate with time would be

$$u = 252 + 102\,(t - t_0) + 0.5\,(t - t_0)^2.$$

In other words, we take the path defined by the stacked frames, and add on the motion which was

compensated for by the stacking process.

That's the principle. In practice, there are many small details to be taken care of!

Practice

Before we can stack the frames, we need the following:

- image alignment array – set up in the usual way.

- PSF information for the files. Use the **psffiles** command, described earlier, to estimate the PSFs for the images. Then load this information into the appropriate array with the **readpsffile** command. This will average the available PSFs for each file, if you fitted PSFs to more than one star. (The **psffiles** command will actually put the information into the array when it is executed, but on subsequent occasions, it is much quicker just to read the PSF data from a pre-existing file, rather run **psffiles** all over again).

- an approximate model for the motion – set this with the **ep** or **estimatepath** command. The **ep** command actually stores the model at two locations in memory – as model 1 and model 2. The reason for this will become apparent in due course. (Until we got involved with this positions-from-stacked-frames business, we only needed one model, and all the commands discussed earlier refer to model 1).

- switch motion on

- open the file objectpath.txt with the **openpathfile** command.

- define the number of frames to stack in each group, using the **maxstack** command.
- define frames to be processed – in the usual way.

We can now stack the images using **stack** or **rstack**. **rstack** is preferred, since although it is slower, it resamples the pixels, effectively allowing the frames to be shifted by a non-integer number of pixels. **stack** just shifts frames by the nearest integer number of pixels.

Having set the number of frames in each group with **maxstack**, the operation of **stack** and **rstack** is changed in an important way. The frames are stacked – until we reach the target number of frames. At this point, the automatic sky subtraction routine is called, followed by the psf position determination routine, to obtain the position of the asteroid. The asteroid is assumed to be near the initial position defined by the path model. The average time for the frames, and the position, is written to objectpath.txt. Then, the image and relevant image stacking variables are cleared, and the program proceeds to stack the next group of images. Whenever the target number of frames is reached, the process is repeated.

The stacked images are not saved, but they are no use for photometry anyway, since the field stars are trailed. In fact, if the time differences between frames are not constant, the stars

may be trailed by different amounts in different images. Remember that the **stack** and **rstack** commands will proceed through the list of files specified, and process each one, provided it has valid alignment information and the integer image width parameter is satisfactory.

To resume normal operation of **stack** and **rstack**, use the **resetmaxstack** command. This resets the target number of frames in a group to 9999 (the default value on starting the program). Since this value cannot be reached, the special procedure just described is not executed.

When we have finished stacking the frames, we must remember to

- close the file objectpath.txt with the **closepathfile** command.

Here is an example script (we assume the list of files has already been set):

```
loadalign
readpsffile
maxstack
5

ep
2005WC1F235
329 338
2005WC1F400
432 89

openpathfile
motion on
rstack
...


closepathfile
resetmaxstack
```

At this end of this script file, the object positions will have been written to `objectpath.txt`. You may wish to examine the contents of this file with Excel.

The next thing to do is to fit a model to the path in `objectpath.txt` – but there is a catch. If we are going to combine the motion models by adding the coefficients, then both models *must* have the same zero point in time. The approximate linear model defined by the **ep** command takes the time origin as that of the first frame specified. The **fitpath** command measures time relative to the first position in `objectpath.txt;` if we working with individual frames, this will indeed be the time of the first frame, but this will not be the case if we have stacked the frames into groups.

So, fit a path to the contents of `objectpath.txt` with the **fitstacked** command. This is the same as **fitpath**, except that it uses the same time origin as model 2 (model 2 was set with **ep**). Obviously, for this to work, you must have defined a model with **ep** first!

Like **fitpath, fitstacked** determines model 1 and stores this as `upath.mdl` and `vpath.mdl`.

Next we must combine model 1 (polynomial fit to positions from stacked images) and model 2 (approximate linear model). This is done simply by using the **addpath** command; the combined (best) model is stored as model 1, and also written to disc as `upath.mdl` and `vpath.mdl`.

It may be convenient to summarise the commands concerning the path models:

ep – used to set approximate linear motion model, which sets *both* model 1 and model 2.

fitpath –fit polynomial to contents of `objectpath.txt`. Time origin is first time in `objectpath.txt`. Store this as model 1 and write to `upath.mdl` and `vpath.mdl`.

fitstacked –fit polynomial to contents of `objectpath.txt`. Time origin is that for model 2. Store this as model 1 and write to `upath.mdl` and `vpath.mdl`.

savepath – saves model 1 to disc as `upath.mdl` and `vpath.mdl`.

loadpath – reads model 1 from disc.

copypath12 – copies model 1 to model 2 (does not save model 2).

savepath2 – saves model 2 to disc as as `upath2.mdl` and `vpath2.mdl`.

loadpath2 – reads model 2 from disc.

fitpath and **fitstacked** automatically save model 1. **ep** does not save a model to disc. No fitting routine automatically saves model 2 (you have to do this with **savepath2**).

