Asteroid Data Hunter MM2: Description of Solution

Kushal Agarwal (TopCoder Handle: Kushal1)
Computer Science
California Institute of Technology
kushal@caltech.edu

September 19, 2014

1 Description of Algorithm

1.1 Source Extraction

The first step in the algorithm is completed by the `findSources()` function. For each pixel i, j in the image, let $g_{i,j}$, be a two dimensional Gaussian function centered on that pixel with standard deviation in each axis of 1.05 pixels. Then, linear regression can be used to find the values of `scaleFac` and `offset` for which the function ($g_{i,j} \cdot$ `scaleFac` + `offset`) best fits the values of the 5x5 grid of pixels centered on i, j.

This results in an `offset` value and a `scaleFac` value for each pixel. Noise in an image follows a Poisson Distribution, and the standard deviation of a Poisson process with mean λ is $\sqrt{\lambda}$. So the expected amount of noise for a specific pixel is proportional to $\sqrt{\texttt{offset}}$ (the proportionality constant depends on the gain of the image, but I have ignored this by assuming all image have the same gain). Therefore I can compute a signal to noise ratio as $\texttt{SNR} = \texttt{scaleFac}/\sqrt{\texttt{offset}}$ for each pixel.

On this new image where each pixel's value is based on its `SNR`, I take two different thresholds and then apply a flood-fill to individually process each contiguous white region. (Contiguous means connected along the x or y directions; diagonally connected is not enough. Taking a threshold of an image means setting pixels above the threshold to white and those below it to black).

This is first done using a low threshold. Then any regions that are either very non square ($|length - width| > 3$ pixels) or very large (area > 16 pixels), are kept white and the rest of the white pixels are made black. After this is complete, the pixels that are remaining white form a mask of points from which we will not make any detections.

Then I repeat the process of thresholding and flood-filling, this time with a higher threshold. This time I keep sources which are close to square ($|length - width| \leq 2$ pixels) and not too large (area ≤ 16 pixels). For each region that meets these criteria I create a detection centered at the pixel from the region that has the the largest `SNR` (as long as the mask at that pixel is black).

1.2 Subpixel Source Centering

The constructor for the `Detection` class next fine tunes the the position of these sources to subpixel accuracy. For each of the detections found above, I use Gauss-Newton minimization to minimize the sum of square error between a Gaussian function given by ($g_{y,x} \cdot$ `scaleFac` + `offset`) and the values of the 5x5 grid of pixels centered on the original detection. (Unlike the linear regression done to this same effect earlier, when using Gauss-Newton minimization I also allow x and y to vary and to do so continuously. Allowing x and y to vary break the linearity so non-linear tools must be used in place of linear regression). The result of Gauss-Newton minimization then yields four values describing the fit (x, y, `scaleFac`, and `offset`), two of which are the new more accurate center.

Finally, this coordinate, (x, y), (in pixel space) is transformed into an RA/Dec space, and then it is transformed into the pixel space of the fourth image. This way all detections from all four images are in the same coordinate space.

1.3 Removal of Stars

From the set of aligned detections generated above, the function `removeStars()` attempts to remove any detections that were created by stationary stars. This helps reduce the total number of detections, reducing the chance of false detections, especially in images with a high density of stars. A set of four detections are labeled as stars if a detection is found in each of the three latter images within 1 pixel of a detection in the first image.

Finally, this process gives me a good opportunity to do a sanity check by checking to see that at least some of the sources were stars. If less than 15% of detections were labeled stars and removed, that is an indication something has gone terribly wrong (the image has really bad artifacts or the WCS data is inaccurate), and so I do not process it to avoid the risk of accidentally seeing thousands of false detections. However, once mid-contest changes were made, I never saw this check ever taking effect.

1.4 Finding Asteroids

This is done by the `findAsteroids()` function. Let the images be taken at times $t_0, \ldots, t_3$. For each detection in the first image (call this detection d_0), I look for all detections within 5 pixels of that one, and for each one of those (call it d_1) I do the following:

The current estimate for the velocity is now

$$v = \frac{\text{position}(d_1) - \text{position}(d_0)}{t_1 - t_0}$$

So I expect d_2 to be at $d_1 + v(t_2 - t_1)$. For each detection within 1 pixel of this expected center, I repeat the process. The new velocity is

$$v = \frac{\text{position}(d_2) - \text{position}(d_0)}{t_2 - t_0}$$

And so the new expected location for d_3 will be $d_2 + v(t_3 - t_2)$. And again I do the following for each detection within 1 pixel of this expected center.

For each tuple of four detections (d_0, d_1, d_2, d_3) found in this manner, I will next check how linear its path is. First, the velocity during each of the three legs can be computed as

$$v_i = \frac{\text{position}(d_{i+1}) - \text{position}(d_i)}{t_{i+1} - t_i} \text{ for } i = 1..3.$$

Let `avgLen` $= (|v_0| + |v_1| + |v_2|)/3$. This tells us about how fast the object is moving. Then I compute the linearity of motion by first computing `error` $= (|v_0 - v_1| + |v_0 - v_2| + |v_1 - v_2|)/$`avgLen`. The numerator indicates how closely the three velocity vectors were pointing in the same direction, the denominator normalizes this for objects that are moving at different speeds. Finally, any tuple of detections with `error` < 2.5 and `avgLen` > 70 (meaning the path is fairly linear and the object is moving faster than 70 pixels per day), is called an asteroid.