Testing the positions-from-stacked-frames algorithms

The previous section may have seemed heavy going, but the procedures described therein are fairly easy to test. Here's how.

We require a dataset in which the asteroid positions can be determined from individual frames. Write the positions to `objectpath.txt`, fit a path model with **fitpath**. Copy this to model 2 (**copypath12**) and save it (**savepath2**).

Next, re-define a linear motion model (**ep**), stack the frames into groups, and determine another path model, as described in the previous section. This will end up as model 1. Take care to ensure that we have the same time origin as the fit to the single frame positions – this basically means that both **ep** commands use the first frame in the series.

Now use **loadpath2** to load the original (single-frames) model from disc. At this point model 1 and model 2 should describe the same path! We may compare them with the **diffpath** command.

I did this for quadratic fits to the 2005WC1 dataset mentioned earlier, comparing the path model obtained from single frames with that from frames stacked in groups of five. The deviation between the two models never exceeded 0.05 pixels.

Miscellaneous Commands

markobject puts cross hairs into an image to indicate an object.

markobject asks for four numbers and then a text string. The numbers are the (u,v) coordinates of the object and the (start, end) of the marks relative to the object. The four character text string contains any or all of the letters `lrud` in any order, depending on whether you want marks to the left, right, up or down from the object.

mask will mask out the outer portion of an image. All pixels outside the box (imin,imax,jmin,jmax) are set to zero.

constrast asks for the parameters *c0* and *scale*. It will linearly scale an image, so pixel value *v* is replaced by $[(v - c0) \times scale]$. I've used it for increasing the contrast in a set of images prior to making an animation.

setgain sets the camera gain (electrons per ADC unit)

noisefloor sets minimum value to the pixel fractional noise. For example: bright pixels might have a calculated error of 0.2%, but this command could be used to limit the error to a larger value. Was used for testing optimal photometry routines.

More commands are likely to added to this section in due course!

Appendix 1: Help! It doesn't work!

or

common mistakes which catch even me out

Alignment procedure won't work – doesn't recognise any stars. Have a look at stars.lst or stars.psf – in particular the first frame. Does it look sensible and contain stars? I've sometimes managed to have no stars in the first file – probably due to a mix up with file naming/numbering (e.g. the first frame was really a dark frame and not an image).

I told it to process a batch of files, but it can't open them. Are you in the correct folder/ directory? If so, is the file numbering system set up correctly? Suppose you recently processed some data which had file names ABC001, ABC002, ABC003…. and have now gone to another data set which have names like XYZ0001, XYZ0002, XYZ0003. The number of digits has changed and you must tell the program this with the **filetype** command.

Appendix 2: Software Testing

Any (good) programmer will tell you that the main problem with software such as Lymm is that, whilst it is good is the results look *sensible*, the real problem is making sure they are *exactly right*. The best way to check this is to use synthetic images, in which we know the positions and relative brightnesses of the stars. Of course, it is important that the generation of the synthetic images is carried out correctly, but from a computational point of view, putting known information into an image is vastly easier than the inverse problem of extracting the information from it.

LYMM includes several fairly low level commands to generate images containing test stars.

setall sets all pixels to the specified value

putstar put a star into the image (specified brightness, shape and position)

skylock lock values for sky and sky variance (these will be the 'override' values returned by the sky determination routine

skyfree resume normal sky determination, using the actual image

To make a test image, first use setall to set pixels to some specified value

Next, use skylock to force the sky determination routines to return this value (yes, you have to enter the sky value twice). Skylock also fixes the standard deviation of the sky which will be returned; this is because the optimal extraction routine needs a finite sky variance to work at all, not zero as is now the case with this image. Just enter a sensible value for the standard deviation.

The stars are specified by five parameters: u, v, height (peak intensity) and two parameters (r0,e0) which describe the width and ellipticity of a Gaussian PSF. Each pixel (within 10 pixels in u or v of the object position) is split into 25x25 subpixels, and the intensity evaluated at each subpixel. The value for the whole pixel is the average of the 625 subpixels. The value of each subpixel is given by routine intensity (height) for each subpixel is derived from a 2-D Gaussian by the routine pxclc:

```
subroutine pxclc(u0,v0,u,v,height,r0,e0,value)
ru=r0*e0
rv=r0/e0
du=u-u0
dv=v-v0
z = ( (du*du)/(ru*ru) ) + ( (dv*dv)/(rv*rv) )
value=height*exp(-z)
return
end
```

The flux from the star is added to that from the sky background, to determine the final pixel value.

Remember that these test images do not check the correct functioning of the low-level routines which read, write, decode and encode the FITS files. These have to be checked by examining individual pixel values. You should check this if the input control variable is set to a previously unused value, as discussed in the section on setting up the software.