1.5 Output Results

The function `getAnswer()` returns all asteroids that were found. Each is output twice because most ground truths are also duplicated, and detections are sorted according to the `error` parameter (described in the previous section) so that the more linear the path, the more confident I am in the detection.

Further, asteroids whose velocity lies outside some very rough bounds (velocity in dec direction is between -0.6 and 0.25 degrees/day, and velocity in the RA direction is between -0.5 and 0.5 degrees/day) are skipped (this really is not recommended, as it leads to a very marginal score increase at the cost of potentially strongly biasing the sample of asteroids detected).

Finally, sources with velocity in the RA direction less than -0.4 degrees/day are considered NEO. Obviously this categorization isn't great, but it's better than nothing.

2 Overview of Implementation

The implementation of such an algorithm is fairly straightforward because it is just a bunch of sequential steps, and most are fairly easy to code in a way that uses time and space efficiently. I have included an explanation of the job of each class I used in the next section (which would be helpful if you want to start reading through code). These are mostly helper classes because I feel the main part of the code has been documented above. Note that a good amount of code is obsolete, meaning it is included in my solution but is not being used because it was made to solve the alignment problems that were fixed mid-contest. So if you are puzzled by a particular function, first check to see that it is actually used.

Next, Section 2.2 has some important assorted implementation details for less obvious parts of the code.

2.1 Classes

`Vector` Class Represents a fixed size vector in $\mathbb{R}^n$ (for doing math) or $\mathbb{Z}^n$ (for indexing pixels in images) and allows you to add, subtract, multiply, and return these vectors. Be careful not to confuse my `Vector` class (which is a vector in the mathematical sense) with the lowercase C++ STL `vector` class which I also use frequently, but which represents an expandable array.

`Matrix` Class Represents a standard matrix.

`Transformation` Class This was left over from older versions of the code and I never bothered to completely get rid of it, so it is unnecessary and rarely used. It represents a transformation (linear and translation) using homogeneous coordinates and is based on the `Matrix` class. It can be applied to `Vectors` using multiplication just like matrices.

`Image` Class Stores an image as a `vector<int>` and provides some 2D accessors into the 1D array. For debugging purposes it also includes a `saveAs` feature to save images in a raw byte format readable by Photoshop.

`Detection` Class Represents a detection in an image and the parameters associated with that detection. It is also responsible for doing the Gauss-Newton minimization described in the previous section to fine tune the position of detections

`DetectionSet` Class Stores a set of 4 `Detection` objects (one from each image) and stores the parameters related to the set of all 4 detections, such as the `error`.

`Header` Class Reads and parses the provided header format so that any field can be accessed by tag (used to read the `"MJD"` of each image to determine the time it was taken)

`WCSTransformation` Class Stores the WCS data for a single image internally and provides functions to convert between pixel coordinates and RA/Dec coordinates using the code provided in the Java Tester.

2.2 Assorted Implementation Details

The linear regression fitting was done by setting the derivative of [the sum of square error between the function and the pixel values of the 5x5 grid] to equal 0 and solving the general case. The results in a system of two equations in two unknowns, which can be solved for a formula that was hardcoded into my code. However, a standard linear regression formula would yield the same result.

In many cases it was important to switch from `Vector` based image indexing to raw `int` based indexing because of performance issues. The point being that it is best to avoid using a wrapper class in inner loops as the compiler is unable to "optimize the warper out." This change led to an approximately 10x speed up, which meant my entire solution runs in 20 sec per set of images, which makes debugging and trial and error much more practical.

Finally, you may have seen that many times while describing my algorithm, I reference finding detections in a certain pixel window. Instead of searching through all the detections each time, I do this more efficiently by creating an `Image` (ie. 2D array) of `Detection*` variables. Then each pixel or element of the array (which is a `Detection*`) is set to the address of a detection if one resides somewhere in that pixel, or otherwise it is set to `null` to indicate no detection there. This makes searches for detections within certain rectangular bounds very fast by just searching through all pixels in that rectangle rather than every single detection. (While this technique limits me to one detection per pixel, this is not any issue because I should not have multiple detections in a pixel because of the way detections are generated)

3 Parameter Derivation

Most parameters were set by hand, using methods that are not worth discussing in detail (such as trial and error, and guessing and being close enough). The few exceptions to this include the following parameters

Asteroid Min and Max Movement Were tuned by looking at images that had been aligned by eye and noticing that most asteroids moved at least 2 pixels over all 4 frames, and at most 5 pixels per frame. This lead to the 5 pixel search radius and the 70 pixels/day minimum motion limits

Gaussian Standard Deviation The standard deviation 1.05 was found using the attached Mathematica file (`PSF.nb`) by fitting a Gaussian to a small 9x9 cropped image of a handpicked sample star from one of the training image (`PSF.png`). (However, I bet NASA knows the true point spread function function of their telescopes and can therefore use this function instead of a Gaussian)

Velocity Bounds These were computed by plotting the velocity (in RA/Dec space) of all the training objects using Mathematica (code: `NEO.nb`) (data: `detections.csv`). The plot allowed me to create the rough velocity bounds for both rejection and NEO classification by looking at the graphs by eye. (Again, I do not recommend this because it was a hack with positive—but very small—benefit)